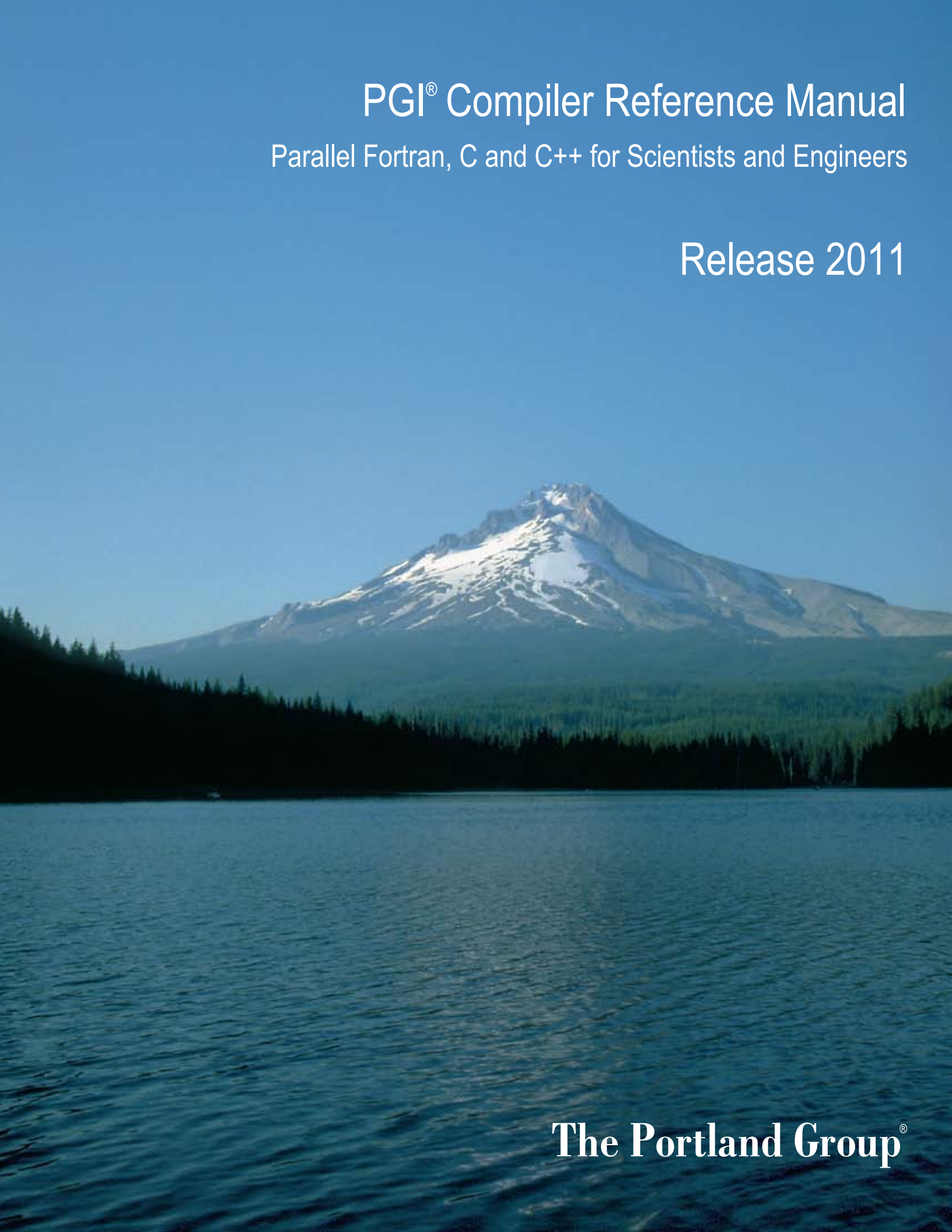
PGI® Compiler Reference Manual

Parallel Fortran, C and C++ for Scientists and Engineers

Release 2011



The Portland Group®

PGI® Compiler Reference Manual

Printed in the United States of America
First printing: Release 2011, 11.0, December, 2010
Second Printing: Release 2011, 11.1, January 2011
Third Printing: Release 2011, 11.3, March 2011
Fourth Printing: Release 2011, 11.4, April 2011
Fifth Printing: Release 2011, 11.5, May 2011

Technical support: trs@pgroup.com
Sales: sales@pgroup.com
Web: www.pgroup.com

ID: 111162228

Contents

Figures

Tables

Examples

Preface

This guide is part of a set of manuals that describe how to use The Portland Group (PGI) Fortran, C, and C++ compilers and program development tools. These compilers and tools include the *PGF77*, *PGF95*, *PGFORTRAN*, *PGHPF*, *PGC++*, and *PGCC ANSI* C compilers, the *PGPROF* profiler, and the *PGDBG* debugger. They work in conjunction with an x86 or x64 assembler and linker. You can use the PGI compilers and tools to compile, debug, optimize, and profile serial and parallel applications for x86 processor-based systems.

The *PGI Compiler Reference Manual* is the reference companion to the *PGI Compiler User's Guide* which provides operating instructions for the PGI command-level development environment. It also contains details concerning the PGI compilers' interpretation of the Fortran language, implementation of Fortran language extensions, and command-level compilation. Users are expected to have previous experience with or knowledge of the Fortran programming language. Neither guide teaches the Fortran programming language.

Audience Description

This manual is intended for scientists and engineers using the PGI compilers. To use these compilers, you should be aware of the role of high-level languages, such as Fortran, C, and C++, as well as assembly-language in the software development process; and you should have some level of understanding of programming. The PGI compilers are available on a variety of x86 or x64 hardware platforms and operating systems. You need to be familiar with the basic commands available on your system.

Compatibility and Conformance to Standards

Your system needs to be running a properly installed and configured version of the compilers. For information on installing PGI compilers and tools, refer to the Release Notes and Installation Guide included with your software.

For further information, refer to the following:

- *American National Standard Programming Language FORTRAN*, ANSI X3. -1978 (1978).
- *ISO/IEC 1539-1 : 1991, Information technology – Programming Languages – Fortran,* Geneva, 1991 (Fortran 90).
- *ISO/IEC 1539-1 : 1997, Information technology – Programming Languages – Fortran,* Geneva, 1997 (Fortran 95).

- *Fortran 95 Handbook Complete ISO/ANSI Reference*, Adams et al, The MIT Press, Cambridge, Mass, 1997.
- *High Performance Fortran Language Specification*, Revision 1.0, Rice University, Houston, Texas (1993), http://www.crpc.rice.edu/HPFF.
- *High Performance Fortran Language Specification*, Revision 2.0, Rice University, Houston, Texas (1997), http://www.crpc.rice.edu/HPFF.
- *OpenMP Application Program Interface*, Version 2.5, May 2005, http://www.openmp.org.
- *Programming in VAX Fortran*, Version 4.0, Digital Equipment Corporation (September, 1984).
- *IBM VS Fortran*, IBM Corporation, Rev. GC26-4119.
- Military Standard, Fortran, DOD Supplement to American National Standard Programming Language Fortran, ANSI x.3-1978, MIL-STD-1753 (November 9, 1978).
- *American National Standard Programming Language C*, ANSI X3.159-1989.
- ISO/IEC 9899:1999, Information technology – Programming Languages – C, Geneva, 1999 (C99).

Organization

Users typically begin by wanting to know how to use a product and often then find that they need more information and facts about specific areas of the product. Knowing *how* as well as *why* you might use certain options or perform certain tasks is key to using the PGI compilers and tools effectively and efficiently. However, once you have this knowledge and understanding, you very likely might find yourself wanting to know much more about specific areas or specific topics.

To facilitate ease of use, this manual contains detailed reference information about specific aspects of the compiler, such as the details of compiler options, directives, and more. This guide contains these chapters:

Chapter 1, "*Fortran, C, and C++ Data Types*" describes the data types that are supported by the PGI Fortran, C, and C++ compilers.

Chapter 2, "*Command-Line Options Reference*" provides a detailed description of each command-line option.

Chapter 3, "*OpenMP Reference Information*" contains detailed descriptions of each of the OpenMP directives and pragmas that PGI supports.

Chapter 4, "*PGI Accelerator Compilers Reference* "contains detailed descriptions of each of the PGI Accelerator directives, runtime routines, and environment variables that PGI supports.

Chapter 5, "*C++ Name Mangling*" describes the name mangling facility and explains the transformations of names of entities to names that include information on aspects of the entity's type and a fully qualified name.

Chapter 6, "*Directives and Pragmas Reference*" contains detailed descriptions of PGI's proprietary directives and pragmas.

Chapter 7, "*Run-time Environment*" describes the programming model supported for compiler code generation, including register conventions and calling conventions for x86 and x64 processor-based systems.

Chapter 8, "*C++ Dialect Supported*" lists more details of the version of the C++ language that PGC++ supports.

Chapter 9, "*Fortran Module/Library Interfaces for Windows*" provides a description of the Fortran module library interfaces that PVF supports.

Chapter 10, "*C/C++ MMX/SSE Inline Intrinsics*" provides tables that list the MMX Inline Intrinsics (mmintrin.h), the SSE1 inline intrinsics (xmmintrin.h), and SSE2 inline intrinsics (emmintrin.h).

Chapter 11, "*Messages*" provides a list of compiler error messages.

Hardware and Software Constraints

This guide describes versions of the PGI compilers that produce assembly code for x86 and x64 processor-based systems. Details concerning environment-specific values and defaults and system-specific features or limitations are presented in the release notes delivered with the PGI compilers.

Conventions

The *PGI Compiler Reference Manual* uses the following conventions:

italic
: Italic font is for emphasis.

`Constant Width`
: Constant width font is for commands, filenames, directories, examples and for language statements in the text, including assembly language statements.

[item1]
: Square brackets indicate optional items. In this case item1 is optional.

{ item2 | item 3}
: Braces indicate that a selection is required. In this case, you must select either item2 or item3.

filename...
: Ellipsis indicate a repetition. Zero or more of the preceding item may occur. In this example, multiple filenames are allowed.

FORTRAN
: Fortran language statements are shown in the text of this guide using upper-case characters and a reduced point size.

The PGI compilers and tools are supported on both 32-bit and 64-bit variants of Linux, Mac OS X, and Windows operating systems on a variety of x86-compatible processors. There are a wide variety of releases and distributions of each of these types of operating systems. Further, the *PGI Compiler Reference Manual* uses a number of terms with respect to these platforms.

For a complete definition of these terms and other terms in this guide with which you may be unfamiliar, PGI provides a glossary of terms which you can access at *www.pgroup.com/support/definitions.htm*.

AMD64	linux86	osx86	static linking
barcelona	linux86-64	osx86-64	Win32
DLL	Mac OS X	shared library	Win64
driver	-mcmodel=small	SSE	Windows
dynamic library	-mcmodel=medium	SSE1	x64
EM64T	MPI	SSE2	x86
hyperthreading (HT)	MPICH	SSE3	x87
IA32	multi-core	SSE4A and ABM	
Large arrays	NUMA	SSSE3	

The following table lists the PGI compilers and tools and their corresponding commands:

Table 1. PGI Compilers and Commands

Compiler or Tool	Language or Function	Command
PGF77	FORTRAN 77	pgf77
PGF95	Fortran 90/95/2003	pgf95
PGFORTRAN	PGI Fortran	pgfortran
PGHPF	High Performance Fortran	pghpf
PGCC C	*ANSI* C99 and K&R C	pgcc
PGC++	*ANSI* C++ with cfront features	pgcpp on Windows pgCC on Linux
PGDBG	Source code debugger	pgdbg
PGPROF	Performance profiler	pgprof

Note

The commands **pgf95** and **pgfortran** are equivalent.

In general, the designation *PGI Fortran* is used to refer to The Portland Group's Fortran 90/95/2003 compiler, and *pgfortran* is used to refer to the command that invokes the compiler. A similar convention is used for each of the PGI compilers and tools.

For simplicity, examples of command-line invocation of the compilers generally reference the *pgfortran* command, and most source code examples are written in Fortran. Usage of the *PGF77* compiler, whose features are a subset of *PGF95* or *PGFORTRAN*, is similar. Usage of *PGHPF, PGC++*, and *PGCC* is consistent with *PGF95, PGFORTRAN*, and *PGF77*, though there are command-line options and features of these compilers that do not apply to *PGF95, PGFORTRAN*, and *PGF77*, and vice versa.

There are a wide variety of x86-compatible processors in use. All are supported by the PGI compilers and tools. Most of these processors are forward-compatible, but not backward-compatible, meaning that code compiled to target a given processor will not necessarily execute correctly on a previous-generation processor.

A table listing the processor options that PGI supports is available in the Release Notes. The table also includes the features utilized by the PGI compilers that distinguish them from a compatibility standpoint.

In this manual, the convention is to use "x86" to specify the group of processors that are "32-bit" but not "64-bit." The convention is to use "x64" to specify the group of processors that are both "32-bit" and "64-bit." x86 processor-based systems can run only 32-bit operating systems. x64 processor-based systems can run either 32-bit or 64-bit operating systems, and can execute all 32-bit x86 binaries in either case. x64 processors have additional registers and 64-bit addressing capabilities that are utilized by the PGI compilers and tools when running on a 64-bit operating system. The prefetch, SSE1, SSE2 and SSE3 processor features further distinguish the various processors. Where such distinctions are important with respect to a given compiler option or feature, it is explicitly noted in this manual.

Note

The default for performing scalar floating-point arithmetic is to use SSE instructions on targets that support SSE1 and SSE2.

Related Publications

The following documents contain additional information related to the x86 and x64 architectures, and the compilers and tools available from The Portland Group.

- *PGI Fortran Reference* manual describes the FORTRAN 77, Fortran 90/95, Fortran 2003, and HPF statements, data types, input/output format specifiers, and additional reference material related to use of the PGI Fortran compilers.
- *System V Application Binary Interface Processor Supplement* by AT&T UNIX System Laboratories, Inc. (Prentice Hall, Inc.).
- *System V Application Binary Interface X86-64 Architecture Processor Supplement*, *www.x86-64.org/abi.pdf*.
- *Fortran 95 Handbook Complete ISO/ANSI Reference*, Adams et al, The MIT Press, Cambridge, Mass, 1997.
- *Programming in VAX Fortran, Version 4.0,* Digital Equipment Corporation (September, 1984).
- *IBM VS Fortran,* IBM Corporation, Rev. GC26-4119.
- *The C Programming Language* by Kernighan and Ritchie (Prentice Hall).
- *C: A Reference Manual* by Samuel P. Harbison and Guy L. Steele Jr. (Prentice Hall, 1987).
- *The Annotated C++ Reference Manual* by Margaret Ellis and Bjarne Stroustrup, AT&T Bell Laboratories, Inc. (Addison-Wesley Publishing Co., 1990).
- *OpenMP Application Program Interface*, Version 2.5 May 2005 (OpenMP Architecture Review Board, 1997-2005).

Chapter 1. Fortran, C, and C++ Data Types

This chapter describes the scalar and aggregate data types recognized by the PGI Fortran, C, and C++ compilers, the format and alignment of each type in memory, and the range of values each type can have on x86 or x64 processor-based systems running a 32-bit operating system. For more information on x86-specific data representation, refer to the System V Application Binary Interface, Processor Supplement, listed in "Related Publications," on page xix. This chapter specifically does not address x64 processor-based systems running a 64-bit operating system, because the application binary interface (ABI) for those systems is still evolving. For the latest version of the ABI, refer to *www.x86-64.org/abi.pdf*.

Fortran Data Types

Fortran Scalars

A scalar data type holds a single value, such as the integer value 42 or the real value 112.6. The next table lists scalar data types, their size, format and range. Table 1.2, "Real Data Type Ranges," on page 2 shows the range and approximate precision for Fortran real data types. Table 1.3, "Scalar Type Alignment," on page 2 shows the alignment for different scalar data types. The alignments apply to all scalars, whether they are independent or contained in an array, a structure or a union.

Table 1.1. Representation of Fortran Data Types

Fortran Data Type	Format	Range
INTEGER	2's complement integer	-2^{31} to $2^{31}-1$
INTEGER*2	2's complement integer	-32768 to 32767
INTEGER*4	2's complement integer	-2^{31} to $2^{31}-1$
INTEGER*8	2's complement integer	-2^{63} to $2^{63}-1$
LOGICAL	32-bit value	true or false
LOGICAL*1	8-bit value	true or false
LOGICAL*2	16-bit value	true or false

Fortran Data Type	Format	Range
LOGICAL*4	32-bit value	true or false
LOGICAL*8	64-bit value	true or false
BYTE	2's complement	-128 to 127
REAL	Single-precision floating point	10^{-37} to $10^{38(1)}$
REAL*4	Single-precision floating point	10^{-37} to $10^{38(1)}$
REAL*8	Double-precision floating point	10^{-307} to $10^{308(1)}$
DOUBLE PRECISION	Double-precision floating point	10^{-307} to $10^{308\ (1)}$
COMPLEX	Single-precision floating point	10^{-37} to $10^{38\ (1)}$
DOUBLE COMPLEX	Double-precision floating point	10^{-307} to $10^{308\ (1)}$
COMPLEX*16	Double-precision floating point	10^{-307} to $10^{308(1)}$
CHARACTER*n	Sequence of n bytes	

(1) Approximate value

The logical constants `.TRUE.` and `.FALSE.` are all ones and all zeroes, respectively. Internally, the value of a logical variable is true if the least significant bit is one and false otherwise. When the option `-Munixlogical` is set, a logical variable with a non-zero value is true and with a zero value is false.

Note

A variable of logical type may appear in an arithmetic context, and the logical type is then treated as an integer of the same size.

Table 1.2. Real Data Type Ranges

Data Type	Binary Range	Decimal Range	Digits of Precision
REAL	-2^{-126} to 2^{128}	10^{-37} to $10^{38\ (1)}$	7-8
REAL*8	-2^{-1022} to 2^{1024}	10^{-307} to $10^{308\ (1)}$	15-16

Table 1.3. Scalar Type Alignment

This Type...	...Is aligned on this size boundary
LOGICAL*1	1-byte
LOGICAL*2	2-byte
LOGICAL*4	4-byte
LOGICAL*8	8-byte
BYTE	1-byte
INTEGER*2	2-byte
INTEGER*4	4-byte

This Type...	...Is aligned on this size boundary
INTEGER*8	8-byte
REAL*4	4-byte
REAL*8	8-byte
COMPLEX*8	4-byte
COMPLEX*16	8-byte

FORTRAN 77 Aggregate Data Type Extensions

The PGF77 compiler supports de facto standard extensions to FORTRAN 77 that allow for aggregate data types. An aggregate data type consists of one or more scalar data type objects. You can declare the following aggregate data types:

- An **array** consists of one or more elements of a single data type placed in contiguous locations from first to last.
- A **structure** can contain different data types. The members are allocated in the order they appear in the definition but may not occupy contiguous locations.
- A **union** is a single location that can contain any of a specified set of scalar or aggregate data types. A union can have only one value at a time. The data type of the union member to which data is assigned determines the data type of the union after that assignment.

The alignment of an array, a structure or union (an aggregate) affects how much space the object occupies and how efficiently the processor can address members. Arrays use the alignment of their members.

Array types
: align according to the alignment of the array elements. For example, an array of INTEGER*2 data aligns on a 2byte boundary.

Structures and Unions
: align according to the alignment of the most restricted data type of the structure or union. In the next example, the union aligns on a 4byte boundary since the alignment of c, the most restrictive element, is four.

```
STRUCTURE /astr/
UNION
 MAP
 INTEGER*2 a ! 2 bytes
 END MAP
 MAP
 BYTE b ! 1 byte
 END MAP
 MAP
 INTEGER*4 c ! 4 bytes
 END MAP
END UNION
END STRUCTURE
```

Structure alignment can result in unused space called *padding*. Padding between members of the structure is called *internal padding*. Padding between the last member and the end of the space is called *tail padding*.

The offset of a structure member from the beginning of the structure is a multiple of the member's alignment. For example, since an INTEGER*2 aligns on a 2-byte boundary, the offset of an INTEGER*2 member from the beginning of a structure is a multiple of two bytes.

Fortran 90 Aggregate Data Types (Derived Types)

The Fortran 90 standard added formal support for aggregate data types. The TYPE statement begins a derived type data specification or declares variables of a specified user-defined type. For example, the following would define a derived type ATTENDEE:

```
TYPE ATTENDEE
 CHARACTER(LEN=30) NAME
 CHARACTER(LEN=30) ORGANIZATION
 CHARACTER (LEN=30) EMAIL
END TYPE ATTENDEE
```

In order to declare a variable of type ATTENDEE and access the contents of such a variable, code such as the following would be used:

```
TYPE (ATTENDEE) ATTLIST(100)
. . .
ATTLIST(1)%NAME = 'JOHN DOE'
```

C and C++ Data Types

C and C++ Scalars

Table 1.4, "C/C++ Scalar Data Types"lists C and C++ scalar data types, providing their size and format. The alignment of a scalar data type is equal to its size. Table 1.5, "Scalar Alignment," on page 5 shows scalar alignments that apply to individual scalars and to scalars that are elements of an array or members of a structure or union. Wide characters are supported (character constants prefixed with an L). The size of each wide character is 4 bytes.

Table 1.4. C/C++ Scalar Data Types

Data Type	Size (bytes)	Format	Range
unsigned char	1	ordinal	0 to 255
[signed] char	1	2's complement integer	-128 to 127
unsigned short	2	ordinal	0 to 65535
[signed] short	2	2's complement integer	-32768 to 32767
unsigned int	4	ordinal	0 to 2^{32} -1
[signed] int	4	2's complement integer	-2^{31} to 2^{31}-1
[signed] long [int] (32-bit operating systems and win64)	4	2's complement integer	-2^{31} to 2^{31}-1
[signed] long [int] (linux86-64 and sua64)	8	2's complement integer	-2^{63} to 2^{63}-1

Data Type	Size (bytes)	Format	Range
unsigned long [int] (32-bit operating systems and win64)	4	ordinal	0 to 2^{32}-1
unsigned long [int] (linux86-64 and sua64)	8	ordinal	0 to 2^{64}-1
[signed] long long [int]	8	2's complement integer	-2^{63} to 2^{63}-1
unsigned long long [int]	8	ordinal	0 to 2^{64}-1
float	4	IEEE single-precision floating-point	10^{-37} to 10^{38} (1)
double	8	IEEE double-precision floating-point	10^{-307} to 10^{308} (1)
long double	8	IEEE double-precision floating-point	10^{-307} to 10^{308} (1)
bit field(2) (unsigned value)	1 to 32 bits	ordinal	0 to 2^{size}-1, where size is the number of bits in the bit field
bit field(2) (signed value)	1 to 32 bits	2's complement integer	-2^{size-1} to 2^{size-1}-1, where size is the number of bits in the bit field
pointer	4	address	0 to 2^{32}-1
enum	4	2's complement integer	-2^{31} to 2^{31}-1

(1) Approximate value

(2) Bit fields occupy as many bits as you assign them, up to 4 bytes, and their length need not be a multiple of 8 bits (1 byte)

Table 1.5. Scalar Alignment

Data Type	Alignment on this size boundary
char	1-byte boundary, signed or unsigned.
short	2-byte boundary, signed or unsigned.
int	4-byte boundary, signed or unsigned.
enum	4-byte boundary.
pointer	4-byte boundary.
float	4-byte boundary.
double	8-byte boundary.
long double	8-byte boundary.
long [int] 32-bit on Win64	4-byte boundary, signed or unsigned.
long [int] linux86-64, sua64	8-byte boundary, signed or unsigned.

Data Type	Alignment on this size boundary
long long [int]	8-byte boundary, signed or unsigned.

C and C++ Aggregate Data Types

An aggregate data type consists of one or more scalar data type objects. You can declare the following aggregate data types:

array
: consists of one or more elements of a single data type placed in contiguous locations from first to last.

class
: (C++ only) is a class that defines an object and its member functions. The object can contain fundamental data types or other aggregates including other classes. The class members are allocated in the order they appear in the definition but may not occupy contiguous locations.

struct
: is a structure that can contain different data types. The members are allocated in the order they appear in the definition but may not occupy contiguous locations. When a struct is defined with member functions, its alignment rules are the same as those for a class.

union
: is a single location that can contain any of a specified set of scalar or aggregate data types. A union can have only one value at a time. The data type of the union member to which data is assigned determines the data type of the union after that assignment.

Class and Object Data Layout

Class and structure objects with no virtual entities and with no base classes, that is just direct data field members, are laid out in the same manner as C structures. The following section describes the alignment and size of these C-like structures. C++ classes (and structures as a special case of a class) are more difficult to describe. Their alignment and size is determined by compiler generated fields in addition to user-specified fields. The following paragraphs describe how storage is laid out for more general classes. The user is warned that the alignment and size of a class (or structure) is dependent on the existence and placement of direct and virtual base classes and of virtual function information. The information that follows is for informational purposes only, reflects the current implementation, and is subject to change. Do not make assumptions about the layout of complex classes or structures.

All classes are laid out in the same general way, using the following pattern (in the sequence indicated):

- First, storage for all of the direct base classes (which implicitly includes storage for non-virtual indirect base classes as well):
  - When the direct base class is also virtual, only enough space is set aside for a pointer to the actual storage, which appears later.
  - In the case of a non-virtual direct base class, enough storage is set aside for its own non-virtual base classes, its virtual base class pointers, its own fields, and its virtual function information, but no space is allocated for its virtual base classes.

- Next, storage for the class's own fields.
- Next, storage for virtual function information (typically, a pointer to a virtual function table).
- Finally, storage for its virtual base classes, with space enough in each case for its own non-virtual base classes, virtual base class pointers, fields, and virtual function information.

Aggregate Alignment

The alignment of an array, a structure or union (an aggregate) affects how much space the object occupies and how efficiently the processor can address members.

Arrays
: align according to the alignment of the array elements. For example, an array of short data type aligns on a 2-byte boundary.

Structures and Unions
: align according to the most restrictive alignment of the enclosing members. In the following example, the union un1 aligns on a 4-byte boundary since the alignment of c, the most restrictive element, is four:

```
union un1 {
 short a; /* 2 bytes */
 char b; /* 1 byte */
 int c; /* 4 bytes */
};
```

Structure alignment can result in unused space, called padding. Padding between members of a structure is called internal padding. Padding between the last member and the end of the space occupied by the structure is called tail padding. Figure 1.1, "Internal Padding in a Structure," on page 7, illustrates structure alignment. Consider the following structure:

```
struct strc1 {
 char a; /* occupies byte 0 */
 short b; /* occupies bytes 2 and 3 */
 char c; /* occupies byte 4 */
 int d; /* occupies bytes 8 through 11 */
};
```

Figure 1.1. Internal Padding in a Structure

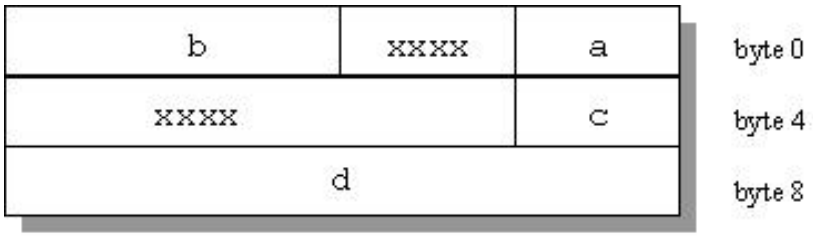


Figure 1.2, "Tail Padding in a Structure," on page 8, shows how tail padding is applied to a structure aligned on a doubleword (8 byte) boundary.

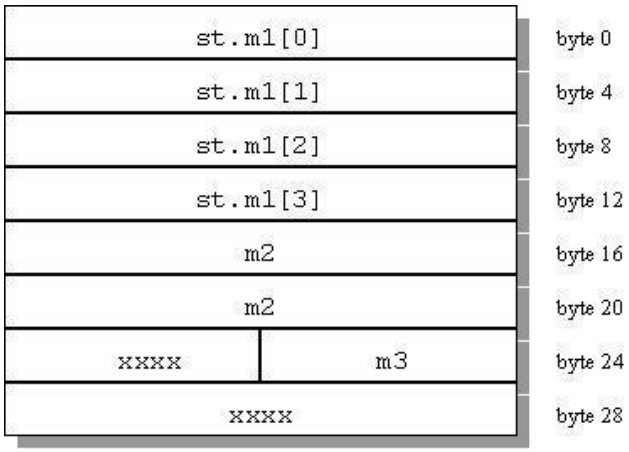
```
struct strc2{
 int m1[4]; /* occupies bytes
0 through 15 */
 double m2; /* occupies bytes 16 through 23 */
 short m3; /* occupies bytes 24 and 25 */
} st;
```

Bit-field Alignment

Bit-fields have the same size and alignment rules as other aggregates, with several additions to these rules:

- Bit-fields are allocated from right to left.
- A bit-field must entirely reside in a storage unit appropriate for its type. Bit-fields never cross unit boundaries.
- Bit-fields may share a storage unit with other structure/union members, including members that are not bit-fields.
- Unnamed bit-field's types do not affect the alignment of a structure or union.
- Items of [signed/unsigned] long long type may not appear in field declarations on 32-bit systems.

Figure 1.2. Tail Padding in a Structure



Other Type Keywords in C and C++

The void data type is neither a scalar nor an aggregate. You can use void or void* as the return type of a function to indicate the function does not return a value, or as a pointer to an unspecified data type, respectively.

The const and volatile type qualifiers do not in themselves define data types, but associate attributes with other types. Use const to specify that an identifier is a constant and is not to be changed. Use volatile to prevent optimization problems with data that can be changed from outside the program, such as memory#mapped I/O buffers.

Chapter 2. Command-Line Options Reference

A command-line option allows you to specify specific behavior when a program is compiled and linked. Compiler options perform a variety of functions, such as setting compiler characteristics, describing the object code to be produced, controlling the diagnostic messages emitted, and performing some preprocessor functions. Most options that are not explicitly set take the default settings. This reference chapter describes the syntax and operation of each compiler option. For easy reference, the options are arranged in alphabetical order.

For an overview and tips on which options are best for which tasks, refer to Chapter 6, "Using Command Line Options" in the PGI Compiler User's Guide, which also provides summary tables of the different options.

This chapter uses the following notation:

[item]
: Square brackets indicate that the enclosed item is optional.

{item | item}
: Braces indicate that you must select one and only one of the enclosed items. A vertical bar (|) separates the choices.

...
: Horizontal ellipses indicate that zero or more instances of the preceding item are valid.

PGI Compiler Option Summary

The following tables include all the PGI compiler options that are not language-specific. The options are separated by category for easier reference.

For a complete description of each option, see the detailed information later in this chapter.

Build-Related PGI Options

The options included in the following table are the ones you use when you are initially building your program or application.

Table 2.1. PGI Build-Related Compiler Options

Option	Description
–#	Display invocation information.
–###	Shows but does not execute the driver commands (same as the option –dryrun).
–Bdynamic	Compiles for and links to the DLL version of the PGI runtime libraries.
–Bstatic_pgi	Compiles for and links to the static version of the PGI runtime libraries.
–c	Stops after the assembly phase and saves the object code in filename.o.
–D<args>	Defines a preprocessor macro.
–dryrun	Shows but does not execute driver commands.
–drystdinc	Displays the standard include directories and then exists the compiler.
–dynamiclib	Invokes the libtool utility program provided by Mac OS X to create the dynamic library. See the libtool man page for more information.
–E	Stops after the preprocessing phase and displays the preprocessed file on the standard output.
–F	Stops after the preprocessing phase and saves the preprocessed file in filename.f (this option is only valid for the PGI Fortran compilers).
--flagcheck	Simply return zero status if flags are correct.
–flags	Display valid driver options.
–fpic	(Linux and Mac OS X only) Generate position-independent code.
–fPIC	(Linux and Mac OS X only) Equivalent to –fpic.
–G	(Linux only) Passed to the linker. Instructs the linker to produce a shared object file.
–g77libs	(Linux only) Allow object files generated by g77 to be linked into PGI main programs.
–help	Display driver help message.
–I<dirname>	Adds a directory to the search path for #include files.
–i2, –i4 and –i8	–i2: Treat INTEGER variables as 2 bytes. –i4: Treat INTEGER variables as 4 bytes. –i8: Treat INTEGER and LOGICAL variables as 8 bytes and use 64-bits for INTEGER*8 operations.
–K<flag>	Requests special compilation semantics with regard to conformance to IEEE 754.

Option	Description
--keeplnk	If the compiler generates a temporary indirect file for a long linker command, preserves the temporary file instead of deleting it.
–L<dirname>	Specifies a library directory.
–l<library>	Loads a library.
–m	Displays a link map on the standard output.
–M<pgflag>	Selects variations for code generation and optimization.
–mcmodel=medium	(–tp k8-64 and –tp p7-64 targets only) Generate code which supports the medium memory model in the linux86-64 environment.
–module <moduledir>	(F90/F95/HPF only) Save/search for module files in directory <moduledir>.
–mp[=all, align,bind,[no]numa]	Interpret and process user-inserted shared-memory parallel programming directives (see Chapters 5 and 6).
–noswitcherror	Ignore unknown command line switches after printing an warning message.
–o	Names the object file.
–pc <val>	(–tp px/p5/p6/piii targets only) Set precision globally for x87 floating-point calculations; must be used when compiling the main program. <val> may be one of 32, 64 or 80.
- -pedantic	Prints warnings from included <system header files>
–pg	Instrument the generated executable to produce a gprof-style gmon.out sample-based profiling trace file (–qp is also supported, and is equivalent).
–pgf77libs	Append PGF77 runtime libraries to the link line.
–pgf90libs	Append PGF90/PGF95/PGFORTRAN runtime libraries to the link line.
–R<directory>	(Linux only) Passed to the Linker. Hard code <directory> into the search path for shared object files.
–r	Creates a relocatable object file.
–r4 and –r8	–r4: Interpret DOUBLE PRECISION variables as REAL. –r8: Interpret REAL variables as DOUBLE PRECISION.
–rc file	Specifies the name of the driver's startup file.
–s	Strips the symbol-table information from the object file.
–S	Stops after the compiling phase and saves the assembly–language code in filename.s.
–shared	(Linux only) Passed to the linker. Instructs the linker to generate a shared object file. Implies –fpic.
–show	Display driver's configuration parameters after startup.

Option	Description
–silent	Do not print warning messages.
–soname	Pass the soname option and its argument to the linker.
–time	Print execution times for the various compilation steps.
–tp <target> [,target...]	Specify the type(s) of the target processor(s).
–u<symbol>	Initializes the symbol table with <symbol>, which is undefined for the linker. An undefined symbol triggers loading of the first member of an archive library.
–U<symbol>	Undefine a preprocessor macro.
–V[release_number]	Displays the version messages and other information, or allows invocation of a version of the compiler other than the default.
–v	Displays the compiler, assembler, and linker phase invocations.
–W	Passes arguments to a specific phase.
–w	Do not print warning messages.

PGI Debug-Related Compiler Options

The options included in the following table are the ones you typically use when you are debugging your program or application.

Table 2.2. PGI Debug-Related Compiler Options

Option	Description
–C	(Fortran only) Generates code to check array bounds.
–c	Instrument the generated executable to perform array bounds checking at runtime.
–E	Stops after the preprocessing phase and displays the preprocessed file on the standard output.
--flagcheck	Simply return zero status if flags are correct.
–flags	Display valid driver options.
–g	Includes debugging information in the object module.
–gopt	Includes debugging information in the object module, but forces assembly code generation identical to that obtained when `-gopt` is not present on the command line.
–K<flag>	Requests special compilation semantics with regard to conformance to IEEE 754.
--keeplnk	If the compiler generates a temporary indirect file for a long linker command, preserves the temporary file instead of deleting it.
–M<pgflag>	Selects variations for code generation and optimization.

Option	Description
–Mprof=time	Instrument the generated executable to produce a gprof-style gmon.out sample-based profiling trace file (–qp is also supported, and is equivalent).
–pc <val>	(–tp px/p5/p6/piii targets only) Set precision globally for x87 floating-point calculations; must be used when compiling the main program. <val> may be one of 32, 64 or 80.
–[no]traceback	Adds debug information for runtime traceback for use with the environment variable `PGI_TERM`.

PGI Optimization-Related Compiler Options

The options included in the following table are the ones you typically use when you are optimizing your program or application code.

Table 2.3. Optimization-Related PGI Compiler Options

Option	Description
–fast	Generally optimal set of flags for targets that support SSE capability.
–fastsse	Generally optimal set of flags for targets that include SSE/SSE2 capability.
–M<pgflag>	Selects variations for code generation and optimization.
–mp[=all, align,bind,[no]numa]	Interpret and process user-inserted shared-memory parallel programming directives (see Chapters 5 and 6).
–O<level>	Specifies code optimization level where <level> is 0, 1, 2, 3, or 4.
–pc <val>	(–tp px/p5/p6/piii targets only) Set precision globally for x87 floating-point calculations; must be used when compiling the main program. <val> may be one of 32, 64 or 80.
–Mprof=time	Instrument the generated executable to produce a gprof-style gmon.out sample-based profiling trace file (-qp is also supported, and is equivalent).

PGI Linking and Runtime-Related Compiler Options

The options included in the following table are the ones you typically use to define parameters related to linking and running your program or application code.

Table 2.4. Linking and Runtime-Related PGI Compiler Options

Option	Description
–Bdynamic	Compiles for and links to the DLL version of the PGI runtime libraries.
–Bstatic_pgi	Compiles for and links to the static version of the PGI runtime libraries.

Option	Description
–byteswapio	(Fortran only) Swap bytes from big-endian to little-endian or vice versa on input/output of unformatted data
–fpic	(Linux only) Generate position-independent code.
–fPIC	(Linux only) Equivalent to –fpic.
–G	(Linux only) Passed to the linker. Instructs the linker to produce a shared object file.
–g77libs	(Linux only) Allow object files generated by g77 to be linked into PGI main programs.
–i2	Treat INTEGER variables as 2 bytes.
–i4	Treat INTEGER variables as 4 bytes.
–i8	Treat INTEGER and LOGICAL variables as 8 bytes and use 64-bits for INTEGER*8 operations.
–K<flag>	Requests special compilation semantics with regard to conformance to IEEE 754.
–M<pgflag>	Selects variations for code generation and optimization.
–mcmodel=medium	(–tp k8-64 and –tp p7-64 targets only) Generate code which supports the medium memory model in the linux86-64 environment.
–shared	(Linux only) Passed to the linker. Instructs the linker to generate a shared object file. Implies –fpic.
–soname	Pass the soname option and its argument to the linker.
–ta=nvidia(,nvidia_suboptions),host	Specify the target accelerator.
–tp <target> [,target...]	Specify the type(s) of the target processor(s).

C and C++ Compiler Options

There are a large number of compiler options specific to the PGCC and PGC++ compilers, especially PGC++. The next table lists several of these options, but is not exhaustive. For a complete list of available options, including an exhaustive list of PGC++ options, use the –help command-line option. For further detail on a given option, use –help and specify the option explicitly. The majority of these options are related to building your program or application.

Table 2.5. C and C++ -specific Compiler Options

Option	Description
–A	(pgcpp only) Accept proposed ANSI C++, issuing errors for non-conforming code.
–a	(pgcpp only) Accept proposed ANSI C++, issuing warnings for non-conforming code.

Option	Description
--[no_]alternative_tokens	(pgcpp only) Enable/disable recognition of alternative tokens. These are tokens that make it possible to write C++ without the use of the , , `[, ]`, #, &, and ^ and characters. The alternative tokens include the operator keywords (e.g., and, bitand, etc.) and digraphs. The default is -–no_alternative_tokens.
–B	Allow C++ comments (using //) in C source.
–b	(pgcpp only) Compile with cfront 2.1 compatibility. This accepts constructs and a version of C++ that is not part of the language definition but is accepted by cfront. EDG option.
–b3	(pgcpp only) Compile with cfront 3.0 compatibility. See –b.
--[no_]bool	(pgcpp only) Enable or disable recognition of bool. The default value is —bool.
--[no_]builtin	Do/don't compile with math subroutine builtin support, which causes selected math library routines to be inlined. The default is —builtin.
--cfront_2.1	(pgcpp only) Enable compilation of C++ with compatibility with cfront version 2.1.
--cfront_3.0	(pgcpp only) Enable compilation of C++ with compatibility with cfront version 3.0.
--compress_names	(pgcpp only) Create a precompiled header file with the name filename.
–d<arg>	(pgcc only) Prints additional information from the preprocessor.
--dependencies (see –M)	(pgcpp only) Print makefile dependencies to stdout.
--dependencies_to_file `filename`	(pgcpp only) Print makefile dependencies to file `filename`.
--display_error_number	(pgcpp only) Display the error message number in any diagnostic messages that are generated.
--diag_error tag	(pgcpp only) Override the normal error severity of the specified diagnostic messages.
--diag_remark tag	(pgcpp only) Override the normal error severity of the specified diagnostic messages.
--diag_suppress tag	(pgcpp only) Override the normal error severity of the specified diagnostic messages.
--diag_warning tag	(pgcpp only) Override the normal error severity of the specified diagnostic messages.

Option	Description
-e<number>	(pgcpp only) Set the C++ front-end error limit to the specified <number>.
--[no_]exceptions	(pgcpp only) Disable/enable exception handling support. The default is —-exceptions
--gnu_extensions	(pgcpp only) Allow GNU extensions like "include next" which are required to compile Linux system header files.
--gnu_version <num>	(pgcpp only) Sets the GNU C++ compatibility version.
--[no]llalign	(pgcpp only) Do/don't align longlong integers on integer boundaries. The default is —-llalign.
–M	Generate make dependence lists.
–MD	Generate make dependence lists.
–MD,filename	(pgcpp only) Generate make dependence lists and print them to file filename.
- -microsoft_version <num>	Sets the Microsoft C++ compatibility version.
--optk_allow_dollar_in_id_chars	(pgcpp only) Accept dollar signs in identifiers.
–P	Stops after the preprocessing phase and saves the preprocessed file in filename.i.
--+p	(pgcpp only) Disallow all anachronistic constructs. cfront option
--pch	(pgcpp only) Automatically use and/or create a precompiled header file.
--pch_dir directoryname	(pgcpp only) The directory dirname in which to search for and/or create a precompiled header file.
--[no_]pch_messages	(pgcpp only) Enable/ disable the display of a message indicating that a precompiled header file was created or used.
--preinclude=<filename>	(pgcpp only) Specify file to be included at the beginning of compilation so you can set system-dependent macros, types, and so on.
–t	Control instantiation of template functions. EDG option
--use_pch filename	(pgcpp only) Use a precompiled header file of the specified name as part of the current compilation.
--[no_]using_std	(pgcpp only) Enable/disable implicit use of the std namespace when standard header files are included.
–X	(pgcpp only) Allow $ in names.

Generic PGI Compiler Options

The following descriptions are for all the PGI options. For easy reference, the options are arranged in alphabetical order. For a list of options by tasks, refer to the tables in the beginning of this chapter.

–#

Displays the invocations of the compiler, assembler and linker.

Default: The compiler does not display individual phase invocations.

Usage: The following command-line requests verbose invocation information.

```
$ pgfortran -# prog.f
```

Description: The –# option displays the invocations of the compiler, assembler and linker. These invocations are command-lines created by the driver from your command-line input and the default value.

Related options: –Minfo, –V, –v.

–###

Displays the invocations of the compiler, assembler and linker, but does not execute them.

Default: The compiler does not display individual phase invocations.

Usage: The following command-line requests verbose invocation information.

```
$ pgfortran -### myprog.f
```

Description: Use the –### option to display the invocations of the compiler, assembler and linker but not to execute them. These invocations are command lines created by the compiler driver from the `rc` files and the command-line options.

Related options: –#, –dryrun, –Minfo, –V

–Bdynamic

Compiles for and links to the DLL version of the PGI runtime libraries.

Default: The compiler uses static libraries.

Usage: You can create the DLL `obj1.dll` and its import library `obj1.lib` using the following series of commands:

```
% pgfortran -Bdynamic -c object1.f
% pgfortran -Mmakedll object1.obj -o obj1.dll
```

Then compile the main program using this command:

```
$ pgfortran -# prog.f
```

For a complete example, refer to the example: “Build a DLL: Fortran” in the “Creating and Using Libraries” chapter of the PGI Compiler User’s Guide.

Description: Use this option to compile for and link to the DLL version of the PGI runtime libraries. This flag is required when linking with any DLL built by the PGI compilers. This flag corresponds to the `/MD` flag used by Microsoft's cl compilers.

Note

On Windows, `-Bdynamic` must be used for *both* compiling and linking.

When you use the PGI compiler flag `-Bdynamic` to create an executable that links to the DLL form of the runtime, the executable built is smaller than one built without `-Bdynamic`. The PGI runtime DLLs, however, must be available on the system where the executable is run. The `-Bdynamic` flag must be used when an executable is linked against a DLL built by the PGI compilers.

Note

C++ on Windows does not support `-Bdynamic`.

Related options:–Bstatic, –Mmakedll

–Bstatic

Compiles for and links to the static version of the PGI runtime libraries.

Default: The compiler uses static libraries.

Usage: The following command line explicitly compiles for and links to the static version of the PGI runtime libraries:

```
% pgfortran -Bstatic -c object1.f
```

Description: You can use this option to explicitly compile for and link to the static version of the PGI runtime libraries.

Note

On Windows, `-Bstatic` must be used for *both* compiling and linking.

For more information on using static libraries on Windows, refer to "Creating and Using Static Libraries on Windows" in the "Creating and Using Libraries" chapter of the PGI Compiler User's Guide.

Related options: –Bdynamic, –Bstatic_pgi, –Mdll

–Bstatic_pgi

Linux only. Compiles for and links to the static version of the PGI runtime libraries. Implies `-Mnorpath`.

Default: The compiler uses static libraries.

Usage: The following command line explicitly compiles for and links to the static version of the PGI runtime libraries:

```
% pgfortran -Bstatic -c object1.f
```

Description: You can use this option to explicitly compile for and link to the static version of the PGI runtime libraries.

Note

On Linux, `-Bstatic_pgi` results in code that runs on most Linux systems without requiring a Portability package.

For more information on using static libraries on Windows, refer to "Creating and Using Static Libraries on Windows" in the "Creating and Using Libraries" chapter of the PGI Compiler User's Guide.

Related options: –Bdynamic, –Bstatic, –Mdll

–byteswapio

Swaps the byte-order of data in unformatted Fortran data files on input/output.

Default: The compiler does not byte-swap data on input/output.

Usage: The following command-line requests that byte-swapping be performed on input/output.

```
$ pgfortran -byteswapio myprog.f
```

Description: Use the `-byteswapio` option to swap the byte-order of data in unformatted Fortran data files on input/output. When this option is used, the order of bytes is swapped in both the data and record control words; the latter occurs in unformatted sequential files.

You can use this option to convert big-endian format data files produced by most RISC workstations and high-end servers to the little-endian format used on x86 or x64 systems on the fly during file reads/writes.

This option assumes that the record layouts of unformatted sequential access and direct access files are the same on the systems. It further assumes that the IEEE representation is used for floating-point numbers. In particular, the format of unformatted data files produced by PGI Fortran compilers is identical to the format used on Sun and SGI workstations; this format allows you to read and write unformatted Fortran data files produced on those platforms from a program compiled for an x86 or x64 platform using the `-byteswapio` option.

Related options: None.

–C

Enables array bounds checking. This option only applies to the PGI Fortran compilers.

Default: The compiler does not enable array bounds checking.

Usage: In this example, the compiler instruments the executable produced from `myprog.f` to perform array bounds checking at runtime:

```
$ pgfortran -C myprog.f
```

Description: Use this option to enable array bounds checking. If an array is an assumed size array, the bounds checking only applies to the lower bound. If an array bounds violation occurs during execution, an error message describing the error is printed and the program terminates. The text of the error message includes the name of the array, the location where the error occurred (the source file and the line number in the source), and information about the out of bounds subscript (its value, its lower and upper bounds, and its dimension).

Related options: –M[no]bounds.

–c

Halts the compilation process after the assembling phase and writes the object code to a file.

Default: The compiler produces an executable file (does not use the –c option).

Usage: In this example, the compiler produces the object file `myprog.o` in the current directory.

```
$ pgfortran -c myprog.f
```

Description: Use the –c option to halt the compilation process after the assembling phase and write the object code to a file. If the input file is `filename.f`, the output file is `filename.o`.

Related options: –E, –Mkeepasm, –o, and –S.

–d<arg>

Prints additional information from the preprocessor. [Valid only for c (pgcc)]

Default: No additional information is printed from the preprocessor.

Syntax:

```
-d[D|I|M|N]
```

-dD
: Print macros and values from source files.

-dI
: Print include file names.

-dM
: Print macros and values, including predefined and command-line macros.

-dN
: Print macro names from source files.

Usage: In the following example, the compiler prints macro names from the source file.

```
$ pgfortran -dN myprog.f
```

Description: Use the -d<arg> option to print additional information from the preprocessor.

Related options: –E, –D, –U.

–D

Creates a preprocessor macro with a given value.

Note

You can use the –D option more than once on a compiler command line. The number of active macro definitions is limited only by available memory.

Syntax:

```
-Dname[=value]
```

Where name is the symbolic name and value is either an integer value or a character string.

Default: If you define a macro name without specifying a value, the preprocessor assigns the string 1 to the macro name.

Usage: In the following example, the macro PATHLENGTH has the value 256 until a subsequent compilation. If the –D option is not used, PATHLENGTH is set to 128.

```
$ pgfortran -DPATHLENGTH=256 myprog.F
```

The source text in `myprog.F` is this:

```
#ifndef PATHLENGTH
#define PATHLENGTH 128
#endif
SUBROUTINE SUB
CHARACTER*PATHLENGTH path
...
END
```

Description: Use the –D option to create a preprocessor macro with a given value. The value must be either an integer or a character string.

You can use macros with conditional compilation to select source text during preprocessing. A macro defined in the compiler invocation remains in effect for each module on the command line, unless you remove the macro with an #undef preprocessor directive or with the –U option. The compiler processes all of the –U options in a command line after processing the –D options.

Related options: –U

–dryrun

Displays the invocations of the compiler, assembler, and linker but does not execute them.

Default: The compiler does not display individual phase invocations.

Usage: The following command line requests verbose invocation information.

```
$ pgfortran -dryrun myprog.f
```

Description: Use the `-dryrun` option to display the invocations of the compiler, assembler, and linker but not have them executed. These invocations are command lines created by the compiler driver from the `rc` files and the command-line supplied with `-dryrun`.

Related options: –Minfo, –V, –###

–drystdinc

Displays the standard include directories and then exits the compiler.

Default: The compiler does not display standard include directores.

Usage: The following command line requests a display for the standard include directories.

```
$ pgfortran -drystdinc myprog.f
```

Description: Use the `–drystdinc` option to display the standard include directories and then exit the compiler.

Related options: None.

–dynamiclib

Invokes the `libtool` utility program provided by Mac OS X to so you can create a dynamic library.

Default: The compiler does not invoke the libtool utility.

Usage: The following command line builds a dynamic library:

```
% pgfortran -dynamiclib world.f90 -o world.dylib
```

Description: Use the –dynamiclib option to invoke the `libtool` utility program provided by Mac OS X to so you can create a dynamic library. For a complete example, refer to "Creating and Using Dynamic Libraries on Mac OS X" on page 289.

For more information on `libtool`, refer to the `libtool` man page.

Related options: –Bdynamic, –Bstatic

–E

Halts the compilation process after the preprocessing phase and displays the preprocessed output on the standard output.

Default: The compiler produces an executable file.

Usage: In the following example the compiler displays the preprocessed `myprog.f` on the standard output.

```
$ pgfortran -E myprog.f
```

Description: Use the –E option to halt the compilation process after the preprocessing phase and display the preprocessed output on the standard output.

Related options: –C, –c, –Mkeepasm, –o, –F, –S.

–F

Stops compilation after the preprocessing phase.

Default: The compiler produces an executable file.

Usage: In the following example the compiler produces the preprocessed file `myprog.f` in the current directory.

```
$ pgfortran -F myprog.F
```

Description: Use the –F option to halt the compilation process after preprocessing and write the preprocessed output to a file. If the input file is `filename.F`, then the output file is `filename.f`.

Related options: –c,–E, –Mkeepasm, –o, –S

–fast

Enables vectorization with SSE instructions, cache alignment, and flushz for 64-bit targets.

Default: The compiler enables vectorization with SSE instructions, cache alignment, and flushz.

Usage: In the following example the compiler produces vector SSE code when targeting a 64-bit machine.

```
$ pgfortran -fast vadd.f95
```

Description: When you use this option, a generally optimal set of options is chosen for targets that support SSE capability. In addition, the appropriate `–tp` option is automatically included to enable generation of code optimized for the type of system on which compilation is performed. This option enables vectorization with SSE instructions, cache alignment, and flushz.

Note

Auto-selection of the appropriate `–tp` option means that programs built using the `–fastsse` option on a given system are not necessarily backward-compatible with older systems.

Note

C/C++ compilers enable `–Mautoinline` with `–fast`.

Related options: –O, –Munroll, –Mnoframe, –Mscalarsse, –Mvect, –Mcache_align, –tp, -M[no]autoinline

–fastsse

Synonymous with –fast.

--flagcheck

Causes the compiler to check that flags are correct then exit without any compilation occuring.

Default: The compiler begins a compile without the additional step to first validate that flags are correct.

Usage: In the following example the compiler checks that flags are correct, and then exits.

```
$ pgfortran --flagcheck myprog.f
```

Description: Use this option to make the compiler check that flags are correct and then exit. If flags are all correct then the compiler returns a zero status. No compilation occurs.

Related options: None

–flags

Displays driver options on the standard output.

Default: The compiler does not display the driver options.

Usage: In the following example the user requests information about the known switches.

```
$ pgfortran -flags
```

Description: Use this option to display driver options on the standard output. When you use this option with –v, in addition to the valid options, the compiler lists options that are recognized and ignored.

Related options: –#, –###, –v

–fpic

(Linux only) Generates position-independent code suitable for inclusion in shared object (dynamically linked library) files.

Default: The compiler does not generate position-independent code.

Usage: In the following example the resulting object file, `myprog.o`, can be used to generate a shared object.

```
$ pgfortran -fpic myprog.f
```

(Linux only) Use the -fpic option to generate position-independent code suitable for inclusion in shared object (dynamically linked library) files.

Related options: –shared, –fPIC, –G, –R

–fPIC

(Linux only) Equivalent to –fpic. Provided for compatibility with other compilers.

–G

(Linux only) Instructs the linker to produce a shared object file.

Default: The compiler does not instruct the linker to produce a shared object file.

Usage: In the following example the linker produces a shared object file.

```
$ pgfortran -G myprog.f
```

Description: (Linux only) Use this option to pass information to the linker that instructs the linker to produce a shared object file.

Related options: –fpic, –shared, –R

–g

Instructs the compiler to include symbolic debugging information in the object module.

Default: The compiler does not put debugging information into the object module.

Usage: In the following example, the object file `myprog.o` contains symbolic debugging information.

```
$ pgfortran -c -g myprog.f
```

Description: Use the –g option to instruct the compiler to include symbolic debugging information in the object module. Debuggers, such as *PGDBG*, require symbolic debugging information in the object module to display and manipulate program variables and source code.

If you specify the –g option on the command-line, the compiler sets the optimization level to –O0 (zero), unless you specify the –O option. For more information on the interaction between the –g and –O options, see the –O entry. Symbolic debugging may give confusing results if an optimization level other than zero is selected.

Note

Including symbolic debugging information increases the size of the object module.

Related options: –O, –gopt

–gopt

Instructs the compiler to include symbolic debugging information in the object file, and to generate optimized code identical to that generated when –g is not specified.

Default: The compiler does not put debugging information into the object module.

Usage: In the following example, the object file`myprog.o` contains symbolic debugging information.

```
$ pgfortran -c -gopt myprog.f
```

Description: Using `–g` alters how optimized code is generated in ways that are intended to enable or improve debugging of optimized code. The `–gopt` option instructs the compiler to include symbolic debugging information in the object file, and to generate optimized code identical to that generated when –g is not specified.

Related options: –g, –M<pgflag>

–g77libs

(Linux only) Used on the link line, this option instructs the pgfortran driver to search the necessary g77 support libraries to resolve references specific to g77 compiled program units.

Note

The g77 compiler must be installed on the system on which linking occurs in order for this option to function correctly.

Default: The compiler does not search g77 support libraries to resolve references at link time.

Usage: The following command-line requests that g77 support libraries be searched at link time:

```
$ pgfortran -g77libs myprog.f g77_object.o
```

Description: (Linux only) Use the –g77libs option on the link line if you are linking g77-compiled program units into a pgfortran-compiled main program using the pgfortran driver. When this option is present, the pgfortran driver searches the necessary g77 support libraries to resolve references specific to g77 compiled program units.

Related options:–pgf77libs

–help

Used with no other options, –help displays options recognized by the driver on the standard output. When used in combination with one or more additional options, usage information for those options is displayed to standard output.

Default: The compiler does not display usage information.

Usage: In the following example, usage information for `-Minline` is printed to standard output.

```
$ pgcc -help -Minline
-Minline[=lib:<inlib>|<func>|except:<func>|
name:<func>|size:<n>|levels:<n>]
Enable function inlining
lib:<extlib> Use extracted functions from extlib
<func> Inline function func
except:<func> Do not inline function func
name:<func> Inline function func
size:<n> Inline only functions smaller than n
levels:<n> Inline n levels of functions
-Minline Inline all functions that were extracted
```

In the following example, usage information for –help shows how groups of options can be listed or examined according to function.

```
$ pgcc -help -help
-help[=groups|asm|debug|language|linker|opt|other|
 overall|phase|prepro|suffix|switch|target|variable]
```

Description: Use the –help option to obtain information about available options and their syntax. You can use –help in one of three ways:

- Use `-help` with no parameters to obtain a list of all the available options with a brief one-line description of each.
- Add a parameter to –help to restrict the output to information about a specific option. The syntax for this usage is this:

  ```
  -help <command line option>
  ```

- Add a parameter to –help to restrict the output to a specific set of options or to a building process. The syntax for this usage is this:

  ```
  -help=<subgroup>
  ```

The following table lists and describes the subgroups available with –help.

Table 2.6. Subgroups for –help Option

Use this –help option	To get this information...
–help=asm	A list of options specific to the assembly phase.
–help=debug	A list of options related to debug information generation.
–help=groups	A list of available switch classifications.
–help=language	A list of language-specific options.
–help=linker	A list of options specific to link phase.
–help=opt	A list of options specific to optimization phase.
–help=other	A list of other options, such as ANSI conformance pointer aliasing for C.
–help=overall	A list of options generic to any PGI compiler.
–help=phase	A list of build process phases and to which compiler they apply.

Use this –help option	To get this information...
–help=prepro	A list of options specific to the preprocessing phase.
–help=suffix	A list of known file suffixes and to which phases they apply.
–help=switch	A list of all known options; this is equivalent to usage of –help without any parameter.
–help=target	A list of options specific to target processor.
–help=variable	A list of all variables and their current value. They can be redefined on the command line using syntax VAR=VALUE.

For more examples of –help, refer to "Help with Command-line Options" on page 103.

Related options: –#, –###, –show, –V, –flags

–I

Adds a directory to the search path for files that are included using either the INCLUDE statement or the preprocessor directive #include.

Default: The compiler searches only certain directories for included files.

- For gcc-lib includes: `/usr/lib64/gcc-lib`
- For system includes: `/usr/linclude`

Syntax:

```
-Idirectory
```

Where directory is the name of the directory added to the standard search path for include files.

Usage: In the following example, the compiler first searches the directory `mydir` and then searches the default directories for include files.

```
$ pgfortran -Imydir
```

Description: Adds a directory to the search path for files that are included using the INCLUDE statement or the preprocessor directive #include. Use the –I option to add a directory to the list of where to search for the included files. The compiler searches the directory specified by the –I option before the default directories.

The Fortran INCLUDE statement directs the compiler to begin reading from another file. The compiler uses two rules to locate the file:

1. If the file name specified in the INCLUDE statement includes a path name, the compiler begins reading from the file it specifies.
2. If no path name is provided in the INCLUDE statement, the compiler searches (in order):
   - Any directories specified using the –I option (in the order specified)
   - The directory containing the source file
   - The current directory

For example, the compiler applies rule (1) to the following statements:

```
INCLUDE '/bob/include/file1' (absolute path name)
INCLUDE '../../file1' (relative path name)
```

and rule (2) to this statement:

```
INCLUDE 'file1'
```

Related options: –Mnostdinc

–i2, –i4 and –i8

Treat INTEGER and LOGICAL variables as either two, four, or eight bytes.

Default: The compiler treats INTERGER and LOGICAL variables as four bytes.

Usage: In the following example, using the –i8 switch causes the integer variables to be treated as 64 bits.

```
$ pgfortran -i8 int.f
```

`int.f` is a function similar to this:

```
int.f
 print *, "Integer size:", bit_size(i)
 end
```

Description: Use this option to treat INTEGER and LOGICAL variables as either two, four, or eight bytes. INTEGER*8 values not only occupy 8 bytes of storage, but operations use 64 bits, instead of 32 bits.

Related options: None

–K<flag>

Requests that the compiler provide special compilation semantics.

Default: The compiler does not provide special compilation semantics.

Syntax:

–K<flag>

Where flag is one of the following:

ieee	Perform floating-point operations in strict conformance with the IEEE 754 standard. Some optimizations are disabled, and on some systems a more accurate math library is linked if –Kieee is used during the link step.
noieee	Default flag. Use the fastest available means to perform floating-point operations, link in faster non-IEEE libraries if available, and disable underflow traps.
PIC	(Linux only) Generate position-independent code. Equivalent to –fpic. Provided for compatibility with other compilers.
pic	(Linux only) Generate position-independent code. Equivalent to –fpic. Provided for compatibility with other compilers.

trap=option [,option]...
: Controls the behavior of the processor when floating-point exceptions occur. Possible options include:
  - fp
  - align (ignored)
  - inv
  - denorm
  - divz
  - ovf
  - unf
  - inexact

Usage: In the following example, the compiler performs floating-point operations in strict conformance with the IEEE 754 standard

```
$ pgfortran -Kieee myprog.f
```

Description: Use `-K` to instruct the compiler to provide special compilation semantics.

The default is `-Knoieee`.

`-Ktrap` is only processed by the compilers when compiling main functions or programs. The options inv, denorm, divz, ovf, unf, and inexact correspond to the processor's exception mask bits: invalid operation, denormalized operand, divide-by-zero, overflow, underflow, and precision, respectively. Normally, the processor's exception mask bits are *on*, meaning that floating-point exceptions are masked—the processor recovers from the exceptions and continues. If a floating-point exception occurs and its corresponding mask bit is *off*, or "unmasked", execution terminates with an arithmetic exception (C's SIGFPE signal).

`-Ktrap=fp` is equivalent to `-Ktrap=inv,divz,ovf`.

> **Note**
>
> The PGI compilers do not support exception-free execution for`-Ktrap=inexact`. The purpose of this hardware support is for those who have specific uses for its execution, along with the appropriate signal handlers for handling exceptions it produces. It is not designed for normal floating point operation code support.

Related options: None.

--keeplnk

(Windows only.) Preserves the temporary file when the compiler generates a temporary indirect file for a long linker command.

Usage: In the following example the compiler preserves each temporary file rather than deleting it.

```
$ pgfortran --keeplnk myprog.f
```

Description: If the compiler generates a temporary indirect file for a long linker command, use this option to instruct the compiler to preserve the temporary file instead of deleting it.

Related options: None.

–L

Specifies a directory to search for libraries.

> **Note**
>
> Multiple –L options are valid. However, the position of multiple –L options is important relative to –l options supplied.

Syntax:

```
-Ldirectory
```

Where `directory` is the name of the library directory.

Default: The compiler searches the standard library directory.

Usage: In the following example, the library directory is `/lib` and the linker links in the standard libraries required by PGFORTRAN from this directory.

```
$ pgfortran -L/lib myprog.f
```

In the following example, the library directory `/lib` is searched for the library file `libx.a` and both the directories `/lib` and `/libz` are searched for `liby.a`.

```
$ pgfortran -L/lib -lx -L/libz -ly myprog.f
```

Use the –L option to specify a directory to search for libraries. Using –L allows you to add directories to the search path for library files.

Related options: -l

–l<library>

Instructs the linker to load the specified library. The linker searches <library> in addition to the standard libraries.

> **Note**
>
> The linker searches the libraries specified with –l in order of appearance *before* searching the standard libraries.

Syntax:

```
-llibrary
```

Where library is the name of the library to search.

Usage: In the following example, if the standard library directory is `/lib` the linker loads the library `/lib/libmylib.a`, in addition to the standard libraries.

```
$ pgfortran myprog.f -lmylib
```

Description: Use this option to instruct the linker to load the specified library. The compiler prepends the characters lib to the library name and adds the .a extension following the library name. The linker searches each library specifies before searching the standard libraries.

Related options: –L

–m

Displays a link map on the standard output.

Default: The compiler does display the link map and does not use the –m option.

Usage: When the following example is executed on Windows, pgfortran creates a link map in the file `myprog.map`.

```
$ pgfortran -m myprog.f
```

Description: Use this option to display a link map.

- On Linux, the map is written to `stdout`.
- On Windows, the map is written to a `.map` file whose name depends on the executable. If the executable is `myprog.f`, the map file is in `myprog.map`.

Related options: –c, –o, -s, –u

–m32

Use the 32-bit compiler for the default processor type.

Usage: When the following example is executed on Windows, pgfortran uses the 32-bit compiler for the default processor type.

```
$ pgfortran -m32 myprog.f
```

Description: Use this option to specify the 32-bit compiler as the default processor type.

–m64

Use the 64-bit compiler for the default processor type.

Usage: When the following example is executed on Windows, pgfortran uses the 64-bit compiler for the default processor type.

```
$ pgfortran -m64 myprog.f
```

Description: Use this option to specify the 64-bit compiler as the default processor type.

–M<pgflag>

Selects options for code generation. The options are divided into the following categories:

Code generation	Fortran Language Controls	Optimization
Environment	C/C++ Language Controls	Miscellaneous
Inlining		

The following table lists and briefly describes the options alphabetically and includes a field showing the category. For more details about the options as they relate to these categories, refer to "–M Options by Category" on page 113.

Table 2.7. –M Options Summary

pgflag	Description	Category
allocatable=95|03	Controls whether to use Fortran 95 or Fortran 2003 semantics in allocatable array assignments.	Fortran Language
anno	Annotate the assembly code with source code.	Miscellaneous
[no]autoinline	C/C++ when a function is declared with the inline keyword, inline it at –O2 and .	Inlining
[no]asmkeyword	Specifies whether the compiler allows the asm keyword in C/C++ source files (pgcc and pgcpp only).	C/C++ Language
[no]backslash	Determines how the backslash character is treated in quoted strings (pgf77, pgf95, pgfortran, and pghpf only).	Fortran Language
[no]bounds	Specifies whether array bounds checking is enabled or disabled.	Miscellaneous
--[no_]builtin	Do/don't compile with math subroutine builtin support, which causes selected math library routines to be inlined (pgcc and pgcpp only).	Optimization
byteswapio	Swap byte-order (big-endian to little-endian or vice versa) during I/O of Fortran unformatted data.	Miscellaneous
cache_align	Where possible, align data objects of size greater than or equal to 16 bytes on cache-line boundaries.	Optimization
chkfpstk	Check for internal consistency of the x87 FP stack in the prologue of a function and after returning from a function or subroutine call (–tp px/p5/p6/piii targets only).	Miscellaneous
chkptr	Check for NULL pointers (pgf95, pgfortran, and pghpf only).	Miscellaneous
chkstk	Check the stack for available space upon entry to and before the start of a parallel region. Useful when many private variables are declared.	Miscellaneous
concur	Enable auto-concurrentization of loops. Multiple processors or cores will be used to execute parallelizable loops.	Optimization
cpp	Run the PGI cpp-like preprocessor without performing subsequent compilation steps.	Miscellaneous

pgflag	Description	Category
cray	Force Cray Fortran (CF77) compatibility (pgf77, pgf95, pgfortran, and pghpf only).	Optimization
cuda	Enables Cuda Fortran.	Fortran Language
[no]daz	Do/don't treat denormalized numbers as zero.	Code Generation
[no]dclchk	Determines whether all program variables must be declared (pgf77, pgf95, pgfortran, and pghpf only).	Fortran Language
[no]defaultunit	Determines how the asterisk character ("*") is treated in relation to standard input and standard output (regardless of the status of I/O units 5 and 6, pgf77, pgf95, pgfortran, and pghpf only).	Fortran Language
[no]depchk	Checks for potential data dependencies.	Optimization
[no]dse	Enables [disables] dead store elimination phase for programs making extensive use of function inlining.	Optimization
[no]dlines	Determines whether the compiler treats lines containing the letter "D" in column one as executable statements (pgf77, pgf95, pgfortran, and pghpf only).	Fortran Language
dll	Link with the DLL version of the runtime libraries (Windows only).	Miscellaneous
dollar,char	Specifies the character to which the compiler maps the dollar sign code (pgf77, pgf95, pgfortran, and pghpf only).	Fortran Language
[no]dwarf	Specifies not to add DWARF debug information.	Code Generation
dwarf1	When used with –g, generate DWARF1 format debug information.	Code Generation
dwarf2	When used with –g, generate DWARF2 format debug information.	Code Generation
dwarf3	When used with –g, generate DWARF3 format debug information.	Code Generation
extend	Instructs the compiler to accept 132-column source code; otherwise it accepts 72-column code (pgf77, pgf95, pgfortran, and pghpf only).	Fortran Language
extract	invokes the function extractor.	Inlining
[no]m128	Instructs the compiler to treat floating-point constants as float data types (pgcc and pgcpp only).	C/C++ Language
fixed	Instructs the compiler to assume F77-style fixed format source code (pgf95, pgfortran, and pghpf only).	Fortran Language

pgflag	Description	Category
[no]flushz	Do/don't set SSE flush-to-zero mode	Code Generation
[no]fpapprox	Specifies not to use low-precision fp approximation operations.	Optimization
[no]f[=option]	Perform certain floating point intrinsic functions using relaxed precision.	Optimization
free	Instructs the compiler to assume F90-style free format source code (pgf95, pgfortran and pghpf only).	Fortran Language
func32	The compiler aligns all functions to 32-byte boundaries.	Code Generation
gccbug[s]	Matches behavior of certain gcc bugs	Miscellaneous
info	Prints informational messages regarding optimization and code generation to standard output as compilation proceeds.	Miscellaneous
inform	Specifies the minimum level of error severity that the compiler displays.	Miscellaneous
inline	Invokes the function inliner.	Inlining
instrumentation	Generates code to enable instrumentation of functions.	Miscellaneous
[no]ipa	Invokes interprocedural analysis and optimization.	Optimization
[no]iomutex	Determines whether critical sections are generated around Fortran I/O calls (pgf77, pgf95, pgfortran, and pghpf only).	Fortran Language
keepasm	Instructs the compiler to keep the assembly file.	Miscellaneous
largeaddressaware	Enables support for 64-bit indexing and single static data objects of size larger than 2GB.	Code Generation
[no]large_arrays	Enables support for 64-bit indexing and single static data objects of size larger than 2GB.	Code Generation
lfs	Links in libraries that allow file I/O to files of size larger than 2GB on 32-bit systems (32-bit Linux only).	Environment
[no]loop32	Aligns/does not align innermost loops on 32 byte boundaries with –tp barcelona	Code Generation
[no]lre	Disable/enable loop-carried redundancy elimination.	Optimization
list	Specifies whether the compiler creates a listing file.	Miscellaneous
[no]m128	Recognizes [ignores] __m128, __m128d, and __m128i datatypes. (C only)	Code Generation

pgflag	Description	Category
makedll	Generate a dynamic link library (DLL) (Windows only).	Miscellaneous
makeimplib	Passes the -def switch to the librarian without a deffile, when used without `-def:deffile`.	Miscellaneous
mpi=option	Link to MPI libraries: MPICH1, MPICH2, or Microsoft MPI libraries	Code Generation
neginfo	Instructs the compiler to produce information on why certain optimizations are not performed.	Miscellaneous
noframe	Eliminates operations that set up a true stack frame pointer for functions.	Optimization
noi4	Determines how the compiler treats INTEGER variables (pgf77, pgf95, pgfortran, and pghpf only).	Optimization
nomain	When the link step is called, don't include the object file that calls the Fortran main program. (pgf77, pgf95, pgfortran, and pghpf only).	Code Generation
noopenmp	When used in combination with the –mp option, the compiler ignores OpenMP parallelization directives or pragmas, but still processes SGI-style parallelization directives or pragmas.	Miscellaneous
nopgdllmain	Do not link the module containing the default DllMain() into the DLL (Windows only).	Miscellaneous
norpath	On Linux, do not add –rpath paths to the link line.	Miscellaneous
nosgimp	When used in combination with the –mp option, the compiler ignores SGI-style parallelization directives or pragmas, but still processes OpenMP directives or pragmas.	Miscellaneous
[no]stddef	Instructs the compiler to not recognize the standard preprocessor macros.	Environment
nostdinc	Instructs the compiler to not search the standard location for include files.	Environment
nostdlib	Instructs the linker to not link in the standard libraries.	Environment
[no]onetrip	Determines whether each DO loop executes at least once (pgf77, pgf95, pgfortran, and pghpf only).	Language
novintr	Disable idiom recognition and generation of calls to optimized vector functions.	Optimization
pfi	Instrument the generated code and link in libraries for dynamic collection of profile and data information at runtime.	Optimization

pgflag	Description	Category
pre	Read a pgfi.out trace file and use the information to enable or guide optimizations.	Optimization
[no]pre	Force/disable generation of non-temporal moves and prefetching.	Code Generation
[no]prefetch	Enable/disable generation of prefetch instructions.	Optimization
preprocess	Perform cpp-like preprocessing on assembly language and Fortran input source files.	Miscellaneous
prof	Set profile options; function-level and line-level profiling are supported.	Code Generation
[no]r8	Determines whether the compiler promotes REAL variables and constants to DOUBLE PRECISION (pgf77, pgf95, pgfortran, and pghpf only).	Optimization
[no]r8intrinsics	Determines how the compiler treats the intrinsics CMPLX and REAL (pgf77, pgf95, pgfortran, and pghpf only).	Optimization
[no]recursive	Allocate / do not allocate local variables on the stack, this allows recursion. SAVEd, data-initialized, or namelist members are always allocated statically, regardless of the setting of this switch (pgf77, pgf95, pgfortran, and pghpf only).	Code Generation
[no]reentrant	Specifies whether the compiler avoids optimizations that can prevent code from being reentrant.	Code Generation
[no]ref_externals	Do/don't force references to names appearing in EXTERNAL statements (pgf77, pgf95, pgfortran and pghpf only).	Code Generation
safeptr	Instructs the compiler to override data dependencies between pointers and arrays (pgcc and pgcpp only).	Optimization
safe_lastval	In the case where a scalar is used after a loop, but is not defined on every iteration of the loop, the compiler does not by default parallelize the loop. However, this option tells the compiler it is safe to parallelize the loop. For a given loop, the last value computed for all scalars make it safe to parallelize the loop.	Code Generation
[no]save	Determines whether the compiler assumes that all local variables are subject to the SAVE statement (pgf77, pgf95, pgfortran, and pghpf only).	Fortran Language
[no]scalarsse	Do/don't use SSE/SSE2 instructions to perform scalar floating-point arithmetic.	Optimization

pgflag	Description	Category
schar	Specifies signed char for characters (pgcc and pgcpp only - also see uchar).	C/C++ Language
[no]second_underscore	Do/don't add the second underscore to the name of a Fortran global if its name already contains an underscore (pgf77, pgf95, pgfortran, and pghpf only).	Code Generation
[no]signextend	Do/don't extend the sign bit, if it is set.	Code Generation
[no]single	Do/don't convert float parameters to double parameter characters (pgcc and pgcpp only).	C/C++ Language
[no]smart	Do/don't enable optional post-pass assembly optimizer.	Optimization
[no]smartalloc[=huge|huge:<n>|hugebss]	Add a call to the routine mallopt in the main routine. Supports large TLBs on Linux and Windows. *Tip.* To be effective, this switch must be specified when compiling the file containing the Fortran, C, or C++ main program.	Environment
standard	Causes the compiler to flag source code that does not conform to the ANSI standard(pgf77, pgf95, pgfortran, and pghpf only).	Fortran Language
[no]stride0	Do/do not generate alternate code for a loop that contains an induction variable whose increment may be zero (pgf77, pgf95, pgfortran, and pghpf only).	Code Generation
uchar	Specifies unsigned char for characters (pgcc and pgcpp only - also see schar).	C/C++ Language
unix	Uses UNIX calling and naming conventions for Fortran subprograms (pgf77, pgf95, pgfortran, and pghpf for Win32 only).	Code Generation
[no]unixlogical	Determines how the compiler treats logical values. (pgf77, pgf95, pgfortran, and pghpf only).	Fortran Language
[no]unroll	Controls loop unrolling.	Optimization
[no]upcase	Determines whether the compiler preserves uppercase letters in identifiers. (pgf77, pgf95, pgfortran, and pghpf only).	Fortran Language
varargs	Forces Fortran program units to assume calls are to C functions with a varargs type interface (pgf77, pgf95, and pgfortran only).	Code Generation
[no]vect	Do/don't invoke the code vectorizer.	Optimization

–mcmodel=medium

(For use only on 64-bit Linux targets) Generates code for the medium memory model in the linux86-64 execution environment. Implies –Mlarge_arrays.

Default: The compiler generates code for the small memory model.

Usage: The following command line requests position independent code be generated, and the option `–mcmodel=medium` be passed to the assembler and linker:

```
$ pgfortran -mcmodel=medium myprog.f
```

Description: The default small memory model of the linux86-64 environment limits the combined area for a user's object or executable to 1GB, with the Linux kernel managing usage of the second 1GB of address for system routines, shared libraries, stacks, and so on. Programs are started at a fixed address, and the program can use a single instruction to make most memory references.

The medium memory model allows for larger than 2GB data areas, or .bss sections. Program units compiled using either `–mcmodel=medium` or `–fpic` require additional instructions to reference memory. The effect on performance is a function of the data-use of the application. The `–mcmodel=medium` switch must be used at both compile time and link time to create 64-bit executables. Program units compiled for the default small memory model can be linked into medium memory model executables as long as they are compiled with the option`–fpic`, or position-independent.

The linux86-64 environment provides `static libxxx.a` archive libraries, that are built both with and without `–fpic`, and `dynamic libxxx.so` shared object libraries that are compiled with `–fpic`. Using the link switch `–mcmodel=medium` implies the `–fpic` switch and utilizes the shared libraries by default. The directory `$PGI/linux86-64/<rel>/lib` contains the libraries for building small memory model codes; and the directory `$PGI/linux86-64/<rel>/libso` contains shared libraries for building both `–fpic` and `–mcmodel=medium` executables.

Note

`–mcmodel=medium -fpic` is not allowed to create shared libraries. However, you can create static archive libraries (.a) that are `–fpic`.

Related options:–Mlarge_arrays

–module <moduledir>

Allows you to specify a particular directory in which generated intermediate `.mod` files should be placed.

Default: The compiler places `.mod` files in the current working directory, and searches only in the current working directory for pre-compiled intermediate `.mod` files.

Usage: The following command line requests that any intermediate module file produced during compilation of `myprog.f` be placed in the directory `mymods`; specifically, the file `./mymods/myprog.mod` is used.

```
$ pgfortran -module mymods myprog.f
```

Description: Use the –module option to specify a particular directory in which generated intermediate .mod files should be placed. If the –module <moduledir> option is present, and USE statements are present in a

compiled program unit, then <moduledir> is searched for `.mod` intermediate files *prior* to a search in the default local directory.

Related options: None.

–mp[=all, align,bind,[no]numa]

Instructs the compiler to interpret user-inserted OpenMP shared-memory parallel programming directives and pragmas, and to generate an executable file which will utilize multiple processors in a shared-memory parallel system.

Default: The compiler ignores user-inserted shared-memory parallel programming directives and pragmas.

Usage: The following command line requests processing of any shared-memory directives present in `myprog.f`:

```
$ pgfortran -mp myprog.f
```

Description: Use the `–mp`option to instruct the compiler to interpret user-inserted OpenMP shared-memory parallel programming directives and to generate an executable file which utilizes multiple processors in a shared-memory parallel system.

The suboptions are one or more of the following:

align
: Forces loop iterations to be allocated to OpenMP processes using an algorithm that maximizes alignment of vector sub-sections in loops that are both parallelized and vectorized for SSE. This allocation can improve performance in program units that include many such loops. It can also result in load-balancing problems that significantly decrease performance in program units with relatively short loops that contain a large amount of work in each iteration. The numa suboption uses libnuma on systems where it is available.

allcores
: Instructs the compiler to all available cores. You specify this suboption at link time.

bind
: Instructs the compiler to bind threads to cores. You specify this suboption at link time.

[no]numa
: Uses [does not use] libnuma on systems where it is available.

For a detailed description of this programming model and the associated directives and pragmas, refer to Chapter 9, "Using OpenMP" of the PGI Compiler User's Guide.

Related options: –Mconcur, –Mvect

–noswitcherror

Issues warnings instead of errors for unknown switches. Ignores unknown command line switches after printing a warning message.

Default: The compiler prints an error message and then halts.

Usage: In the following example, the compiler ignores unknown command line switches after printing a warning message.

```
$ pgfortran -noswitcherror myprog.f
```

Description: Use this option to instruct the compiler to ignore unknown command line switches after printing an warning message.

Tip

You can configure this behavior in the `siterc` file by adding: `set NOSWITCHERROR=1.`

Related options: None.

–O<level>

Invokes code optimization at the specified level.

Default: The compiler optimizes at level 2.

Syntax:

–O [level]

Where level is an integer from 0 to 4.

Usage: In the following example, since no –O option is specified, the compiler sets the optimization to level 1.

```
$ pgfortran myprog.f
```

In the following example, since no optimization level is specified and a –O option is specified, the compiler sets the optimization to level 2.

```
$ pgfortran -O myprog.f
```

Description: Use this option to invoke code optimization at the specified level - one of the following:

0
: creates a basic block for each statement. Neither scheduling nor global optimization is done. To specify this level, supply a 0 (zero) argument to the –O option.

1
: schedules within basic blocks and performs some register allocations, but does no global optimization.

2
: performs all level-1 optimizations, and also performs global scalar optimizations such as induction variable elimination and loop invariant movement.

3
: level-three specifies aggressive global optimization. This level performs all level-one and level-two optimizations and enables more aggressive hoisting and scalar replacement optimizations that may or may not be profitable.

4
: level-four performs all level-one, level-two, and level-three optimizations and enables hoisting of guarded invariant floating point expressions.

Table 2-8 shows the interaction between the -O option, -g option, -Mvect, and -Mconcur options.

Table 2.8. Optimization and –O, –g, –Mvect, and –Mconcur Options

Optimize Option	Debug Option	–M Option	Optimization Level
none	none	none	1
none	none	–Mvect	2
none	none	–Mconcur	2
none	–g	none	0
–O	none or –g	none	2
–Olevel	none or –g	none	level
–Olevel < 2	none or –g	–Mvect	2
–Olevel < 2	none or –g	–Mconcur	2

Unoptimized code compiled using the option -O0 can be significantly slower than code generated at other optimization levels. Like the -Mvect option, the -Munroll option sets the optimization level to level-2 if no –O or -g options are supplied. The -gopt option is recommended for generation of debug information with optimized code. For more information on optimization, refer to Chapter 7, "Optimizing and Parallelizing" in the PGI Compiler User's Guide.

Related options: –g, –M<pgflag>, –gopt

–o

Names the executable file. Use the –o option to specify the filename of the compiler object file. The final output is the result of linking.

Syntax:

–o filename

Where filename is the name of the file for the compilation output. The filename must not have a .f extension.

Default: The compiler creates executable filenames as needed. If you do not specify the -o option, the default filename is the linker output file a.out.

Usage: In the following example, the executable file is myprog instead of the default a.outmyprog.exe.

```
$ pgfortran myprog.f -o myprog
```

Related options: –c, –E, –F, –S

–pc

Note

This option is available only for –tp px/p5/p6/piii targets.

Allows you to control the precision of operations performed using the x87 floating point unit, and their representation on the x87 floating point stack.

Syntax:

–pc { 32 | 64 | 80 }

Usage:

```
$ pgfortran -pc 64 myprog.f
```

Description: The x87 architecture implements a floating-point stack using 8 80-bit registers. Each register uses bits 0-63 as the significant, bits 64-78 for the exponent, and bit 79 is the sign bit. This 80-bit real format is the default format, called the *extended format*. When values are loaded into the floating point stack they are automatically converted into extended real format. The precision of the floating point stack can be controlled, however, by setting the precision control bits (bits 8 and 9) of the floating control word appropriately. In this way, you can explicitly set the precision to standard IEEE double-precision using 64 bits, or to single precision using 32 bits.[1] The default precision is system dependent. To alter the precision in a given program unit, the main program must be compiled with the same -pc option. The command line option `–pc val` lets the programmer set the compiler's precision preference.

Valid values for val are:

32 single precision 64 double precision 80 extended precision

Extended Precision Option – Operations performed exclusively on the floating-point stack using extended precision, without storing into or loading from memory, can cause problems with accumulated values within the extra 16 bits of extended precision values. This can lead to answers, when rounded, that do not match expected results.

For example, if the argument to `sin` is the result of previous calculations performed on the floating-point stack, then an 80-bit value used instead of a 64-bit value can result in slight discrepancies. Results can even change sign due to the sin curve being too close to an x-intercept value when evaluated. To maintain consistency in this case, you can assure that the compiler generates code that calls a function. According to the x86 ABI, a function call must push its arguments on the stack (in this way memory is guaranteed to be accessed, even if the argument is an actual constant). Thus, even if the called function simply performs the inline expansion, using the function call as a wrapper to sin has the effect of trimming the argument precision down to the expected size. Using the `–Mnobuiltin` option on the command line for C accomplishes this task by resolving all math routines in the library `libm`, performing a function call of necessity. The other method of generating a function call for math routines, but one that may still produce the inline instructions, is by using the `–Kieee` switch.

A second example illustrates the precision control problem using a section of code to determine machine precision:

```
 program find_precision

 w = 1.0
100 w=w+w
 y=w+1
 z=y-w
 if (z .gt. 0) goto 100
```

[1]According to Intel documentation, this only affects the x87 operations of add, subtract, multiply, divide, and square root. In particular, it does not appear to affect the x87 transcendental instructions.

```
C now w is just big enough that |((w+1)-w)-1| >= 1
...
 print*,w
 end
```

In this case, where the variables are implicitly real*4, operations are performed on the floating-point stack where optimization removes unnecessary loads and stores from memory. The general case of copy propagation being performed follows this pattern:

```
a = x
y = 2.0 + a
```

Instead of storing x into a, then loading a to perform the addition, the value of x can be left on the floating-point stack and added to 2.0. Thus, memory accesses in some cases can be avoided, leaving answers in the extended real format. If copy propagation is disabled, stores of all left-hand sides will be performed automatically and reloaded when needed. This will have the effect of rounding any results to their declared sizes.

The find_precision program has a value of 1.8446744E+19 when executed using default (extended) precision. If, however, `-Kieee` is set, the value becomes 1.6777216E+07 (single precision.) This difference is due to the fact that `-Kieee` disables copy propagation, so all intermediate results are stored into memory, then reloaded when needed. Copy propagation is only disabled for floating-point operations, not integer. With this particular example, setting the `-pc` switch will also adjust the result.

The `-Kieee` switch also has the effect of making function calls to perform all transcendental operations. Except when the `-Mnobuiltin` switch is set in C, the function still produces the x86 machine instruction for computation, and arguments are passed on the stack, which results in a memory store and load.

Finally, `-Kieee` also disables reciprocal division for constant divisors. That is, for a/b with unknown a and constant b, the expression is usually converted at compile time to a*(1/b), thus turning an expensive divide into a relatively fast scalar multiplication. However, numerical discrepancies can occur when this optimization is used.

Understanding and correctly using the `-pc`, `-Mnobuiltin`, and `-Kieee` switches should enable you to produce the desired and expected precision for calculations which utilize floating-point operations.

Related options: –Kieee, –Mnobuiltin

- -pedantic

Prints warnings from included <system header files> .

Default: The compiler prints the warnings from the included system header files.

Usage: In the following example, the compiler prints the warnings from the included system header files.

```
$ pgfortran --pedantic myprog.f
```

Related options:

–pg

(Linux only) Instructs the compiler to instrument the generated executable for gprof-style sample-based profiling.

Default: The compiler does not instrument the generated executable for gprof-style profiling.

Usage: In the following example the program is compiled for profiling using pgdbg or gprof.

```
$ pgfortran -pg myprog.c
```

Description: Use this option to instruct the compiler to instrument the generated executable for gprof-style sample-based profiling. You must use this option at both the compile and link steps. A `gmon.out` style trace is generated when the resulting program is executed, and can be analyzed using gprof or pgprof.

–pgcpplibs

Instructs the compiler to append C++ runtime libraries to the link line for programs built using either PGF90 or PGF77.

Default: The C/C++ compilers do not append the C++ runtime libraries to the link line.

Usage: In the following example the C++ runtime libraries are linked with an object file compiled with pgf77.

```
$ pgf90 main.f90 mycpp.o -pgcpplibs
```

Description: Use this option to instruct the compiler to append C++ runtime libraries to the link line for programs built using either PGF90 or PGF77.

Related options:–pgf90libs, –pgf77libs

–pgf77libs

Instructs the compiler to append PGF77 runtime libraries to the link line.

Default: The C/C++ compilers do not append the PGF77 runtime libraries to the link line.

Usage: In the following example a .c main program is linked with an object file compiled with pgf77.

```
$ pgcc main.c myf77.o -pgf77libs
```

Description: Use this option to instruct the compiler to append PGF77 runtime libraries to the link line.

Related options:–pgcpplibs, –pgf90libs

–pgf90libs

Instructs the compiler to append PGF90/PGF95/PGFORTRAN runtime libraries to the link line.

Default: The C/C++ compilers do not append the PGF90/PGF95/PGFORTRAN runtime libraries to the link line.

Usage: In the following example a .c main program is linked with an object file compiled with pgfortran.

```
$ pgcc main.c myf95.o -pgf90libs
```

Description: Use this option to instruct the compiler to append PGF90/PGF95/PGFORTRAN runtime libraries to the link line.

Related options:–pgcpplibs, –pgf77libs

–R<directory>

(Linux only) Instructs the linker to hard-code the pathname <directory> into the search path for generated shared object (dynamically linked library) files.

Note

There cannot be a space between R and <directory>.

Usage: In the following example, at runtime the a.out executable searches the specified directory, in this case `/home./Joe/myso`, for shared objects.

```
$ pgfortran -Rm/home/Joe/myso myprog.f
```

Description: Use this option to instruct the compiler to pass information to the linker to hard-code the pathname <directory> into the search path for shared object (dynamically linked library) files.

Related options: –fpic, –shared, –G

–r

Linux only. Creates a relocatable object file.

Default: The compiler does not create a relocatable object file and does not use the –r option.

Usage: In this example, pgfortran creates a relocatable object file.

```
$ pgfortran -r myprog.f
```

Description: Use this option to create a relocatable object file.

Related options: –c, –o, –s, –u

–r4 and –r8

Interprets DOUBLE PRECISION variables as REAL (–r4), or interprets REAL variables as DOUBLE PRECISION (–r8).

Usage: In this example, the double precision variables are interpreted as REAL.

```
$ pgfortran -r4 myprog.f
```

Description: Interpret DOUBLE PRECISION variables as REAL (–r4) or REAL variables as DOUBLE PRECISION (–r8).

Related options: –i2, –i4, –i8, –nor8

–rc

Specifies the name of the driver startup configuration file. If the file or pathname supplied is not a full pathname, the path for the configuration file loaded is relative to the $DRIVER path (the path of the currently executing driver). If a full pathname is supplied, that file is used for the driver configuration file.

Syntax:

```
-rc [path] filename
```

Where path is either a relative pathname, relative to the value of $DRIVER, or a full pathname beginning with "/". Filename is the driver configuration file.

Default: The driver uses the configuration file `.pgirc`.

Usage: In the following example, the file `.pgfortranrctest`, relative to `/usr/pgi/linux86/bin`, the value of $DRIVER, is the driver configuration file.

```
$ pgfortran -rc .pgfortranrctest myprog.f
```

Description: Use this option to specify the name of the driver startup configuration file. If the file or pathname supplied is not a full pathname, the path for the configuration file loaded is relative to the $DRIVER path - the path of the currently executing driver. If a full pathname is supplied, that file is used for the driver configuration file.

Related options: –show

–rpath

(Linux only) Specifies the name of the driver startup configuration file.

Syntax:

```
-rpath path <ldarg>
```

where path is either a relative pathname, or a full pathname beginning with "/".

Default: The driver uses the configuration file `.pgirc`.

Usage: In the following example, the file `.pgfortranrctest`, relative to `/usr/pgi/linux86/bin`, the value of $DRIVER, is the driver configuration file.

```
$ pgfortran -rc .pgfortranrctest myprog.f
```

Description: Use this option to specify the name of the driver startup configuration file. If the file or pathname supplied is not a full pathname, the path for the configuration file loaded is relative to the $DRIVER path - the path of the currently executing driver. If a full pathname is supplied, that file is used for the driver configuration file.

With the ldarg option (Linux only), the ldarg information is passed to the linker and the directory is added to the runtime shared library search path.

Related options: –show

–s

(Linux only) Strips the symbol-table information from the executable file.

Default: The compiler includes all symbol-table information and does not use the –s option.

Usage: In this example, pgfortran strips symbol-table information from the `a.out`. executable file.

```
$ pgfortran -s myprog.f
```

Description: Use this option to strip the symbol-table information from the executable.

Related options: –c, –o, –u

–S

Stops compilation after the compiling phase and writes the assembly-language output to a file.

Default: The compiler does not produce a `.s` file.

Usage: In this example, pgfortran produces the file `myprog.s` in the current directory.

```
$ pgfortran -S myprog.f
```

Description: Use this option to stop compilation after the compiling phase and then write the assembly-language output to a file. If the input file is `filename.f`, then the output file is `filename.s`.

Related options: –c, –E, –F, –Mkeepasm, –o

–shared

(Linux only) Instructs the compiler to pass information to the linker to produce a shared object (dynamically linked library) file.

Default: The compiler does not pass information to the linker to produce a shared object file.

Usage: In the following example the compiler passes information to the linker to produce the shared object file: `myso.so`.

```
$ pgfortran -shared myprog.f -o myso.so
```

Description: Use this option to instruct the compiler to pass information to the linker to produce a shared object (dynamically linked library) file.

Related options: –fpic, –G, –R

–show

Produces driver help information describing the current driver configuration.

Default: The compiler does not show driver help information.

Usage: In the following example, the driver displays configuration information to the standard output after processing the driver configuration file.

```
$ pgfortran -show myprog.f
```

Description: Use this option to produce driver help information describing the current driver configuration.

Related options: –V, –v, –###, –help, –rc

–silent

Do not print warning messages.

Default: The compiler prints warning messages.

Usage: In the following example, the driver does not display warning messages.

```
$ pgfortran -silent myprog.f
```

Description: Use this option to suppress warning messages.

Related options: –v, –V, –w

–soname

(Linux only.) The compiler recognizes the –soname option and passes it through to the linker.

Default: The compiler does not recognize the –soname option.

Usage: In the following example, the driver passes the soname option and its argument through to the linker.

```
$ pgfortran -soname library.so myprog.f
```

Description: Use this option to instruct the compiler to recognize the –soname option and pass it through to the linker.

Related options:

–stack

(Windows only.) Allows you to explicitly set stack properties for your program.

Default: If `-stack` is not specified, then the defaults are as followed:

Win32
: Setting is `-stack:2097152,2097152`, which is approximately 2MB for reserved and committed bytes.

Win64
: No default setting

Syntax:

```
-stack={ (reserved bytes)[,(committed bytes)] }{, [no]check }
```

Usage: The following example demonstrates how to reserve 524,288 stack bytes (512KB), commit 262,144 stack bytes for each routine (256KB), and disable the stack initialization code with the nocheck argument.

```
$ pgfortran -stack=524288,262144,nocheck myprog.f
```

Description: Use this option to explicitly set stack properties for your program. The `-stack` option takes one or more arguments: (reserved bytes), (committed bytes), [no]check.

reserved bytes
: Specifies the total stack bytes required in your program.

committed bytes
: Specifies the number of stack bytes that the Operating System will allocate for each routine in your program. This value must be less than or equal to the stack *reserved bytes* value.

  Default for this argument is 4096 bytes

[no]check
: Instructs the compiler to generate or not to generate stack initialization code upon entry of each routine. Check is the default, so stack initialization code is generated.

Stack initialization code is required when a routine's stack exceeds the *committed bytes* size. When your *committed bytes* is equal to the *reserved bytes* or equal to the stack bytes required for each routine, then you can turn off the stack initialization code using the `-stack=nocheck` compiler option. If you do this, the compiler assumes that you are specifying enough committed stack space; and therefore, your program does not have to manage its own stack size.

For more information on determining the amount of stack required by your program, refer to `–Mchkstk` compiler option, described in "Miscellaneous Controls" on page 151.

Note

`-stack=(reserved bytes),(committed bytes)` are linker options.

`-stack=[no]check` is a compiler option.

If you specify `-stack=(reserved bytes),(committed bytes)` on your compile line, it is only used during the link step of your build. Similarly, `-stack=[no]check` can be specified on your link line, but its only used during the compile step of your build.

Related options: –Mchkstk

–ta=nvidia(,nvidia_suboptions),host

Defines the target accelertator.

Note

This flag is valid only for Fortran and C.

Default: The compiler uses NVIDIA.

Usage: In the following example, NVIDEA is the accelerator target architecture and the accelerator generates code for compute capability 1.3.

```
$ pgfortran -ta=nvidia,cc13
```

Description: Use this option to select the accelerator target and, optionally, to define the type of code to genertate.

The `–ta` flag has the following options:

nvidia
: Select NVIDIA accelerator target. This option has the following nvidia-suboptions:

  analysis
  : Perform loop analysis only; do not generate GPU code.

  cc10
  : Generate code for compute capability 1.0.

  cc11
  : Generate code for compute capability 1.1.

cc12
Generate code for compute capability 1.2.

cc13
Generate code for compute capability 1.3.

cc20
Generate code for compute capability 2.0.

cuda2.3 or 2.3
Specify the NVIDIA CUDA 2.3 version of the toolkit.

cuda3.0 or 3.0
Specify the NVIDIA CUDA 3.0 version of the toolkit.

cuda3.1 or 3.1
Specify the NVIDIA CUDA 3.1 version of the toolkit.

cuda3.2 or 3.2
Specify the NVIDIA CUDA 3.2 version of the toolkit.

Note

Compiling with the CUDA 3.1 or CUDA 3.2 toolkit, either by using the `-ta=nvidia:cuda3.1` or `-ta=nvidia:cuda3.2`option or by adding `set CUDAVERSION=3.1` or `set CUDAVERSION=3.2` to the `siterc` file, generates binaries that may not work on machines with a 2.3 CUDA driver.

`pgaccelinfo` prints the driver version as the first line of output.

For a 2.3 driver: `CUDA Driver Version 2030`
For a 3.0 driver: `CUDA Driver Version 3000`
For a 3.1 driver: `CUDA Driver Version 3010`
For a 3.2 driver: `CUDA Driver Version 3020`

fastmath
Use routines from the fast math library.

[no]flushz
Enable[disable] flush-to-zero mode for floating point computations in the GPU code generated for for PGI Accelerator model compute regions.

keepbin
Keep the binary (.bin) files.

keepgpu
Keep the kernel source (.gpu) files.

keepptx
Keep the portable assembly (.ptx) file for the GPU code.

maxregcount:n
Specify the maximum number of registers to use on the GPU. Leaving this blank indicates no limit.

mul24
: Use 24-bit multiplication for subscripting.

nofma
: Do not generate fused multiply-add instructions.

time
: Link in a limited-profiling library, as described in "Profiling Accelerator Kernels" in the Using an Accelerator chapter of the PGI Compiler User's Guide.

[no]wait
: Wait [do not wait] for each kernel to finish before continuing in the host program.

host
: Select NO accelerator target. Generate PGI Unified Binary Code, as described in "PGi Unified Binaries for Accelerators" in the Using an Accelerator chapter of the PGI Compiler User's Guide.

Related options: –#

–time

Print execution times for various compilation steps.

Default: The compiler does not print execution times for compilation steps.

Usage: In the following example, pgfortran prints the execution times for the various compilation steps.

```
$ pgfortran -time myprog.f
```

Description: Use this option to print execution times for various compilation steps.

Related options: –#

–tp <target> [,target...]

Sets the target architecture.

Default: The PGI compilers produce code specifically targeted to the type of processor on which the compilation is performed. In particular, the default is to use all supported instructions wherever possible when compiling on a given system.

The default style of code generation is auto-selected depending on the type of processor on which compilation is performed. Further, the –tp x64 style of unified binary code generation is only enabled by an explicit –tp x64 option.

Note

Executables created on a given system may not be usable on previous generation systems. (For example, executables created on a Pentium 4 may fail to execute on a Pentium III or Pentium II.)

Usage: In the following example, pgfortran sets the target architecture to EM64T:

```
$ pgfortran -tp p7-64 myprog.f
```

Description: Use this option to set the target architecture. By default, the PGI compiler uses all supported instructions wherever possible when compiling on a given system. As a result, executables created on a given system may not be usable on previous generation systems. For example, executables created on a Pentium 4 may fail to execute on a Pentium III or Pentium II.

Processor-specific optimizations can be specified or limited explicitly by using the `-tp` option. Thus, it is possible to create executables that are usable on previous generation systems. With the exception of k8-64, k8-64e, p7-64, and x64, any of these suboptions are valid on any x86 or x64 processor-based system. The k8-64, k8-64e, p7-64 and x64 options are valid only on x64 processor-based systems.

The `-tp x64` option generates unified binary object and executable files, as described in "Using –tp to Generate a Unified Binary" on page 90.

The following list contains the possible suboptions for `-tp` and the processors that each suboption is intended to target. Options without a bit-length suffix use the current width associated with the driver on your path.

athlon
: generate code for AMD Athlon XP/MP and compatible processors.

barcelona
: generate code for AMD Opteron/Quadcore and compatible processors. The

barcelona-32
: generate 32-bit code for AMD Opteron/Quadcore and compatible processors.

barcelona-64
: generate 64-bit code for AMD Opteron/Quadcore and compatible processors.

core2
: generate code for Intel Core 2 Duo and compatible processors.

core2-32
: generate 32-bit code for Intel Core 2 Duo and compatible processors.

core2-64
: generate 64-bit code for Intel Core 2 Duo EM64T and compatible processors.

istanbul
: generate code that is usable on any Istanbul processor-based system.

istanbul-32
: generate 32-bit code that is usable on any Istanbul processor-based system.

istanbul-64
: generate 64-bit code that is usable on any Istanbul processor-based system.

k8-32
: generate 32-bit code for AMD Athlon64, AMD Opteron and compatible processors.

k8-64
: generate 64-bit code for AMD Athlon64, AMD Opteron and compatible processors.

k8-64e
: generate 64-bit code for AMD Opteron Revision E, AMD Turion, and compatible processors.

nehalem
: generate code that is usable on any Nehalem processor-based system.

nehalem-32
: generate 32-bit code that is usable on any Nehalem processor-based system.

nehalem-64
: generate 64-bit code that is usable on any Nehalem processor-based system.

p6
: generate code for Pentium Pro/II/III and AthlonXP compatible processors.

p7
: generate code for Pentium 4 and compatible processors.

p7-32
: generate 32-bit code for Pentium 4 and compatible processors.

p7-64
: generate 64-bit code for Intel P4/Xeon EM64T and compatible processors.

penryn
: generate code for Intel Penryn Architecture and compatible processors.

penryn-32
: generate 32-bit code for Intel Penryn Architecture and compatible processors.

penryn-64
: generate 64-bit code for Intel Penryn Architecture and compatible processors.

piii
: generate code for Pentium III and compatible processors, including support for single-precision vector code using SSE instructions.

px
: generate code that is usable on any x86 processor-based system.

px-32
: generate 32-bit code that is usable on any x86 processor-based system.

shanghai
: generate code that is usable on any AMD Shanghai processor-based system.

shanghai-32
: generate 32-bit code that is usable on any AMD Shanghai processor-based system.

shanghai-64
: generate 64-bit code that is usable on any AMD Shanghai processor-based system.

x64
: generate 64-bit unified binary code including full optimizations and support for both AMD and Intel x64 processors.

Refer to the PGI Release Notes for a concise list of the features of these processors that distinguish them as separate targets when using the PGI compilers and tools.

The syntax for 64-bit and 32-bit targets is similar, even though the target information varies.

Syntax for 64-bit targets:

```
-tp {k8-64 | k8-64e | p7-64 | core2-64 | x64}
```

Syntax for 32-bit targets:

```
-tp {k8-32 | p6 | p7 | core2 | piii | px}
```

Using –tp to Generate a Unified Binary

Different processors have differences, some subtle, in hardware features such as instruction sets and cache size. The compilers make architecture-specific decisions about such things as instruction selection, instruction scheduling, and vectorization. Any of these decisions can have significant effects on performance and compatibility. PGI unified binaries provide a low-overhead means for a single program to run well on a number of hardware platforms.

You can use the –tp option to produce PGI Unified Binary programs. The compilers generate, and combine into one executable, multiple binary code streams, each optimized for a specific platform. At runtime, this one executable senses the environment and dynamically selects the appropriate code stream.

The target processor switch, –tp, accepts a comma-separated list of 64-bit targets and will generate code optimized for each listed target. For example, the following switch generates optimized code for three targets: k8-64, p7-64, and core2-64.

Syntax for optimizing for multiple targets:

```
-tp k8-64,p7-64,core2-64
```

The –tp k8-64 and –tp k8-64e options result in generation of code supported on and optimized for AMD x64 processors, while the –tp p7-64 option results in generation of code that is supported on and optimized for Intel x64 processors. Performance of k8-64 or k8-64e code executed on Intel x64 processors, or of p7-64 code executed on AMD x64 processors, can often be significantly less than that obtained with a native binary.

The special –tp x64 option is equivalent to –tp k8-64,p7-64. This switch produces PGI Unified Binary programs containing code streams fully optimized and supported for *both* AMD64 and Intel EM64T processors.

For more information on unified binaries, refer tothe section ”Processor-Specific Optimization and the Unified Binary” in the PGI Compiler User’s Guide.

“Processor-Specific Optimization & the Unified Binary” on page 137.

Related options: –M<pgflag> options that control environments

–[no]traceback

Adds debug information for runtime traceback for use with the environment variable `PGI_TERM`.

Default: The compiler enables traceback for FORTRAN 77 and Fortran 90/95 and disables traceback for C and C++.

Syntax:

```
-traceback
```

Usage: In this example, pgfortran enables traceback for the program `myprog.f`.

```
$ pgfortran -traceback myprog.f
```

Description: Use this option to enable or disable runtime traceback information for use with the environment variable `PGI_TERM`.

Setting `setTRACEBACK=OFF;` in `siterc` or `.mypg*rc` also disables default traceback.

Using ON instead of OFF enables default traceback.

–u

Initializes the symbol-table with <symbol>, which is undefined for the linker.

Default: The compiler does not use the –u option.

Syntax:

```
-usymbol
```

Where *symbol* is a symbolic name.

Usage: In this example, pgfortran initializes symbol-table with `test`.

```
$ pgfortran -utest myprog.f
```

Description: Use this option to initialize the symbol-table with <symbol>, which is undefined for the linker. An undefined symbol triggers loading of the first member of an archive library.

Related options: –c, –o, –s

–U

Undefines a preprocessor macro.

Syntax:

```
-Usymbol
```

Where *symbol* is a symbolic name.

Usage: The following examples undefine the macro test.

```
$ pgfortran -Utest myprog.F
$ pgfortran -Dtest -Utest myprog.F
```

Description: Use this option to undefine a preprocessor macro. You can also use the #undef preprocessor directive to undefine macros.

Related options: –D, –Mnostddef.

–V[release_number]

Displays additional information, including version messages. Further, if a `release_number` is appended, the compiler driver attempts to compile using the specified release instead of the default release.

Note

There can be no space between `-V` and `release_number`.

Default: The compiler does not display version information and uses the release specified by your path to compile.

Usage: The following command-line shows the output using the `-V` option.

```
% pgfortran -V myprog.f
```

The following command-line causes pgcc to compile using the 5.2 release instead of the default release.

```
% pgcc -V5.2 myprog.c
```

Description: Use this option to display additional information, including version messages or, if a release_number is appended, to instruct the compiler driver to attempt to compile using the specified release instead of the default release.

The specified release must be co-installed with the default release, and must have a release number greater than or equal to 4.1, which was the first release that supported this functionality.

Related options: –Minfo, –v

–v

Displays the invocations of the compiler, assembler, and linker.

Default: The compiler does not display individual phase invocations.

Usage: In the following example you use –v to see the commands sent to compiler tools, assembler, and linker.

```
$ pgfortran -v myprog.f90
```

Description: Use the –v option to display the invocations of the compiler, assembler, and linker. These invocations are command lines created by the compiler driver from the files and the –W options you specify on the compiler command-line.

Related options: –dryrun, –Minfo, –V, –W

–W

Passes arguments to a specific phase.

Syntax:

```
-W{0 | a | l },option[,option...]
```

Note

You cannot have a space between the –W and the single-letter pass identifier, between the identifier and the comma, or between the comma and the option.

0

(the number zero) specifies the compiler.

a
: specifies the assembler.

l
: (lowercase letter l) specifies the linker.

option
: is a string that is passed to and interpreted by the compiler, assembler or linker. Options separated by commas are passed as separate command line arguments.

Usage: In the following example the linker loads the text segment at address `0xffc00000` and the data segment at address `0xffe00000`.

```
$ pgfortran -Wl,-k,-t,0xffc00000,-d,0xffe00000 myprog.f
```

Description: Use this option to pass arguments to a specific phase. You can use the –W option to specify options for the assembler, compiler, or linker.

> **Note**
>
> A given PGI compiler command invokes the compiler driver, which parses the command-line, and generates the appropriate commands for the compiler, assembler, and linker.

Related options: –Minfo, –V, –v

–w

Do not print warning messages.

Default: The compiler prints warning messages.

Usage: In the following example no warning messages are printed.

```
$ pgfortran -w myprog.f
```

Description: Use the –w option to not print warning messages. Sometimes the compiler issues many warning in which you may have no interest. You can use this option to not issue those warnings.

Related options: –silent

–Xs

Use legacy standard mode for C and C++.

Default: None.

Usage: In the following example the compiler uses legacy standard mode.

```
$ pgcc -Xs myprog.c
```

Description: Use this option to use legacy standard mode for C and C++. Further, this option implies -alias=traditional.

Related options: –alias, –Xt

–Xt

Use legacy transitional mode for C and C++.

Default: None.

Usage: In the following example the compiler uses legacy transitional mode.

```
$ pgcc -Xt myprog.c
```

Description: Use this option to use legacy transitional mode for C and C++. Further, this option implies -alias=traditional.

Related options: –alias, –Xs

C and C++ -specific Compiler Options

There are a large number of compiler options specific to the PGCC and PGC++ compilers, especially PGC++. This section provides the details of several of these options, but is not exhaustive. For a complete list of available options, including an exhaustive list of PGC++ options, use the –help command-line option. For further detail on a given option, use –help and specify the option explicitly, as described in "–help" on page 44.

–A

(pgcpp only) Instructs the PGC++ compiler to accept code conforming to the proposed ANSI C++ standard, issuing errors for non-conforming code.

Default: By default, the compiler accepts code conforming to the standard C++ Annotated Reference Manual.

Usage: The following command-line requests ANSI conforming C++.

```
$ pgcpp -A hello.cc
```

Description: Use this option to instruct the PGC++ compiler to accept code conforming to the proposed ANSI C++ standard and to issues errors for non-conforming code.

Related options:–a, –b and +p.

–a

(pgcpp only) Instructs the PGC++ compiler to accept code conforming to the proposed ANSI C++ standard, issuing warnings for non-conforming code.

Default: By default, the compiler accepts code conforming to the standard C++ Annotated Reference Manual.

Usage: The following command-line requests ANSI conforming C++, issuing warnings for non-conforming code.

```
$ pgcpp -a hello.cc
```

Description: Use this option to instruct the PGC++ compiler to accept code conforming to the proposed ANSI C++ standard and to issues warnings for non-conforming code.

Related options:–A, –b and +p.

–alias

select optimizations based on type-based pointer alias rules in C and C++.

Syntax:

```
-alias=[ansi|traditional]
```

Default: None.

Usage: The following command-line enables optimizations.

```
$ pgcpp -alias=ansi hello.cc
```

Description: Use this option to select optimizations based on type-based pointer alias rules in C and C++.

ansi
: Enable optimizations using ANSI C type-based pointer disambiguation

traditional
: Disable type-based pointer disambiguation

Related options: –Xt

--[no_]alternative_tokens

(pgcpp only) Enables or disables recognition of alternative tokens. These are tokens that make it possible to write C++ without the use of the comma (,) , [,], #, &, ^, and characters. The alternative tokens include the operator keywords (e.g., and, bitand, etc.) and digraphs.

Default:. The default behavior is --no_alternative_tokens, that is, to disable recognition of alternative tokens.

Usage: The following command-line enables alternative token recognition.

```
$ pgcpp --alternative_tokens hello.cc
```

(pgcpp only) Use this option to enable or disable recognition of alternative tokens. These tokens make it possible to write C++ without the use of the comma (,), [,], #, &, ^, and characters. The alternative tokens include digraphs and the operator keywords, such as *and*, *bitand*, and so on. The default behavior is disabled recognition of alternative tokens: --no_alternative_tokens.

Related options:

–B

(pgcc and pgcpp only) Enables use of C++ style comments starting with // in C program units.

Default: The PGCC ANSI and K&R C compiler does not allow C++ style comments.

Usage: In the following example the compiler accepts C++ style comments.

```
$ pgcc -B myprog.cc
```

Description: Use this option to enable use of C++ style comments starting with // in C program units.

Related options: –Mcpp

–b

(pgcpp only) Enables compilation of C++ with cfront 2.1 compatibility and acceptance of anachronisms.

Default: The compiler does not accept cfront language constructs that are not part of the C++ language definition.

Usage: In the following example the compiler accepts cfront constructs.

```
$ pgcpp -b myprog.cc
```

Description: Use this option to enable compilation of C++ with cfront 2.1 compatibility. The compiler then accepts language constructs that, while not part of the C++ language definition, are accepted by the AT&T C++ Language System (cfront release 2.1).

This option also enables acceptance of anachronisms.

Related options: —cfront2.1, –b3 , —cfront3.0, +p, –A

–b3

(pgcpp only) Enables compilation of C++ with cfront 3.0 compatibility and acceptance of anachronisms.

Default: The compiler does not accept cfront language constructs that are not part of the C++ language definition.

Usage: In the following example, the compiler accepts cfront constructs.

```
$ pgcpp -b3 myprog.cc
```

Description: Use this option to enable compilation of C++ with cfront 3.0 compatibility. The compiler then accepts language constructs that, while not part of the C++ language definition, are accepted by the AT&T C++ Language System (cfront release 3.0).

This option also enables acceptance of anachronisms.

Related options: —cfront2.1, –b, —cfront3.0, +p, –A

--[no_]bool

(pgcpp only) Enables or disables recognition of bool.

Default: The compile recognizes bool: --bool.

Usage: In the following example, the compiler does not recognize bool.

```
$ pgcpp --no_bool myprog.cc
```

Description: Use this option to enable or disable recognition of bool.

Related options: None.

--[no_]builtin

Compile with or without math subroutine builtin support.

Default: The default is to compile with math subroutine support: --builtin.

Usage: In the following example, the compiler does not build with math subroutine support.

```
$ pgcpp --no_builtin myprog.cc
```

Description: Use this option to enable or disable compiling with math subroutine builtin support. When you compile with math subroutine builtin support, the selected math library routines are inlined.

Related options:

--cfront_2.1

(pgcpp only) Enables compilation of C++ with cfront 2.1 compatibility and acceptance of anachronisms.

Default: The compiler does not accept cfront language constructs that are not part of the C++ language definition.

Usage: In the following example, the compiler accepts cfront constructs.

```
$ pgcpp --cfront_2.1 myprog.cc
```

Description: Use this option to enable compilation of C++ with cfront 2.1 compatibility. The compiler then accepts language constructs that, while not part of the C++ language definition, are accepted by the AT&T C++ Language System (cfront release 2.1).

This option also enables acceptance of anachronisms.

Related options: –b, –b3, —cfront3.0, +p, –A

--cfront_3.0

(pgcpp only) Enables compilation of C++ with cfront 3.0 compatibility and acceptance of anachronisms.

Default: The compiler does not accept cfront language constructs that are not part of the C++ language definition.

Usage: In the following example, the compiler accepts cfront constructs.

```
$ pgcpp --cfront_3.0 myprog.cc
```

Description: Use this option to enable compilation of C++ with cfront 3.0 compatibility. The compiler then accepts language constructs that, while not part of the C++ language definition, are accepted by the AT&T C++ Language System (cfront release 3.0).

This option also enables acceptance of anachronisms.

Related options: --cfront2.1, –b, –b3, +p, –A

--compress_names

Compresses long function names in the file.

Default: The compiler does not compress names: --no_compress_names.

Usage: In the following example, the compiler compresses long function names.

```
$ pgcpp --ccmpress_names myprog.cc
```

Description: Use this option to specify to compress long function names. Highly nested template parameters can cause very long function names. These long names can cause problems for older assemblers. Users encountering these problems should compile all C++ code, including library code with the switch `--compress_names`. Libraries supplied by PGI work with --compress_names.

Related options: None.

--create_pch filename

(pgcpp only) If other conditions are satisfied, create a precompiled header file with the specified name.

Note

If --pch (automatic PCH mode) appears on the command line following this option, its effect is erased.

Default: The compiler does not create a precompiled header file.

Usage: In the following example, the compiler creates a precompiled header file, `hdr1`.

```
$ pgcpp --create_pch hdr1 myprog.cc
```

Description: If other conditions are satisfied, use this option to create a precompiled header file with the specified name.

Related options: --pch

--diag_error tag

(pgcpp only) Overrides the normal error severity of the specified diagnostic messages.

Default: The compiler does not override normal error severity.

Description: Use this option to override the normal error severity of the specified diagnostic messages. The message(s) may be specified using a mnemonic error tag or using an error number.

Related options:--diag_remark tag, --diag_suppress tag, --diag_warning tag, --display_error_number

--diag_remark tag

(pgcpp only) Overrides the normal error severity of the specified diagnostic messages.

Default: The compiler does not override normal error severity.

Description: Use this option to override the normal error severity of the specified diagnostic messages. The message(s) may be specified using a mnemonic error tag or using an error number.

Related options: --diag_error tag, --diag_suppress tag, --diag_warning tag, --display_error_number

--diag_suppress tag

(pgcpp only) Overrides the normal error severity of the specified diagnostic messages.

Default: The compiler does not override normal error severity.

Usage: In the following example, the compiler overrides the normal error severity of the specified diagnostic messages.

```
$ pgcpp --diag_suppress error_tag prog.cc
```

Description: Use this option to override the normal error severity of the specified diagnostic messages. The message(s) may be specified using a mnemonic error tag or using an error number.

Related options:--diag_error tag, --diag_remark tag, --diag_warning tag, --diag_error_number

--diag_warning tag

(pgcpp only) Overrides the normal error severity of the specified diagnostic messages.

Default: The compiler does not override normal error severity.

Usage: In the following example, the compiler overrides the normal error severity of the specified diagnostic messages.

```
$ pgcpp --diag_suppress an_error_tag myprog.cc
```

Description: Use this option to override the normal error severity of the specified diagnostic messages. The message(s) may be specified using a mnemonic error tag or using an error number.

Related options: --diag_error tag, --diag_remark tag, --diag_suppress tag, --diag_error_number

--display_error_number

(pgcpp only) Displays the error message number in any diagnostic messages that are generated. The option may be used to determine the error number to be used when overriding the severity of a diagnostic message.

Default: The compiler does not display error message numbers for generated diagnostic messages.

Usage: In the following example, the compiler displays the error message number for any generated diagnostic messages.PLEASE PROVIDE ONE

```
$ pgcpp --display_error_number myprog.cc
```

Description: Use this option to display the error message number in any diagnostic messages that are generated. You can use this option to determine the error number to be used when overriding the severity of a diagnostic message.

Related options: --diag_error tag, --diag_remark tag, --diag_suppress tag, --diag_warning tag

-e<number>

(pgcpp only) Set the C++ front-end error limit to the specified <number>.

--[no_]exceptions

(pgcpp only) Enables or disables exception handling support.

Default: The compiler provides exception handling support: --exceptions.

Usage: In the following example, the compiler does not provide exception handling support.

```
$ pgcpp --no_exceptions myprog.cc
```

Description: Use this option to enable or disable exception handling support.

Related options:--zc_eh

--gnu_extensions

(pgcpp only) Allows GNU extensions.

Default: The compiler does not allow GNU extensions.

Usage: In the following example, the compiler allows GNU extensions.

```
$ pgcpp --gnu_extensions myprog.cc
```

Description: Use this option to allow GNU extensions, such as "include next", which are required to compile Linux system header files.

Related options:--zc_eh, --gnu_version

--gnu_version <num>

(pgcpp only) Sets the GNU C++ compatibility version.

Default: The compiler uses the latest version.

Usage: In the following example, the compiler sets the GNU version to 4.3.4.

```
$ pgcpp --gnu_version 4.3.4 myprog.cc
```

Description: Use this option to set the GNU C++ compatibility version to use when you compile.

Related options:--gnu_extensions

--[no]llalign

(pgcpp only) Enables or disables alignment of long long integers on long long boundaries.

Default: The compiler aligns long long integers on long long boundaries: --llalign.

Usage: In the following example, the compiler does not align long long integers on long long boundaries.

```
$ pgcpp --nollalign myprog.cc
```

Description: Use this option to allow enable or disable alignment of long long integers on long long boundaries.

Related options: –Mipa=[no]align

–M

Generates a list of make dependencies and prints them to stdout.

Note

The compilation stops after the preprocessing phase.

Default: The compiler does not generate a list of make dependencies.

Usage: In the following example, the compiler generates a list of make dependencies.

```
$ pgcpp -M myprog.cc
```

Description: Use this option to generate a list of make dependencies and prints them to stdout.

Related options:–MD, –P

–MD

Generates a list of make dependencies and prints them to a file.

Default: The compiler does not generate a list of make dependencies.

Usage: In the following example, the compiler generates a list of make dependencies and prints them to the file myprog.d.

```
$ pgcpp -MD myprog.cc
```

Description: Use this option to generate a list of make dependencies and prints them to a file. The name of the file is determined by the name of the file under compilation.dependencies_file<file>.

Related options:–M, –P

--optk_allow_dollar_in_id_chars

(pgcpp only) Accepts dollar signs ($) in identifiers.

Default: The compiler does not accept dollar signs ($) in identifiers.

Usage: In the following example, the compiler allows dollar signs ($) in identifiers.

```
$ pgcpp -optk_allow_dollar_in_id_chars myprog.cc
```

Description: Use this option to instruct the compiler to accept dollar signs ($) in identifiers.

- -microsoft_version <num>

Sets the Microsoft C++ compatibility version.

Default: The compiler uses the latest version.

Usage: In the following example, the compiler sets the Microsoft C++ version to 1.5.

```
$ pgcpp -microsoft_version 1.5 myprog.cc
```

Description: Use this option to set the GNU C++ compatibility version to use when you compile.

Related options:--gnu_extensions

–P

Halts the compilation process after preprocessing and writes the preprocessed output to a file.

Default: The compiler produces an executable file.

Usage: In the following example, the compiler produces the preprocessed file myprog.i in the current directory.

```
$ pgcpp -P myprog.cc
```

Description: Use this option to halt the compilation process after preprocessing and write the preprocessed output to a file. If the input file is `filename.c` or `filename.cc.`, then the output file is `filename.i`.

Related options: –C,–c,–E, –Mkeepasm, –o, –S

-+p

(pgcpp only) Disallow all anachronistic constructs.

Default: The compiler disallows all anachronistic constructs.

Usage: In the following example, the compiler disallows all anachronistic constructs.

```
$ pgcpp -+p myprog.cc
```

Description: Use this option to disallow all anachronistic constructs.

Related options: None.

--pch

(pgcpp only) Automatically use and/or create a precompiled header file.

Note

If --use_pch or --create_pch (manual PCH mode) appears on the command line following this option, this option has no effect.

Default: The compiler does not automatically use or create a precompiled header file.

Usage: In the following example, the compiler automatically uses a precompiled header file.

```
$ pgcpp --pch myprog.cc
```

Description: Use this option to automatically use and/or create a precompiled header file.

Related options:--create_pch, --pc_dir, --use_pch

--pch_dir directoryname

(pgcpp only) Specifies the directory in which to search for and/or create a precompiled header file.

The compiler searches your PATH for precompiled header files / use or create a precompiled header file.

Usage: In the following example, the compiler searches in the directory `myhdrdir` for a precompiled header file.

```
$ pgcpp --pch_dir myhdrdir myprog.cc
```

Description: Use this option to specify the directory in which to search for and/or create a precompiled header file. You may use this option with automatic PCH mode (--pch) or manual PCH mode (--create_pch or --use_pch).

Related options:--create_pch, --pch, --use_pch

--[no_]pch_messages

(pgcpp only) Enables or disables the display of a message indicating that the current compilation used or created a precompiled header file.

The compiler displays a message when it uses or creates a precompiled header file.

In the following example, no message is displayed when the precompiled header file located in `myhdrdir` is used in the compilation.

```
$ pgcpp --pch_dir myhdrdir --no_pch_messages myprog.cc
```

Description: Use this option to enable or disable the display of a message indicating that the current compilation used or created a precompiled header file.

Related options:--pch_dir

--preinclude=<filename>

(pgcpp only) Specifies the name of a file to be included at the beginning of the compilation.

In the following example, the compiler includes the file `incl_file.c` at the beginning of the compilation. me

```
$ pgcpp --preinclude=incl_file.c myprog.cc
```

Description: Use this option to specify the name of a file to be included at the beginning of the compilation. For example, you can use this option to set system-dependent macros and types.

Related options: None.

--use_pch filename

(pgcpp only) Uses a precompiled header file of the specified name as part of the current compilation.

Note

If --pch (automatic PCH mode) appears on the command line following this option, its effect is erased.

Default: The compiler does not use a precompiled header file.

In the following example, the compiler uses the precompiled header file, `hdr1` as part of the current compilation.

```
$ pgcpp --use_pch hdr1 myprog.cc
```

Use a precompiled header file of the specified name as part of the current compilation. If --pch (automatic PCH mode) appears on the command line following this option, its effect is erased.

Related options:--create_pch, --pch_dir, --pch_messages

--[no_]using_std

(pgcpp only) Enables or disables implicit use of the std namespace when standard header files are included.

Default: The compiler uses std namespace when standard header files are included: --using_std.

Usage: The following command-line disables implicit use of the std namespace:

```
$ pgcpp --no_using_std hello.cc
```

Description: Use this option to enable or disable implicit use of the std namespace when standard header files are included in the compilation.

Related options: –M[no]stddef

–t

(pgcpp only) Control instantiation of template functions.

–t [arg]

Default: No templates are instantiated.

Usage: In the following example, all templates are instantiated.

```
$ pgcpp -tall myprog.cc
```

Description: Use this option to control instantiation of template functions. The argument is one of the following:

all
: Instantiates all functions whether or not they are used.

local
: Instantiates only the functions that are used in this compilation, and forces those functions to be local to this compilation.

  Note: This may cause multiple copies of local static variables. If this occurs, the program may not execute correctly.

none
: Instantiates no functions. This is the default.

used
: Instantiates only the functions that are used in this compilation.

Example: In the following example, all templates are instantiated.

```
$ pgcpp  -tall myprog.cc
```

–X

(pgcpp only) Generates cross-reference information and places output in the specified file.

Syntax:

–Xfoo

where foo is the specifies file for the cross reference information.

Default: The compiler does not generate cross-reference information.

Usage: In the following example, the compiler generates cross-reference information, placing it in the file: `xreffile`.

```
$ pgcpp -Xxreffile myprog.cc
```

Description: Use this option to generate cross-reference information and place output in the specified file. This is an EDG option.

Related options: None.

--[no]zc_eh

(Linux only) Generates zero-overhead exception regions.

Default: The compiler generates zero-overhead exception regions. To use exception handling with setjmp and longjmp, use the --nozc_eh flag.

Usage: The following command-line enables zero-overhead exception regions:

```
$ pgcpp --zc_eh ello.cc
```

Description: Use this option to generate zero-overhead exception regions. The --zc_eh option defers the cost of exception handling until an exception is thrown. For a program with many exception regions and few throws, this option may lead to improved run-time performance.

This option is compatible with C++ code that was compiled with previous versions of PGI C++. pgCC uses this option for low cost exception handling by default so code compiled prior to 11.0 must be recompiled.

To use exception handling with setjmp and longjmp, use the --nozc_eh flag.

Note

The --zc_eh option is available only on newer Linux systems that supply the system unwind libraries in libgcc_eh and on Windows.

Related options: --[no]exceptions, --nozc_eh

–M Options by Category

This section describes each of the options available with –M by the categories:

Code generation	Fortran Language Controls	Optimization	Environment

C/C++ Language Controls Inlining Miscellaneous

For a complete alphabetical list of all the options, refer to "–M Options Summary" on page 55.

The following sections provide detailed descriptions of several, but not all, of the –M<pgflag> options. For a complete alphabetical list of all the options, refer to "–M Options Summary" on page 55. These options are grouped according to categories and are listed with exact syntax, defaults, and notes concerning similar or related options. For the latest information and description of a given option, or to see all available options, use the –help command-line option, described in"–help" on page 44.

Code Generation Controls

This section describes the –M<pgflag> options that control code generation.

Default: For arguments that you do not specify, the default code generation controls are these:

nodaz	norecursive	nosecond_underscore
noflushz	noreentrant	nostride0
largeaddressaware	noref_externals	signextend

Related options: –D, –I, –L, –l, –U

The following list provides the syntax for each–M<pgflag> option that controls code generation. Each option has a description and, if appropriate, any related options.

–Mdaz
: Set IEEE denormalized input values to zero; there is a performance benefit but misleading results can occur, such as when dividing a small normalized number by a denormalized number. To take effect, this option must be set for the main program.

–Mnodaz
: Do not treat denormalized numbers as zero.To take effect, this option must be set for the main program.

–Mnodwarf
: Specifies not to add DWARF debug information; must be used in combination with `-g`.

–Mdwarf1
: Generate DWARF1 format debug information; must be used in combination with `-g`.

–Mdwarf2
: Generate DWARF2 format debug information; must be used in combination with `-g`.

–Mdwarf3
: Generate DWARF3 format debug information; must be used in combination with `-g`.

–Mflushz
: Set SSE flush-to-zero mode; if a floating-point underflow occurs, the value is set to zero.To take effect, this option must be set for the main program.

–Mnoflushz
: Do not set SSE flush-to-zero mode; generate underflows.To take effect, this option must be set for the main program.

–Mfunc32
Align functions on 32-byte boundaries.

–Minstrument[=functions] linx86-64 only
Generate additional code to enable instrumentation of functions. The option –Minstrument=functions is the same as –Minstrument.

Implies –Minfo=ccff and –Mframe.

–Mlargeaddressaware=[no]
[Win64 only] Generates code that allows for addresses greater than 2GB, using RIP-relative addressing.

Use–Mlargeaddressaware=no for a direct addressing mechanism that restricts the total addressable memory.

Note

Do not use –Mlargeaddressaware=no if the object file will be placed in a DLL.

If –Mlargeaddressaware=no is used to compile any object file, it must also be used when linking.

–Mlarge_arrays
Enable support for 64-bit indexing and single static data objects larger than 2GB in size. This option is default in the presence of –mcmodel=medium. Can be used separately together with the default small memory model for certain 64-bit applications that manage their own memory space. For more information, refer tothe "Programming Considerations for 64-Bit Enviornments" chapter of the PGI Compiler User's Guide.

–Mmpi=option
–Mmpi adds the include and library options to the compile and link commands necessary to build an MPI application using MPI header files and libraries.

To use –Mmpi, you must have a version of MPI installed on your system.

This option tells the compiler to use the headers and libraries for the specified version of MPI.

On Windows, PGI compilers and tools support Microsoft's implementation of MPI, MSMPI. This version of MPI is available with Microsoft's HPC Pack 2008 SDK.

The -Mmpi options are as specified:

- –Mmpi=hpmpi - (Linux only) Select the HP-MPI communication libraries and associated header files if they are installed.
- –Mmpi=mpich1 - Selects preconfigured MPICH-1 communication libraries.
- –Mmpi=mpich2 - Selects preconfigured MPICH-2 communication libraries.
- –Mmpi=msmpi - Select Microsoft MSMPI libraries.
- –Mmpi=mvapich1 - (Linux only) Selects default MVAPICH communication libraries that are available only from the PGI Cluster Development Kit

For more information on these options, refer to the chapter "Using MPI in PVF in the PGI Compiler User's Guide.

Note

The user can set the environment variables `MPIDIR` and `MPILIBNAME` to override the default locations for the MPI directory and library name.

On Windows, the user can set the appropriate environment variable, either `CCP_HOME` or `CCP_SDK` to override the default location for the directory associated with using MSMPI.

For `-Mmpi=msmpi` to work, the `CCP_HOME` environment variable must be set. When the Microsoft HPC Pack 2008 SDK is installed, this variable is typically set to point to the MSMPI library directory.

–Mnolarge_arrays
Disable support for 64-bit indexing and single static data objects larger than 2GB in size. When placed after `-mcmodel=medium` on the command line, disables use of 64-bit indexing for applications that have no single data object larger than 2GB.

–Mnomain
Instructs the compiler not to include the object file that calls the Fortran main program as part of the link step. This option is useful for linking programs in which the main program is written in C/C++ and one or more subroutines are written in Fortran (pgf77, pgf95, pgfortran, and pghpf only).

–M[no]movnt
Instructs the compiler to generate nontemporal move and prefetch instructions even in cases where the compiler cannot determine statically at compile-time that these instructions will be beneficial.

–M[no]pre
enables or disables partial redundancy elimination.

–Mprof[=option[,option,...]]
Set performance profiling options. Use of these options causes the resulting executable to create a performance profile that can be viewed and analyzed with the *PGPROF* performance profiler. In the descriptions that follow, PGI-style profiling implies compiler-generated source instrumentation. MPICH-style profiling implies the use of instrumented wrappers for MPI library routines.

The option argument can be any of the following:

`[no]ccff`
Enable [disable] common compiler feedback format, CCFF, information.

`dwarf`
Generate limited DWARF symbol information sufficient for most performance profilers.

`func`
Perform PGI-style function-level profiling.

`hpmpi`
Use the profiled HPMPI communication library. Implies `-Mpfi=hpmpi`. For more information, refer to "Using HP-MPI on Linux" on page 213.

`hwcts`
Generate a profile using event-based sampling of hardware counters via the PAPI interface. (linux86-64 platform only; PAPI must be installed).

`lines`
Perform PGI-style line-level profiling.

`mpich1`
Perform MPICH-style profiling for MPICH-1. Implies `–Mmpi=mpich1`. (Linux only).

`mpich2`
Perform MPICH-style profiling for MPICH-2. Implies `–Mmpi=mpich2`. (Linux with MPI support licence privileges only.) For more information, refer to "Using MPICH-2 on Linux" on page 212.

`msmpi`
Perform MPICH-style profiling for Microsoft MPI. Implies `–Mmpi=msmpi`.

This option is valid only if Microsoft HPC Pack 2008 SDK is installed.

For more information, refer to "Using MSMPI on Windows" in the PGI Compiler User's Guide"Using MPI in PVF" in the PVF User's Guide..

`mvapich1`
Use profiled MVAPICH communication library. Implies `–Mmpi=mvapich1`. (Linux only). For or more information, refer to "Using MVAPICH on Linux" on page 213

`time`
Generate a profile using time-based instruction-level statistical sampling. This is equivalent to `–pg`, except that the profile is saved to a file names `pgprof.out` rather than `gmon.out`.

`–Mrecursive`
instructs the compiler to allow Fortran subprograms to be called recursively.

`–Mnorecursive`
Fortran subprograms may not be called recursively.

`–Mref_externals`
force references to names appearing in EXTERNAL statements (pgf77, pgf95, pgfortran, and pghpf only).

`–Mnoref_externals`
do not force references to names appearing in EXTERNAL statements (pgf77, pgf95, pgfortran, and pghpf only).

`–Mreentrant`
instructs the compiler to avoid optimizations that can prevent code from being reentrant.

`–Mnoreentrant`
instructs the compiler not to avoid optimizations that can prevent code from being reentrant.

`–Msecond_underscore`
instructs the compiler to add a second underscore to the name of a Fortran global symbol if its name already contains an underscore. This option is useful for maintaining compatibility with object code compiled using g77, which uses this convention by default (pgf77, pgf95, pgfortran, and pghpf only).

`–Mnosecond_underscore`
instructs the compiler not to add a second underscore to the name of a Fortran global symbol if its name already contains an underscore (pgf77, pgf95, pgfortran, and pghpf only).

–Msignextend
instructs the compiler to extend the sign bit that is set as a result of converting an object of one data type to an object of a larger signed data type.

–Mnosignextend
instructs the compiler not to extend the sign bit that is set as the result of converting an object of one data type to an object of a larger data type.

–Msafe_lastval
When a scalar is used after a loop, but is not defined on every iteration of the loop, the compiler does not by default parallelize the loop. However, this option tells the compiler it's safe to parallelize the loop. For a given loop the last value computed for all scalars makes it safe to parallelize the loop.

–Mstride0
instructs the compiler to inhibit certain optimizations and to allow for stride 0 array references. This option may degrade performance and should only be used if zero-stride induction variables are possible.

–Mnostride0
instructs the compiler to perform certain optimizations and to disallow for stride 0 array references.

–Munix
use UNIX symbol and parameter passing conventions for Fortran subprograms (pgf77, pgf95, pgfortran, and pghpf for Win32 only).

–Mvarargs
force Fortran program units to assume procedure calls are to C functions with a varargs-type interface (pgf77, pgf95, and pgfortran only).

C/C++ Language Controls

This section describes the –M<pgflag> options that affect C/C++ language interpretations by the PGI C and C++ compilers. These options are only valid to the pgcc and pgcpp compiler drivers.

Default: For arguments that you do not specify, the defaults are as follows:

noasmkeyword	nosingle
dollar,_	schar

Usage:

In this example, the compiler allows the asm keyword in the source file.

```
$ pgcc -Masmkeyword myprog.c
```

In the following example, the compiler maps the dollar sign to the dot character.

```
$ pgcc -Mdollar,. myprog.c
```

In the following example, the compiler treats floating-point constants as float values.

```
$ pgcc -Mfcon myprog.c
```

In the following example, the compiler does not convert float parameters to double parameters.

```
$ pgcc -Msingle myprog.c
```

Without –Muchar or with –Mschar, the variable ch is a signed character:

```
char ch;
signed char sch;
```

If –Muchar is specified on the command line:

```
$ pgcc -Muchar myprog.c
```

char ch in the preceding declaration is equivalent to:

```
unsigned char ch;
```

The following list provides the syntax for each–M<pgflag> option that controls code generation. Each option has a description and, if appropriate, any related options.

–Masmkeyword
: instructs the compiler to allow the asm keyword in C source files. The syntax of the asm statement is as follows:

  ```
  asm("statement");
  ```

  Where statement is a legal assembly-language statement. The quote marks are required.

  Note. The current default is to support gcc's extended asm, where the syntax of extended asm includes asm strings. The –M[no]asmkeyword switch is useful only if the target device is a Pentium 3 or older cpu type (–tp piii|p6|k7|athlon|athlonxp|px).

–Mnoasmkeyword
: instructs the compiler not to allow the asm keyword in C source files. If you use this option and your program includes the asm keyword, unresolved references will be generated

–Mdollar,char
: char specifies the character to which the compiler maps the dollar sign ($). The PGCC compiler allows the dollar sign in names; ANSI C does not allow the dollar sign in names.

–M[no]eh_frame
: instructs the linker to keep eh_frame call frame sections in the executable.

  Note

  The eh_frame option is available only on newer Linux or Windows systems that supply the system unwind libraries.

–Mfcon
: instructs the compiler to treat floating-point constants as float data types, instead of double data types. This option can improve the performance of single-precision code.

–M[no]m128
: instructs the compiler to recognize [ignore] __m128, __m128d, and __m128i datatypes. floating-point constants as float data types, instead of double data types. This option can improve the performance of single-precision code.

–Mschar
: specifies signed char characters. The compiler treats "plain" char declarations as signed char.

–Msingle
do not to convert float parameters to double parameters in non-prototyped functions. This option can result in faster code if your program uses only float parameters. However, since ANSI C specifies that routines must convert float parameters to double parameters in non-prototyped functions, this option results in non#ANSI conformant code.

–Mnosingle
instructs the compiler to convert float parameters to double parameters in non-prototyped functions.

–Muchar
instructs the compiler to treat "plain" char declarations as unsigned char.

Environment Controls

This section describes the –M<pgflag> options that control environments.

Default: For arguments that you do not specify, the default environment option depends on your configuration.

The following list provides the syntax for each –M<pgflag> option that controls environments. Each option has a description and, if appropriate,a list of any related options.

–Mlfs
(32-bit Linux only) link in libraries that enable file I/O to files larger than 2GB (Large File Support).

–Mnostartup
instructs the linker not to link in the standard startup routine that contains the entry point (_start) for the program.

> **Note**
>
> If you use the –Mnostartup option and do not supply an entry point, the linker issues the following error message: Warning: cannot find entry symbol _start

–M[no]smartalloc[=huge|h[uge:<n>|hugebss|nohuge]
adds a call to the routine mallopt in the main routine. This option supports large TLBs on Linux and Windows. This option must be used to compile the main routine to enable optimized malloc routines.

The option arguments can be any of the following:

huge
Link in the huge page runtime library.

Enables large 2-megabyte pages to be allocated. The effect is to reduce the number of TLB entries required to execute a program. This option is most effective on Barcelona and Core 2 systems; older architectures do not have enough TLB entries for this option to be beneficial. By itself, the huge suboption tries to allocate as many huge pages as required.

huge:<n>
Link the huge page runtime library and allocate n huge pages. Use this suboption to limit the number of huge pages allocated to n.

You can also limit the pages allocated by using the environment variable PGI_HUGE_PAGES.

hugebss
(64-bit only) Puts the BSS section in huge pages; attempts to put a program's uninitialized data section into huge pages.

Note

This flag dynamically links the library `libhugetlbfs_pgi` even if `-Bstatic` is used.

nohuge
Overrides a previous `-Msmartalloc=huge` setting.

Tip

To be effective, this switch must be specified when compiling the file containing the Fortran, C, or C++ main program.

`–M[no]stddef`
instructs the compiler not to predefine any macros to the preprocessor when compiling a C program.

`–Mnostdinc`
instructs the compiler to not search the standard location for include files.

`–Mnostdlib`
instructs the linker not to link in the standard libraries `libpgftnrtl.a`, `libm.a`, `libc.a`, and `libpgc.a` in the library directory `lib` within the standard directory. You can link in your own library with the –l option or specify a library directory with the –L option.

Fortran Language Controls

This section describes the –M<pgflag> options that affect Fortran language interpretations by the PGI Fortran compilers. These options are valid only for the pghpf, pgf77, pgf95, and pgfortran compiler drivers.

Default: For arguments that you do not specify, the defaults are as follows:

nobackslash	nodefaultunit	dollar,_	noonetrip	nounixlogical
nodclchk	nodlines	noiomutex	nosave	noupcase

The following list provides the syntax for each–M<pgflag> option that affect Fortran language interpretations. Each option has a description and, if appropriate, a list of any related options.

`–Mallocatable=95|03`
controls whether Fortran 95 or Fortran 2003 semantics are used in allocatable array assignments. The default behavior is to use Fortran 95 semantics; the 03 option instructs the compiler to use Fortran 2003 semantics.

`–Mbackslash`
the compiler treats the backslash as a normal character, and not as an escape character in quoted strings.

`–Mnobackslash`
the compiler recognizes a backslash as an escape character in quoted strings (in accordance with standard C usage).

–Mcuda
the compiler enables Cuda Fortran.

The following suboptions exist:

Note

If more than one option is on the command line, all the specified options occur.

cc10
Generate code for compute capability 1.0.

cc11
Generate code for compute capability 1.1.

cc12
Generate code for compute capability 1.2.

cc13
Generate code for compute capability 1.3.

cc20
Generate code for compute capability 2.0.

cuda2.3 or 2.3
Sets the toolkit compatibility version to 2.3.

cuda3.0 or 3.0
Sets the toolkit compatibility version to 3.0.

cuda3.1 or 3.1
Sets the toolkit compatibility version to 3.0.

cuda3.2 or 3.2
Sets the toolkit compatibility version to 3.1.

Note

Compiling with the CUDA 3.1 or CUDA 3.2 toolkit, either by using the `-ta=nvidia:cuda3.2` or `-ta=nvidia:cuda3.1` option or by adding `set CUDAVERSION=3.2` or `set CUDAVERSION=3.1` to the `siterc` file, generates binaries that may not work on machines with a 2.3 CUDA driver.

`pgaccelinfo` prints the driver version as the first line of output.

For a 2.3 driver: `CUDA Driver Version 2030`
For a 3.0 driver: `CUDA Driver Version 3000`
For a 3.1 driver: `CUDA Driver Version 3010`
For a 3.2 driver: `CUDA Driver Version 3020`

emu
Enable Cuda Fortran emulation mode.

fastmath
: Use routines from the fast math library.

[no]flushz
: Enable[disable] flush-to-zero mode for floating point computations in the GPU code generated for CUDA Fortran kernels.

keepbin
: Keep the generated binary (.bin) file for CUDA Fortran.

keepgpu
: Keep the generated GPU code for CUDA Fortran.

keepptx
: Keep the portable assembly (.ptx) file for the GPU code.

maxregcount:n
: Specify the maximum number of registers to use on the GPU. Leaving this blank indicates no limit.

nofma
: Do not generate fused multiply-add instructions.

ptxinfo
: Show PTXAS informational messages during compilation.

–Mdclchk
: the compiler requires that all program variables be declared.

–Mnodclchk
: the compiler does not require that all program variables be declared.

–Mdefaultunit
: the compiler treats "*" as a synonym for standard input for reading and standard output for writing.

–Mnodefaultunit
: the compiler treats "*" as a synonym for unit 5 on input and unit 6 on output.

–Mdlines
: the compiler treats lines containing "D" in column 1 as executable statements (ignoring the "D").

–Mnodlines
: the compiler does not treat lines containing "D" in column 1 as executable statements (does not ignore the "D").

–Mdollar,char
: char specifies the character to which the compiler maps the dollar sign. The compiler allows the dollar sign in names.

–Mextend
: the compiler accepts 132-column source code; otherwise it accepts 72-column code.

–Mfixed
: the compiler assumes input source files are in FORTRAN 77-style fixed form format.

–Mfree
the compiler assumes the input source files are in Fortran 90/95 freeform format.

–Miomutex
the compiler generates critical section calls around Fortran I/O statements.

–Mnoiomutex
the compiler does not generate critical section calls around Fortran I/O statements.

–Monetrip
the compiler forces each DO loop to execute at least once.

–Mnoonetrip
the compiler does not force each DO loop to execute at least once. This option is useful for programs written for earlier versions of Fortran.

–Msave
the compiler assumes that all local variables are subject to the SAVE statement. Note that this may allow older Fortran programs to run, but it can greatly reduce performance.

–Mnosave
the compiler does not assume that all local variables are subject to the SAVE statement.

–Mstandard
the compiler flags non-ANSI-conforming source code.

–Munixlogical
directs the compiler to treat logical values as true if the value is non-zero and false if the value is zero (UNIX F77 convention.) When –Munixlogical is enabled, a logical value or test that is non-zero is .TRUE., and a value or test that is zero is .FALSE.. In addition, the value of a logical expression is guaranteed to be one (1) when the result is .TRUE..

–Mnounixlogical
directs the compiler to use the VMS convention for logical values for true and false. Even values are true and odd values are false.

–Mupcase
the compiler preserves uppercase letters in identifiers. With –Mupcase, the identifiers "X" and "x" are different. Keywords must be in lower case. This selection affects the linking process. If you compile and link the same source code using –Mupcase on one occasion and –Mnoupcase on another, you may get two different executables - depending on whether the source contains uppercase letters. The standard libraries are compiled using the default –Mnoupcase .

–Mnoupcase
the compiler converts all identifiers to lower case. This selection affects the linking process: If you compile and link the same source code using –Mupcase on one occasion and –Mnoupcase on another, you may get two different executables (depending on whether the source contains uppercase letters). The standard libraries are compiled using –Mnoupcase.

Inlining Controls

This section describes the –M<pgflag> options that control function inlining. Before looking at all the options, let's look at a couple examples.

Usage: In the following example, the compiler extracts functions that have 500 or fewer statements from the source file `myprog.f` and saves them in the file `extract.il`.

```
$ pgfortran -Mextract=500 -o extract.il myprog.f
```

In the following example, the compiler inlines functions with fewer than approximately 100 statements in the source file `myprog.f`.

```
$ pgfortran -Minline=size:100 myprog.f
```

Related options: –o, –Mextract

The following list provides the syntax for each–M<pgflag> option that controls function inlining. Each option has a description and, if appropriate, a list of any related options.

– M[no]autoinline[=option[,option,...]]
: instructs the compiler to inline [not to inline] a C/C++ function at –O2, where the option can be any of these:

  levels:n
  : instructs the compiler to perform *n* levels of inlining. The default number of levels is 10.

  maxsize:n
  : instructs the compiler not to inline functions of size > *n*. The default size is 100.

  totalsize:n
  : instructs the compiler to stop inlining when the size equals *n*. The default size is 800.

–Mextract[=option[,option,...]]
: Extracts functions from the file indicated on the command line and creates or appends to the specified extract directory where option can be any of:

  name:func
  : instructs the extractor to extract function func from the file.

  size:number
  : instructs the extractor to extract functions with number or fewer statements from the file.

  lib:filename.ext
  : Use directory filename.ext as the extract directory (required in order to save and re-use inline libraries).

  If you specify both name and size, the compiler extracts functions that match func, or that have number or fewer statements. For examples of extracting functions, refer to "Using Function Inlining" in the PGI Compiler User's Guide.

–Minline[=option[,option,...]]
: This passes options to the function inliner, where the option can be any of these:

except:func
: instructs the inliner to inline all eligible functions except func, a function in the source text. Multiple functions can be listed, comma-separated.

[name:]func
: instructs the inliner to inline the function func. The func name should be a non-numeric string that does not contain a period. You can also use a name: prefix followed by the function name. If name: is specified, what follows is always the name of a function.

[lib:]filename.ext
: instructs the inliner to inline the functions within the library file `filename.ext`. The compiler assumes that a `filename.ext` option containing a period is a library file. Create the library file using the `–Mextract` option. You can also use a lib: prefix followed by the library name. If lib: is specified, no period is necessary in the library name. Functions from the specified library are inlined. If no library is specified, functions are extracted from a temporary library created during an extract prepass.

levels:number
: instructs the inliner to perform number levels of inlining. The default number is 1.

[no]reshape
: instructs the inliner to allow (disallow) inlining in Fortran even when array shapes do not match. The default is `–Minline=noreshape`, except with `–Mconcur` or `–mp`, where the default is `–Minline=reshape`,=reshape.

[size:]number
: instructs the inliner to inline functions with number or fewer statements. You can also use a size: prefix followed by a number. If size: is specified, what follows is always taken as a number.

If you specify both func and number, the compiler inlines functions that match the function name or have number or fewer statements. For examples of inlining functions, refer to "Using Function Inlining" in the PGI Compiler User's Guide.

Optimization Controls

This section describes the –M<pgflag> options that control optimization. Before looking at all the options, let's look at the defaults.

Default: For arguments that you do not specify, the default optimization control options are as follows:

depchk	noipa	nounroll	nor8
i4	nolre	novect	nor8intrinsics
nofprelaxed	noprefetch		

Note

If you do not supply an option to `–Mvect`, the compiler uses defaults that are dependent upon the target system.

Usage: In this example, the compiler invokes the vectorizer with use of packed SSE instructions enabled.

```
$ pgfortran -Mvect=sse -Mcache_align myprog.f
```

Related options: –g, –O

The following list provides the syntax for each–M<pgflag> option that controls optimization. Each option has a description and, if appropriate, a list of any related options.

–Mcache_align
Align unconstrained objects of length greater than or equal to 16 bytes on cache-line boundaries. An unconstrained object is a data object that is not a member of an aggregate structure or common block. This option does not affect the alignment of allocatable or automatic arrays.

Note

To effect cache-line alignment of stack-based local variables, the main program or function must be compiled with –Mcache_align.

–Mconcur[=option [,option,...]]
Instructs the compiler to enable auto-concurrentization of loops. If –Mconcur is specified, multiple processors will be used to execute loops that the compiler determines to be parallelizable. Where option is one of the following:

allcores
Instructs the compiler to use all available cores. Use this option at link time.

[no]altcode:n
Instructs the parallelizer to generate alternate serial code for parallelized loops. If altcode is specified without arguments, the parallelizer determines an appropriate cutoff length and generates serial code to be executed whenever the loop count is less than or equal to that length. If altcode:n is specified, the serial altcode is executed whenever the loop count is less than or equal to n. If noaltcode is specified, the parallelized version of the loop is always executed regardless of the loop count.

bind
Instructs the parallelizer to bind threads to cores. Use this option at link time.

cncall
Calls in parallel loops are safe to parallelize. Loops containing calls are candidates for parallelization. Also, no minimum loop count threshold must be satisfied before parallelization will occur, and last values of scalars are assumed to be safe.

dist:block
Parallelize with block distribution (this is the default). Contiguous blocks of iterations of a parallelizable loop are assigned to the available processors.

dist:cyclic
Parallelize with cyclic distribution. The outermost parallelizable loop in any loop nest is parallelized. If a parallelized loop is innermost, its iterations are allocated to processors cyclically. For example, if there are 3 processors executing a loop, processor 0 performs iterations 0, 3, 6, etc.; processor 1 performs iterations 1, 4, 7, etc.; and processor 2 performs iterations 2, 5, 8, etc.

[no]innermost
Enable parallelization of innermost loops. The default is to not parallelize innermost loops, since it is usually not profitable on dual-core processors.

noassoc
: Disables parallelization of loops with reductions.

When linking, the `-Mconcur` switch must be specified or unresolved references will result. The `NCPUS` environment variable controls how many processors or cores are used to execute parallelized loops.

> **Note**
>
> This option applies only on shared-memory multi-processor (SMP) or multi-core processor-based systems.

–`Mcray[=option[,option,...]]`
: (pgf77 , pgf95, and pgfortran only) Force Cray Fortran (CF77) compatibility with respect to the listed options. Possible values of option include:

  pointer
  : for purposes of optimization, it is assumed that pointer-based variables do not overlay the storage of any other variable.

–`Mdepchk`
: instructs the compiler to assume unresolved data dependencies actually conflict.

–`Mnodepchk`
: Instructs the compiler to assume potential data dependencies do not conflict. However, if data dependencies exist, this option can produce incorrect code.

–`Mdse`
: Enables a dead store elimination phase that is useful for programs that rely on extensive use of inline function calls for performance. This is disabled by default.

–`Mnodse`
: Disables the dead store elimination phase. This is the default.

–`M[no]fpapprox [=option]`
: Perform certain fp operations using low-precision approximation.

  `-Mnofpapprox` specifies not to use low-precision fp approximation operations.

  By default `-Mfpapprox` is not used.

  If `-Mfpapprox` is used without suboptions, it defaults to use approximate `div`, `sqrt`, and `rsqrt`. The available suboptions are these:

  div
  : Approximate floating point division

  sqrt
  : Approximate floating point square root

  rsqrt
  : Approximate floating point reciprocal square root

–M[no]fpmisalign
Instructs the compiler to allow (not allow) vector arithmetic instructions with memory operands that are not aligned on 16-byte boundaries. The default is –Mnofpmisalign on all processors.

Note

Applicable only with one of these options: –tp barcelona or –tp barcelona-64

–M[no]fprelaxed[=option]
Instructs the compiler to use (not use) relaxed precision in the calculation of some intrinsic functions. Can result in improved performance at the expense of numerical accuracy.

The possible values for option are:

div
Perform divide using relaxed precision.

noorder
Do not allow expression reordering or factoring.

order
Allow expression reordering, including factoring.

recip
Perform reciprocal using relaxed precision.

rsqrt
Perform reciprocal square root (1/sqrt) using relaxed precision.

sqrt
Perform square root with relaxed precision.

With no options, –Mfprelaxed generates relaxed precision code for those operations that generate a significant performance improvement, depending on the target processor.

The default is –Mnofprelaxed which instructs the compiler to not use relaxed precision in the calculation of intrinsic functions.

–Mi4
(pgf77 , pgf95, and pgfortran only) the compiler treats INTEGER variables as INTEGER*4.

–Mipa=<option>[,<option>[,...]]
Pass options to the interprocedural analyzer.

Note

–Mipa implies –O2, and the minimum optimization level that can be specified in combination with –Mipa is –O2.

For example, if you specify –Mipa –O1 on the command line, the optimization level is automatically elevated to –O2 by the compiler driver. Typically, as recommended, you would use –Mipa=fast.

Many of the following suboptions can be prefaced with no, which reverses or disables the effect of the suboption if it's included in an aggregate suboption such as –Mipa=fast. The choices of option are:

[no]align
: recognize when targets of a pointer dummy are aligned. The default is noalign.

[no]arg
: remove arguments replaced by const, ptr. The default is noarg.

[no]cg
: generate call graph information for viewing using the `pgicg` command-line utility. The default is nocg.

[no]const
: perform interprocedural constant propagation. The default is const.

except:<func>
: used with inline to specify functions which should not be inlined. The default is to inline all eligible functions according to internally defined heuristics. Valid only immediately following the inline suboption.

[no]f90ptr
: F90/F95 pointer disambiguation across calls. The default is nof90ptr.

fast
: choose IPA options generally optimal for the target. To see settings for `–Mipa=fast` on a given target, use `–help`.

force
: force all objects to re-compile regardless of whether IPA information has changed.

[no]globals
: optimize references to global variables. The default is noglobals.

inline[:n]
: perform automatic function inlining. If the optional :n is provided, limit inlining to at most n levels. IPA-based function inlining is performed from leaf routines upward.

ipofile
: save IPA information in an `.ipo` file rather than incorporating it into the object file.

jobs[:n]
: recompile n jobs in parallel and print source file names as they are compiled.

[no]keepobj
: keep the optimized object files, using file name mangling, to reduce re-compile time in subsequent builds. The default is keepobj.

[no]libc
: optimize calls to certain standard C library routines. The default is nolibc.

[no]libinline
: allow inlining of routines from libraries; implies `–Mipa=inline`. The default is nolibinline.

[no]libopt
: allow recompiling and optimization of routines from libraries using IPA information. The default is nolibopt.

[no]localarg
: equivalent to arg plus externalization of local pointer targets. The default is nolocalarg.

main:<func>
: specify a function to appear as a global entry point; may appear multiple times; disables linking.

rsqrt
: Perform reciprocal square root (1/sqrt) using relaxed precision.

[no]pfo
: enable profile feedback information. The nopfo option is valid only immediately following the inline suboption. `-Mipa=inline,nopfo` tells IPA to ignore PFO information when deciding what functions to inline, if PFO information is available.

[no]ptr
: enable pointer disambiguation across procedure calls. The default is noptr.

[no]pure
: pure function detection. The default is nopure.

required
: return an error condition if IPA is inhibited for any reason, rather than the default behavior of linking without IPA optimization.

[no]reshape
: enables or disables Fortran inline with mismatched array shapes. Valid only immediately following the inline suboption.

safe:[<function>|<library>]
: declares that the named function, or all functions in the named library, are safe; a safe procedure does not call back into the known procedures and does not change any known global variables.

  Without `-Mipa=safe`, any unknown procedures will cause IPA to fail.

[no]safeall
: declares that all unknown procedures are safe; see `-Mipa=safe`. The default is nosafeall.

[no]shape
: perform Fortran 90 array shape propagation. The default is noshape.

summary
: only collect IPA summary information when compiling; this prevents IPA optimization of this file, but allows optimization for other files linked with this file.

[no]vestigial
: remove uncalled (vestigial) functions. The default is novestigial.

`-M[no]loop32`
: Aligns or does not align innermost loops on 32 byte boundaries with `-tp` barcelona.

  Small loops on barcelona may run fast if aligned on 32-byte boundaries; however, in practice, most assemblers do not yet implement efficient padding causing some programs to run more slowly with this default. Use `-Mloop32` on systems with an assembler tuned for barcleona. The default is `-Mnoloop32`.

–Mlre[=array | assoc | noassoc]
Enables loop-carried redundancy elimination, an optimization that can reduce the number of arithmetic operations and memory references in loops.

array
treat individual array element references as candidates for possible loop-carried redundancy elimination. The default is to eliminate only redundant expressions involving two or more operands.

assoc
allow expression re-association; specifying this suboption can increase opportunities for loop-carried redundancy elimination but may alter numerical results.

noassoc
disallow expression re-association.

–Mnolre
Disables loop-carried redundancy elimination.

–Mnoframe
Eliminates operations that set up a true stack frame pointer for every function. With this option enabled, you cannot perform a traceback on the generated code and you cannot access local variables.

–Mnoi4
(pgf77 , pgf95, and pgfortran only) the compiler treats INTEGER variables as INTEGER*2.

–Mpfi[=indirect]
generate profile-feedback instrumentation; this includes extra code to collect run-time statistics and dump them to a trace file for use in a subsequent compilation.

When you use the indirect option,–Mpfi=indirect saves indirect function call targets.

–Mpfi must also appear when the program is linked. When the resulting program is executed, a profile feedback trace file pgfi.out is generated in the current working directory; see –Mpfo.

Note

Compiling and linking with –Mpfi adds significant runtime overhead to almost any executable. You should use executables compiled with –Mpfi only for execution of training runs.

–Mpfo[=indirect | nolayout]
enable profile-feedback optimizations; requires the presence of a pgfi.out profile-feedback trace file in the current working directory. See –Mpfi.

indirect
enable indirect function call inlining

nolayout
disable dynamic code layout.

–Mpre
enables partial redundancy elimination.

–Mprefetch[=option [,option...]]
enables generation of prefetch instructions on processors where they are supported. Possible values for option include:

d:m
set the fetch-ahead distance for prefetch instructions to m cache lines.

n:p
set the maximum number of prefetch instructions to generate for a given loop to p.

nta
use the prefetch instruction.

plain
use the prefetch instruction (default).

t0
use the prefetcht0 instruction.

w
use the AMD-specific prefetchw instruction.

–Mnoprefetch
Disables generation of prefetch instructions.

–M[no]propcond
Enables or disables constant propagation from assertions derived from equality conditionals.

The default is enabled.

–Mr8
(pgf77, pgf95, pgfortran,and pghpf only) the compiler promotes REAL variables and constants to DOUBLE PRECISION variables and constants, respectively. DOUBLE PRECISION elements are 8 bytes in length.

–Mnor8
(pgf77, pgf95, pgfortran, and pghpf only) the compiler does not promote REAL variables and constants to DOUBLE PRECISION. REAL variables will be single precision (4 bytes in length).

–Mr8intrinsics
(pgf77, pgf95, and pgfortran only) the compiler treats the intrinsics CMPLX and REAL as DCMPLX and DBLE, respectively.

–Mnor8intrinsics
(pgf77, pgf95, and pgfortran only) the compiler does not promote the intrinsics CMPLX and REAL to DCMPLX and DBLE, respectively.

–Msafeptr[=option[,option,...]]
(pgcc and pgcpp only) instructs the C/C++ compiler to override data dependencies between pointers of a given storage class. Possible values of option include:

all
assume all pointers and arrays are independent and safe for aggressive optimizations, and in particular that no pointers or arrays overlap or conflict with each other.

arg
: instructs the compiler that arrays and pointers are treated with the same copyin and copyout semantics as Fortran dummy arguments.

global
: instructs the compiler that global or external pointers and arrays do not overlap or conflict with each other and are independent.

local/auto
: instructs the compiler that local pointers and arrays do not overlap or conflict with each other and are independent.

static
: instructs the compiler that static pointers and arrays do not overlap or conflict with each other and are independent.

–Mscalarsse
: Use SSE/SSE2 instructions to perform scalar floating-point arithmetic. (This option is valid only on option `-tp` {p7 | k8-32 | k8-64} targets).

–Mnoscalarsse
: Do not use SSE/SSE2 instructions to perform scalar floating-point arithmetic; use x87 instructions instead. (This option is not valid in combination with the `-tp k8-64` option).

–Msmart
: instructs the compiler driver to invoke a post-pass assembly optimization utility.

–Mnosmart
: instructs the compiler not to invoke an AMD64-specific post-pass assembly optimization utility.

–Munroll[=option [,option...]]
: invokes the loop unroller to execute multiple instances of the loop during each iteration. This also sets the optimization level to 2 if the level is set to less than 2, or if no `-O` or `-g` options are supplied. The option is one of the following:

  c:m
  : instructs the compiler to completely unroll loops with a constant loop count less than or equal to m, a supplied constant. If this value is not supplied, the m count is set to 4.

  m:<n>
  : instructs the compiler to unroll multi-block loops n times. This option is useful for loops that have conditional statements. If n is not supplied, then the default value is 4. The default setting is not to enable `-Munroll=m`.

  n:<n>
  : instructs the compiler to unroll single-block loops n times, a loop that is not completely unrolled, or has a non-constant loop count. If n is not supplied, the unroller computes the number of times a candidate loop is unrolled.

–Mnounroll
: instructs the compiler not to unroll loops.

-M[no]vect[=option [,option,...]]

(disable) enable the code vectorizer, where option is one of the following:

altcode

Instructs the vectorizer to generate alternate code (altcode) for vectorized loops when appropriate. For each vectorized loop the compiler decides whether to generate altcode and what type or types to generate, which may be any or all of: altcode without iteration peeling, altcode with non-temporal stores and other data cache optimizations, and altcode based on array alignments calculated dynamically at runtime. The compiler also determines suitable loop count and array alignment conditions for executing the altcode. This option is enabled by default.

noaltcode

This disables alternate code generation for vectorized loops.

assoc

Instructs the vectorizer to enable certain associativity conversions that can change the results of a computation due to roundoff error. A typical optimization is to change an arithmetic operation to an arithmetic operation that is mathematically correct, but can be computationally different, due to round-off error

noassoc

Instructs the vectorizer to disable associativity conversions.

cachesize:n

Instructs the vectorizer, when performing cache tiling optimizations, to assume a cache size of n. The default is set per processor type, either using the `–tp` switch or auto-detected from the host computer.

[no]gather

Vectorize loops containing indirect array references, such as this one:

```
sum = 0.d0
do k=d(j),d(j+1)-1
sum = sum + a(k)*b(c(k))
enddo
```

The default is gather.

partial

Instructs the vectorizer to enable partial loop vectorization through innemost loop distribution.

prefetch

Instructs the vectorizer to search for vectorizable loops and, wherever possible, make use of prefetch instructions.

[no]short

Enable [disable] short vector operations. `-Mvect=short` enables generation of packed SSE instructions for short vector operations that arise from scalar code outside of loops or within the body of a loop iteration.

[no]sizelimit

Generate vector code for all loops where possible regardless of the number of statements in the loop. This overrides a heuristic in the vectorizer that ordinarily prevents vectorization of loops with a number of statements that exceeds a certain threshold. The default is nosizelimit.

smallvect[:n]
Instructs the vectorizer to assume that the maximum vector length is less than or equal to n. The vectorizer uses this information to eliminate generation of the stripmine loop for vectorized loops wherever possible. If the size n is omitted, the default is 100.

Note

No space is allowed on either side of the colon (:).

[no]sse
Instructs the vectorizer to search for vectorizable loops and, wherever possible, make use of SSE, SSE2, and prefetch instructions. The default is nosse.

[no]uniform
Instructs the vectorizer to perform the same optimizations in the vectorized and residual loops.

Note

This option may affect the performance of the residual loop.

–Mnovect
instructs the compiler not to perform vectorization; can be used to override a previous instance of –Mvect on the command-line, in particular for cases in which –Mvect is included in an aggregate option such as –fastsse.

–Mvect=[*option*]
instructs the compiler to enable loop vectorization, where option is one of the following:

partial
Enable partial loop vectorization through innermost loop distribution.

[no]short
Enable [disable] short vector operations. Enables [disables] generation of packed SSE instructions for short vector operations that arise from scalar code outside of loops or within the body of a loop iteration.

tile
Enable tiling/blocking over multiple nested loops for more efficient cache utilization.

–Mnovintr
instructs the compiler not to perform idiom recognition or introduce calls to hand-optimized vector functions.

Miscellaneous Controls

Default: For arguments that you do not specify, the default miscellaneous options are as follows:

inform nobounds nolist warn

Usage: In the following example, the compiler includes Fortran source code with the assembly code.

```
$ pgfortran -Manno -S myprog.f
```

In the following example, the assembler does not delete the assembly file myprog.s after the assembly pass.

```
$ pgfortran -Mkeepasm myprog.f
```

In the following example, the compiler displays information about inlined functions with fewer than approximately 20 source lines in the source file `myprog.f`.

```
$ pgfortran -Minfo=inline -Minline=20 myprog.f
```

In the following example, the compiler creates the listing file `myprog.lst`.

```
$ pgfortran -Mlist myprog.f
```

In the following example, array bounds checking is enabled.

```
$ pgfortran -Mbounds myprog.f
```

Related options: –m, –S, –V, –v

The following list provides the syntax for each miscellaneous –M<pgflag> option. Each option has a description and, if appropriate, a list of any related options.

–Manno
: annotate the generated assembly code with source code. Implies `-Mkeepasm` .

–Mbounds
: enables array bounds checking. If an array is an assumed size array, the bounds checking only applies to the lower bound. If an array bounds violation occurs during execution, an error message describing the error is printed and the program terminates. The text of the error message includes the name of the array, the location where the error occurred (the source file and the line number in the source), and information about the out of bounds subscript (its value, its lower and upper bounds, and its dimension). The following is a sample error message:

```
PGFTN-F-Subscript out of range for array a (a.f: 2)
subscript=3, lower bound=1, upper bound=2, dimension=2
```

–Mnobounds
: disables array bounds checking.

–Mbyteswapio
: swap byte-order from big-endian to little-endian or vice versa upon input/output of Fortran unformatted data files.

–Mchkfpstk (32-bit only)
: instructs the compiler to check for internal consistency of the x87 floating-point stack in the prologue of a function and after returning from a function or subroutine call. Floating-point stack corruption may occur in many ways, one of which is Fortran code calling floating-point functions as subroutines (i.e., with the CALL statement). If the `PGI_CONTINUE` environment variable is set upon execution of a program compiled with `-Mchkfpstk`, the stack will be automatically cleaned up and execution will continue. There is a performance penalty associated with the stack cleanup. If `PGI_CONTINUE` is set to verbose, the stack will be automatically cleaned up and execution will continue after printing the warning message.

  Note

  This switch is only valid for 32-bit. On 64-bit it is ignored.

–Mchkptr
: instructs the compiler to check for pointers that are dereferenced while initialized to NULL (pgf95, pgfortran, and pghpf only).

–Mchkstk
instructs the compiler to check the stack for available space in the prologue of a function and before the start of a parallel region. Prints a warning message and aborts the program gracefully if stack space is insufficient. Useful when many local and private variables are declared in an OpenMP program.

If the user also sets the PGI_STACK_USAGE environment variable to any value, then the program displays the stack space allocated and used after the program exits. For example, you might see something similar to the following message:

```
thread 0 stack: max 8180KB, used 48KB
```

This message indicates that the program used 48KB of a 8180KB allocated stack. For more information on the PGI_STACK_USAGE, refer to "PGI_STACK_USAGE" in the PGI Compiler User's Guide.

This information is useful when you want to explicitly set a reserved and committed stack size for your programs, such as using the -stack option on Windows.

–Mcpp[=option [,option,...]]
run the PGI cpp-like preprocessor without execution of any subsequent compilation steps. This option is useful for generating dependence information to be included in makefiles.

Note

Only one of the m, md, mm or mmd options can be present; if multiple of these options are listed, the last one listed is accepted and the others are ignored.

The option is one or more of the following:

m
print makefile dependencies to stdout.

md
print makefile dependencies to filename.d, where filename is the root name of the input file being processed.

mm
print makefile dependencies to stdout, ignoring system include files.

mmd
print makefile dependencies to filename.d, where filename is the root name of the input file being processed, ignoring system include files.

[no]comment
(don't) retain comments in output.

[suffix:]<suff>
use <suff> as the suffix of the output file containing makefile dependencies.

–Mdll
This Windows-only flag has been deprecated. Refer to -Bdynamic. This flag was used to link with the DLL versions of the runtime libraries, and it was required when linking with any DLL built by any of The Portland Group compilers. This option implied –D_DLL, which defines the preprocessor symbol _DLL.

`–Mgccbug[s]`
match the behavior of certain gcc bugs.

`–Miface[=option]`
adjusts the calling conventions for Fortran, where option is one of the following:

unix
(Win32 only) uses UNIX calling conventions, no trailing underscores.

cref
uses CREF calling conventions, no trailing underscores.

mixed_str_len_arg
places the lengths of character arguments immediately after their corresponding argument. Has affect only with the CREF calling convention.

nomixed_str_len_arg
places the lengths of character arguments at the end of the argument list. Has affect only with the CREF calling convention.

`–Minfo[=option [,option,...]]`
instructs the compiler to produce information on standard error, where option is one of the following:

all
instructs the compiler to produce all available `–Minfo` information. Implies a number of suboptions:
`–Mneginfo=accel,inline,ipa,loop,lre,mp,opt,par,vect`

accel
instructs the compiler to enable accelerator information.

ccff
instructs the compiler to append common compiler feedback format information, such as optimization information, to the object file.

ftn
instructs the compiler to enable Fortran-specific information.

hpf
instructs the compiler to enable HPF-specific information.

inline
instructs the compiler to display information about extracted or inlined functions. This option is not useful without either the `–Mextract` or `–Minline` option.

intensity
instructs the compiler to provide informational messages about the intensity of the loop. Specify <n> to get messages on nested loops.

- For floating point loops, intensity is defined as the number of floating point operations divided by the number of floating point loads and stores.
- For integer loops, the loop intensity is defined as the total number of integer arithmetic operations, which may include updates of loop counts and addresses, divided by the total number of integer loads and stores.

- By default, the messages just apply to innermost loops.

ipa
: instructs the compiler to display information about interprocedural optimizations.

loop
: instructs the compiler to display information about loops, such as information on vectorization.

lre
: instructs the compiler to enable LRE, loop-carried redundancy elimination, information.

mp
: instructs the compiler to display information about parallelization.

opt
: instructs the compiler to display information about optimization.

par
: instructs the compiler to enable parallelizer information.

pfo
: instructs the compiler to enable profile feedback information.

time
: instructs the compiler to display compilation statistics.

unroll
: instructs the compiler to display information about loop unrolling.

vect
: instructs the compiler to enable vectorizer information.

`–Minform=level`
: instructs the compiler to display error messages at the specified and higher levels, where level is one of the following:

fatal
: instructs the compiler to display fatal error messages.

[no]file
: instructs the compiler to print or not print source file names as they are compiled. The default is to print the names: `–Minform=file`.

inform
: instructs the compiler to display all error messages (inform, warn, severe and fatal).

severe
: instructs the compiler to display severe and fatal error messages.

warn
: instructs the compiler to display warning, severe and fatal error messages.

`–Minstrumentation=option`
: specifies the level of instrumentation calls generated. This option implies `-Minfo=ccff`, `-Mframe`.

`option` is one of the following:

level
: specifies the level of instrumentation calls generated.

function (default)
: generates instrumentation calls for entry and exit to functions.

  Just after function entry and just before function exit, the following profiling functions are called with the address of the current function and its call site. (linux86-64 only).

```
void __cyg_profile_func_enter (void *this_fn, void *call_site);
void __cyg_profile_func_exit (void *this_fn, void *call_site);
```

  In these calls, the first argument is the address of the start of the current function.

`–Mkeepasm`
: instructs the compiler to keep the assembly file as compilation continues. Normally, the assembler deletes this file when it is finished. The assembly file has the same filename as the source file, but with a .s extension.

`–Mlist`
: instructs the compiler to create a listing file. The listing file is `filename.lst`, where the name of the source file is `filename.f`.

`–Mmakedll`
: (Windows only) generate a dynamic link library (DLL).

`–Mmakeimplib`
: (Windows only) generate an import library for a DLL without creating the DLL. When used without -def:deffile, passes the switch `-def` to the librarian without a deffile.

`–Mnames=lowercase|uppercase`
: specifies the case for the names of Fortran externals .

  - lowercase - Use lowercase for Fortran externals.
  - uppercase - Use uppercase for Fortran externals.

`–Mneginfo[=option [,option,...]]`
: instructs the compiler to produce information on standard error, where option is one of the following:

  all
  : instructs the compiler to produce all available information on why various optimizations are not performed.

  accel
  : instructs the compiler to enable accelerator information.

  ccff
  : instructs the compiler to append information, such as optimization information, to the object file.

  concur
  : instructs the compiler to produce all available information on why loops are not automatically parallelized. In particular, if a loop is not parallelized due to potential data dependence, the

variable(s) that cause the potential dependence are listed in the messages that you see when using the option `–Mneginfo`.

ftn
: instructs the compiler to enable Fortran-specific information.

hpf
: instructs the compiler to enable HPF-specific information.

inline
: instructs the compiler to display information about extracted or inlined functions. This option is not useful without either the `–Mextract` or `–Minline` option.

ipa
: instructs the compiler to display information about interprocedural optimizations.

loop
: instructs the compiler to display information about loops, such as information on vectorization.

lre
: instructs the compiler to enable LRE, loop-carried redundancy elimination, information.

mp
: instructs the compiler to display information about parallelization.

opt
: instructs the compiler to display information about optimization.

par
: instructs the compiler to enable parallelizer information.

pfo
: instructs the compiler to enable profile feedback information.

vect
: instructs the compiler to enable vectorizer information.

`–Mnolist`
: the compiler does not create a listing file. This is the default.

`–Mnoopenmp`
: when used in combination with the `–mp` option, the compiler ignores OpenMP parallelization directives or pragmas, but still processes SGI-style parallelization directives or pragmas.

`–Mnosgimp`
: when used in combination with the `–mp` option, the compiler ignores SGI-style parallelization directives or pragmas, but still processes OpenMP parallelization directives or pragmas.

`–Mnopgdllmain`
: (Windows only) do not link the module containing the default DllMain() into the DLL. This flag applies to building DLLs with the PGFORTRAN and PGHPF compilers. If you want to replace the default DllMain() routine with a custom DllMain(), use this flag and add the object containing the custom DllMain() to the link line. The latest version of the default DllMain() used by PGFORTRAN and PGHPF is included in

the Release Notes for each release; the PGFORTRAN- and PGHPF-specific code in this routine must be incorporated into the custom version of DllMain() to ensure the appropriate function of your DLL.

–Mnorpath
(Linux only) Do not add –rpath to the link line.

–Mpreprocess
perform cpp-like preprocessing on assembly and Fortran input source files.

–Mwritable_strings
stores string constants in the writable data segment.

Note

Options –Xs and –Xst include –Mwritable_strings.

Chapter 3. OpenMP Reference Information

The PGF77, PGF95, and PGFORTRAN Fortran compilers support the OpenMP Fortran Application Program Interface. The PGCC ANSI C and C++ compilers support the OpenMP C/C++ Application Program Interface.

This chapter contains detailed descriptions of each of the OpenMP Fortran directives and C/C++ pragmas that PGI supports. In addition, the section "Directive and Pragma Clauses" in the PGI Compiler User's Guide contains information about the clauses associated with these directives and pragmas.

Tasks

Every part of an OpenMP program is part of a task. The Task Overview section of the PGI Compiler User's Guide provides a general overview of tasks and general terminology associated with tasks. This section provides more detailed information about tasks, including tasks scheduling points and the task construct.

Task Characteristics and Activities

A task, whose execution can be performed immediately or delayed, has these characteristics:

- Code to execute
- A data environment - that is, it owns its data
- An assigned thread that executes the code and uses the data.

There are two activities associated with tasks: packaging and execution.

- Packaging: Each encountering thread packages a new instance of a task - code and data.
- Execution: Some thread in the team executes the task at some later time.

Task Scheduling Points

PGI currently supports four task scheduling points: at the beginning of a task, at the end of a task, a taskwait, and at a barrier.

- Beginning of a task.

  At the beginning of a task, the task can be executed immediately or registered for later execution. A programmer-specified "if" clause that is FALSE forces immediate execution of the task. The implementation can also force immediate execution; for example, a task within a task is never registered for later execution, it executes immediately.

- End of a task

  At the end of a task, the behavior of the scheduling point depends on how the task was executed. If the task was immediately executed, execution continues to the next statement. If it was previously registered and is being executed "out of sequence", control returns to where the task was executed.

- Taskwait

  A taskwait executes all registered tasks at the time it is called. In addition to executing all tasks registered by the calling thread, it also executes tasks previously registered by other threads. Let's take a quick look at this process; suppose the following is true:

  - Thread 0 called taskwait and is executing tasks.
  - Thread 1 is registering tasks.

  Depending on the timing between thread 0 and thread 1, thread 0 may execute none of the tasks, all of the tasks, or some of tasks.

  > **Note**
  >
  > Taskwait waits only for immediate children tasks, not for descendant tasks. You can achieve waiting on descendants but ensuring that each child also waits on its children.

- Barrier

  A barrier can be explicit or implicit. An example of an implicit barrier is the end of a parallel region.

  The barrier effectively contains taskwaits. All threads must arrive at the barrier for the barrier to complete. This rule guarantees that all tasks have been executed at the completion of the barrier.

Task Construct

A task construct is a task directive plus a structured block, with the following syntax:

```
#pragma omp task [clause[[,]clause] ...]
     structured-block
```

where clause can be one of the following:

```
if (expression)
untied
shared (list)
private (list)
firstprivate (list)
default( shared | none )
```

Consider the following simple example of a program using tasks. This example illustrates the difference between registering tasks and executing tasks, a concept that is fundamental to understanding tasks.

This program contains a parallel region that contains a single region. The single region contains a loop that registers 10 tasks. Before reading the explanation that follows the example, consider what happens if you use four threads with this example.

Example 3.1. OpenMP Task C Example

```
int
main(int argc,char *argv[])
{
     int i;
#pragma omp parallel private(i)
{
#pragma omp single
{
     for(i=0;i<10;i++) {
         sleep(i%2);
         printf("task %2d registered by thread %d\n",i,
                 omp_get_thread_num());
#pragma omp task firstprivate(i)
{
         sleep(i%5);
         printf("task %2d executed by thread %d\n",i,
                 omp_get_thread_num());
} /* end task */
     } /* end for */
} /* end single */
} /* end parallel */
} /* end main */
```

Example 3.2. OpenMP Task Fortran Example

```
     PROGRAM MAIN
     INTEGER I
     INTEGER omp_get_thread_num
!$OMP PARALLEL PRIVATE(I)
!$OMP SINGLE
     DO I = 1, 10
        CALL SLEEP(MOD(I,2))
        PRINT *,"TASK ",I," REGISTERED BY THREAD ",omp_get_thread_num()
!$OMP TASK FIRSTPRIVATE(I)
        CALL SLEEP(MOD(I,5))
        PRINT *,"TASK ",I," EXECUTED BY THREAD   ",omp_get_thread_num()
!$OMP END TASK
     ENDDO
!$OMP END SINGLE
!$OMP END PARALLEL
     END
```

If you run this program with four threads, 0 through 3, one thread is in the single region registering tasks. The other three threads are in the implied barrier at the end of the single region executing tasks. Further, when the thread executing the single region completes registering the tasks, it joins the other threads and executes tasks.

The program includes calls to `sleep` to slow the program and allow all threads to participate.

The output for the Fortran example is similar to the following. In this output, thread 1 was registering tasks while the other three threads - 0,2, and 3 - were executing tasks When all 10 tasks were registered, thread 1 began executing tasks as well.

```
TASK 1  REGISTERED BY THREAD 1
TASK 2  REGISTERED BY THREAD 1
TASK 1  EXECUTED BY THREAD 0
TASK 3  REGISTERED BY THREAD 1
TASK 4  REGISTERED BY THREAD 1
TASK 2  EXECUTED BY THREAD 3
TASK 5  REGISTERED BY THREAD 1
TASK 6  REGISTERED BY THREAD 1
TASK 6  EXECUTED BY THREAD 3
TASK 5  EXECUTED BY THREAD 3
TASK 7  REGISTERED BY THREAD 1
TASK 8  REGISTERED BY THREAD 1
TASK 3  EXECUTED BY THREAD 0
TASK 9  REGISTERED BY THREAD 1
TASK 10  REGISTERED BY THREAD 1
TASK 10  EXECUTED BY THREAD 1
TASK 4  EXECUTED BY THREAD 2
TASK 7  EXECUTED BY THREAD 0
TASK 8  EXECUTED BY THREAD 3
TASK 9  EXECUTED BY THREAD 1
```

Parallelization Directives and Pragmas

Parallelization directives, as described in "Using OpenMP" in the PGI Compiler User's Guide, are comments in a program that are interpreted by the PGI Fortran compilers when the option `-mp` is specified on the command line. The form of a parallelization directive is:

```
sentinel directive_name [clauses]
```

Parallelization pragmas are #pragma statements in a C or C++ program that are interpreted by the PGCC C and C++ compilers when the option -mp is specified on the command line. The form of a parallelization pragma is:

```
#pragma omp pragma_name [clauses]
```

The examples given with each section use the routines `omp_get_num_threads()` and `omp_get_thread_num()`. They return the number of threads currently in the team executing the parallel region and the thread number within the team, respectively. For more information, refer to "run-time Library Routines" in the PGI Compiler User's Guide.

Note

Directives which are presented in pairs must be used in pairs.

This section describes the details of these directives and pragmas that were summarized in "Using OpenMP" in the PGI Compiler User's Guide. For each directive and pragma, this section describes the overall purpose, the syntax, the clauses associated with it, the usage, and examples of how to use it.

ATOMIC and atomic

The OpenMP ATOMIC directive or the omp critical pragma is semantically equivalent to a single statement in a CRITICAL...END CRITICAL directive or the omp critical pragma.

Syntax:

`!$OMP ATOMIC`	`#pragma omp atomic` `< C/C++ expression statement >`

Usage:

The ATOMIC directive is semantically equivalent to enclosing the following single statement in a CRITICAL / END CRITICAL directive pair. The omp atomic pragma is semantically equivalent to subjecting the following single C/C++ expression statement to an omp critical pragma.

The statements must be one of the following forms:

For Directives:

```
x = x operator expr
x = expr operator x
x = intrinsic (x, expr)
x = intrinsic (expr, x)
```

For Pragmas:

```
x <binary_operator>= expr
x++
++x
x--
--x
```

where `x` is a scalar variable of intrinsic type, `expr` is a scalar expression that does not reference `x`, intrinsic is one of MAX, MIN, IAND, IOR, or IEOR, operator is one of +, *, -, /, .AND., .OR., .EQV., or .NEQV., and <binary_operator> is not overloaded and is one of +, *, -, /, &, ^, |, << or >>.

BARRIER and barrier

The OpenMP BARRIER directive defines a point in a program where each thread waits for all other threads to arrive before continuing with program execution.

Syntax:

`!$OMP BARRIER`	`#pragma omp barrier`

Usage:

There may be occasions in a parallel region when it is necessary that all threads complete work to that point before any thread is allowed to continue. The BARRIER directive or omp barrier pragma synchronizes all threads at such a point in a program. Multiple barrier points are allowed within a parallel region. The BARRIER directive and omp barrier pragma must either be executed by all threads executing the parallel region or by none of them.

CRITICAL ... END CRITICAL and critical

The CRITICAL...END CRITICAL directive and omp critical pragma require a thread to wait until no other thread is executing within a critical section.

Syntax:

`!$OMP CRITICAL [(name)]`
` < Fortran code executed in body`
`of critical section >`
`!$OMP END CRITICAL [(name)]`
	`#pragma omp critical [(name)]`
	`< C/C++ structured block >`

Usage:

Within a parallel region, there may exist subregions of code that will not execute properly when executed by multiple threads simultaneously. This issue is often due to a shared variable that is written and then read again.

The CRITICAL... END CRITICAL directive pair and the omp critical pragma define a subsection of code within a parallel region, referred to as a critical section, which is executed one thread at a time.

The first thread to arrive at a critical section is the first to execute the code within the section. The second thread to arrive does not begin execution of statements in the critical section until the first thread exits the critical section. Likewise, each of the remaining threads waits to execute the statements in the critical section until the previous thread exits the critical section.

You can use the optional *name argument* to identify the critical region. Names that identify critical regions have external linkage and are in a name space separate from the name spaces used by labels, tags, members, and ordinary identifiers. If a name argument appears on a CRITICAL directive, the same name must appear on the END CRITICAL directive.

Note

Critical sections cannot be nested, and any such specifications are ignored. Branching into or out of a critical section is illegal.

Fortran Example of Critical...End Critical directive:

```
 PROGRAM CRITICAL_USE
 REAL A(100,100),MX, LMX
 INTEGER I, J
 MX = -1.0
 LMX = -1.0
 CALL RANDOM_SEED()
 CALL RANDOM_NUMBER(A)
!$OMP PARALLEL PRIVATE(I), FIRSTPRIVATE(LMX)
!$OMP DO
   DO J=1,100
    DO I=1,100
     LMX = MAX(A(I,J),LMX)
    ENDDO
   ENDDO
!$OMP CRITICAL
   MX = MAX(MX,LMX)
!$OMP END CRITICAL
!$OMP END PARALLEL
   PRINT *,"MAX VALUE OF A IS ", MX
END
```

C Example of **omp critical** pragma

```
#include <stdlib.h>
main(){
int a[100][100], mx=-1,lmx=-1, i, j;
 for (j=0; j<100; j++)
 for (i=0; i<100; i++)
 a[i][j]=1+(int)(10.0*rand()/(RAND_MAX+1.0));
#pragma omp parallel private(i) firstprivate(lmx)
 {
#pragma omp for
```

```
 for (j=0; j<100; j++)
 for (i=0; i<100; i++)
 lmx = (lmx > a[i][j]) ? lmx : a[i][j];
#pragma omp critical
 mx = (mx > lmx) ? mx : lmx;
 }
 printf ("max value of a is %d\n",mx);
}
```

This program could also be implemented without the critical region by declaring MX as a reduction variable and performing the MAX calculation in the loop using MX directly rather than using LMX. Refer to "PARALLEL ... END PARALLEL and parallel " and "DO...END DO and for " for more information on how to use the REDUCTION clause on a parallel DO loop.

C$DOACROSS

The C$DOACROSS directive, while not part of the OpenMP standard, is supported for compatibility with programs parallelized using legacy SGI-style directives.

Syntax:

```
C$DOACROSS [ Clauses ]
< Fortran DO loop to be executed
 in parallel >
```

```
#pragma omp parallel [clauses]
< C/C++ structured block >
```

Clauses:

For Directives:	For Pragmas:
[{PRIVATE | LOCAL} (list)]	private | local(list)
[{SHARED | SHARE} (list)]	shared | share(list)
[MP_SCHEDTYPE={SIMPLE | INTERLEAVE}]	mp_schedule (simple | interleave)
[CHUNK=<integer_expression>]	chunk=<integer_expression>
[IF (logical_expression)]	if (logical_expression)

Usage:

The C$DOACROSS directive has the effect of a combined parallel region and parallel DO loop applied to the loop immediately following the directive. It is very similar to the OpenMP PARALLEL DO directive, but provides for backward compatibility with codes parallelized for SGI systems prior to the OpenMP standardization effort.

The C$DOACROSS directive must not appear within a parallel region. It is a shorthand notation that tells the compiler to parallelize the loop to which it applies, even though that loop is not contained within a parallel region. While this syntax is more convenient, it should be noted that if multiple successive DO loops are to be parallelized it is more efficient to define a single enclosing parallel region and parallelize each loop using the OpenMP DO directive.

A variable declared PRIVATE or LOCAL to a C$DOACROSS loop is treated the same as a private variable in a parallel region or DO. A variable declared SHARED or SHARE to a C$DOACROSS loop is shared among the threads, meaning that only 1 copy of the variable exists to be used and/or modified by all of the threads. This is equivalent to the default status of a variable that is not listed as PRIVATE in a parallel region or DO. This same default status is used in C$DOACROSS loops as well. For more information on clauses, refer to "Directive and Pragma Clauses," on page 121.

DO...END DO and for

The OpenMP DO...END DO directive and omp for pragma support parallel execution and the distribution of loop iterations across available threads in a parallel region.

Syntax:

```
!$OMP DO [Clauses]
< Fortran DO loop to be executed
 in parallel
 !$OMP END DO [NOWAIT]
```

```
#pragma omp for [Clauses]
< C/C++ for loop to be executed
in parallel >
```

Clauses:

For Directives:	For Pragmas:
PRIVATE(list)	private(list)
FIRSTPRIVATE(list)	firstprivate(list)
LASTPRIVATE(list)	lastprivate(list)
REDUCTION({operator | intrinsic } : list)	reduction(operator: list)
SCHEDULE (type [, chunk])	schedule (kind[, chunk])
COLLAPSE (n)	collapse (n)
ORDERED	ordered
	nowait

Usage:

The real purpose of supporting parallel execution is the distribution of work across the available threads. The DO... END DO directive pair and the omp for pragma provide a convenient mechanism for the distribution of loop iterations across the available threads in a parallel region.

While you can explicitly manage work distribution with constructs such as the following one, these constructs are not in the form of directives or pragmas.

Examples:

For Directives:

```
IF (omp_get_thread_num() .EQ. 0)
THEN
   ...
ELSE IF (omp_get_thread_num() .EQ. 1)
THEN
   ...
ENDIF
```

For Pragmas:

```
if (omp_get_thread_num() == 0) {
 ...
}
else if (omp_get_thread_num() == 1) {
 ...
}
```

Tips

Remember these items about clauses in the DO...END DO directives and omp for pragmas:

- Variables declared in a PRIVATE list are treated as private to each thread participating in parallel execution of the loop, meaning that a separate copy of the variable exists with each thread.
- Variables declared in a FIRSTPRIVATE list are PRIVATE, and are initialized from the original object existing before the construct.
- Variables declared in a LASTPRIVATE list are PRIVATE, and the thread that executes the sequentially last iteration updates the version of the object that existed before the construct.
- The REDUCTION clause for the directive is described in "REDUCTION," on page 124 and the reduction clause for the pragma is described in "Directive and Pragma Clauses," on page 121.
- The SCHEDULE clause specifies how iterations of the DO or for loop are divided up between threads. For more information on this clause, refer to "SCHEDULE," on page 125.
- If ORDERED code blocks are contained in the dynamic extent of the DO directive, the ORDERED clause must be present. For more information on ORDERED code blocks, refer to "ORDERED and ordered ".
- The DO... END DO directive pair directs the compiler to distribute the iterative DO loop immediately following the !$OMP DO directive across the threads available to the program. The DO loop is executed in parallel by the team that was started by an enclosing parallel region. If the !$OMP END DO directive is not specified, the !$OMP DO is assumed to end with the enclosed DO loop. DO... END DO directive pairs may not be nested. Branching into or out of a !$OMP DO loop is not supported.
- The **omp for** pragma directs the compiler to distribute the iterative for loop immediately following across the threads available to the program. The for loop is executed in parallel by the team that was started by an enclosing parallel region. Branching into or out of an omp for loop is not supported, and omp for pragmas may not be nested.
- By default, there is an implicit barrier after the end of the parallel loop; the first thread to complete its portion of the work waits until the other threads have finished their portion of work. If NOWAIT is specified, the threads will not synchronize at the end of the parallel loop.

In addition to the preceding items, remember these items about !$OMP DO loops and **omp for** loops:

- The DO loop index variable is always private.
- The for loop index variable is always private.
- !$OMP DO loops and omp for loops must be executed by all threads participating in the parallel region or none at all.
- The END DO directive is optional, but if it is present it must appear immediately after the end of the enclosed DO loop.
- The for loop must be a structured block and its execution must not be terminated by break.
- Values of the loop control expressions and the chunk expressions must be the same for all threads executing the loop.

Examples:

Fortran Example of **DO...END DO** directive

```
PROGRAM DO_USE
   REAL A(1000), B(1000)
   DO I=1,1000
      B(I) = FLOAT(I)
   ENDDO
!$OMP PARALLEL
!$OMP DO
   DO I=1,1000
      A(I) = SQRT(B(I));
   ENDDO
  ...
!$OMP END PARALLEL
   ...
   END
```

C Example of **omp for** pragma

```
#include <stdio.h>
#include <math.h>
main(){
float a[1000], b[1000];
int i;
 for (i=0; i<1000; i++)
 b[i] = i;
#pragma omp parallel
 {
#pragma omp for
 for (i=0; i<1000; i++)
 a[i] = sqrt(b[i]);
 ...
 }
 ...
}
```

FLUSH and flush

The OpenMP FLUSH directive and omp flush pragma ensure that processor-visible data item are written back to memory at the point at which the directive appears.

Syntax:

```
!$OMP FLUSH [(list)]
```

```
#pragma omp flush [(list)]
```

Usage:

The OpenMP FLUSH directive ensures that all processor-visible data items, or only those specified in `list`, when it is present, are written back to memory at the point at which the directive or pragma appears.

MASTER ... END MASTER and master

The MASTER...END MASTER directive and omp master pragma allow the user to designate code that must execute on a master thread and that is skipped by other threads in the team of threads.

Syntax:

```
!$OMP MASTER
< Fortran code executed in body of
 MASTER section >
!$OMP END MASTER
```

```
#pragma omp master
< C/C++ structured block >
```

Usage:

A master thread is a single thread of control that begins an OpenMP program and which is present for the duration of the program. In a parallel region of code, there may be a sub-region of code that should execute only on the master thread. Instead of ending the parallel region before this subregion and then starting it up again after this subregion, the MASTER... END MASTER directive pair or omp master pragma allows the user to conveniently designate code that executes on the master thread and is skipped by the other threads.

- There is no implied barrier on entry to or exit from a master section of code.

- Nested master sections are ignored.
- Branching into or out of a master section is not supported.

Examples:

Example of Fortran **MASTER...END MASTER** directive

```
PROGRAM MASTER_USE
     INTEGER A(0:1)
     INTEGER omp_get_thread_num
     A=-1
!$OMP PARALLEL
     A(omp_get_thread_num()) = omp_get_thread_num()
!$OMP MASTER
     PRINT *, "YOU SHOULD ONLY SEE THIS ONCE"
!$OMP END MASTER
!$OMP END PARALLEL
     PRINT *, "A(0)=", A(0), " A(1)=", A(1)
 END
```

Example of **omp master** pragma

```
#include <stdio.h>
#include <omp.h>
main(){
 int a[2]={-1,-1};
#pragma omp parallel
 {
 a[omp_get_thread_num()] = omp_get_thread_num();
#pragma omp master
 printf("YOU SHOULD ONLY SEE THIS ONCE\n");
 }
 printf("a[0]=%d, a[1]=%d\n",a[0],a[1]);
}
```

ORDERED and ordered

The OpenMP ORDERED directive and omp ordered pragma allow the user to identify a portion of code within an ordered code block that must be executed in the original, sequential order, while allowing parallel execution of statements outside the code block.

Syntax:

```
!$OMP ORDERED
< Fortran code block executed
 by processor >
!$OMP END ORDERED
```

```
#pragma omp ordered
< C/C++ structured block >
```

Usage:

The ORDERED directive can appear only in the dynamic extent of a DO or PARALLEL DO directive that includes the ORDERED clause. The ordered pragma can appear only in the dynamic extent of a for or parallel for pragma that includes the ordered clause. The structured code block between the ORDERED / END ORDERED directives or after the ordered pragma is executed by only one thread at a time, and in the order of the loop iterations. This sequentializes the ordered code block while allowing parallel execution of statements outside the code block. The following additional restrictions apply to the ORDERED directive and ordered pragma:

- The ordered code block must be a structured block.

- It is illegal to branch into or out of the block.
- A given iteration of a loop with a DO directive or omp for pragma cannot execute the same ORDERED directive or omp ordered pragma more than once, and cannot execute more than one ORDERED directive or omp ordered pragma.

PARALLEL ... END PARALLEL and parallel

The OpenMP PARALLEL...END PARALLEL directive and OpenMP omp parallel pragma support a fork/join execution model in which a single thread executes all statements until a parallel region is encountered.

Syntax:

```
!$OMP PARALLEL [Clauses]
   < Fortran code executed in body
 of parallel region >
!$OMP END PARALLEL
```

```
#pragma omp parallel [clauses]
   < C/C++ structured block >
```

Clauses:

For Directives:	For Pragmas:
PRIVATE(list)	private(list)
SHARED(list)	shared(list)
DEFAULT(PRIVATE | SHARED | NONE)	default(shared | none)
FIRSTPRIVATE(list)	firstprivate(list)
REDUCTION([{operator | intrinsic}:] list)	reduction(operator: list)
COPYIN(list)	copyin (list)
IF(scalar_logical_expression)	if (scalar_expression)
NUM_THREADS(scalar_integer_expression)	num_threads(scalar_integer_expression)

Usage:

This directive pair or pragma declares a region of parallel execution. It directs the compiler to create an executable in which the statements within the structured block, such as between PARALLEL and PARALLEL END for directives, are executed by multiple lightweight threads. The code that lies within this structured block is called a *parallel region*.

The OpenMP parallelization directives or pragmas support a fork/join execution model in which a single thread executes all statements until a parallel region is encountered. At the entrance to the parallel region, a system-dependent number of symmetric parallel threads begin executing all statements in the parallel region redundantly. These threads share work by means of work-sharing constructs such as parallel DO loops or FOR loops.

- The number of threads in the team is controlled by the `OMP_NUM_THREADS` environment variable. If `OMP_NUM_THREADS` is not defined, the program executes parallel regions using only one processor.
- Branching into or out of a parallel region is not supported.
- All other shared-memory parallelization directives or pragmas must occur within the scope of a parallel region. Nested PARALLEL... END PARALLEL directive pairs or omp parallel pragmas are not supported and are ignored.

- There is an implicit barrier at the end of the parallel region, which, in the directive, is denoted by the END PARALLEL directive. When all threads have completed execution of the parallel region, a single thread resumes execution of the statements that follow.

  Note

  By default, there is no work distribution in a parallel region. Each active thread executes the entire region redundantly until it encounters a directive or pragma that specifies work distribution. For work distribution, see the DO, PARALLEL DO, or DOACROSS directives or the omp for pragma.

Examples:

PARALLEL...END PARALLEL directive example:

```
 PROGRAM WHICH_PROCESSOR_AM_I
   INTEGER A(0:1)
   INTEGER omp_get_thread_num
   A(0) = -1
   A(1) = -1
!$OMP PARALLEL
   A(omp_get_thread_num())
        = omp_get_thread_num()
!$OMP END PARALLEL
   PRINT *, "A(0)=",A(0),
            " A(1)=",A(1)
 END
```

omp parallel pragma example

```
#include <stdio.h>
#include <omp.h>
main(){
   int a[2]={-1,-1};
#pragma omp parallel
  {
     a[omp_get_thread_num()] =
 omp_get_thread_num();
  }
  printf("a[0] = %d,
a[1] = %d",a[0],a[1]);
}
```

Clause Usage:

COPYIN: The COPYIN clause applies only to THREADPRIVATE common blocks. In the presence of the COPYIN clause, data from the master thread's copy of the common block is copied to the THREADPRIVATE copies upon entry to the parallel region.

IF: In the presence of an IF clause, the parallel region is executed in parallel only if the corresponding `scalar_logical_expression` evaluates to .TRUE.. Otherwise, the code within the region is executed by a single processor, regardless of the value of the environment variable `OMP_NUM_THREADS`.

NUM_THREADS: If the NUM_THREADS clause is present, the corresponding `scalar_integer_expression` must evaluate to a positive integer value. This value sets the maximum number of threads used during execution of the parallel region. A NUM_THREADS clause overrides either a previous call to the library routine omp_set_num_threads() or the setting of the OMP_NUM_THREADS environment variable.

PARALLEL DO

The OpenMP PARALLEL DO directive is a shortcut for a PARALLEL region that contains a single DO directive.

Note

The OpenMP PARALLEL DO or DO directive must be immediately followed by a DO statement (as defined by R818 of the ANSI Fortran standard). If you place another statement or an OpenMP directive between the PARALLEL DO or DO directive and the DO statement, the compiler issues a syntax error.

Syntax:

```
!$OMP PARALLEL DO [CLAUSES]
< Fortran DO loop to be executed
 in parallel >
[!$OMP END PARALLEL DO]
```

```
#pragma omp parallel [clauses]
< C/C++ structured block >
```

Clauses:

For Directives:	For Pragmas:
PRIVATE(list)	private(list)
SHARED(list)	shared(list)
DEFAULT(PRIVATE | SHARED | NONE)	default(shared | none)
FIRSTPRIVATE(list)	firstprivate(list)
LASTPRIVATE(list)	reduction(operator: list)
REDUCTION([{operator | intrinsic}:] list)	copyin (list)
COPYIN(list)	if (scalar_expression)
IF(scalar_logical_expression)	num_threads(scalar_integer_expression)
NUM_THREADS(scalar_integer_expression)	
SCHEDULE (type [, chunk])	
COLLAPSE (n)	
ORDERED	

Usage:

The semantics of the PARALLEL DO directive are identical to those of a parallel region containing only a single parallel DO loop and directive. The available clauses are the same as those defined in "PARALLEL ... END PARALLEL and parallel ," on page 112 and "DO...END DO and for ".

Note

The END PARALLEL DO directive is optional.

PARALLEL SECTIONS and parallel sections

The OpenMP PARALLEL SECTIONS / END SECTIONS directive pair and the omp parallel sections pragma define tasks to be executed in parallel; that is, they define a non-iterative work-sharing construct without the need to define an enclosing parallel region.

Syntax:

```
!$OMP PARALLEL SECTIONS [CLAUSES]
[!$OMP SECTION]
< Fortran code block executed
 by processor i >
[!$OMP SECTION]
< Fortran code block executed
 by processor j >
 ...
!$OMP END SECTIONS [NOWAIT]
```

```
#pragma omp parallel sections [clauses]
{
 [#pragma omp section]
 < C/C++ structured block executed
 by processor i >
 [#pragma omp section]
 < C/C++ structured block executed
 by processor j >
 ...
}
```

Clauses:

For Directives:	For Pragmas:
PRIVATE(list)	private(list)
SHARED(list)	shared(list)
DEFAULT(PRIVATE | SHARED | NONE)	default(shared | none)
FIRSTPRIVATE(list)	firstprivate(list)
LASTPRIVATE(list)	lastprivate (list)
REDUCTION({operator | intrinsic} : list)	reduction({operator: list)
COPYIN (list)	copyin (list)
IF(scalar_logical_expression)	if (scalar_expression)
NUM_THREADS(scalar_integer_expression)	num_threads(scalar_integer_expression)
	nowait

Usage:

The PARALLEL SECTIONS / END SECTIONS directive pair and the omp parallel sections pragma define a non-iterative work-sharing construct without the need to define an enclosing parallel region. Each section is executed by a single processor. If there are more processors than sections, some processors will have no work and will jump to the implied barrier at the end of the construct. If there are more sections than processors, one or more processors will execute more than one section.

A SECTION directive may only appear within the lexical extent of the enclosing PARALLEL SECTIONS / END SECTIONS directives. In addition, the code within the PARALLEL SECTIONS / END SECTIONS directives must be a structured block, and the code in each SECTION must be a structured block.

Semantics are identical to a parallel region containing only an omp sections pragma and the associated structured block. The available clauses are as defined in “PARALLEL ... END PARALLEL and parallel ,” on page 112 and “DO...END DO and for ”.

PARALLEL WORKSHARE ... END PARALLEL WORKSHARE

The OpenMP PARALLEL WORKSHARE Fortran directive provides a short form method of including a WORKSHARE directive inside a PARALLEL construct. The END PARALLEL WORKSHARE directive is optional.

Syntax:

```
!$OMP PARALLEL WORKSHARE [CLAUSES]
   < Fortran structured block to be executed in parallel >
[!$OMP END PARALLEL WORKSHARE]
```

```
!$OMP PARALLEL DO [CLAUSES]
   < Fortran DO loop to be executed in parallel >
[!$OMP END PARALLEL DO]
```

Clauses:

PRIVATE(list)	COPYIN (list)
SHARED(list)	IF(scalar_logical_expression)
DEFAULT(PRIVATE | SHARED | NONE)	NUM_THREADS(scalar_integer_expression)
FIRSTPRIVATE(list)	SCHEDULE (type [, chunk])
LASTPRIVATE(list)	COLLAPSE (n)
REDUCTION({operator | intrinsic} : list)	ORDERED

Usage:

The OpenMP PARALLEL WORKSHARE directive provides a short form method of including a WORKSHARE directive inside a PARALLEL construct. The semantics of the PARALLEL WORKSHARE directive are identical to those of a parallel region containing a single WORKSHARE construct.

The END PARALLEL WORKSHARE directive is optional, and NOWAIT may not be specified on an END PARALLEL WORKSHARE directive. The available clauses are as defined in "PARALLEL ... END PARALLEL and parallel ," on page 112.

SECTIONS … END SECTIONS and sections

The OpenMP SECTIONS / END SECTIONS directive pair and the omp sections pragma define a non-iterative work-sharing construct within a parallel region in which each section is executed by a single processor.

Syntax:

```
!$OMP SECTIONS [ Clauses ]
[!$OMP SECTION]
   < Fortran code block executed
  by processor i >
[!$OMP SECTION]
   < Fortran code block executed
  by processor j >
 ...
!$OMP END SECTIONS [NOWAIT]
```

```
#pragma omp sections [ Clauses ]
{
 [#pragma omp section]
   < C/C++ structured block executed
   by processor i >
 [#pragma omp section]
   < C/C++ structured block executed
   by processor j >
 ...
}
```

Clauses:

For Directives:	For Pragmas:
PRIVATE(list)	private(list)
FIRSTPRIVATE(list)	firstprivate(list)
LASTPRIVATE(list)	lastprivate (list)
REDUCTION({operator | intrinsic} : list)	reduction({operator: list)
	nowait

Usage:

The SECTIONS / END SECTIONS directive pair and the omp sections pragma define a non-iterative work-sharing construct within a parallel region. Each section is executed by a single processor. If there are more processors than sections, some processors have no work and thus jump to the implied barrier at the end of the construct. If there are more sections than processors, one or more processors must execute more than one section.

A SECTION directive or omp sections pragma may only appear within the lexical extent of the enclosing SECTIONS / END SECTIONS directives or omp sections pragma. In addition, the code within the SECTIONS / END SECTIONS directives or omp sections pragma must be a structured block.

The available clauses are as defined in "PARALLEL ... END PARALLEL and parallel ," on page 112 and "DO...END DO and for ".

SINGLE ... END SINGLE and single

The SINGLE...END SINGLE directive or omp single pragma designate code that executes on a single thread and that is skipped by the other threads.

Syntax:

```
!$OMP SINGLE [Clauses]
   < Fortran code executed in body of
  SINGLE processor section >
!$OMP END SINGLE [NOWAIT]
```

```
#pragma omp single [clauses]
   < C/C++ structured block >
```

Clauses:

For Directives:	For Pragmas:
PRIVATE(list)	private(list)
FIRSTPRIVATE(list)	firstprivate(list)
COPYPRIVATE(list)	copyprivate (list)
	nowait

Usage:

In a parallel region of code, there may be a sub-region of code that only executes correctly on a single thread. Instead of ending the parallel region before this subregion and then starting it up again after this subregion, the SINGLE...END SINGLE directive pair lets you conveniently designate code that executes on a single thread and is skipped by the other threads. There is an implied barrier on exit from a SINGLE...END SINGLE section of code unless the optional NOWAIT clause is specified.

Nested single process sections are ignored. Branching into or out of a single process section is not supported.

Examples:

For Directives:

```
   PROGRAM SINGLE_USE
   INTEGER A(0:1)
   INTEGER omp_get_thread_num()
!$OMP PARALLEL
   A(omp_get_thread_num()) =
     omp_get_thread_num()
!$OMP SINGLE
   PRINT *, "YOU ONLY SEE THIS ONCE"
!$OMP END SINGLE
!$OMP END PARALLEL
   PRINT *,"A(0)=",A(0),
          " A(1)=", A(1)
   END
```

For Pragmas:

```
single_use() {
    int a[2]={0,1};
#pragma omp parallel
{
      a(omp_get_thread_num()) =
     omp_get_thread_num();
#pragma omp single
{
   printf("You only see this once"
} /* end single */
} /* end parallel */
printf("a[0] = %d, a[1] = %d",
        a[0], a[1]);
} /* end single_use */
```

TASK and task

The OpenMP TASK directive and the omp task pragma define an explicit task.

Syntax:

```
!$OMP TASK [Clauses]
< Fortran code executed as task>
!$OMP END TASK
```

```
#pragma omp task [clauses]
< C/C++ structured block >
```

Clauses:

For Directives:	For Pragmas:
IF(scalar_logical_expression)	if (scalar_expression)
UNTIED	untied
DEFAULT(private | firstprivate | shared | none)	default(shared | none)
PRIVATE(list)	private(list)
FIRSTPRIVATE(list)	firstprivate(list)
SHARED(list)	shared(list)

Usage:

The TASK / END TASK directive pair and the omp task pragma define an explicit task.

When a thread encounters a task construct, a task is generated from the code for the associated structured block. The data environment of the task is created according to the data-sharing attribute clauses on the task construct and any defaults that apply. The encountering thread may immediately execute the task, or delay its execution. If the task execution is delayed, then any thread in the team may be assigned the task. Completion of the task can be guaranteed using task synchronization constructs.

A task construct may be nested inside an outer task, but the task region of the inner task is not a part of the task region of the outer task.

When an if clause is present on a task construct and the if clause expression evaluates to false, the encountering thread must suspend the current task region and begin execution of the generated task immediately, and the suspended task region may not be resumed until the generated task is completed. The task still behaves as a distinct task region with respect to data environment, lock ownership, and synchronization constructs.

Note

Use of a variable in an if clause expression of a task construct causes an implicit reference to the variable in all enclosing constructs.

A thread that encounters a task scheduling point within the task region may temporarily suspend the task region. By default, a task is tied and its suspended task region can only be resumed by the thread that started its execution. If the untied clause is present on a task construct, any thread in the team can resume the task region after a suspension.

The task construct includes a task scheduling point in the task region of its generating task, immediately following the generation of the explicit task. Each explicit task region includes a task scheduling point at its point of completion. An implementation may add task scheduling points anywhere in untied task regions.

Note

When storage is shared by an explicit task region, it is the programmer's responsibility to ensure, by adding proper synchronization, that the storage does not reach the end of its lifetime before the explicit task region completes its execution.

Restrictions:

The following restrictions apply to the TASK directive or omp task pragma:

- A program that branches into or out of a task region is non-conforming.
- A program must not depend on any ordering of the evaluations of the clauses of the task directive, or on any side effects of the evaluations of the clauses.
- At most one *if* clause can appear on the directive.
- Unsynchronized use of Fortran I/O statements by multiple tasks on the same unit has unspecified behavior.
- In C/C++, a throw executed inside a task region must cause execution to resume within the same task region, and the same thread that threw the exception must catch it.

TASKWAIT and taskwait

The OpenMP TASKWAIT directive and the omp taskwait pragma specify a wait on the completion of child tasks generated since the beginning of the current task.

Syntax:

```
!$OMP TASKWAIT
```

```
#pragma omp taskwait >
```

Clauses:

For Directives:	For Pragmas:
IF(scalar_logical_expression)	if (scalar_expression)
UNTIED	untied
DEFAULT(private | firstprivate | shared | none)	default(shared | none)
PRIVATE(list)	private(list)
FIRSTPRIVATE(list)	firstprivate(list)
SHARED(list)	shared(list)

Usage:

The OpenMP TASKWAIT directive and the omp taskwait pragma specify a wait on the completion of child tasks generated since the beginning of the current task.

Restrictions:

The following restrictions apply to the TASKWAIT directive or omp taskwait pragma:

- The TASKWAIT directive and the omp taskwait pragma may be placed only at a point where a base language statement is allowed.
- The taskwait directive may not be used in place of the statement following an *if*, *while*, *do*, *switch*, or *label*.

THREADPRIVATE and threadprivate

The OpenMP THREADPRIVATE directive identifies a Fortran common block as being private to each thread. The omp threadprivate pragma identifies a global variable as being private to each thread.

Syntax:

```
!$OMP THREADPRIVATE (list)          #pragma omp threadprivate (list)
```

Usage:

The `list` is a list of variables to be made private to each thread but global within the thread. For directives, common block names must appear between slashes, such as `/common_block_name/`.

This directive or pragma must appear in the declarations section of a program unit after the declaration of any variables listed. On entry to a parallel region, data in a threadprivate variable is undefined unless copyin is specified on the parallel directive or pragma. When a variable appears in an threadprivate directive or pragma, each thread's copy is initialized once at an unspecified point prior to its first use as the master copy would be initialized in a serial execution of the program.

Restrictions:

The following restrictions apply to the THREADPRIVATE directive or omp threadprivate pragma:

- The THREADPRIVATE directive must appear after every declaration of a thread private common block.
- The omp threadprivate pragma must appear after the declaration of every threadprivate variable included in list.
- Only named common blocks can be made thread private.
- It is illegal for a THREADPRIVATE common block or its constituent variables to appear in any clause other than a COPYIN clause.
- A variable can appear in a THREADRIVATE directive only in the scope in which it is declared. It must not be an element of a common block or be declared in an EQUIVALENCE statement.
- A variable that appears in a THREADPRIVATE directive and is not declared in the scope of a module must have the SAVE attribute.
- If a variable is specified in an omp threadprivate pragma in one translation unit, it must be specified in an omp threadprivate pragma in every translation unit in which it appears.
- The address of an omp threadprivate variable is not an address constant.

- An omp threadprivate variable must not have an incomplete type or a reference type.

WORKSHARE ... END WORKSHARE

The OpenMP WORKSHARE … END WORKSHARE Fortran directive pair provides a mechanism to effect parallel execution of non-iterative but implicitly data parallel constructs.

Syntax:

```
!$OMP WORKSHARE
   < Fortran structured block to be executed in parallel >
!$OMP END WORKSHARE [NOWAIT]
```

Usage:

The Fortran structured block enclosed by the WORKSHARE … END WORKSHARE directive pair can consist only of the following types of statements and constructs:

- Array assignments
- Scalar assignments
- FORALL statements or constructs
- WHERE statements or constructs
- OpenMP ATOMIC, CRITICAL or PARALLEL constructs

The work implied by these statements and constructs is split up between the threads executing the WORKSHARE construct in a way that is guaranteed to maintain standard Fortran semantics. The goal of the WORKSHARE construct is to effect parallel execution of non-iterative but implicitly data parallel array assignments, FORALL, and WHERE statements and constructs intrinsic to the Fortran language beginning with Fortran 90. The Fortran structured block contained within a WORKSHARE construct must not contain any user-defined function calls unless the function is ELEMENTAL.

Directive and Pragma Clauses

Some directives and C/C++ pragmas accept clauses that further allow a user to control the scope attributes of variables for the duration of the directive or pragma. Not all clauses are allowed on all directives or pragmas, so the clauses that are valid are included with the description of the directive or pragma. Typically, if no data scope clause is specified for variables, the default scope is *shared*.

The PGI Compiler User's Guide provides a table that contains a brief summary of the clauses associated with OpenMP directives and pragmas that PGI supports. This section contains more information about each of these clauses. For complete information and more details related to use of these clauses, refer to the OpenMP documentation available on the WorldWide Web.

COLLAPSE (n)

The COLLAPSE(n) clause specifies how many loops are associated with the loop construct.

The parameter of the collapse clause must be a constant positive integer expression. If no COLLAPSE clause is present, the only loop that is associated with the loop construct is the one that immediately follows the construct.

If more than one loop is associated with the loop construct, then the iterations of all associated loops are collapsed into one larger iteration space, which is then divided according to the schedule clause. The sequential execution of the iterations in all associated loops determines the order of the iterations in the collapsed iteration space.

If the loop directive contains a COLLAPSE clause then there may be more than one associated loop.

COPYIN (list)

The COPYIN(list) clause allows threads to access the master thread's value, for a threadprivate variable. You assign the same value to threadprivate variables for each thread in the team executing the parallel region; that is, for each variable specified, the value of the variable in the master thread of the team is copied to the threadprivate copies at the beginning of the parallel region.

The COPYIN clause applies only to THREADPRIVATE common blocks. If you specify a COPYIN clause, here are a few tips:

- You cannot specify the same entity name more than once in the list.
- You cannot specify the same entity name in separate COPYIN clauses of the same directive.
- You cannot specify both a common block name and any variable within that same named common block in the list.
- You cannot specify both a common block name and any variable within that same named common block in separate COPYIN clauses of the same directive.

COPYPRIVATE(list)

The COPYPRIVATE(list) clause specifies that one or more variables should be shared among all threads. This clause provides a mechanism to use a private variable to broadcast a value from one member of a team to the other members.

You use a COPYPRIVATE(list) clause on an END SINGLE directive to cause the variables in the list to be copied from the private copies in the single thread that executes the SINGLE region to the other copies in all other threads of the team at the end of the SINGLE region.

Note

The COPYPRIVATE clause must not appear on the same END SINGLE directive as a NOWAIT clause.

The compiler evaluates a COPYPRIVATE clause before any threads have passed the implied BARRIER directive at the end of that construct.

DEFAULT

The DEFAULT clause specifies the behavior of unscoped variables in a parallel region, such as the data-sharing attributes of variables. The DEFAULT clause lets you specify the default attribute for variables in the lexical

extent of the parallel region. Individual clauses specifying PRIVATE, SHARED, and so on, override the declared DEFAULT.

Specifying DEFAULT(NONE) declares that there is no implicit default. With this declaration, each variable in the parallel region must be explicitly listed with an attribute of PRIVATE, SHARED, FIRSTPRIVATE, LASTPRIVATE, or REDUCTION.

FIRSTPRIVATE(list)

The FIRSTPRIVATE(list) clause specifies that each thread should have its own instance of a variable, and that each variable in the list should be initialized with the value of the original variable, because it exists before the parallel construct.

Variables that appear in the list of a FIRSTPRIVATE clause are subject to the same semantics as PRIVATE variables; however, these variables are initialized from the original object that exists prior to entering the parallel region.

If a directive construct contains a FIRSTPRIVATE argument to a Message Passing Interface (MPI) routine performing non-blocking communication, the MPI communication must complete before the end of the construct.

IF()

The IF() clause specifies whether a loop should be executed in parallel or in serial.

In the presence of an IF clause, the parallel region is executed in parallel only if the corresponding `scalar_logical_expression` evaluates to .TRUE.. Otherwise, the code within the region is executed by a single processor, regardless of the value of the environment variable `OMP_NUM_THREADS`.

LASTPRIVATE(list)

The LASTPRIVATE(list) clause specifies that the enclosing context's version of the variable is set equal to the *private* version of whichever thread executes the final iteration (for-loop construct) or last section (#pragma sections).

NOWAIT

The NOWAIT clause overrides the barrier implicit in a directive. When you specify NOWAIT, it removes the implicit barrier synchronization at the end of a for or sections construct.

NUM_THREADS

The NUM_THREADS clause sets the number of threads in a thread team. The num_threads clause allows a user to request a specific number of threads for a parallel construct. If the num_threads clause is present, then

ORDERED

The ORDERED clause specifies that a loop is executed in the order of the loop iterations. This clause is required on a parallel FOR statement when an ordered directive is used in the loop.

You use this clause in conjunction with a DO or SECTIONS construct to impose a serial order on the execution of a section of code. If ORDERED constructs are contained in the dynamic extent of the DO construct, the ordered clause must be present on the DO directive.

PRIVATE

The PRIVATE clause specifies that each thread should have its own instance of a variable. Therefore, variables specified in a PRIVATE list are private to each thread in a team. In effect, the compiler creates a separate copy of each of these variables for each thread in the team. When an assignment to a private variable occurs, each thread assigns to its local copy of the variable. When operations involving a private variable occur, each thread performs the operations using its local copy of the variable.

Tips about private variables:

- Variables declared private in a parallel region are undefined upon entry to the parallel region. If the first use of a private variable within the parallel region is in a right-hand-side expression, the results of the expression are undefined, indicating the probability of a coding error.
- Variables declared private in a parallel region are undefined when serial execution resumes at the end of the parallel region.

REDUCTION

The REDUCTION clause specifies that one or more variables that are private to each thread are the subject of a reduction operation at the end of the parallel region. It updates named variables declared on the clause within the directive construct.

- Intermediate values of REDUCTION variables are not used within the parallel construct, other than in the updates themselves.
- Variables that appear in the list of a REDUCTION clause must be SHARED.
- A private copy of each variable in `list` is created for each thread as if the PRIVATE clause had been specified.

Each private copy is initialized according to the operator as specified in the following table:

Table 3.1. Initialization of REDUCTION Variables

For Directives	
Operator / Intrinsic	**Initialization**
+	0
*	1
-	0

For Pragmas	
Operator	**Initialization**
+	0
*	1
-	0

For Directives	
Operator / Intrinsic	**Initialization**
.AND.	.TRUE.
.OR.	.FALSE.
.EQV.	.TRUE.
.NEQV.	.FALSE.
MAX	Smallest Representable Number
MIN	Largest Representable Number
IAND	All bits on
IOR	0
IEOR	0

For Pragmas	
Operator	**Initialization**
&	~0
|	0
^	0
&&	1
||	0

At the end of the parallel region, a reduction is performed on the instances of variables appearing in `list` using operator or intrinsic as specified in the REDUCTION clause. The initial value of each REDUCTION variable is included in the reduction operation. If the `{operator | intrinsic}:` portion of the REDUCTION clause is omitted, the default reduction operator is "+" (addition).

SCHEDULE

The SCHEDULE clause specifies how iterations of the DO or for loop are divided up between processors. Given a SCHEDULE (type [, chunk]) clause, the type can be STATIC, DYNAMIC, GUIDED, or RUNTIME, defined in the following list.

Note

For pragmas, the values for the clause are lower case static, dynamic, guided, or runtime. For simplicity, we use the directive uppercase value in the following information.

- When SCHEDULE (STATIC, chunk) is specified, iterations are allocated in contiguous blocks of size chunk. The blocks of iterations are statically assigned to threads in a round-robin fashion in order of the thread ID numbers. The chunk must be a scalar integer expression. If chunk is not specified, a default chunk size is chosen equal to:

```
(number_of_iterations + omp_num_threads() - 1) / omp_num_threads()
```

- When SCHEDULE (DYNAMIC, chunk) is specified, iterations are allocated in contiguous blocks of size chunk. As each thread finishes a piece of the iteration space, it dynamically obtains the next set of iterations. The chunk must be a scalar integer expression. If no chunk is specified, a default chunk size is chosen equal to 1.
- When SCHEDULE (GUIDED, chunk) is specified, the chunk size is reduced in an exponentially decreasing manner with each dispatched piece of the iteration space. Chunk specifies the minimum number of iterations to dispatch each time, except when there are less than chunk iterations remaining to be processed, at which point all remaining iterations are assigned. If no chunk is specified, a default chunk size is chosen equal to 1.

- When SCHEDULE (RUNTIME) is specified, the decision regarding iteration scheduling is deferred until runtime. The schedule type and chunk size can be chosen at runtime by setting the OMP_SCHEDULE environment variable. If this environment variable is not set, the resulting schedule is equivalent to SCHEDULE(STATIC).

SHARED

The SHARED clause specifies variables that must be available to all threads. If you specify a variable as SHARED, you are stating that all threads can safely share a single copy of the variable. When one or more variables are shared among all threads, all threads access the same storage area for the shared variables.

UNTIED

The UNTIED clause specifies that any thread in the team can resume the task region after a suspension.

Note

The thread number may change at any time during the execution of an untied task. Therefore, the value returned by `omp_get_thread_num` is generally not useful during execution of such a task region.

OpenMP Environment Variables

OpenMP environment variables allow you to control the behavior of OpenMP programs. These environment variables allow you to set and pass information that can alter the behavior of directives and pragmas.

The PGI Compiler User's Guide contains a table that provides a brief summary of these variables. This section contains more information about each of them. For complete information and more details related to these environment variables, refer to the OpenMP documentation available on the WorldWide Web.

OMP_DYNAMIC

`OMP_DYNAMIC` currently has no effect. Typically this variable enables (TRUE) or disables (FALSE) the dynamic adjustment of the number of threads.

OMP_NESTED

`OMP_NESTED` currently has no effect. Typically this variable enables (TRUE) or disables (FALSE) nested parallelism.

OMP_MAX_ACTIVE_LEVELS

`OMP_MAX_ACTIVE_LEVELS` currently has no effect. Typically this variable specifies the maximum number of nested parallel regions. PGI ignores this variable value since nested parallelism is not supported.

OMP_NUM_THREADS

`OMP_NUM_THREADS` specifies the number of threads to use during execution of parallel regions. The default value for this variable is 1. For historical reasons, the environment variable `NCPUS` is supported with

the same functionality. In the event that both OMP_NUM_THREADS and NCPUS are defined, the value of OMP_NUM_THREADS takes precedence.

Note

OMP_NUM_THREADS defines the threads that are used to execute the program, regardless of the number of physical processors available in the system. As a result, you can run programs using more threads than physical processors and they execute correctly. However, performance of programs executed in this manner can be unpredictable, and oftentimes will be inefficient.

OMP_SCHEDULE

OMP_SCHEDULE specifies the type of iteration scheduling to use for DO and PARALLEL DO loop directives and for omp for and omp parallel for loop pragmas that include the SCHEDULE(RUNTIME) clause, described in "SCHEDULE," on page 125. The default value for this variable is STATIC.

If the optional chunk size is not set, a chunk size of 1 is assumed except in the case of a static schedule. For a static schedule, the default is as defined in "DO...END DO and for ," on page 108.

Examples of the use of OMP_SCHEDULE are as follows:

For Fortran:

```
% setenv OMP_SCHEDULE "STATIC, 5"
% setenv OMP_SCHEDULE "GUIDED, 8"
% setenv OMP_SCHEDULE "DYNAMIC"
```

For C/C++:

```
% setenv OMP_SCHEDULE "static, 5"
% setenv OMP_SCHEDULE "guided, 8"
% setenv OMP_SCHEDULE "dynamic"
```

OMP_STACKSIZE

OMP_STACKSIZE is an OpenMP 3.0 feature that controls the size of the stack for newly-created threads. This variable overrides the default stack size for a newly created thread. The value is a decimal integer followed by an optional letter B, K, M, or G, to specify bytes, kilobytes, megabytes, and gigabytes, respectively. If no letter is used, the default is kilobytes. There is no space between the value and the letter; for example, one megabyte is specified 1M. The following example specifies a stack size of 8 megabytes.

```
% setenv OMP_STACKSIZE 8M
```

The API functions related to OMP_STACKSIZE are omp_set_stack_size and omp_get_stack_size.

The environment variable OMP_STACKSIZE is read on program start-up. If the program changes its own environment, the variable is not re-checked.

This environment variable takes precedence over MPSTKZ. Once a thread is created, its stack size cannot be changed.

In the PGI implementation, threads are created prior to the first parallel region and persist for the life of the program. The stack size of the main thread (thread 0) is set at program start-up and is not affected by OMP_STACKSIZE.

OMP_THREAD_LIMIT

You can use the OMP_THREAD_LIMIT environment variable to specify the absolute maximum number of threads that can be used in a parallel program. Attempts to dynamically set the number of processes or threads

to a higher value, for example using set_omp_num_threads(), cause the number of processes or threads to be set at the value of `OMP_THREAD_LIMIT` rather than the value specified in the function call.

OMP_WAIT_POLICY

`OMP_WAIT_POLICY` sets the behavior of idle threads - specifically, whether they spin or sleep when idle. The values are ACTIVE and PASSIVE, with ACTIVE the default. The behavior defined by `OMP_WAIT_POLICY` is also shared by threads created by auto-parallelization.

- Threads are considered idle when waiting at a barrier, when waiting to enter a critical region, or when unemployed between parallel regions.
- Threads waiting for critical sections always busy wait (ACTIVE).
- Barriers always busy wait (ACTIVE), with calls to `sched_yield` determined by the environment variable `MP_SPIN`.
- Unemployed threads during a serial region can either busy wait using the barrier (ACTIVE) or politely wait using a mutex (PASSIVE). This choice is set by `OMP_WAIT_POLICY`, so the default is ACTIVE.

When ACTIVE is set, idle threads consume 100% of their CPU allotment spinning in a busy loop waiting to restart in a parallel region. This mechanism allows for very quick entry into parallel regions, a condition which is good for programs that enter and leave parallel regions frequently.

When PASSIVE is set, idle threads wait on a mutex in the operating system and consume no CPU time until being restarted. Passive idle is best when a program has long periods of serial activity or when the program runs on a multi-user machine or otherwise shares CPU resources.

Chapter 4. PGI Accelerator Compilers Reference

The chapter "Using an Accelerator" in the PGI Compiler User's Guide describes the programming model that uses a collection of compiler directives to specify regions of code in Fortran and C programs that can be offloaded from a *host* CPU to an attached *accelerator*. The method described provides a model for accelerator programming that is portable across operating systems and various types of host CPUs and accelerators.

PGI provides a set of Fortran and C accelerator compilers and tools for 64-bit x86-compatible processor-based workstations and servers with an attached NVIDIA CUDA-enabled GPU or Tesla card.

Note

The PGI Accelerator compilers require a separate license key in addition to a normal PGI Workstation, Server, or CDK license.

This chapter contains detailed descriptions of each of the PGI Accelerator directives and C pragmas that PGI supports. In addition, the section "PGI Accelerator Directive Clauses," on page 135 contains information about the clauses associated with these directives and pragmas.

PGI Accelerator Directives

This section provides detailed descriptions of the Fortran and C directives used to delineate accelerator regions and to augment information available to the compiler for scheduling of loops and classification of data.

- In C, PGI Accelerator directives are specified using the #pragma mechanism provided by the standard.
- In Fortran, PGI Accelerator directives are specified using special comments that are identified by a unique sentinel.

This syntax enables compilers to ignore accelerator directives if support is disabled or not provided.

PGI currently supports these types of accelerator directives:

- An "Accelerator Compute Region Directive" defines information about the region of a program. These directives are either an accelerator compute region directive, that defines the region of a program to be compiled for execution on the accelerator device, or an accelerator data region directive that

- An "Accelerator Loop Mapping Directive" describes the type of parallelism to use to execute the loop and declare loop-private variables and arrays.
- A "Combined Directive" is a combination of the Accelerator region and loop mapping directives, and specifies a loop directive nested immediately inside an accelerator region directive.
- An "Accelerator Declarative Data Directive" specifies an array or arrays are to be allocated in the device memory for the duration of the implicit data region of a function, subroutine or program, and specifies whether the data values are to be transferred from the host to the device memory upon entry to the implicit data region, and from the device to the host memory upon exit from the implicit data region.
- An "Accelerator Update Directive" used within an explicit or implicit data region, updates all or part of a host memory array with values from the corresponding array in device memory, or updates all or part of a device memory array with values from the corresponding array in host memory.

Accelerator Compute Region Directive

This directive defines the region of the program that should be compiled for execution on an accelerator device.

Syntax

In C, the syntax of an accelerator region directive is:

```
#pragma acc region [clause [, clause]…] new-line
 structured block
```

In Fortran, the syntax is:

```
!$acc region [clause [, clause]…]
    structured block
!$acc end region
```

where clause is one of the following, described in more detail in "PGI Accelerator Directive Clauses":

```
if( condition )
copy( list )
copyin( list )
copyout( list )
deviceptr( list )   [Available in C only]
local( list )
updatein( list )
updateout( list )
```

Description

Loops within the structured block are compiled into accelerator kernels. Data is copied from the host memory to the accelerator memory, as required, and result data is copied back. Any computation that cannot be executed on the accelerator, perhaps because of limitations of the device, is executed on the host. This approach may require data to move back and forth between the host and device.

At the end of the region, all results stored on the device that are needed on the host are copied back to the host memory, and accelerator memory is deallocated.

Restrictions

The following restrictions apply to the accelerator compute region directive:

- Accelerator regions may not be nested.
- A program may not branch into or out of an accelerator region.
- A program must not depend on the order of evaluation of the clauses, or on any side effects of the evaluations.
- At most one `if` clause may appear. In Fortran, the condition must evaluate to a scalar logical value; in C, the condition must evaluate to a scalar integer value.
- A variable may appear in only one of the `local`, `copy`, `copyin` or `copyout` lists.

Accelerator Data Region Directive

This directive defines data, typically arrays, that should be allocated in the device memory for the duration of the data region. Further, it defines whether data should be copied from the host to the device memory upon region entry, and copied from the device to host memory upon region exit.

Syntax

In C, the syntax of an accelerator region directive is:

```
#pragma acc data region [clause [, clause]…] new-line
 structured block
```

In Fortran, the syntax is:

```
!$acc data region [clause [, clause]…]
    structured block
!$acc end data region
```

where clause is one of the following, described in more detail in "PGI Accelerator Directive Clauses":

```
copy( list )
copyin( list )
copyout( list )
deviceptr( list )   [Available in C only]
local( list )
mirror( list )
updatein( list )
updateout( list )
```

Description

Data is allocated in the device memory and copied from the host memory to the device, or copied back, as required. The *list* argument to each data clause is a comma-separated collection of variable names, array names, or subarray specifications. In all cases, the compiler allocates and manages a copy of the variable or array in device memory, creating a visible device copy of that variable or array.

In C, a subarray is an array name followed by a range specification in brackets, such as this:

```
arr[2:high][low:100]
```

In Fortran, a subarray is an array name followed by a comma-separated list of range specifications in parentheses, such as this:

```
arr(2:high,low:100)
```

If either the lower or upper bounds are missing, the declared or allocated bounds of the array, if known, are used. Using an array name in a data clause on a compute region directive without bounds tells the compiler to analyze the references to the array to determine what bounds to use. Thus, every array reference is equivalent to some subarray of that array.

Restrictions

The following restrictions apply to the accelerator data region directive:

- A variable, array, or subarray may appear at most once in all data clauses for a compute or data region.
- Only one subarray of an array may appear in all data clauses for a region.
- If a variable, array, or subarray appears in a data clause for a region, the same variable, array, or any subarray of the same array may not appear in a data clause for any enclosed region.
- In Fortran, the upper bound for the last dimension of an assumed-size dummy array must be specified.
- In C, a missing lower bound is assumed to be zero. A missing upper bound for a dynamically allocated array must be specified.
- If a subarray is specified in a data clause, the compiler may choose to allocate memory for only that subarray on the accelerator.
- The compiler may pad dimensions of arrays on the accelerator to improve memory alignment and program performance.
- The mirror clause is valid only in Fortran. The *list* argument to the mirror clause, a comma-separated list of array names. The arrays may be explicit shape, assumed shape, or allocatable; pointer arrays are not allowed.
- In Fortran, pointer arrays may be specified, but pointer association is not preserved in the device memory.

Accelerator Loop Mapping Directive

An accelerator loop mapping directive specifies the type of parallelism to use to execute the loop and declare loop-private variables and arrays.

Syntax

In C, the syntax of an accelerator loop mapping directive is

```
#pragma acc for [clause [,clause]…] new-line
     for loop
```

In Fortran, the syntax of an accelerator loop mapping directive is

```
!$acc do [clause [,clause]…]
         do loop
```

where clause is one of the following, described in more detail in “PGI Accelerator Directive Clauses”:

```
    cache (list)]
    host [(width)]
    independent
    kernel
    parallel [(width)]
    private( list)
    seq [(width)]
    unroll [(width)]
    vector [(width)]
```

Description

An accelerator loop mapping directive applies to a loop which must appear on the following line. It can describe what type of parallelism to use to execute the loop and declare loop-private variables and arrays.

Combined Directive

The combined accelerator region and loop mapping directive is a shortcut for specifying a loop directive nested immediately inside an accelerator region directive. The meaning is identical to explicitly specifying a region construct containing a loop directive. Any clause that is allowed in a region directive or a loop directive is allowed in a combined directive.

Syntax

In C, the syntax of the combined accelerator region and loop directive is:

```
#pragma acc region for [clause [, clause]...] new-line
            for loop
```

In Fortran the syntax of the combined accelerator region and loop directive is:

```
  !$acc region do [clause [, clause]...]
            do loop
```

where *clause* is any of the region or loop clauses described previously in this chapter.

The associated region is the body of the loop which must immediately follow the directive.

Restrictions

The following restrictions apply to a combined directive:

- The combined accelerator region and loop directive may not appear within the body of another accelerator region.
- All restrictions for the region directive and the loop directive apply.

Accelerator Declarative Data Directive

Declarative data directives specify that an array or arrays are to be allocated in the device memory for the duration of the implicit data region of a function, subroutine, or program. They also specify whether the data

values are to be transferred from the host to the device memory upon entry to the implicit data region, and from the device to the host memory upon exit from the implicit data region.

You use the declarative data directives in the declaration section of a Fortran subroutine, function, or module, or just following an array declaration in C.

These directives create a visible device copy of the variable or array.

Syntax

In C, the syntax of the declarative data directive is:

```
#pragma acc declclause [, declclause]... new-line
```

In Fortran the syntax of the declarative data directive is:

```
!$acc declclause [, declclause]...
```

where `declclause` is one of the following:

```
copy( list )
copyin( list )
copyout( list )
deviceptr( list )
local( list )
mirror( list )
reflected( list )
```

Description

The associated region is the implicit region associated with the function, subroutine, or program in which the directive appears.

If the directive appears in a Fortran MODULE subprogram, the associated region is the implicit region for the whole program.

Restrictions

- A variable or array may appear at most once in all declarative data clauses for a function, subroutine, program, or module.
- Subarrays are not allowed in declarative data clauses.
- If variable or array appears in a declarative data clause, the same variable or array may not appear in a data clause for any region where the declaration of the variable is visible.
- In Fortran, assumed-size dummy arrays may not appear in declarative data clauses.
- The compiler may pad dimensions of arrays on the accelerator to improve memory alignment and program performance.
- The mirror and reflected clauses are valid only in Fortran.
- In Fortran, pointer arrays may be specified, but pointer association is not preserved in the device memory.

Accelerator Update Directive

The update directive is used within an explicit or implicit data region to do one of the following:

- Update all or part of a host memory array with values from the corresponding array in device memory.
- Update all or part of a device memory array with values from the corresponding array in host memory.

Syntax

In C, the syntax of the update directive is:

```
#pragma acc update updateclause[, updateclause]... new-line
```

In Fortran the syntax of the update data directive is:

```
!$acc update updateclause [, updateclause]...
```

where `updateclause` is one of the following:

```
host( list )
device( list )
```

Description

The effect of an update clause is to copy data from the device memory to the host memory for updateout, and from host memory to device memory for `updatein`. The following is true:

- The *list* argument to an update clause is a comma-separated collection of variable names, array names, or subarray specifications.
- Multiple subarrays of the same array may appear in a list.
- The updates are done in the order in which they appear on the directive.

Restrictions

These restrictions apply:

- The update directive is executable. It must not appear in place of the statement following an `if`, `while`, `do`, `switch`, or `label` in C, or in place of the statement following a `logical if` in Fortran.
- A variable or array which appears in the list of an update directive must have a visible device copy.

PGI Accelerator Directive Clauses

Accelerator directives accept clauses that further clarify the use of the directive. Some of these clauses are specific to certain directives.

Accelerator Region Directive Clauses

The following clauses further clarify the use of the Accelerator Region directive.

if (condition)

The `if` clause is optional.

- When there is no `if` clause, the compiler generates code to execute as much of the region on the accelerator as possible.
- When an `if` clause appears, the compiler generates two copies of the region, one copy to execute on the accelerator and one copy to execute on the host. The condition in the `if` clause determines whether the host or accelerator copy is executed.
  - When the condition in the `if` clause evaluates to zero in C, or `.false.` in Fortran, the host copy is executed.
  - When the condition in the `if` clause evaluates to nonzero in C, or `.true.` in Fortran, the accelerator copy is executed.

Data Clauses

The data clauses for an accelerator region directive are one of the following:

```
copy( list )
copyout( list )
copyin( list )
deviceptr( list)
local( list )
mirror( list )
updatein( list )
updateout( list )
```

Data clauses are optional, but may assist the compiler in generating code for an accelerator or in generating more optimal accelerator kernels.

Note

By default, the PGI Accelerator compilers attempt to minimize data movement between the host and accelerator. As a result, for many accelerator regions the compilers choose to copy sub-arrays which may be non-contiguous. Performance of an accelerator may improve in these cases if the user inserts explicit copy/copyin/copyout clauses on the accelerator region directive to specify to copy whole arrays rather than sub-arrays. Depending on the architecture of the target accelerator memory, performance also may improve if one or more dimensions of copied arrays are padded.

For each variable or array used in the region that does not appear in any data clause, the compiler analyzes all references to the variable or array and determines:

- For arrays, how much memory needs to be allocated in the accelerator memory to hold the array;
- Whether the value in host memory needs to be copied to the accelerator memory;
- Whether a value computed on the accelerator will be needed again on the host, and therefore needs to be copied back to the host memory.

When compiler analysis is unable to determine these items, it may fail to generate code for the accelerator; in that case, it issues a message to notify the programmer why it failed. You can use data clauses to augment or override this compiler analysis.

List arguments

When a data clause is used, the *list* argument is a comma-separated collection of variable names, array names, or subarray specifications.

- Using an array name without bounds tells the compiler to use the whole array. Thus, every array reference is equivalent to some subarray of that array.
- In C, a subarray is an array name followed by a range specification in brackets, such as the following:

```
arr[2:high]
```

- In Fortran, a subarray is an array name followed by a comma-separated list of range specifications in parentheses, such as the following:

```
arr(2:high,low:100)
```

Array Restrictions

An accelerator region data clause has these restrictions related to arrays:

- If either the lower or upper bounds of an array are missing, the declared or allocated bounds of the array, if known, are used.
- In Fortran, the upper bound for the last dimension of an assumed-size dummy array must be specified.
- In C, a missing lower bound of an array is assumed to be zero. A missing upper bound for a dynamically allocated array must be specified.
- If a subarray is specified, then only that subarray of the array needs to be copied.
- Only one subarray for an array may appear in any data clause for a region.
- The compiler may pad dimensions of allocated arrays or subarrays to improve memory alignment and program performance.

copy (list)

You use the `copy` clause to declare that the variables, arrays or subarrays in the *list* have values in the host memory that need to be copied to the accelerator memory, and are assigned values on the accelerator that need to be copied back to the host.

- The data is copied to the device memory upon entry to the region.
- Data is copied back to the host memory upon exit from the region.

copyin (list)

You use the `copyin` clause to declare that the variables, arrays or subarrays in the *list* have values in the host memory that need to be copied to the accelerator memory.

- The data is copied to the device memory upon entry to the implicit region associated with the directive.
- If a variable, array ,or subarray appears in a copyin clause then that data need not be copied back from the device memory to the host memory, even if those values were changed on the accelerator.

copyout (list)

You use the `copyout` clause to declare that the variables, arrays, or subarrays in the *list* are assigned or contain values in the accelerator memory that need to be copied back to the host memory.

- The data is copied back to the host memory upon exit from the region.
- If a variable, array or subarray appears in a copyout clause, then that data need not be copied to the device memory from the host memory, even if those values are used on the accelerator.

deviceptr (list)

You use the `deviceptr` clause to declare that the pointers in the list are device pointers so that the compiler need not move data between the host and device for accesses using these base pointers.

local (list)

You use the `local` clause to declare that the variables, arrays or subarrays in the *list* need to be allocated in the accelerator memory, but the values in the host memory are not needed on the accelerator, and the values computed and assigned on the accelerator are not needed on the host.

mirror (list)

You use the `mirror` clause to declare that the arrays in the list need to mirror the allocation state of the host array within the implicit region.

- If the host array is allocated upon region entry, the device copy of the array is allocated at region entry to the same size.
- If the host array is not allocated, the device copy is initialized to an unallocated state.
- If the host array is allocated or deallocated within the region, the device copy is allocated to the same size, or deallocated, at the same point in the region.
- If it is still allocated at region exit, the device copy is automatically deallocated.
- When used in a Fortran module subprogram, the associated region is the implicit region for the whole program.

Mirror Clause Restrictions

- The mirror clause is valid only in Fortran.
- The *list* argument to the mirror clause is a comma-separated list of array names. The arrays may be explicit shape, assumed shape, or allocatable; pointer arrays are not allowed.
- The mirror clause may be used for Fortran allocatable arrays in module subprograms; the `copy`, `copyin`, `copyout`, `local`, and `reflected` clauses may not be.

updatein|updateout (list)

The `update` clauses allow you to update values of variables, arrays, or subarrays.

- The *list* argument to each update clause is a comma-separated collection of variable names, array names, or subarray specifications.
- All variables or arrays that appear in the *list* argument of an update clause must have a visible device copy outside the compute or data region.
- Multiple subarrays of the same array may appear in update clauses for the same region, potentially causing updates of different subarrays in each direction.

updatein (list)

The `updatein` clause copies the variables, arrays, or subarrays in the *list* argument from host memory to the visible device copy of the variables, arrays, or subarrays in device memory, before beginning execution of the compute or data region.

updateout (list)

The `updateout` clause copies the visible device copies of the variables, arrays, or subarrays in the *list* argument to the associated host memory locations, after completion of the compute or data region.

Loop Scheduling Clauses

The loop scheduling clauses tell the compiler about loop level parallelism and how to map the parallelism onto the accelerator parallelism.

The loop scheduling clauses for the accelerator loop mapping directive are one of the following:

```
cache (list)]
host [(width)]
independent
kernel
parallel [(width)]
private( list )
seq [(width)]
unroll [(width)]
vector [(width)]
```

The loop scheduling clauses tell the compiler about loop level parallelism and how to map the parallelism onto the accelerator parallelism.

The loop scheduling clauses are optional.

For each loop without a scheduling clause, the compiler determines an appropriate schedule automatically.

loop scheduling clauses restrictions

The loop scheduling clauses have these restrictions:

- In some cases, there is a limit on the trip count of a parallel loop on the accelerator. For instance, some accelerators have a limit on the maximum length of a vector loop. In such cases, the compiler strip-mines the loop, so one of the loops has a maximum trip count that satisfies the limit.

  For example, if the maximum vector length is 256, the compiler uses strip-mining to compile a vector loop like the following one:

  ```
  !$acc do vector
    do i = 1,n
  ```

  into the following pair of loops:

  ```
  do is = 1,n,256
   !$acc do vector
     do i = is,max(is+255,n)
  ```

  The compiler then chooses an appropriate schedule for the outer, strip loop.

- If more than one scheduling clause appears on the loop directive, the compiler strip-mines the loop to get at least that many nested loops, applying one loop scheduling clause to each level.
- If a loop scheduling clause has a *width* argument, the compiler strip-mines the loop to that width, applying the scheduling clause to the outer strip or inner element loop, and then determines the appropriate schedule for the other loop.
- The *width* argument must be a compile-time positive constant integer.
- If two or more loop scheduling clauses appear on a single loop mapping directive, all but one must have a *width* argument.
- Some implementations or targets may require the *width* argument for the vector clause to be a compile-time constant.
- Some implementations or targets may require the *width* argument for the vector or parallel clauses to be a power of two, or a multiple of some power of two. If so, the behavior when the restriction is violated is implementation-defined.

loop scheduling clause examples

In the following example, the compiler strip-mines the loop to 16 host iterations:

```
!$acc do host(16), parallel
  do i = 1,n
```

The parallel clause applies to the inner loop, as follows:

```
ns = ceil(n/16)
!$acc do host
  do is = 1, n, ns
  !$acc do parallel
    do i = is, min(n,is+ns-1)
```

cache (list)

The `cache` clause provides a hint to the compiler to try to move the variables, arrays, or subarrays in the *list* to the highest level of the memory hierarchy.

Many accelerators have a software-managed fast cache memory, and the cache clause can help the compiler choose what data to keep in that fast memory for the duration of the loop. The compiler is not required to store all or even any of the data items in the cache memory.

host [(width)]

The `host` clause tells the compiler to execute the loop sequentially on the host processor. There is no maximum number of iterations on a host schedule. If the `host` clause has a *width* argument, the compiler strip mines the loop to that many strips, and determines an appropriate schedule for the remaining loop.

independent

The `independent` clause tells the compiler that the iterations of this loop are data-independent of each other. This allows the compiler to generate code to examine the iterations in parallel, without synchronization.

Note

It is an error to use the `independent` clause if any iteration writes to a variable or array element that any other iterations also writes or reads.

kernel

The `kernel` clause tells the compiler that the body of this loop is to be the body of the computational kernel. Any loops contained within the kernel loop are executed sequentially on the accelerator.

kernel clause restrictions

The `kernel` clause has this restriction:

- Loop mapping directives must not appear on any loop contained within the kernel loop.

parallel [(width)]

The `parallel` clause tells the compiler to execute this loop in parallel mode on the accelerator. There may be a target-specific limit on the number of iterations in a parallel loop or on the number of parallel loops allowed in a given kernel. If there is a limit:

- If there is no *width* argument, or the value of the *width* argument is greater than the limit, the compiler enforces the limit.
- If there is a *width* argument or a limit on the number of iterations in a parallel loop, then only that many iterations can run in parallel at a time.

private (list)

You use the `private` clause to declare that the variables, arrays, or subarrays in the *list* argument need to be allocated in the accelerator memory with one copy for each iteration of the loop.

Any value of the variable or array used in the loop must have been computed and assigned in that iteration of the loop, and the values computed and assigned in any iteration are not needed after completion of the loop.

Using an array name without bounds tells the compiler to analyze the references to the array to determine what bounds to use. If the lower or upper bounds are missing, the declared or allocated bounds, if known, are used.

private clause restrictions

The `private` clause has these restrictions:

- A variable, array or subarray may only appear once in any `private` clause for a loop.
- Only one subarray for an array may appear in any `private` clause for a loop.
- If a subarray appears in a `private` clause, then the compiler only needs to allocate memory to hold that subarray in the accelerator memory.
- The compiler may pad dimensions of allocated arrays or subarrays to improve memory alignment and program performance.
- If a subarray appears in a `private` clause, it is an error to refer to any element of the array in the loop outside the bounds of the subarray.
- It is an error to refer to a variable or any element of an array or subarray that appears in a `private` clause and that has not been assigned in this iteration of the loop.
- In Fortran, the upper bound for the last dimension of an assumed-size dummy array must be specified.
- In C, a missing lower bound is assumed to be zero. A missing upper bound for a dynamically allocated array must be specified.

seq [(width)]

The `seq` clause tells the compiler to execute this loop sequentially on the accelerator. There is no maximum number of iterations for a seq schedule. If the `seq` clause has a *width* argument, the compiler strip mines the loop and determines an appropriate schedule for the remaining loop.

unroll [(width)]

The `unroll` clause tells the compiler to unroll *width* iterations for sequential execution on the accelerator. The *width* argument must be a compile time positive constant integer.

unroll clause restrictions

The `unroll` clause has these restrictions:

- If two or more loop scheduling clauses appear on a single loop mapping directive, all but one must have a *width* argument.
- Some implementations or targets may require the *width* expression for the vector clause to be a compile-time constant.
- Some implementations or targets may require the *width* expression for the vector or parallel clauses to be a power of two, or a multiple of some power of two. If this is the case, the behavior when the restriction is violated is implementation-defined.

vector [(width)]

The `vector` clause tells the compiler to execute this loop in vector mode on the accelerator. There may be a target-specific limit on the number of iterations in a vector loop, the aggregate number of iterations in all vector loops, or the number of vector loops allowed in a kernel.

When there is a limit:

- If there is no *width* argument, or the value of the *width* argument is greater than the limit, the compiler strip mines the loop to enforce the limit.

Declarative Data Directive Clauses

The clauses for a declarative data directive are one of the following:

```
copy( list )
copyout( list )
copyin( list )
local( list )
mirror( list )
reflected( list )
```

All of these clauses, except the `reflected(list)` clause are the same as the clauses defined for the accelerator region directive.

reflected (list)

You use the `reflected` clause to declare that the actual argument arrays that are bound to the dummy argument arrays in the *list* need to have a visible copy at the call site.

- This clause is only valid in a Fortran subroutine or function.
- The *list* argument to the reflected clause is a comma-separated list of dummy argument array names. The arrays may be explicit shape, assumed shape, or allocatable.
- If the reflected declarative clause is used, the caller must have an explicit interface to this subprogram.
- If a Fortran interface block is used to describe the explicit interface, a matching reflected directive must appear in the interface block.
- The device copy of the array used within the subroutine or function is the device copy that is visible at the call site.

Update Directive Clauses

The clauses for an accelerator update directive are one of the following:

```
device( list )
host( list )
```

The *list* argument to each update clause is a comma-separated collection of variable names, array names, or subarray specifications. All variables or arrays that appear in the *list* argument of an update clause must have a visible device copy outside the compute or data region.

Multiple subarrays of the same array may appear in update clauses for the same region, potentially causing updates of different subarrays in each direction.

device (list)

The device clause for the update directive copies the variables, arrays, or subarrays in the *list* argument from host memory to the visible device copy of the variables, arrays, or subarrays in device memory. The copy occurs before beginning execution of the compute or data region.

This clause has the same function as the updatein clause for an acclerator compute region directive.

host (list)

The host clause for the update directive copies the visible device copies of the variables, arrays, or subarrays in the *list* argument to the associated host memory locations. The copy occurs after completion of the compute or data region.

This clause has the same function as the updateout clause for an acclerator compute region directive.

PGI Accelerator Runtime Routines

This section defines specific details related to user-callable functions and library routines that are available for use by programmers to query the accelerator features and to control behavior of accelerator-enabled programs at runtime.

To access these routines, add the following to your program:

For a Fortran program:

```
use accel_lib
```

For a C program:

```
#include “accel.h”
```

acc_allocs

The `acc_allocs` routine returns the number of arrays allocated in data or compute regions.

Syntax

In C, the syntax is this:

```
unsigned int acc_allocs();
```

In Fortran, the syntax is this:

```
integer function acc_allocs()
```

Description

A call to `acc_allocs` returns the number of arrays allocated on the accelerator by data or compute regions since the start of the program.

acc_bytesalloc

The `acc_bytesalloc` routine returns the total bytes allocated by data or compute regions.

Syntax

In C, the syntax is this:

```
unsigned long acc_bytesalloc();
```

In Fortran, the syntax is this:

```
integer(8) function acc_bytesalloc()
```

Description

A call to `acc_bytesalloc` returns the total bytes allocated by data or compute regions since the start of the program.

Important

This is not the bytes currently allocated on the device.

acc_bytesin

The `acc_bytesin` routine returns the total bytes copied in to the accelerator by data or compute regions.

Syntax

In C, the syntax is this:

```
unsigned long acc_bytesin();
```

In Fortran, the syntax is this:

```
integer(8) function acc_bytesin()
```

Description

A call to `acc_bytesin` returns the total bytes copied in to the accelerator by data or compute regions since the start of the program. This total includes data implicitly copied through compiler analysis as well as copies directed by the user through the use of `copy`, `copyin`, or `updatein` clauses, or update device directives.

acc_bytesout

The `acc_bytesout` routine returns the total bytes copied out from the accelerator by data or compute regions.

Syntax

In C, the syntax is this:

```
unsigned long acc_bytesout();
```

In Fortran, the syntax is this:

```
integer(8) function acc_bytesout()
```

Description

A call to `acc_bytesin` returns the total bytes copied out from the accelerator by data or compute regions since the start of the program. This total includes data implicitly copied through compiler analysis as well as copies directed by the user through the use of `copy`, `copyin`, or `updatein` clauses, or update host directives.

acc_copyins

The `acc_copyins` routine returns the number of arrays copied in to the accelerator by data or compute regions.

Syntax

In C, the syntax is this:

```
unsigned int acc_copyins();
```

In Fortran, the syntax is this:

```
integer function acc_copyins()
```

Description

A call to `acc_copyins` returns the number of arrays copied in to the accelerator by data or compute regions since the start of the program. This number is the count of arrays, not the number of DMA copies from the host memory to the device memory. The number includes arrays implicitly copied through compiler analysis as well as copies directed by the user through the use of `copy`, `copyin`, or `updatein` clauses, or update device directives.

acc_copyouts

The `acc_copyouts` routine returns the number of arrays copied out from the accelerator by data or compute regions.

Syntax

In C, the syntax is this:

```
unsigned int acc_copyouts();
```

In Fortran, the syntax is this:

```
integer function acc_copyouts()
```

Description

A call to `acc_copyouts` returns the number of arrays copied out from the accelerator by data or compute regions since the start of the program. This number is the count of arrays, not the number of DMA copies from the device memory to the host memory. The number includes arrays implicitly copied through compiler analysis as well as copies directed by the user through the use of `copy`, `copyin`, or `updatein` clauses, or update host directives.

acc_disable_time

The `acc_disable_time` routine tells the runtime to stop profiling accelerator regions and kernels.

Syntax

In C, the syntax is this:

```
void acc_disable_time( acc_device_t );
```

In Fortran, the syntax is this:

```
subroutine acc_disable_time( devicetype )
integer(acc_device_kind) devicetype
```

Description

The `acc_disable_time` routine decrements the profile counter in the runtime. If the profile counter reaches zero, accelerator region and kernel profiling is disabled.

This routine is only implements when the argument has the value `acc_device_nvidia`. If the program is linked with `-ta=nvidia,time`, an implicit call to `acc_enable_time(acc_device_nvidia)` is made at program startup. A subsequent call to `acc_disable_time(acc_device_nvidia` disables that profiling.

acc_enable_time

The `acc_enable_time` routine tells the runtime to start profiling accelerator regions and kernels, if it is not already doing so.

Syntax

In C, the syntax is this:

```
void acc_enable_time( acc_device_t );
```

In Fortran, the syntax is this:

```
subroutine acc_enable_time( devicetype )
integer(acc_device_kind) devicetype
```

Description

The `acc_enable_time` routine increments the profile counter in the runtime. If the profile counter is greater than zero, accelerator region and kernel profiling is enabled.

This routine is only implements when the argument has the value `acc_device_nvidia`. If the program is linked with `-ta=nvidia,time`, an implicit call to `acc_enable_time(acc_device_nvidia)` is made at program startup.

acc_exec_time

The `acc_exec_time` routine returns the number of microseconds spent on the accelerator executing kernels.

Syntax

In C, the syntax is this:

```
long acc_exec_time( acc_device_t );
```

In Fortran, the syntax is this:

```
integer(8) function acc_exec_time( devicetyoe()
integer(acc_device_kind) devicetype
```

Description

A call to `acc_exec_time` returns the number of microseconds spent since the program started on the accelerator executing kernels while accelerator profiling was active. This routine is only implemented when the argument has the value `acc_device_nvidia`.

acc_free

The `acc_free` routine frees the memory on the attached accelerator.

Syntax

Available only in C, the syntax is this:

```
void acc_free(void*);
```

Description

A call to `acc_free` frees the memory on the attached accelerator, essentially calling `cudaFree()` on NVIDIA devices. If no device is attached, `acc_free` assumes the pointer is for host memory and calls free().

The pointer may have been allocated with `acc_malloc` or with `cudaMalloc`.

acc_frees

The `acc_frees` routine returns the number of arrays freed or deallocated in data or compute regions.

Syntax

In C, the syntax is this:

```
unsigned int acc_frees();
```

In Fortran, the syntax is this:

```
integer function acc_frees()
```

Description

A call to `acc_frees` returns the number of arrays freed on the accelerator at the end of data or compute regions since the start of the program. Outside of all accelerator regions, the value returned by `acc_frees` should equal the value returned by `acc_allocs`.

acc_get_device

The `acc_get_device` routine returns the type of accelerator device being used.

Syntax

In C, the syntax is this:

```
int acc_get_device(void);
```

In Fortran, the syntax is this:

```
integer function acc_get_device()
```

Description

The `acc_get_device` routine returns the type of accelerator device to use when executing an accelerator compute region. Its return value is one of the predefined values in the C include file `accel.h`, the Fortran include file `accel_lib.h` or the Fortran module `accel_lib`.

This routine is useful when a program is compiled to use more than one type of accelerator.

Restrictions

The `acc_get_device` routine has the following restrictions:

- The routine may not be called during execution of an accelerator compute or data region.
- If the device type has not yet been selected, the value `acc_device_none` is returned.

acc_get_device_num

The `acc_get_device_num` routine returns the number of the device being used to execute an accelerator region.

Syntax

In C, the syntax is this:

```
int acc_get_device_num(acc_device_t);
```

In Fortran, the syntax is this:

```
integer function acc_get_device_num(devicetype)
integer(acc_device_kind) devicetype
```

Description

The `acc_get_device_num` routine returns the number of the device being used to execute an accelerator region.

Restrictions

The `acc_get_device_num` routine has the following restrictions:

- The routine may not be called during execution of an accelerator compute or data region.
- If the device has not yet been selected, the value `-1` is returned.
- The argument must have the value `acc_device_nvidia`.

acc_get_free_memory

The `acc_get_free_memory` routine returns the total available memory on the attached accelerator.

Syntax

In C, the syntax is this:

```
unsigned long acc_get_free_memory();
```

In Fortran, the syntax is this:

```
integer(8) function acc_get_free_memory()
```

Description

A call to `acc_get_free_memory` returns the total memory on the attached accelerator. This rountine must have been preceded by a call to `acc_init`, or by an accelerator region that initializes the device.

acc_get_memory

The `acc_get_memory` routine returns the total memory on the attached accelerator.

Syntax

In C, the syntax is this:

```
unsigned long acc_get_memory();
```

In Fortran, the syntax is this:

```
integer(8) function acc_get_free_memory()
```

Description

A call to `acc_get_free_memory` returns the total memory on the attached accelerator. This rountine must have been preceded by a call to `acc_init`, or by an accelerator region that initializes the device.

acc_get_num_devices

The `acc_get_num_devices` routine returns the number of accelerator devices of the given type attached to the host.

Syntax

In C, the syntax is this:

```
int acc_get_num_devices(acc_device_t);
```

In Fortran, the syntax is this:

```
 integer function acc_get_num_devices(devicetype)
 integer(acc_device_kind) devicetype
```

Description

The `acc_get_num_devices` routine returns the number of accelerator devices of the given type attached to the host. The *devicetype* argument determines what kind of device to count. The possible values for *devicetype* are implementation-specific, and are listed in the C include file `accel.h`, the Fortran include file `accel_lib.h` and the Fortran module `accel_lib`.

acc_init

The `acc_init` routine connects to and initializes the accelerator device and allocates the control structures in the accelerator library.

Syntax

In C, the syntax is this:

```
      void acc_init(acc_device_t);
```

In Fortran, the syntax is this:

```
      subroutine acc_init( devicetype )
        integer(acc_device_kind) devicetype
```

Description

The `acc_init` routine connects to and initializes the accelerator device and allocates the control structures in the accelerator library.

Restrictions

The `acc_init` routine has the following restrictions:

- The `acc_init` routine must be called before entering any accelerator regions or after an `acc_shutdown` call.
- The argument must be one of the predefined values in the C include file `accel.h`, the Fortran include file `accel_lib.h` or the Fortran module `accel_lib`.
- The routine may not be called during execution of an accelerator region.
- If the device type specified is not available, the behavior is implementation-defined; in particular, the program may abort.
- If the routine is called more than once with a different value for the device type argument and without an intervening `acc_shutdown` call, the behavior is implementation-defined.
- If some accelerator regions are compiled to only use one device type, calling this routine with a different device type may produce undefined behavior.

acc_kernels

The `acc_kernels` routine returns the number of accelerator kernels launched since the start of the program.

Syntax

In C, the syntax is this:

```
unsigned int acc_kernels();
```

In Fortran, the syntax is this:

```
integer function acc_kernels()
```

Description

A call to `acc_kernels` returns the number of accelerator kernels launched since the start of the program.

acc_malloc

The `acc_malloc` routine allocates memory on the attached accelerator.

Syntax

Available only in C, the syntax is this:

```
void* acc_malloc( size_t );
```

Description

A call to `acc_malloc` allocates the specified number of bytes on the attached device, essentially calling `cudaMalloc` on NVIDIA devices.

- If no device is attached, `acc_malloc` calls `malloc()`, returning a host pointer.
- If the allocation fails, `acc_malloc` returns `NULL`.

Note

The program must initialize the device before calling `acc_malloc`. Initialization occurs either by calling `acc_init()` or by entering an accelerator region.

acc_on_device

The `acc_on_device` routine tells the program whether it is executing on a particular device.

Syntax

In C, the syntax is this:

```
int acc_on_device (acc_device_t);
```

In Fortran, the syntax is this:

```
logical function acc_on_device( devicetype )
  integer(acc_device_kind) devicetype
```

Description

The `acc_on_device` routine may be used to execute different paths depending on whether the code is running on the host or on some accelerator.

- If the `acc_on_device` routine has a compile-time constant argument, it evaluates at compile time to a constant. The argument must be one of the defined accelerator types.
- If `acc_on_device` has the argument `acc_device_host`, then outside of an accelerator compute region, or in an accelerator compute region that is compiled for the host processor, this routine evaluates to nonzero for C, and `.true.` for Fortran; otherwise, it evaluates to zero for C and `.false.` for Fortran.

acc_regions

The `acc_regions` routine returns the number of accelerator data and compure regions entered since the start of the program.

Syntax

In C, the syntax is this:

```
unsigned int acc_regions();
```

In Fortran, the syntax is this:

```
integer function acc_regions()
```

Description

A call to `acc_regions` returns the number of accelerator data and compure regions entered since the start of the program.

acc_set_device

The `acc_set_device` routine specifies which type of device the runtime uses when executing an accelerator compute region.

Syntax

In C, the syntax is this:

```
void acc_set_device( acc_device_t );
```

In Fortran, the syntax is this:

```
subroutine acc_set_device( devicetype )
integer(acc_device_kind) devicetype
```

Description

The `acc_set_device` routine specifies which type of device the runtime uses when executing an accelerator compute region. This is useful when the program has been compiled to use more than one type of accelerator.

Restrictions

The `acc_set_device` routine has the following restrictions:

- The routine may not be called during execution of an accelerator compute or data region.
- If the device type specified is not available, the behavior is implementation-defined; in particular, the program may abort.
- If the routine is called more than once with a different value for the device type argument and without an intervening `acc_shutdown` call, the behavior is implementation-defined.
- If some accelerator regions are compiled to only use one device type, calling this routine with a different device type may produce undefined behavior.

acc_set_device_num

The `acc_set_device_num` routine tells the runtime which device to use when executing an accelerator region.

Syntax

In C, the syntax is this:

```
int acc_set_device_num(int, acc_device_t);
```

In Fortran, the syntax is this:

```
subroutine acc_set_device_num( devicenum, devicetype )
   integer devicenum
  integer(acc_device_kind) devicetype
```

Description

The `acc_set_device_num` routine tells the runtime which device to use among those attached of the given type.

- If the value of devicenum is zero, the runtime reverts to its default behavior, which is implementation-defined.
- If the value of devicenum is greater than the value returned by `acc_get_num_devices` for that device type, the behavior is implementation-defined.
- If the value of the second argument is zero, the selected device number is used for all attached accelerator types.
- Calling `acc_set_device_num` implies a call to `acc_set_device` with the devicetype specified by this routine.

Restrictions

The `acc_set_device_num` routine has the following restrictions:

- The routine may not be called during execution of an accelerator region.

acc_shutdown

The `acc_shutdown` routine tells the runtime to shutdown the connection to the given accelerator device, and free up any runtime resources.

Syntax

In C, the syntax is this:

```
void acc_shutdown (acc_device_t);
```

In Fortran, the syntax is this:

```
subroutine acc_shutdown( devicetype )
  integer(acc_device_kind) devicetype
```

Description

The `acc_shutdown` routine disconnects the program from the accelerator device, and frees up any runtime resources. If the program is built to run on different device types, you can use this routine to connect to a different device.

Restrictions

The `acc_shutdown` routine has the following restrictions:

- The routine may not be called during execution of an accelerator region.

acc_total_time

The `acc_total_time` routine returns the number of microseconds spent in accelerator compute regions and moving data for accelerator data regions.

Syntax

In C, the syntax is this:

```
long acc_total_time( acc_device_t );
```

In Fortran, the syntax is this:

```
integer(8) function acc_total_time( devicetyoe()
integer(acc_device_kind) devicetype
```

Description

A call to `acc_total_time` returns the number of microseconds spent since the program started in computer regions or in transferring data to and from the accelerator while accelerator profiling was active. This routine is only implemented when the argument has the value `acc_device_nvidia`.

Accelerator Environment Variables

This section describes the environment variables that PGI supports to control behavior of accelerator-enabled programs at execution and to modify the behavior of accelerator regions. The following are TRUE for all these variables:

- The names of the environment variables must be upper case.
- The values assigned environment variables are case insensitive and may have leading and trailing white space.
- The behavior is implementation-defined if the values of the environment variables change after the program has started, even if the program itself modifies the values.

ACC_DEVICE

The ACC_DEVICE environment variable controls the default device type to use when executing accelerator regions, if the program has been compiled to use more than one different type of device. The value of this environment variable is implementation-defined.

When a program is compiled with the PGI Unified Binary, the ACC_DEVICE environment variable controls the default device to use when executing a program. The value of this environment variable must be set to `NVIDIA` or `nvidia`, indicating to run on the NVIDIA GPU. Currently, any other value of the environment variable causes the program to use the host version.

Example

The following example indicates to use the NVIDIA GPU when executing the program:

```
setenv ACC_DEVICE nvidia
export ACC_DEVICE=nvidia
```

ACC_DEVICE_NUM

The ACC_DEVICE_NUM environment variable controls the default device number to use when executing accelerator regions. The value of this environment variable must be a nonnegative integer between zero and the number of devices attached to the host.

- If the value is zero, the implementation-defined default is used.
- If the value is greater than the number of devices attached, the behavior is implementation-defined.

Example

The following example indicates how to set the default device number to use when executing accelerator regions:

```
setenv ACC_DEVICE_NUM 1
export ACC_DEVICE_NUM=1
```

ACC_NOTIFY

The ACC_NOTIFY environment variable, when set to a non-negative integer, indicates to print a short message to the standard output when a kernel is executed on an accelerator. The value of this environment variable must be a nonnegative integer.

- If the value is zero, no message is printed (the default behavior).
- If the value is nonzero, a one-line message is printed whenever an accelerator kernel is executed.

Example

The following example indicates to print a message for each kernel launched on the device:

```
setenv ACC_NOTIFY 1
export ACC_NOTIFY=1
```

pgcudainit Utility

PGI includes a utility program **pgcudainit**. If you run this program in background mode, it holds open a CUDA connection to the device driver, significantly reducing initialization time for subsequent programs.

Chapter 5. C++ Name Mangling

Name mangling transforms the names of entities so that the names include information on aspects of the entity's type and fully qualified name. This ability is necessary since the intermediate language into which a program is translated contains fewer and simpler name spaces than there are in the C++ language; specifically:

- Overloaded function names are not allowed in the intermediate language.
- Classes have their own scopes in C++, but not in the generated intermediate language. For example, an entity x from inside a class must not conflict with an entity x from the file scope.
- External names in the object code form a completely flat name space. The names of entities with external linkage must be projected onto that name space so that they do not conflict with one another. A function f from a class A, for example, must not have the same external name as a function f from class B.
- Some names are not names in the conventional sense of the word, they're not strings of alphanumeric characters, for example: operator=.

There are two main problems here:

1. Generating external names that will not clash.
2. Generating alphanumeric names for entities with strange names in C++.

Name mangling solves these problems by generating external names that will not clash, and alphanumeric names for entities with strange names in C++. It also solves the problem of generating hidden names for some behind-the-scenes language support in such a way that they match up across separate compilations.

You see mangled names if you view files that are translated by PGC++, and you do not use tools that demangle the C++ names. Intermediate files that use mangled names include the assembly and object files created by the pgcpp command. To view demangled names, use the tool pgdecode, which takes input from stdin.

```
prompt> pgdecode
g__1ASFf
A::g(float)
```

The name mangling algorithm for the PGC++ compiler is the same as that for cfront, and, except for a few minor details, also matches the description in Section 8.0, Function Name Encoding, of The Annotated C++ Reference Manual (ARM). Refer to the ARM for a complete description of name mangling.

Types of Mangling

The following entity names are mangled:

- Function names including non-member function names are mangled, to deal with overloading. Names of functions with extern "C" linkage are not mangled.
- Mangled function names have the function name followed by __ followed by F followed by the mangled description of the types of the parameters of the function. If the function is a member function, the mangled form of the class name precedes the F. If the member function is static, an S also precedes the F.

```
int f(float); // f__Ff
class A
 int f(float); // f__1AFf
 static int g(float); // g__1ASFf
;
```

- Special and operator function names, like constructors and operator=(). The encoding is similar to that for normal functions, but a coded name is used instead of the routine name:

```
class A
 int operator+(float); // __pl__1Aff
 A(float); // __ct__1Aff
;
int operator+(A, float); // __pl__F1Af
```

- Static data member names. The mangled form is the member name followed by __ followed by the mangled form of the class name:

```
class A
 static int i; // i__1A
;
```

- Names of variables generated for virtual function tables. These have names like `vtblmangled-class-name` or `vtblmangled-base-class-namemangled-class-name`.
- Names of variables generated to contain runtime type information. These have names like `Ttype-encoding` and `TIDtype-encoding`.

Mangling Summary

This section lists some of the C++ entities that are mangled and provides some details on the mangling algorithm. For more details, refer to The Annotated C++ Reference Manual.

Type Name Mangling

Using PGC++, each type has a corresponding mangled encoding. For example, a class type is represented as the class name preceded by the number of characters in the class name, as in `5abcde` for `abcde`. Simple types are encoded as lower-case letters, as in i for int or f for float. Type modifiers and declarators are encoded as upper-case letters preceding the types they modify, as in U for unsigned or P for pointer.

Nested Class Name Mangling

Nested class types are encoded as a Q followed by a digit indicating the depth of nesting, followed by a _, followed by the mangled-form names of the class types in the fully-qualified name of the class, from outermost to innermost:

```
class A
 class B // Q2_1A1B
 ;
;
```

Local Class Name Mangling

The name of the nested class itself is mangled to the form described previously with a prefix ___, which serves to make the class name distinct from all user names. Local class names are encoded as L followed by a number, followed by ___, followed by the mangled name of the class. The number has no special meaning; it's just an identifying number assigned to the class. The name of the class is not in the ARM, and cfront encodes local class names slightly differently.

```
void f()
 class A // L1__1A}
 ;
;
```

This form is used when encoding the local class name as a type. It's not necessary to mangle the name of the local class itself unless it's also a nested class.

Template Class Name Mangling

Template classes have mangled names that encode the arguments of the template:

```
template<class T1, class T2> class abc ;
abc<int, int> x;
abc__pt__3_ii
```

This describes two template arguments of type int with the total length of template argument list string, including the underscore, and a fixed string, indicates parameterized type as well, the name of the class template.

Chapter 6. Directives and Pragmas Reference

PGI Fortran compilers support proprietary directives and pragmas. These directives and pragmas override corresponding command-line options. For usage information such as the scope and related command-line options, refer to the PGI Compiler User's Guide.

This chapter contains detailed descriptions of PGI's proprietary directives and pragmas.

PGI Proprietary Fortran Directive and C/C++ Pragma Summary

Directives are Fortran comments and pragmas are C/C++ comments that the user may supply in a source file to provide information to the compiler. These comments alter the effects of certain command line options or default behavior of the compiler. They provide pragmatic information that control the actions of the compiler in a particular portion of a program without affecting the program as a whole. That is, while a command line option affects the entire source file that is being compiled, directives and pragmas apply, or disable, the effects of a command line option to selected subprograms or to selected loops in the source file, for example, to optimize a specific area of code. Use directives and pragmas to tune selected routines or loops.

The Fortran directives may have any of the following forms:

```
!pgi$g directive
!pgi$r directive
!pgi$l directive
!pgi$ directive
```

where the scope indicator follows the $ and is either g (global), r (routine), or l (loop). This indicator controls the scope of the directive, though some directives ignore the scope indicator.

Note

If the input is in fixed format, the comment character, !, * or C, must begin in column 1.

Directives and pragmas override corresponding command-line options. For usage information such as the scope and related command-line options, refer to the "Using Directives and Pragmas in the PGI Compiler User's Guide.

altcode (noaltcode)

The `altcode` directive or pragma instructs the compiler to generate alternate code for vectorized or parallelized loops.

The `noaltcode` directive or pragma disables generation of alternate code.

Scope: This directive or pragma affects the compiler only when `-Mvect=sse` or `-Mconcur` is enabled on the command line.

cpgi$ altcode
: Enables alternate code (altcode) generation for vectorized loops. For each loop the compiler decides whether to generate altcode and what type(s) to generate, which may be any or all of: altcode without iteration peeling, altcode with non-temporal stores and other data cache optimizations, and altcode based on array alignments calculated dynamically at runtime. The compiler also determines suitable loop count and array alignment conditions for executing the alternate code.

cpgi$ altcode alignment
: For a vectorized loop, if possible, generates an alternate vectorized loop containing additional aligned moves which is executed if a runtime array alignment test is passed.

cpgi$ altcode [(n)] concur
: For each auto-parallelized loop, generates an alternate serial loop to be executed if the loop count is less than or equal to n. If n is omitted or n is 0, the compiler determines a suitable value of n for each loop.

cpgi$ altcode [(n)] concurreduction
: Sets the loop count threshold for parallelization of reduction loops to n. For each auto-parallelized reduction loop, generate an alternate serial loop to be executed if the loop count is less than or equal to n. If n is omitted or n is 0, the compiler determines a suitable value of n for each loop.

cpgi$ altcode [(n)] nontemporal
: For a vectorized loop, if possible, generates an alternate vectorized loop containing non-temporal stores and other cache optimizations to be executed if the loop count is greater than n. If n is omitted or n is 1, the compiler determines a suitable value of n for each loop. The alternate code is optimized for the case when the data referenced in the loop does not all fit in level 2 cache.

cpgi$ altcode [(n)] nopeel
: For a vectorized loop where iteration peeling is performed by default, if possible, generates an alternate vectorized loop without iteration peeling to be executed if the loop count is less than or equal to n. If n is omitted or n is 1, the compiler determines a suitable value of n for each loop, and in some cases it may decide not to generate an alternate unpeeled loop.

cpgi$ altcode [(n)] vector
: For each vectorized loop, generates an alternate scalar loop to be executed if the loop count is less than or equal to n. If n is omitted or n is 1, the compiler determines a suitable value of n for each loop.

cpgi$ noaltcode
: Sets the loop count thresholds for parallelization of all innermost loops to 0, and disables alternate code generation for vectorized loops.

assoc (noassoc)

This directive or pragma toggles the effects of the `-Mvect=noassoc` command-line option, an optimization `-M` control.

Scope: This directive or pragma affects the compiler only when `-Mvect=sse` is enabled on the command line.

By default, when scalar reductions are present the vectorizer may change the order of operations, such as dot product, so that it can generate better code. Such transformations may change the result of the computation due to roundoff error. The noassoc directive disables these transformations.

bounds (nobounds)

This directive or pragma alters the effects of the `-Mbounds` command line option. This directive enables the checking of array bounds when subscripted array references are performed. By default, array bounds checking is not performed.

cncall (nocncall)

This directive or pragma indicates that loops within the specified scope are considered for parallelization, even if they contain calls to user-defined subroutines or functions. A nocncall directive cancels the effect of a previous cncall.

concur (noconcur)

This directive or pragma alters the effects of the `-Mconcur` command-line option. The directive instructs the auto-parallelizer to enable auto-concurrentization of loops.

Scope: This directive or pragma affects the compiler only when `-Mconcur` is enabled on the command line.

If concur is specified, the compiler uses multiple processors to execute loops which the auto-parallelizer determines to be parallelizable. The noconcur directive disables these transformations; however, use of concur overrides previous noconcur statements.

depchk (nodepchk)

This directive or pragma alters the effects of the `-Mdepchk` command line option. When potential data dependencies exist, the compiler, by default, assumes that there is a data dependence that in turn may inhibit certain optimizations or vectorizations. nodepchk directs the compiler to ignore unknown data dependencies.

eqvchk (noeqvchk)

The eqvchk directive or pragma specifies to check dependencies between EQUIVALENCE associated elements. When examining data dependencies, noeqvchk directs the compiler to ignore any dependencies between variables appearing in `EQUIVALENCE` statements.

fcon (nofcon)

This C/C++ pragma alters the effects of the `-Mfcon` (a –M Language control) command-line option.

The pragma instructs the compiler to treat non-suffixed floating-point constants as float rather than double. By default, all non-suffixed floating-point constants are treated as double.

Note

Only routine or global scopes are allowed for this C/C++ pragma.

invarif (noinvarif)

This directive or pragma has no corresponding command-line option. Normally, the compiler removes certain invariant if constructs from within a loop and places them outside of the loop. The directive noinvarif directs the compiler not to move such constructs. The directive invarif toggles a previous noinvarif.

ivdep

The ivdep directive assists the compiler's dependence analysis and is equivalent to the directive nodepchk.

lstval (nolstval)

This directive or pragma has no corresponding command-line option. The compiler determines whether the last values for loop iteration control variables and promoted scalars need to be computed. In certain cases, the compiler must assume that the last values of these variables are needed and therefore computes their last values. The directive nolstval directs the compiler not to compute the last values for those cases.

prefetch

The prefetch directive or pragma the compiler emits prefetch instructions whereby elements are fetched into the data cache prior to first use. By varying the prefetch distance, it is sometimes possible to reduce the effects of main memory latency and improve performance.

The syntax of this directive or pragma is:

```
c$mem prefetch <var1>[,<var2>[,...]]
```

where `<varn>` is any valid variable, member, or array element reference.

opt

The opt directive or pragma overrides the value specified by the command line option `-On`.

The syntax of this directive or pragma is:

```
cpgi$<scope> opt=<level>
```

where the optional <scope> is *r* or *g* and <level> is an integer constant representing the optimization level to be used when compiling a subprogram (routine scope) or all subprograms in a file (global scope).

safe (nosafe)

This C/C++ pragma has no corresponding command-line option. By default, the compiler assumes that all pointer arguments are unsafe. That is, the storage located by the pointer can be accessed by other pointers.

The formats of the safe pragma are:

```
#pragma [scope] [no]safe
#pragma safe (variable [, variable]...)
```

where scope is either global or routine.

- When the pragma safe is not followed by a variable name or a list of variable names:
  - If the scope is routine, then the compiler treats all pointer arguments appearing in the routine as safe.
  - If the scope is global, then the compiler treats all pointer arguments appearing in all routines as safe.
- When the pragma safe is followed by a variable name or a list of variable names, each name is the name of a pointer argument in the current function, and the compiler considers that named argument to be safe.

  Note

  If only one variable name is specified, you may omit the surrounding parentheses.

safe_lastval

During parallelization, scalars within loops need to be privatized. Problems are possible if a scalar is accessed outside the loop. If you know that a scalar is assigned on the last iteration of the loop, making it safe to parallelize the loop, you use the safe_lastval directive or pragma to let the compiler know the loop is safe to parallelize.

For example, use the following C pragma to tell the compiler that for a given loop the last value computed for all scalars make it safe to parallelize the loop:

```
cpgi$l safe_lastval
#pragma loop safe_lastval
```

The command-line option `-Msafe_lastval` provides the same information for all loops within the routines being compiled, essentially providing global scope.

In the following example, the value of `t` may not be computed on the last iteration of the loop.

```
do i = 1, N
if( f(x(i)) > 5.0 then)
t = x(i)
endif
enddo
v = t
```

If a scalar assigned within a loop is used outside the loop, we normally save the last value of the scalar. Essentially the value of the scalar on the "last iteration" is saved, in this case when `i=N`.

If the loop is parallelized and the scalar is not assigned on every iteration, it may be difficult to determine on what iteration `t` is last assigned, without resorting to costly critical sections. Analysis allows the compiler to

determine if a scalar is assigned on every iteration, thus the loop is safe to parallelize if the scalar is used later. An example loop is:

```
do i = 1, N
if( x(i) > 0.0 ) then
t = 2.0
else
t = 3.0
endif
y(i) = ...t...
enddo
v = t
```

where `t` is assigned on every iteration of the loop. However, there are cases where a scalar may be privatizable. If it is used after the loop, it is unsafe to parallelize. Examine this loop:

```
do i = 1,N
 if( x(i) > 0.0 ) then
 t = x(i)
 ...
 ...
 y(i) = ...t..
 endif
enddo
v = t
```

where each use of `t` within the loop is reached by a definition from the same iteration. Here `t` is privatizable, but the use of `t` outside the loop may yield incorrect results since the compiler may not be able to detect on which iteration of the parallelized loop `t` is assigned last.

The compiler detects these cases. When a scalar is used after the loop, but is not defined on every iteration of the loop, parallelization does not occur.

safeptr (nosafeptr)

The pragma safeptr directs the compiler to treat pointer variables of the indicated storage class as safe. The pragma nosafeptr directs the compiler to treat pointer variables of the indicated storage class as unsafe. This pragma alters the effects of the –Msafeptr command-line option.

The syntax of this pragma is:

```
cpgi$[] [no]safeptr={arg|local|auto|global|static|all},..
#pragma [scope] [no]safeptr={arg|local|auto|global|static|all},...
```

where scope is global, routine, or loop.

Note

The values local and auto are equivalent.

- all - All pointers are safe
- arg - Argument pointers are safe
- local - local pointers are safe
- global - global pointers are safe

- static - static local pointers are safe

In a file containing multiple functions, the command-line option –Msafeptr might be helpful for one function, but can't be used because another function in the file would produce incorrect results. In such a file, the safeptr pragma, used with routine scope could improve performance and produce correct results.

single (nosingle)

The pragma single directs the compiler not to implicitly convert float values to double non-prototyped functions. This can result in faster code if the program uses only float parameters.

Note

Since ANSI C specifies that floats must be converted to double, this pragma results in non-ANSI conforming code. Valid only for routine or global scope.

tp

You use the directive or pragma tp to specify one or more processor targets for which to generate code.

```
cpgi$ tp [target]...
```

Note

The tp directive or pragma can only be applied at the routine or global level. For more information about these levels, refer to the PGi Compiler User's Guide"Scope of C/C++ Pragmas and Command-Line Options" on page 273.

Refer to the PGI Workstation Release Notes for a list of targets that can be used as parameters to the tp directive.

unroll (nounroll)

The unroll directive or pragma enables loop unrolling while nounroll disables loop unrolling.

Note

The unroll directive or pragma has no effect on vectorized loops.

The directive or pragma takes arguments c and n.

- c specifies that c complete unrolling should be turned on or off.
- n specifies that n (count) unrolling should be turned on or off. In addition, the following arguments may be added to the unroll directive:

In addition, the following arguments may be added to the unroll directive:

c:v sets the threshold to which c unrolling applies. v is a constant; and a loop whose constant loop count isless than or equal to (<=) v is completely unrolled.

```
cpgi$ unroll = c:v
```

n:v adjusts threshold to which n unrolling applies. v is a constant. A loop to which n unrolling applies is unrolled v times.

```
cpgi$ unroll = n:v
```

The directives unroll and nounroll only apply if–Munroll is selected on the command line.

vector (novector)

The directive or pragma novector disables vectorization. The directive or pragma vector re-enables vectorization after a previous novector directive. The directives vector and novector only apply if –Mvect has been selected on the command line.

vintr (novintr)

The directive or pragma novintr directs the vectorizer to disable recognition of vector intrinsics. The directive vintr is re-enables recognition of vector intrinsics after a previous novintr directive. The directives vintr and novintr only apply if –Mvect has been selected on the command line.

Prefetch Directives and Pragmas

Prefetch instructions can increase the speed of an application substantially by bringing data into cache so that it is available when the processor needs it. The PGI prefetch directive takes the form:

The syntax of a prefetch directive is as follows:

```
c$mem prefetch <var1>[,<var2>[,...]]
```

where <var*n*> is any valid variable, member, or array element reference.

The syntax of a prefetch pragma is as follows:

```
#pragma mem prefetch <var1>[,<var2>[,...]]
```

where <var*n*> is any valid variable, member, or array element reference.

For examples on how to use the prefetch directive or pragma, refer to the PGI Compiler USer's Guide.

!DEC$ Directives

PGI Fortran compilers for Microsoft Windows support directives that help with inter-language calling and importing and exporting routines to and from DLLs. These directives all take the form:

```
!DEC$ directive
```

For specific format requirements, refer to the section "!DEC$ Directives" in the PGI Compiler User's Guide.

ALIAS Directive

This directive specifies an alternative name with which to resolve a routine.

The syntax for the ALIAS directive is either of the following:

```
!DEC$ ALIAS routine_name , external_name
!DEC$ ALIAS routine_name : external_name
```

In this syntax, `external_name` is used as the external name for the specified `routine_name`.

If `external_name` is an identifier name, the name (in uppercase) is used as the external name for the specified `routine_name`. If `external_name` is a character constant, it is used as-is; the string is not changed to uppercase, nor are blanks removed.

You can also supply an alias for a routine using the ATTRIBUTES directive, described in the next section:

```
!DEC$ ATTIRIBUTES ALIAS : 'alias_name' :: routine_name
```

This directive specifies an alternative name with which to resolve a routine, as illustrated in the following code fragment that provides external names for three routines. In this fragment, the external name for `sub1` is `name1`, for `sub2` is `name2`, and for `sub3` is `name3`.

```
 subroutine sub
!DEC$ alias sub1 , 'name1'
!DEC$ alias sub2 : 'name2'
!DEC$ attributes alias : 'name3' :: sub3
```

ATTRIBUTES Directive

This directive lets you specify properties for data objects and procedures.

The syntax for the ATTRIBUTES directive is this:

```
!DEC$ ATTRIBUTES <list>
```

where <list> is one of the following:

ALIAS : 'alias_name' :: routine_name
: Specifies an alternative name with which to resolve `routine_name`.

C :: routine_name
: Specifies that the routine `routine_name` will have its arguments passed by value. When a routine marked C is called, arguments, except arrays, are sent by value. For characters, only the first character is passed. The standard Fortran calling convention is pass by reference.

DLLEXPORT :: name
: Specifies that `name` is being exported from a DLL.

DLLIMPORT :: name
: Specifies that `name` is being imported from a DLL.

NOMIXED_STR_LEN_ARG
: Specifies that hidden lengths are placed in sequential order at the end of the list, like `-Miface=unix`.

> **Note**
>
> This attribute only applies to routines that are CREF-style or that use the default Windows calling conventions.

REFERENCE :: name
: Specifies that the argument `name` is being passed by reference. Often this attribute is used in conjunction with `STDCALL`, where `STDCALL` refers to an entire routine; then individual arguments are modified with `REFERENCE`.

STDCALL :: routine_name
: Specifies that routine `routine_name` will have its arguments passed by value. When a routine marked `STDCALL` is called, arguments (except arrays and characters) will be sent by value. The standard Fortran calling convention is pass by reference.

VALUE :: name
: Specifies that the argument 'name' is being passed by value.

DECORATE Directive

The DECORATE directive specifies that the name specified in the ALIAS directive should have the prefix and postfix decorations performed on it that are associated with the calling conventions that are in effect. These declarations are the same ones performed on the name when ALIAS is not specified.

The syntax for the DECORATE directive is this:

```
!DEC$ DECORATE
```

Note

When ALIAS is not specified, this directive has no effect.

DISTRIBUTE Directive

This directive is front-end based, and tells the compiler at what point within a loop to split into two loops.

The syntax for the DISTRIBUTE directive is either of the following:

```
!DEC$ DISTRIBUTE POINT
!DEC$ DISTRIBUTEPOINT
```

Example:

```
 subroutine dist(a,b,n)
 integer i
 integer n
 integer a(*)
 integer b(*)
 do i = 1,n
 a(i) = a(i)+2
!DEC$ DISTRIBUTE POINT
 b(i) = b(i)*4
 enddo
 end subroutine
```

IGNORE_TKR Directive

This directive indicates to the compiler to ignore the type, kind, and/or rank of the specified dummy arguments in an interface of a procedure. The compiler also ignores the type, kind, and/or rank of the actual arguments when checking all the specifics in a generic call for ambiguities.

The syntax for the IGNORE_TKR directive is this:

```
!DIR$ IGNORE_TKR [ [(<letter>) <dummy_arg>] ... ]
```

<letter>
is one or any combination of the following:

T - type **K** - kind **R** - rank

For example, KR indicates to ignore both kind and rank rules and TKR indicates to ignore the type, kind, and rank arguments.

<dummy_arg>
if specified, indicates the dummy argument for which TKR rules should be ignored. If not specified, TKR rules are ignored for all dummy arguments in the procedure that contains the directive.

Rules

The following rules apply to this directive:

- IGNORE_TKR must not specify dummy arguments that are allocatable, Fortran 90 pointers, or assumed-shape arrays.
- IGNORE_TKR may only appear in the body of an interface block and may specify dummy argument names only.
- IGNORE_TKR may appear before or after the declarations of the dummy arguments it specifies.
- If dummy argument names are specified, IGNORE_TKR applies only to those particular dummy arguments.
- If no dummy argument names are specified, IGNORE_TKR applies to all dummy arguments except those that are allocatable objects, Fortran 90 pointers, or assumed-shape arrays.

Example:

Consider this subroutine fragment:

```
subroutine example(A,B,C,D)
!DIR$ IGNORE_TKR A, (R) B, (TK) C, (K) D
```

Table 6.1 indicates which rules are ignored for which dummy arguments in the sample subroutine fragment:

Table 6.1. IGNORE_TKR Example

Dummy Argument	Ignored Rules
A	Type, Kind and Rank
B	Only rank
C	Type and Kind
D	Only Kind

Notice that no letters were specified for A, so all type, kind, and rank rules are ignored.

Chapter 7. Run-time Environment

This chapter describes the programming model supported for compiler code generation, including register conventions and calling conventions for x86 and x64 processor-based systems. It addresses these conventions for processors running linux86 or Win32 operating systems, for processors running linux86-64 operating systems, and for processors running Win64 operating systems.

Note

In this chapter we sometimes refer to word, halfword, and double word. The equivalent byte information is word (4 byte), halfword (2 byte), and double word (8 byte).

Linux86 and Win32 Programming Model

This section defines compiler and assembly language conventions for the use of certain aspects of an x86 processor running a linux86 or Win32 operating system. These standards must be followed to guarantee that compilers, application programs, and operating systems written by different people and organizations will work together. The conventions supported by the PGCC ANSI C compiler implement the application binary interface (ABI) as defined in the System V Application Binary Interface: Intel Processor Supplement and the System V Application Binary Interface, listed in the "Related Publications" section in the Preface.

Function Calling Sequence

This section describes the standard function calling sequence, including the stack frame, register usage, and parameter passing.

Register Usage Conventions

The following table defines the standard for register allocation. The 32-bit x86 Architecture provides a number of registers. All the integer registers and all the floating-point registers are global to all procedures in a running program.

Table 7.1. Register Allocation

Type	Name	Purpose
General	%eax	integer return value
	%edx	dividend register (for divide operations)
	%ecx	count register (shift and string operations)
	%ebx	local register variable
	%ebp	optional stack frame pointer
	%esi	local register variable
	%edi	local register variable
	%esp	stack pointer
Floating-point	%st(0)	floating-point stack top, return value
	%st(1)	floating-point next to stack top
	%st(...)	
	%st(7)	floating-point stack bottom

In addition to the registers, each function has a frame on the run-time stack. This stack grows downward from high addresses. The next table shows the stack frame organization.

Table 7.2. Standard Stack Frame

Position	Contents	Frame
4n+8 (%ebp)	argument word n	previous
	argument words 1 to n-1	
8 (%ebp)	argument word 0	
4 (%ebp)	return address	
0 (%ebp)	caller's %ebp	current
-4 (%ebp)	n bytes of local	
-n (%ebp)	variables and temps	

Key points concerning the stack frame include:

- The stack is kept double word aligned.
- Argument words are pushed onto the stack in reverse order so the rightmost argument in C call syntax has the highest address. A dummy word may be pushed ahead of the rightmost argument in order to preserve doubleword alignment. All incoming arguments appear on the stack, residing in the stack frame of the caller.
- An argument's size is increased, if necessary, to make it a multiple of words. This may require tail padding, depending on the size of the argument.

All registers on an x86 system are global and thus visible to both a calling and a called function. Registers %ebp, %ebx, %edi, %esi, and %esp are non-volatile across function calls. Therefore, a function must preserve these registers' values for its caller. Remaining registers are volatile (scratch). If a calling function wants to preserve such a register value across a function call, it must save its value explicitly.

Some registers have assigned roles in the standard calling sequence:

%esp
: The stack pointer holds the limit of the current stack frame, which is the address of the stack's bottom-most, valid word. At all times, the stack pointer should point to a word-aligned area.

%ebp
: The frame pointer holds a base address for the current stack frame. Consequently, a function has registers pointing to both ends of its frame. Incoming arguments reside in the previous frame, referenced as positive offsets from %ebp, while local variables reside in the current frame, referenced as negative offsets from %ebp. A function must preserve this register value for its caller.

%eax
: Integral and pointer return values appear in %eax. A function that returns a structure or union value places the address of the result in %eax. Otherwise, this is a scratch register.

%esi, %edi
: These local registers have no specified role in the standard calling sequence. Functions must preserve their values for the caller.

%ecx, %edx
: Scratch registers have no specified role in the standard calling sequence. Functions do not have to preserve their values for the caller.

%st(0)
: Floating-point return values appear on the top of the floating point register stack; there is no difference in the representation of single or double-precision values in floating point registers. If the function does not return a floating point value, then the stack must be empty.

%st(1) - %st(7)
: Floating point scratch registers have no specified role in the standard calling sequence. These registers must be empty before entry and upon exit from a function.

EFLAGS
: The flags register contains the system flags, such as the direction flag and the carry flag. The direction flag must be set to the "forward" (i.e., zero) direction before entry and upon exit from a function. Other user flags have no specified role in the standard calling sequence and are not reserved.

Floating Point Control Word
: The control word contains the floating-point flags, such as the rounding mode and exception masking. This register is initialized at process initialization time and its value must be preserved.

Signals can interrupt processes. Functions called during signal handling have no unusual restriction on their use of registers. Moreover, if a signal handling function returns, the process resumes its original execution path with registers restored to their original values. Thus, programs and compilers may freely use all registers without danger of signal handlers changing their values.

Function Return Values

Functions Returning No Value

Functions that return no value are also called procedures or void functions. These functions put no particular value in any register.

Functions Returning Scalars

- A function that returns an integral or pointer value places its result in register %eax.
- A function that returns a long long integer value places its result in the registers %edx and %eax. The most significant word is placed in %edx and the least significant word is placed in %eax.
- A floating-point return value appears on the top of the floating point stack. The caller must then remove the value from the floating point stack, even if it does not use the value. Failure of either side to meet its obligations leads to undefined program behavior. The standard calling sequence does not include any method to detect such failures nor to detect return value type mismatches. Therefore, the user must declare all functions properly. There is no difference in the representation of single-, double- or extended-precision values in floating-point registers.
- A call instruction pushes the address of the next instruction (the return address) onto the stack. The return instruction pops the address off the stack and effectively continues execution at the next instruction after the call instruction. A function that returns a scalar or no value must preserve the caller's registers. Additionally, the called function must remove the return address from the stack, leaving the stack pointer (%esp) with the value it had before the call instruction was executed.

Functions Returning Structures or Unions

If a function returns a structure or union, then the caller provides space for the return value and places its address on the stack as argument word zero. In effect, this address becomes a hidden first argument.

A function that returns a structure or union also sets %eax to the value of the original address of the caller's area before it returns. Thus, when the caller receives control again, the address of the returned object resides in register %eax and can be used to access the object. Both the calling and the called functions must cooperate to pass the return value successfully:

- The calling function must supply space for the return value and pass its address in the stack frame;
- The called function must use the address from the frame and copy the return value to the object so supplied;
- The called function must remove this address from the stack before returning.

Failure of either side to meet its obligation leads to undefined program behavior. The standard function calling sequence does not include any method to detect such failures nor to detect structure and union type mismatches. Therefore, you must declare the function properly.

The following table illustrates the stack contents when the function receives control, after the call instruction, and when the calling function again receives control, after the `ret` instruction.

Table 7.3. Stack Contents for Functions Returning struct/union

Position	After Call	After Return	Position
4n+8 (%esp)	argument word n	argument word n	4n-4 (%esp)
8 (%esp)	argument word 1	argument word 1	0 (%esp)
4 (%esp)	value address	undefined	
0 (%esp)	return address		

The following sections of this chapter describe where arguments appear on the stack. The examples in this chapter are written as if the function prologue is used.

Argument Passing

Integral and Pointer Arguments

As mentioned, a function receives all its arguments through the stack; the last argument is pushed first. In the standard calling sequence, the first argument is at offset 8(%ebp), the second argument is at offset 12(%ebp), as previously shown in Table 7.3, "Stack Contents for Functions Returning struct/union". Functions pass all integer-valued arguments as words, expanding or padding signed or unsigned bytes and halfwords as needed.

Table 7.4. Integral and Pointer Arguments

Call	Argument	Stack Address
g(1, 2, 3, (void *)0);	1	8 (%ebp)
	2	12 (%ebp)
	3	16 (%ebp)
	(void *) 0	20 (%ebp)

Floating-Point Arguments

The stack also holds floating-point arguments: single-precision values use one word and double-precision use two. The following example uses only double-precision arguments.

Table 7.5. Floating-point Arguments

Call	Argument	Stack Address
h(1.414, 1, 2.998e10);	word 0, 1.414	8 (%ebp)
	word 1, 1.414	12 (%ebp)
	1	16 (%ebp)
	word 0 2.998e10	20 (%ebp)
	word 1, 2.998e10	24 (%ebp)

Structure and Union Arguments

Structures and unions can have byte, halfword, or word alignment, depending on the constituents. An argument's size is increased, if necessary, to make it a multiple of words. This size increase may require tail padding, depending on the size of the argument. Structure and union arguments are pushed onto the stack in the same manner as integral arguments. This process provides call-by-value semantics, letting the called function modify its arguments without affecting the calling function's object. In the following example, the argument, s, is a structure consisting of more than 2 words.

Table 7.6. Structure and Union Arguments

Call	Argument	Stack Address
i(1,s);	1	8 (%ebp)
	word 0, s	12 (%ebp)
	word 1, s	16 (%ebp)
	...	...

Implementing a Stack

In general, compilers and programmers must maintain a software stack. Register %esp is the stack pointer. Register %esp is set by the operating system for the application when the program is started. The stack must be a grow-down stack.

A separate frame pointer enables calls to routines that change the stack pointer to allocate space on the stack at run-time (e.g. alloca). Some languages can also return values from a routine allocated on stack space below the original top-of-stack pointer. Such a routine prevents the calling function from using %esp-relative addressing to get at values on the stack. If the compiler does not call routines that leave %esp in an altered state when they return, a frame pointer is not needed and is not used if the compiler option `-Mnoframe` is specified.

Although not required, the stack should be kept aligned on 8-byte boundaries so that 8-byte locals are favorably aligned with respect to performance. PGI's compilers allocate stack space for each routine in multiples of 8 bytes.

Variable Length Parameter Lists

Parameter passing in registers can handle a variable number of parameters. The C language uses a special method to access variable-count parameters. The `stdarg.h` and `varargs.h` files define several functions to access these parameters. A C routine with variable parameters must use the `va_start` macro to set up a data structure before the parameters can be used. The `va_arg` macro must be used to access the successive parameters.

C Parameter Conversion

In C, for a called prototyped function, the parameter type in the called function must match the argument type in the calling function. If the called function is not prototyped, the calling convention uses the types of the arguments but promotes char or short to int, and unsigned char or unsigned short to unsigned int and promotes float to double, unless you use the `-Msingle` option. For more information on the `-Msingle` option, refer to Chapter 2, *"Command-Line Options Reference"*. If the called function is prototyped, the unused bits of a register containing a char or short parameter are undefined and the called function must extend the sign of the unused bits when needed.

Calling Assembly Language Programs

The following example shows a C program calling an assembly-language routine `sum_3`.

Example 7.1. C Program Calling an Assembly-language Routine

```
/* File: testmain.c */
main(){
 long l_para1 = 0x3f800000;
 float f_para2 = 1.0;
 double d_para3 = 0.5;
 float f_return;
 extern float sum_3 (long para1, float para2, double para3);
 f_return = sum_3(l_para1,f_para2, d_para3);
 printf("Parameter one, type long = %08x\n",l_para1);
 printf("Parameter two, type float = %f\n",f_para2);
 printf("Parameter three, type double = %g\n",d_para3);
 printf("The sum after conversion = %f\n",f_return);
}
```

```
# File: sum_3.s
# Computes ( para1 + para2 ) + para3
 .text
 .align 4
 .long .EN1-sum_3+0xc8000000
 .align 16
 .globl sum_3
sum_3:
 pushl %ebp
 movl %esp,%ebp
 subl $8,%esp
..EN1:
 fildl 8(%ebp)
 fadds 12(%ebp)
 faddl 16(%ebp)
 fstps -4(%ebp)
 flds -4(%ebp)
 addl $8,%esp
 leave
 ret
 .type sum_3,@function
 .size sum_3,.-sum_3
```

Linux86-64 Programming Model

This section defines compiler and assembly language conventions for the use of certain aspects of an x64 processor running a linux86-64 operating system. These standards must be followed to guarantee that compilers, application programs, and operating systems written by different people and organizations will work together. The conventions supported by the PGCC ANSI C compiler implement the application binary interface (ABI) as defined in the System V Application Binary Interface: AMD64 Architecture Processor Supplement and the System V Application Binary Interface, listed in the "Related Publications" section in the Preface.

Note

The programming model used for Win64 differs from the Linux86-64 model. For more information, refer to "Win64 Programming Model," on page 194.

Function Calling Sequence

This section describes the standard function calling sequence, including the stack frame, register usage, and parameter passing.

Register Usage Conventions

The following table defines the standard for register allocation. The x64 Architecture provides a variety of registers. All the general purpose registers, XMM registers, and x87 registers are global to all procedures in a running program.

Table 7.7. Register Allocation

Type	Name	Purpose
General	%rax	1st return register
	%rbx	callee-saved; optional base pointer
	%rcx	pass 4th argument to functions
	%rdx	pass 3rd argument to functions; 2nd return register
	%rsp	stack pointer
	%rbp	callee-saved; optional stack frame pointer
	%rsi	pass 2nd argument to functions
	%rdi	pass 1st argument to functions
	%r8	pass 5th argument to functions
	%r9	pass 6th argument to functions
	%r10	temporary register; pass a function's static chain pointer
	%r11	temporary register
	%r12-r15	callee-saved registers

Type	Name	Purpose
XMM	%xmm0-%xmm1	pass and return floating point arguments
	%xmm2-%xmm7	pass floating point arguments
	%xmm8-%xmm15	temporary registers
x87	%st(0)	temporary register; return long double arguments
	%st(1)	temporary register; return long double arguments
	%st(2) - %st(7)	temporary registers

In addition to the registers, each function has a frame on the run-time stack. This stack grows downward from high addresses. Table 7.8 shows the stack frame organization.

Table 7.8. Standard Stack Frame

Position	Contents	Frame
8n+16 (%rbp)	argument eightbyte n	previous
	. . .	
16 (%rbp)	argument eightbyte 0	
8 (%rbp)	return address	current
0 (%rbp)	caller's %rbp	current
-8 (%rbp)	unspecified	
	. . .	
0 (%rsp)	variable size	
-128 (%rsp)	red zone	

Key points concerning the stack frame:

- The end of the input argument area is aligned on a 16-byte boundary.
- The 128-byte area beyond the location of %rsp is called the red zone and can be used for temporary local data storage. This area is not modified by signal or interrupt handlers.
- A call instruction pushes the address of the next instruction (the return address) onto the stack. The return instruction pops the address off the stack and effectively continues execution at the next instruction after the call instruction. A function must preserve non-volatile registers, a register whose contents must be preserved across subroutine calls. Additionally, the called function must remove the return address from the stack, leaving the stack pointer (%rsp) with the value it had before the call instruction was executed.

All registers on an x64 system are global and thus visible to both a calling and a called function. Registers %rbx, %rsp, %rbp, %r12, %r13, %r14, and %r15 are non-volatile across function calls. Therefore, a function must preserve these registers' values for its caller. Remaining registers are volatile (scratch) registers, that is a register whose contents need not be preserved across subroutine calls. If a calling function wants to preserve such a register value across a function call, it must save its value explicitly.

Registers are used extensively in the standard calling sequence. The first six integer and pointer arguments are passed in these registers (listed in order): %rdi, %rsi, %rdx, %rcx, %r8, %r9. The first eight floating point arguments are passed in the first eight XMM registers: %xmm0, %xmm1, …, %xmm7. The registers %rax and %rdx are used to return integer and pointer values. The registers %xmm0 and %xmm1 are used to return floating point values.

Additional registers with assigned roles in the standard calling sequence:

%rsp
: The stack pointer holds the limit of the current stack frame, which is the address of the stack's bottom-most, valid word. The stack must be 16-byte aligned.

%rbp
: The frame pointer holds a base address for the current stack frame. Consequently, a function has registers pointing to both ends of its frame. Incoming arguments reside in the previous frame, referenced as positive offsets from %rbp, while local variables reside in the current frame, referenced as negative offsets from %rbp. A function must preserve this register value for its caller.

RFLAGS
: The flags register contains the system flags, such as the direction flag and the carry flag. The direction flag must be set to the "forward" (i.e., zero) direction before entry and upon exit from a function. Other user flags have no specified role in the standard calling sequence and are not preserved.

Floating Point Control Word
: The control word contains the floating-point flags, such as the rounding mode and exception masking. This register is initialized at process initialization time and its value must be preserved.

Signals can interrupt processes. Functions called during signal handling have no unusual restriction on their use of registers. Moreover, if a signal handling function returns, the process resumes its original execution path with registers restored to their original values. Thus, programs and compilers may freely use all registers without danger of signal handlers changing their values.

Function Return Values

Functions Returning Scalars or No Value

- A function that returns an integral or pointer value places its result in the next available register of the sequence %rax, %rdx.
- A function that returns a floating point value that fits in the XMM registers returns this value in the next available XMM register of the sequence %xmm0, %xmm1.
- An X87 floating-point return value appears on the top of the floating point stack in %st(0) as an 80-bit X87 number. If this X87 return value is a complex number, the real part of the value is returned in %st(0) and the imaginary part in %st(1).
- A function that returns a value in memory also returns the address of this memory in %rax.
- Functions that return no value (also called procedures or void functions) put no particular value in any register.

Functions Returning Structures or Unions

A function can use either registers or memory to return a structure or union. The size and type of the structure or union determine how it is returned. If a structure or union is larger than 16 bytes, it is returned in memory allocated by the caller.

To determine whether a 16-byte or smaller structure or union can be returned in one or more return registers, examine the first eight bytes of the structure or union. The type or types of the structure or union's fields making up these eight bytes determine how these eight bytes will be returned. If the eight bytes contain at least one integral type, the eight bytes will be returned in %rax even if non-integral types are also present in the eight bytes. If the eight bytes only contain floating point types, these eight bytes will be returned in %xmm0.

If the structure or union is larger than eight bytes but smaller than 17 bytes, examine the type or types of the fields making up the second eight bytes of the structure or union. If these eight bytes contain at least one integral type, these eight bytes will be returned in %rdx even if non-integral types are also present in the eight bytes. If the eight bytes only contain floating point types, these eight bytes will be returned in %xmm1.

If a structure or union is returned in memory, the caller provides the space for the return value and passes its address to the function as a "hidden" first argument in %rdi. This address will also be returned in %rax.

Argument Passing

Integral and Pointer Arguments

Integral and pointer arguments are passed to a function using the next available register of the sequence %rdi, %rsi, %rdx, %rcx, %r8, %r9. After this list of registers has been exhausted, all remaining integral and pointer arguments are passed to the function via the stack.

Floating-Point Arguments

Float and double arguments are passed to a function using the next available XMM register taken in the order from %xmm0 to %xmm7. After this list of registers has been exhausted, all remaining float and double arguments are passed to the function via the stack.

Structure and Union Arguments

Structure and union arguments can be passed to a function in either registers or on the stack. The size and type of the structure or union determine how it is passed. If a structure or union is larger than 16 bytes, it is passed to the function in memory.

To determine whether a 16-byte or smaller structure or union can be passed to a function in one or two registers, examine the first eight bytes of the structure or union. The type or types of the structure or union's fields making up these eight bytes determine how these eight bytes will be passed. If the eight bytes contain at least one integral type, the eight bytes will be passed in the first available general purpose register of the sequence %rdi, %rsi, %rdx, %rcx, %r8, %r9 even if non-integral types are also present in the eight bytes. If the eight bytes only contain floating point types, these eight bytes will be passed in the first available XMM register of the sequence from %xmm0 to %xmm7.

If the structure or union is larger than eight bytes but smaller than 17 bytes, examine the type or types of the fields making up the second eight bytes of the structure or union. If the eight bytes contain at least one integral type, the eight bytes will be passed in the next available general purpose register of the sequence %rdi, %rsi,

%rdx, %rcx, %r8, %r9 even if non-integral types are also present in the eight bytes. If these eight bytes only contain floating point types, these eight bytes will be passed in the next available XMM register of the sequence from %xmm0 to %xmm7.

If the first or second eight bytes of the structure or union cannot be passed in a register for some reason, the entire structure or union must be passed in memory.

Passing Arguments on the Stack

If there are arguments left after every argument register has been allocated, the remaining arguments are passed to the function on the stack. The unassigned arguments are pushed on the stack in reverse order, with the last argument pushed first.

Table 7.9, "Register Allocation for Example A-2" shows the register allocation and stack frame offsets for the function declaration and call shown in the following example. Both table and example are adapted from System V Application Binary Interface: AMD64 Architecture Processor Supplement.

Example 7.2. Parameter Passing

```
typedef struct {
 int a, b;
 double d;
} structparam;
structparam s;
int e, f, g, h, i, j, k;
float flt;  double m, n;
extern void func(int e, int f, structparam s, int g, int h,
 float flt, double m, double n, int i, int j, int k);
void func2()
{
 func(e, f, s, g, h, flt, m, n, i, j, k);
}
```

Table 7.9. Register Allocation for Example A-2

General Purpose Registers	Floating Point Registers	Stack Frame Offset
%rdi: e	%xmm0: s.d	0: j
%rsi: f	%xmm1: flt	8: k
%rdx: s.a,s.b	%xmm2: m	
%rcx: g	%xmm3: n	
%r8: h		
%r9: i		

Implementing a Stack

In general, compilers and programmers must maintain a software stack. The stack pointer, register %rsp, is set by the operating system for the application when the program is started. The stack must grow downwards from high addresses.

A separate frame pointer enables calls to routines that change the stack pointer to allocate space on the stack at run-time (e.g. alloca). Some languages can also return values from a routine allocated on stack space below the original top-of-stack pointer. Such a routine prevents the calling function from using %rsp-relative addressing for values on the stack. If the compiler does not call routines that leave %rsp in an altered state when they return, a frame pointer is not needed and may not be used if the compiler option –Mnoframe is specified.

The stack must be kept aligned on 16-byte boundaries.

Variable Length Parameter Lists

Parameter passing in registers can handle a variable number of parameters. The C language uses a special method to access variable-count parameters. The `stdarg.h` and `varargs.h` files define several functions to access these parameters. A C routine with variable parameters must use the va_start macro to set up a data structure before the parameters can be used. The `va_arg` macro must be used to access the successive parameters.

For calls that use `varargs` or `stdargs`, the register %rax acts as a hidden argument whose value is the number of XMM registers used in the call.

C Parameter Conversion

In C, for a called prototyped function, the parameter type in the called function must match the argument type in the calling function. If the called function is not prototyped, the calling convention uses the types of the arguments but promotes char or short to int, and unsigned char or unsigned short to unsigned int and promotes float to double, unless you use the `–Msingle` option. For more information on the `–Msingle` option, refer to Chapter 3.

Calling Assembly Language Programs

The following example shows a C program calling an assembly-language routine `sum_3`.

Example 7.3. C Program Calling an Assembly-language Routine

```
/* File: testmain.c */
#include <stdio.h>
int
main() {
 long l_para1 = 2;
 float f_para2 = 1.0;
 double d_para3 = 0.5;
 float f_return;
 extern float sum_3(long para1, float para2, double para3);
 f_return = sum_3(l_para1, f_para2, d_para3);
 printf("Parameter one, type long = %ld\n", l_para1);
 printf("Parameter two, type float = %f\n", f_para2);
 printf("Parameter three, type double = %f\n", d_para3);
 printf("The sum after conversion = %f\n", f_return);
 return 0;
}
# File: sum_3.s
# Computes ( para1 + para2 ) + para3
 .text
```

```
 .align 16
 .globl sum_3
sum_3:
 pushq %rbp
 movq %rsp, %rbp
 cvtsi2ssq %rdi, %xmm2
 addss %xmm0, %xmm2
 cvtss2sd %xmm2,%xmm2
 addsd %xmm1, %xmm2
 cvtsd2ss %xmm2, %xmm2
 movaps %xmm2, %xmm0
 popq %rbp
 ret
 .type sum_3, @function
 .size sum_3,.-sum_3
```

Linux86-64 Fortran Supplement

Sections A2.4.1 through A2.4.4 of the ABI for x64 Linux and Mac OS X define the Fortran supplement. The register usage conventions set forth in that document remain the same for Fortran.

Fortran Fundamental Types

Table 7.10. Linux86-64 Fortran Fundamental Types

Fortran Type	Size (bytes)	Alignment (bytes)
INTEGER	4	4
INTEGER*1	1	1
INTEGER*2	2	2
INTEGER*4	4	4
INTEGER*8	8	8
LOGICAL	4	4
LOGICAL*1	1	1
LOGICAL*2	2	2
LOGICAL*4	4	4
LOGICAL*8	8	8
BYTE	1	1
CHARACTER*n	n	1
REAL	4	4
REAL*4	4	4
REAL*8	8	8
DOUBLE PRECISION	8	8

Fortran Type	Size (bytes)	Alignment (bytes)
COMPLEX	8	4
COMPLEX*8	8	4
COMPLEX*16	16	8
DOUBLE COMPLEX	16	8

A logical constant is one of:

- .TRUE.
- .FALSE.

The logical constants .TRUE. and .FALSE. are defined to be the four-byte values -1 and 0 respectively. A logical expression is defined to be .TRUE. if its least significant bit is 1 and .FALSE. otherwise.

Note that the value of a character is not automatically NULL-terminated.

Naming Conventions

By default, all globally visible Fortran symbol names (subroutines, functions, common blocks) are converted to lower-case. In addition, an underscore is appended to Fortran global names to distinguish the Fortran name space from the C/C++ name space.

Argument Passing and Return Conventions

Arguments are passed by reference (i.e. the address of the argument is passed, rather than the argument itself). In contrast, C/C++ arguments are passed by value.

When passing an argument declared as Fortran type CHARACTER, an argument representing the length of the CHARACTER argument is also passed to the function. This length argument is a four-byte integer passed by value, and is passed at the end of the parameter list following the other formal arguments. A length argument is passed for each CHARACTER argument; the length arguments are passed in the same order as their respective CHARACTER arguments.

A Fortran function, returning a value of type CHARACTER, adds two arguments to the beginning of its argument list. The first additional argument is the address of the area created by the caller for the return value; the second additional argument is the length of the return value. If a Fortran function is declared to return a character value of constant length, for example CHARACTER*4 FUNCTION CHF(), the second extra parameter representing the length of the return value must still be supplied.

A Fortran complex function returns its value in memory. The caller provides space for the return value and passes the address of this storage as if it were the first argument to the function.

Alternate return specifiers of a Fortran function are not passed as arguments by the caller. The alternate return function passes the appropriate return value back to the caller in %rax.

The handling of the following Fortran 90 features is implementation-defined: internal procedures, pointer arguments, assumed-shape arguments, functions returning arrays, and functions returning derived types.

Inter-language Calling

Inter-language calling between Fortran and C/C++ is possible if function/subroutine parameters and return values match types. If a C/C++ function returns a value, call it from Fortran as a function, otherwise, call it as a subroutine. If a Fortran function has type CHARACTER or COMPLEX, call it from C/C++ as a void function. If a Fortran subroutine has alternate returns, call it from C/C++ as a function returning int; the value of such a subroutine is the value of the integer expression specified in the alternate RETURN statement. If a Fortran subroutine does not contain alternate returns, call it from C/C++ as a void function.

Table 7.11 provides the C/C++ data type corresponding to each Fortran data type.

Table 7.11. Fortran and C/C++ Data Type Compatibility

Fortran Type	C/C++ Type	Size (bytes)
CHARACTER*n x	char x[n]	n
REAL x	float x	4
REAL*4 x	float x	4
REAL*8 x	double x	8
DOUBLE PRECISION x	double x	8
INTEGER x	int x	4
INTEGER*1 x	signed char x	1
INTEGER*2 x	short x	2
INTEGER*4 x	int x	4
INTEGER*8 x	long x, or long long x	8
LOGICAL x	int x	4
LOGICAL*1 x	char x	1
LOGICAL*2 x	short x	2
LOGICAL*4 x	int x	4
LOGICAL*8 x	long x, or long long x	8

Table 7.12. Fortran and C/C++ Representation of the COMPLEX Type

Fortran Type (lower case)	C/C++ Type	Size (bytes)
complex x	struct {float r,i;} x; float complex x;	8
complex*8 x	struct {float r,i;} x; float complex x;	8 8

Fortran Type (lower case)	C/C++ Type	Size (bytes)
double complex x	struct {double dr,di;} x; double complex x;	16 16
complex *16 x	struct {double dr,di;} x; double complex x;	16 16

Note

For C/C++, the `complex` type implies C99 or later.

Arrays

C/C++ arrays and Fortran arrays use different default initial array index values. By default, C/C++ arrays start at 0 and Fortran arrays start at 1. A Fortran array can be declared to start at zero.

Another difference between Fortran and C/C++ arrays is the storage method used. Fortran uses column-major order and C/C++ use row-major order. For one-dimensional arrays, this poses no problems. For two-dimensional arrays, where there are an equal number of rows and columns, row and column indexes can simply be reversed. Inter-language function mixing is not recommended for arrays other than single dimensional arrays and square two-dimensional arrays.

Structures, Unions, Maps, and Derived Types

Fields within Fortran structures and derived types, and multiple map declarations within a Fortran union, conform to the same alignment requirements used by C structures.

Common Blocks

A named Fortran common block can be represented in C/C++ by a structure whose members correspond to the members of the common block. The name of the structure in C/C++ must have the added underscore.

For example, the Fortran common block:

```
INTEGER I, J
COMPLEX C
DOUBLE COMPLEX CD
DOUBLE PRECISION D
COMMON /COM/ i, j, c, cd, d
```

is represented in C with the following equivalent:

```
extern struct {
int i;
int j;
struct {float real, imag;} c;
struct {double real, imag;} cd;
double d;
} com_;
```

and in C++ with the following equivalent:

```
extern "C" struct {
int i;
int j;
struct {float real, imag;} c;
struct {double real, imag;} cd;
double d;
} com_;
```

Note

The compiler-provided name of the BLANK COMMON block is implementation specific.

Calling Fortran COMPLEX and CHARACTER functions from C/C++ is not as straightforward as calling other types of Fortran functions. Additional arguments must be passed to the Fortran function by the C/C++ caller. A Fortran COMPLEX function returns its value in memory; the first argument passed to the function must contain the address of the storage for this value. A Fortran CHARACTER function adds two arguments to the beginning of its argument list. The following example of calling a Fortran CHARACTER function from C/C++ illustrates these caller-provided extra parameters:

```
CHARACTER*(*) FUNCTION CHF(C1, I)
CHARACTER*(*) C1
INTEGER I
END
```

```
extern void chf_();
char tmp[10];
char c1[9];
int i;
chf_(tmp, 10, c1, &i, 9);
```

The extra parameters tmp and 10 are supplied for the return value, while 9 is supplied as the length of c1. Refer to Section 2.8, Argument Passing and Return Conventions, for additional information.

Win64 Programming Model

This section defines compiler and assembly language conventions for the use of certain aspects of an x64 processor running a Win64 operating system. These standards must be followed to guarantee that compilers, application programs, and operating systems written by different people and organizations will work together. The conventions supported by the PGCC ANSI C compiler implement the application binary interface (ABI) as defined in the AMD64 Software Conventions document.

Function Calling Sequence

This section describes the standard function calling sequence, including the stack frame, register usage, and parameter passing.

Register Usage Conventions

The following table defines the standard for register allocation. The 64-bit AMD64 and Intel 64 architectures provide a number of registers. All the general purpose registers, XMM registers, and x87 registers are global to all procedures in a running program.

Table 7.13. Register Allocation

Type	Name	Purpose
General	%rax	return value register
	%rbx	callee-saved
	%rcx	pass 1st argument to functions
	%rdx	pass 2nd argument to functions
	%rsp	stack pointer
	%rbp	callee-saved; optional stack frame pointer
	%rsi	callee-saved
	%rdi	callee-saved
	%r8	pass 3rd argument to functions
	%r9	pass 4th argument to functions
	%r10-%r11	temporary registers; used in syscall/sysret instructions
	%r12-r15	callee-saved registers
XMM	%xmm0	pass 1st floating point argument; return value register
	%xmm1	pass 2nd floating point argument
	%xmm2	pass 3rd floating point argument
	%xmm3	pass 4th floating point argument
	%xmm4-%xmm5	temporary registers
	%xmm6-%xmm15	callee-saved registers

In addition to the registers, each function has a frame on the run-time stack. This stack grows downward from high addresses. Table 7.14 shows the stack frame organization.

Table 7.14. Standard Stack Frame

Position	Contents	Frame
8n-120 (%rbp)	argument eightbyte n	previous
	. . .	
-80 (%rbp)	argument eightbyte 5	
-88 (%rbp)	%r9 home	
-96 (%rbp)	%r8 home	
-104 (%rbp)	%rdx home	
-112 (%rbp)	%rcx home	
-120 (%rbp)	return address	current
-128 (%rbp)	caller's %rbp	
	. . .	
0 (%rsp)	variable size	

Key points concerning the stack frame:

- The parameter area at the bottom of the stack must contain enough space to hold all the parameters needed by any function call. Space must be set aside for the four register parameters to be "homed" to the stack even if there are less than four register parameters used in a given call.
- Sixteen-byte alignment of the stack is required except within a function's prolog and within leaf functions.

All registers on an x64 system are global and thus visible to both a calling and a called function. Registers %rbx, %rsp, %rbp, %rsi, %rdi, %r12, %r13, %r14, and %r15 are non-volatile. Therefore, a called function must preserve these registers' values for its caller. Remaining registers are scratch. If a calling function wants to preserve such a register value across a function call, it must save a value in its local stack frame.

Registers are used in the standard calling sequence. The first four arguments are passed in registers. Integral and pointer arguments are passed in these general purpose registers (listed in order): %rcx, %rdx, %r8, %r9. Floating point arguments are passed in the first four XMM registers: %xmm0, %xmm1, %xmm2, %xmm3. Registers are assigned using the argument's ordinal position in the argument list. For example, if a function's first argument is an integral type and its second argument is a floating-point type, the first argument will be passed in the first general purpose register (%rcx) and the second argument will be passed in the second XMM register (%xmm1); the first XMM register and second general purpose register are ignored. Arguments after the first four are passed on the stack.

Integral and pointer type return values are returned in %rax. Floating point return values are returned in %xmm0.

Additional registers with assigned roles in the standard calling sequence:

%rsp
: The stack pointer holds the limit of the current stack frame, which is the address of the stack's bottom-most, valid word. The stack pointer should point to a 16-byte aligned area unless in the prolog or a leaf function.

%rbp
: The frame pointer, if used, can provide a way to reference the previous frames on the stack. Details are implementation dependent. A function must preserve this register value for its caller.

MXCSR
: The flags register MXCSR contains the system flags, such as the direction flag and the carry flag. The six status flags (MXCSR[0:5]) are volatile; the remainder of the register is nonvolatile.

x87 - Floating Point Control Word (FPCSR)
: The control word contains the floating-point flags, such as the rounding mode and exception masking. This register is initialized at process initialization time and its value must be preserved.

Signals can interrupt processes. Functions called during signal handling have no unusual restriction on their use of registers. Moreover, if a signal handling function returns, the process resumes its original execution path with registers restored to their original values. Thus, programs and compilers may freely use all registers without danger of signal handlers changing their values.

Function Return Values

Functions Returning Scalars or No Value

- A function that returns an integral or pointer value that fits in 64 bits places its result in %rax.
- A function that returns a floating point value that fits in the XMM registers returns this value in %xmm0.
- A function that returns a value in memory via the stack places the address of this memory (passed to the function as a "hidden" first argument in %rcx) in %rax.
- Functions that return no value (also called procedures or void functions) put no particular value in any register.
- A call instruction pushes the address of the next instruction (the return address) onto the stack. The return instruction pops the address off the stack and effectively continues execution at the next instruction after the call instruction. A function that returns a scalar or no value must preserve the caller's registers as previously described. Further, the called function must remove the return address from the stack, leaving the stack pointer (%rsp) with the value it had before the call instruction was executed.

Functions Returning Structures or Unions

A function can use either registers or the stack to return a structure or union. The size and type of the structure or union determine how it is returned. A structure or union is returned in memory if it is larger than 8 bytes or if its size is 3, 5, 6, or 7 bytes. A structure or union is returned in %rax if its size is 1, 2, 4, or 8 bytes.

If a structure or union is to be returned in memory, the caller provides space for the return value and passes its address to the function as a "hidden" first argument in %rcx. This address will also be returned in %rax.

Argument Passing

Integral and Pointer Arguments

Integral and pointer arguments are passed to a function using the next available register of the sequence %rcx, %rdx, %r8, %r9. After this list of registers has been exhausted, all remaining integral and pointer arguments are passed to the function via the stack.

Floating-Point Arguments

Float and double arguments are passed to a function using the next available XMM register of the sequence %xmm0, %xmm1, %xmm2, %xmm3. After this list of registers has been exhausted, all remaining XMM floating-point arguments are passed to the function via the stack.

Array, Structure, and Union Arguments

Arrays and strings are passed to functions using a pointer to caller-allocated memory.

Structure and union arguments of size 1, 2, 4, or 8 bytes will be passed as if they were integers of the same size. Structures and unions of other sizes will be passed as a pointer to a temporary, allocated by the caller, and whose value contains the value of the argument. The caller-allocated temporary memory used for arguments of aggregate type must be 16-byte aligned.

Passing Arguments on the Stack

Registers are assigned using the argument's ordinal position in the argument list. For example, if a function's first argument is an integral type and its second argument is a floating-point type, the first argument will be passed in the first general purpose register (%rcx) and the second argument will be passed in the second XMM register (%xmm1); the first XMM register and second general purpose register are ignored. Arguments after the first four are passed on the stack; they are pushed on the stack in reverse order, with the last argument pushed first.

Table 7.15, "Register Allocation for Example A-4" shows the register allocation and stack frame offsets for the function declaration and call shown in the following example.

Example 7.4. Parameter Passing

```
typedef struct {
 int i;
 float f;
} struct1;
int i;
float f;
double d;
long l;
long long ll;
struct1 s1;
extern void func (int i, float f, struct1 s1, double d, long long ll, long l);
func (i, f, s1, d, ll, l);
```

Table 7.15. Register Allocation for Example A-4

General Purpose Registers	Floating Point Registers	Stack Frame Offset
%rcx: i	%xmm0: <ignored>	32: ll
%rdx: <ignored>	%xmm1: f	40: l
%r8: s1.i, s1.f	%xmm2: <ignored>	
%r9: <ignored>	%xmm3: d	

Implementing a Stack

In general, compilers and programmers must maintain a software stack. The stack pointer, register %rsp, is set by the operating system for the application when the program is started. The stack must grow downwards from high addresses.

A separate frame pointer enables calls to routines that change the stack pointer to allocate space on the stack at run-time (e.g. alloca). Some languages can also return values from a routine allocated on stack space below the original top-of-stack pointer. Such a routine prevents the calling function from using %rsp-relative addressing to get at values on the stack. If the compiler does not call routines that leave %rsp in an altered state when they return, a frame pointer is not needed and is not used if the compiler option `-Mnoframe` is specified.

The stack must always be 16-byte aligned except within the prolog and within leaf functions.

Variable Length Parameter Lists

Parameter passing in registers can handle a variable number of parameters. The C language uses a special method to access variable-count parameters. The `stdarg.h` and `varargs.h` files define several functions to access these parameters. A C routine with variable parameters must use the `va_start` macro to set up a data structure before the parameters can be used. The `va_arg` macro must be used to access the successive parameters.

For unprototyped functions or functions that use varargs, floating-point arguments passed in registers must be passed in both an XMM register and its corresponding general purpose register.

C Parameter Conversion

In C, for a called prototyped function, the parameter type in the called function must match the argument type in the calling function.

- If the called function is not prototyped, the calling convention uses the types of the arguments but promotes char or short to int, and unsigned char or unsigned short to unsigned int and promotes float to double, unless you use the `-Msingle` option.

  For more information on the `-Msingle` option, refer to Chapter 2, "*Command-Line Options Reference*".

- If the called function is prototyped, the unused bits of a register containing a char or short parameter are undefined and the called function must extend the sign of the unused bits when needed.

Calling Assembly Language Programs

Example 7.5. C Program Calling an Assembly-language Routine

```
/* File: testmain.c */
main() {
 long l_para1 = 0x3f800000;
 float f_para2 = 1.0;
 double d_para3 = 0.5;
 float f_return;
 extern float sum_3 (long para1, float para2, double para3);
 f_return = sum_3(l_para1,f_para2, d_para3);
 printf("Parameter one, type long = %08x\n",l_para1);
 printf("Parameter two, type float = %f\n",f_para2);
 printf("Parameter three, type double = %g\n",d_para3);
 printf("The sum after conversion = %f\n",f_return);
}
# File: sum_3.s
# Computes ( para1 + para2 ) + para3
 .text
 .align 16
 .globl sum_3
sum_3:
 pushq %rbp
 leaq 128(%rsp), %rbp
 cvtsi2ss %ecx, %xmm0
 addss %xmm1, %xmm0
 cvtss2sd %xmm0, %xmm0
 addsd %xmm2, %xmm0
 cvtsd2ss %xmm0, %xmm0
 popq %rbp
 ret
 .type sum_3,@function
 .size sum_3,.-sum_3
```

Win64 Fortran Supplement

Sections A3.4.1 through A3.4.4 of the AMD64 Software Conventions for Win64 define the Fortran supplement. The register usage conventions set forth in that document remain the same for Fortran.

Fortran Fundamental Types

Table 7.16. Win64 Fortran Fundamental Types

Fortran Type	Size (bytes)	Alignment (bytes)
INTEGER	4	4
INTEGER*1	1	1
INTEGER*2	2	2
INTEGER*4	4	4
INTEGER*8	8	8
LOGICAL	4	4
LOGICAL*1	1	1
LOGICAL*2	2	2
LOGICAL*4	4	4
LOGICAL*8	8	8
BYTE	1	1
CHARACTER*n	n	1
REAL	4	4
REAL*4	4	4
REAL*8	8	8
DOUBLE PRECISION	8	8
COMPLEX	8	4
COMPLEX*8	8	4
COMPLEX*16	16	8
DOUBLE COMPLEX	16	8

A logical constant is one of:

- .TRUE.
- .FALSE.

The logical constants .TRUE. and .FALSE. are defined to be the four-byte value 1 and 0 respectively. A logical expression is defined to be .TRUE. if its least significant bit is 1 and .FALSE. otherwise.

Note that the value of a character is not automatically NULL-terminated.

Fortran Naming Conventions

By default, all globally visible Fortran symbol names (subroutines, functions, common blocks) are converted to lower-case. In addition, an underscore is appended to Fortran global names to distinguish the Fortran name space from the C/C++ name space.

Fortran Argument Passing and Return Conventions

Arguments are passed by reference, meaning the address of the argument is passed rather than the argument itself. In contrast, C/C++ arguments are passed by value.

When passing an argument declared as Fortran type CHARACTER, an argument representing the length of the CHARACTER argument is also passed to the function. This length argument is a four-byte integer passed by value, and is passed at the end of the parameter list following the other formal arguments. A length argument is passed for each CHARACTER argument; the length arguments are passed in the same order as their respective CHARACTER arguments.

A Fortran function, returning a value of type CHARACTER, adds two arguments to the beginning of its argument list. The first additional argument is the address of the area created by the caller for the return value; the second additional argument is the length of the return value. If a Fortran function is declared to return a character value of constant length, for example `CHARACTER*4 FUNCTION CHF()`, the second extra parameter representing the length of the return value must still be supplied.

A Fortran complex function returns its value in memory. The caller provides space for the return value and passes the address of this storage as if it were the first argument to the function.

Alternate return specifiers of a Fortran function are not passed as arguments by the caller. The alternate return function passes the appropriate return value back to the caller in %rax.

The handling of the following Fortran 90 features is implementation-defined: internal procedures, pointer arguments, assumed-shape arguments, functions returning arrays, and functions returning derived types.

Inter-language Calling

Inter-language calling between Fortran and C/C++ is possible if function/subroutine parameters and return values match types. If a C/C++ function returns a value, call it from Fortran as a function, otherwise, call it as a subroutine. If a Fortran function has type CHARACTER or COMPLEX, call it from C/C++ as a void function. If a Fortran subroutine has alternate returns, call it from C/C++ as a function returning int; the value of such a subroutine is the value of the integer expression specified in the alternate RETURN statement. If a Fortran subroutine does not contain alternate returns, call it from C/C++ as a void function.

Table 7.17 provides the C/C++ data type corresponding to each Fortran data type.

Table 7.17. Fortran and C/C++ Data Type Compatibility

Fortran Type	C/C++ Type	Size (bytes)
CHARACTER*n x	char x[n]	n
REAL x	float x	4
REAL*4 x	float x	4

Fortran Type	C/C++ Type	Size (bytes)
REAL*8 x	double x	8
DOUBLE PRECISION x	double x	8
INTEGER x	int x	4
INTEGER*1 x	signed char x	1
INTEGER*2 x	short x	2
INTEGER*4 x	int x	4
INTEGER*8 x	long long x	8
LOGICAL x	int x	4
LOGICAL*1 x	char x	1
LOGICAL*2 x	short x	2
LOGICAL*4 x	int x	4
LOGICAL*8 x	long long x	8

The PGI Compiler User's Guide contains a table that provides the Fortran and C/C++ representation of the COMPLEX type.

Table 7.18. Fortran and C/C++ Representation of the COMPLEX Type

Fortran Type (lower case)	C/C++ Type	Size (bytes)
complex x	struct {float r,i;} x; float complex x;	8 8
complex*8 x	struct {float r,i;} x; float complex x;	8 8
double complex x	struct {double dr,di;} x; double complex x;	16 16
complex *16 x	struct {double dr,di;} x; double complex x;	16 16

Note

For C/C++, the `complex` type implies C99 or later.

Arrays

For a number of reasons inter-language function mixing is not recommended for arrays other than single dimensional arrays and square two-dimensional arrays.

- C/C++ arrays and Fortran arrays use different default initial array index values. By default, C/C++ arrays start at 0 and Fortran arrays start at 1. However, a Fortran array can be declared to start at zero.

- Fortran and C/C++ arrays use different storage methods. Fortran uses column-major order and C/C++ use row-major order. For one-dimensional arrays, this poses no problems. For two-dimensional arrays, where there are an equal number of rows and columns, row and column indexes can simply be reversed.

Structures, Unions, Maps, and Derived Types.

Fields within Fortran structures and derived types, and multiple map declarations within a Fortran union, conform to the same alignment requirements used by C structures.

Common Blocks

A named Fortran common block can be represented in C/C++ by a structure whose members correspond to the members of the common block. The name of the structure in C/C++ must have the added underscore. Here is an example.

Fortran common block:

```
INTEGER I, J
COMPLEX C
DOUBLE COMPLEX CD
DOUBLE PRECISION D
COMMON /COM/ i, j, c, cd, d
```

C equivalent:

```
extern struct {
int i;
int j;
struct {float real, imag;} c;
struct {double real, imag;} cd;
double d;
} com_;
```

C++ equivalent:

```
extern "C" struct {
int i;
int j;
struct {float real, imag;} c;
struct {double real, imag;} cd;
double d;
} com_;
```

Note

The compiler-provided name of the BLANK COMMON block is implementation-specific.

Calling Fortran COMPLEX and CHARACTER functions from C/C++ is not as straightforward as calling other types of Fortran functions. Additional arguments must be passed to the Fortran function by the C/C++ caller. A Fortran COMPLEX function returns its value in memory; the first argument passed to the function must contain the address of the storage for this value. A Fortran CHARACTER function adds two arguments to the beginning of its argument list. The following example of calling a Fortran CHARACTER function from C/C++ illustrates these caller-provided extra parameters:

```
CHARACTER*(*) FUNCTION CHF(C1, I)
CHARACTER*(*) C1
INTEGER I
 END
```

```
 extern void chf_();
char tmp[10];
char c1[9];
int i;
chf_(tmp, 10, c1, &i, 9);
```

The extra parameters `tmp` and `10` are supplied for the return value, while `9` is supplied as the length of `c1`.

Chapter 8. C++ Dialect Supported

The PGC++ compiler accepts the C++ language of the ISO/IEC 14882:1998 C++ standard, except for Exported Templates.PGC++ optionally accepts a number of features erroneously accepted by cfront version 2.1 or 3.0. Using the -b option, PGC++ accepts these features, which may never have been legal C++, but have found their way into some user's code.

Command-line options provide full support of many C++ variants, including strict standard conformance. PGC++ provides command line options that enable the user to specify whether anachronisms and/or cfront 2.1/3.0 compatibility features should be accepted.

Extensions Accepted in Normal C++ Mode

The following extensions are accepted in all modes, except when strict ANSI violations are diagnosed as errors, described in the –A option:

- A friend declaration for a class may omit the class keyword:

```
class A {
friend B; // Should be "friend class B"
};
```

- Constants of scalar type may be defined within classes:

```
class A {
const int size = 10;
int a[size];
};
```

- In the declaration of a class member, a qualified name may be used:

```
struct A{
int A::f(); // Should be int f();
}
```

- The preprocessing symbol c_plusplus is defined in addition to the standard __cplusplus.
- An assignment operator declared in a derived class with a parameter type matching one of its base classes is treated as a "default'' assignment operator --- that is, such a declaration blocks the implicit generation of a copy assignment operator. (This is cfront behavior that is known to be relied upon in at least one widely used library.)

  Here's an example:

```
struct A { } ;
struct B : public A {
B& operator=(A&);
};
```

- By default, as well as in cfront-compatibility mode, there will be no implicit declaration of B::operator=(const B&), whereas in strict-ANSI mode B::operator=(A&) is not a copy assignment operator and B::operator=(const B&) is implicitly declared.
- Implicit type conversion between a pointer to an extern "C" function and a pointer to an extern "C++" function is permitted. Here's an example:

```
extern "C" void
f(); // f's type has extern "C" linkage
 void (*pf) () // pf points to an extern
"C++" function
 = &f; // error unless
implicit conv is allowed
```

cfront 2.1 Compatibility Mode

The following extensions are accepted in cfront 2.1 compatibility mode in addition to the extensions listed in the following section. These things were corrected in the 3.0 release of cfront:

- The dependent statement of an if, while, do-while, or for is not considered to define a scope. The dependent statement may not be a declaration. Any objects constructed within the dependent statement are destroyed at exit from the dependent statement.
- Implicit conversion from integral types to enumeration types is allowed.
- A non-const member function may be called for a const object. A warning is issued.
- A const void * value may be implicitly converted to a void * value, e.g., when passed as an argument.
- When, in determining the level of argument match for overloading, a reference parameter is initialized from an argument that requires a non-class standard conversion, the conversion counts as a user-defined conversion. (This is an outright bug, which unfortunately happens to be exploited in some class libraries.)
- When a builtin operator is considered alongside overloaded operators in overload resolution, the match of an operand of a builtin type against the builtin type required by the builtin operator is considered a standard conversion in all cases (e.g., even when the type is exactly right without conversion).
- A reference to a non-const type may be initialized from a value that is a const-qualified version of the same type, but only if the value is the result of selecting a member from a const class object or a pointer to such an object.
- A cast to an array type is allowed; it is treated like a cast to a pointer to the array element type. A warning is issued.
- When an array is selected from a class, the type qualifiers on the class object (if any) are not preserved in the selected array. (In the normal mode, any type qualifiers on the object are preserved in the element type of the resultant array.)
- An identifier in a function is allowed to have the same name as a parameter of the function. A warning is issued.

- An expression of type void may be supplied on the return statement in a function with a void return type. A warning is issued.
- cfront has a bug that causes a global identifier to be found when a member of a class or one of its base classes should actually be found. This bug is not emulated in cfront compatibility mode.
- A parameter of type "const void *'' is allowed on operator delete; it is treated as equivalent to "void *".
- A period (".") may be used for qualification where "::" should be used. Only "::'' may be used as a global qualifier. Except for the global qualifier, the two kinds of qualifier operators may not be mixed in a given name (i.e., you may say A::B::C or A.B.C but not A::B.C or A.B::C). A period may not be used in a vacuous destructor reference nor in a qualifier that follows a template reference such as A<T>::B.
- cfront 2.1 does not correctly look up names in friend functions that are inside class definitions. In this example function f should refer to the functions and variables (e.g., f1 and a1) from the class declaration. Instead, the global definitions are used.

```
int a1;
int e1;
void f1();
class A {
 int a1;
 void f1();
 friend void f()
 {
 int i1 = a1; // cfront uses global a1
 f1(); // cfront uses global f1
 }
};
```

- Only the innermost class scope is (incorrectly) skipped by cfront as illustrated in the following example.

```
int a1;
int b1;
struct A {
 static int a1;
 class B {
 static int b1;
 friend void f()
 {
 int i1 = a1; // cfront uses A::a1
 int j1 = b1; // cfront uses global b1
 }
 };
};
```

- operator= may be declared as a nonmember function. (This is flagged as an anachronism by cfront 2.1)
- A type qualifier is allowed (but ignored) on the declaration of a constructor or destructor. For example:

```
class A {
 A() const; // No error in cfront 2.1 mode
};
```

cfront 2.1/3.0 Compatibility Mode

The following extensions are accepted in both cfront 2.1 and cfront 3.0 compatibility mode (i.e., these are features or problems that exist in both cfront 2.1 and 3.0):

- Type qualifiers on this parameter may to be dropped in contexts such as this example:

```
struct
A {
 void f() const;
};
void (A::*fp)() = &A::f;
```

  This is actually a safe operation. A pointer to a const function may be put into a pointer to non-const, because a call using the pointer is permitted to modify the object and the function pointed to will actually not modify the object. The opposite assignment would not be safe.

- Conversion operators specifying conversion to void are allowed.
- A nonstandard friend declaration may introduce a new type. A friend declaration that omits the elaborated type specifier is allowed in default mode, but in cfront mode the declaration is also allowed to introduce a new type name.

```
 struct A {
 friend B;
 };
```

- The third operator of the ? operator is a conditional expression instead of an assignment expression.
- A reference to a pointer type may be initialized from a pointer value without use of a temporary even when the reference pointer type has additional type qualifiers above those present in the pointer value. For example,

```
 int *p;
 const int *&r = p; // No
temporary used
```

- A reference may be initialized with a null.

Chapter 9. Fortran Module/Library Interfaces for Windows

PGI Fortran for Windows provides access to a number of libraries that export C interfaces by using Fortran modules. PGI uses this mechanism to support the Win32 API and Unix/Linux/Mac OS X portability libraries. This chapter describes the Fortran module library interfaces that PGI supports, describing each property available.

Source Files

All routines described in this chapter have their prototypes and interfaces described in source files that are included in the PGI Windows compiler installation. The location of these files depends on your operating system version, either win32 or win64, and the release version that you have installed, such as 7.2-5 or 10.0-0. These files are typically located in this directory:

```
C:/Program Files/PGI/{win32,win64}/[release_version]/src
```

For example, if you have installed the Win32 version of the 11.0-0 release, look for your files in this location:

```
C:/Program Files/PGI/win32/11.0-0/src
```

Data Types

Because the Win32 API and Portability interfaces resolve to C language libraries, it is important to understand how the data types compare within the two languages. Here is a table summarizing how C types correspond with Fortran types for some of the more common data types:

Table 9.1. Fortran Data Type Mappings

Windows Data Type	Fortran Data Type
BOOL	LOGICAL(4)
BYTE	BYTE
CHAR	CHARACTER
SHORT, WORD	INTEGER(2)

Windows Data Type	Fortran Data Type
DWORD, INT, LONG	INTEGER(4)
LONG LONG	INTEGER(8)
FLOAT	REAL(4)
DOUBLE	REAL(8)
x86 Pointers	INTEGER(4)
x64 Pointers	INTEGER(8)

For more information on data types, refer to "Fortran Data Types," on page 1.

Using DFLIB, LIBM, and DFPORT

PGI includes Fortran module interfaces to libraries supporting some standard C library, C math library, and Unix/Linux/Mac OS X system call functionality. These functions are provided by the `DFLIB`, `LIBM`, and `DFPORT` modules. To utilize these modules, add the appropriate `USE` statement:

```
use dflib
```

```
use libm
```

```
use dfport
```

DFLIB

Table 9.2 lists the functions that `DFLIB` includes. In the table `[Generic]` refers to a generic routine. To view the prototype and interfaces, look in the location described in "Source Files," on page 211.

Table 9.2. DFLIB Function Summary

Routine	Result	Description
commitqq	LOGICAL*4	Executes any pending write operations for the file associated with the specified unit to the file's physical device.
delfilesqq	INTEGER*4	Deletes the specified files in a specified directory.
findfileqq	INTEGER*4	Searches for a file in the directories specified in the PATH environment variable.
fullpathqq	INTEGER*4	Returns the full path for a specified file or directory.
getdat	INTEGER*2,*4,*8	[Generic] Returns the date.
getdrivedirqq	INTEGER*4	Returns the current drive and directory path.
getenvqq	INTEGER*4	Returns a value from the current environment.
getfileinfoqq	INTEGER*4	Returns information about files with names that match the specified string.
getfileinfoqqi8	INTEGER*4	Returns information about files with names that match the specified string.
gettim	INTEGER*2,*4,*8	[Generic] Returns the time.

Routine	Result	Description
packtimeqq	INTEGER*4	Packs the time and date values for use by setfiletimeqq
renamefileqq	LOGICAL*4	Renames the specified file.
runqq	INTEGER*2	Calls another program and waits for it to execute.
setenvqq	LOGICAL*4	Sets the values of an existing environment variable or adds a new one.
setfileaccessqq	LOGICAL*4	Sets the file access mode for the specified file.
setfiletimeqq	LOGICAL*4	Sets the modification time for the specified file.
signalqq	INTEGER*8	Controls signal handling.
sleepqq	None	Delays execution of the program for a specified time.
splitpathqq	LOGICAL*4	Breaks a full path into components.
systemqq	LOGICAL*4	Executes a command by passing a command string to the operating system's command interpreter.
unpacktimeqq	Multiple INTEGERS	Unpacks a file's packed time and date value into its component parts.

LIBM

A Fortran module is available to declare interfaces to many of the routines in the standard C math library: `libm` Table 9.3 lists the `LIBM` routines that are available. To view the prototype and interfaces, look in the location described in "Source Files," on page 211.

Some `libm` routine names conflct with Fortran intrinsics. These routines are not listed in this table because they resolve to Fortran intrinsics.

asin	acos	atan2	cos	cosh
exp	log	log10	sin	sinh
sqrt	tan	tanh		

You can also use `libm` routines in CUDA Fortran global and device subprograms, in CUF kernels, and in PGI Accelerator compute regions. When targeting NVIDIA devices, the `libm` routines translate to the corresponding `libm` device routine.

Table 9.3. LIBM Functions

acosf	erfc	frexp	log1p	remquo
acosh	erff	frexpf	log1pf	remquof
acoshf	erfcf	ilog	log2	rint
asinf	expf	ilogbf	log2f	rintf
asinh	exp10	ldexp	logb	scalbn
asinhf	exp10f	ldexpf	logbf	scalbnf
atan2f	exp2	lgamma	logf	scalbln

atanh	exp2f	lgammaf	modf	scalblnf
atanhf	expf	llrint	modff	sinf
cbrt	expm1	llrintf	nearbyint	sinhf
cbrtf	expm1f	lrint	nearbyintf	sqrtf
ceil	floor	lrint	nextafter	tanf
ceilf	floorf	llround	nextafterf	tanhf
copysign	fma	llroundf	pow	tgamma
copysignf	fmaf	lround	powf	tgammaf
cosf	fmax	lroundf	remainder	trunc
coshf	fmaxf	log10f	remainderf	truncf
erf	fminf			

DFPORT

Table 9.4 lists the functions that `DFPORT` includes. In the table `[Generic]` refers to a generic routine. To view the prototype and interfaces, look in the location described in "Source Files," on page 211.

Table 9.4. DFPORT Functions

Routine	Result	Description
abort	None	Immediately terminates the program. If the operating systems supports a core dump, abort produces one that can be used for debugging.
access	INTEGER*4	Determines access mode or existence of a file.
alarm	INTEGER*4	Executes a routine after a specified time.
besj0	REAL*4	Computes the BESSEL function of the first kind of order 0 of X, where X is real.
besj1	REAL*4	Computes the BESSEL function of the first kind of order 1 of X, where X is real.
besjn	REAL*4	Computes the BESSEL function of the first kind of order N of X, where N is an integer and X is real.
besy0	REAL*4	Computes the BESSEL function of the second kind of order 0 of X, where X is real.
besy1	REAL*4	Computes the BESSEL function of the second kind of order 1 of X, where X is real.
besyn	REAL*4	Computes the BESSEL function of the second kind of order N of X, where N is an integer and X is real.
chdir	INTEGER*4	Changes the current directory to the directory specified. Returns 0, if successful or an error

Routine	Result	Description
chmod	INTEGER*4	Changes the mode of a file by setting the access permissions of the specified file to the specified mode. Returns 0 if successful, or error
ctime	STRING(24)	Converts and returns the specified time and date as a string.
date	STRING	Returns the date as a character string: dd-mm-yy.
dbesj0	REAL*8	Computes the double-precision BESSEL function of the first kind of order 0 of X, where X is a double-precision argument.
dbesj1	REAL*8	Computes the double-precision BESSEL function of the first kind of order 1 of X, where X is a double-precision argument.
dbesjn	REAL*8	Computes the double-precision BESSEL function of the first kind of order N of X, where N is an integer and X is a double-precision argument.
dbesy0	REAL*8	Computes the double-precision BESSEL function of the second kind of order 0 of X, where X, where X is a double-precision argument.
dbesy1	REAL*8	Computes the double-precision BESSEL function of the second kind of order 1 of X, where X, where X is a double-precision argument.
dbesyn	REAL*8	Computes the double-precision BESSEL function of the second kind of order N of X, where N is an integer and X, where X is a double-precision argument.
derf	REAL*8	Computes the double-precision error function of X, where X is a double-precision argument.
derfc	REAL*8	Computes the complementary double-precision error function of X, where X is a double-precision argument.
dffrac	REAL*8	Returns fractional accuracy of a REAL*8 floating-point value.
dflmax	REAL*8	Returns the maximum positive REAL*8 floating-point value.
dflmin	REAL*8	Returns the minimum positive REAL*8 floating-point value.
drandm	REAL*8	Generates a REAL*8 random number.
dsecnds	REAL*8	Returns the number of real time seconds since midnight minus the supplied argument value.
dtime	REAL*4	Returns the elapsed user and system time in seconds since the last call to dtime.
erf	REAL*4	Computes the error function of X, where X is Real.

Routine	Result	Description
erfc	REAL	Computes the complementary error function of X, where X is Real.
etime	REAL*4	Returns the elapsed time in seconds since the start of program execution.
exit	None	Immediately terminates the program and passes a status to the parent process.
fdate	STRING	Returns the current date and time as an ASCII string.
ffrac	REAL*4	Returns the fractional accuracy of a REAL*4 floating-point value.
fgetc	INTEGER*4	Gets a character or word from an input stream. Returns the next byte or and integer
flmax	REAL*4	Returns the maximum positive REAL*4 floating-point value.
flmin	REAL*4	Returns the minimum positive REAL*4 floating-point value.
flush	None	Writes the output to a logical unit.
fputc	INTEGER*4	Writes a character or word from an input stream to a logical unit. Returns 0 if successful or an error.
free	None	Frees memory previously allocated by MALLOC(). Intended for users compiling legacy code. Use DEALLOCATE for newer code.
fseek	INTEGER*4	Repositions the file pointer associated with the specified file. Returns 0 if successful, 1 otherwise.
fseek64	INTEGER*4	Repositions the file pointer associated with the specified stream. Returns 0 if successful, 1 otherwise.
fstat	INTEGER*4	Returns file status information about the referenced open file or shared memory object.
fstat64	INTEGER*4	Returns information in a 64-bit structure about the referenced open file or shared memory object.
ftell	INTEGER*4	Returns the current value of the file pointer associated with the specified stream.
ftell64	INTEGER*8	Returns the current value of the file pointer associated with the specified stream.
gerror	STRING	Writes system error messages.
getarg	STRING	Returns the list of parameters that were passed to the current process when it was started.
getc	INTEGER*4	Retrieves the character at the front of the specified character list, or -1 if empty
getcwd	INTEGER*4	Retrieves the pathname of the current working directory or null if fails.

Routine	Result	Description
getenv		Returns the value of the specified environment variable(s).
getfd	INTEGER*4	Returns the file descriptor associated with a Fortran logical unit.
getgid	INTEGER*4	Returns the numerical group ID of the current process.
getlog	STRING	Stores the user's login name in NAME. If the login name is not found, then NAME is filled with blanks.
getpid	INTEGER*4	Returns the process numerical identifier of the current process.
getuid	INTEGER*4	Returns the numerical user ID of the current process.
gmtime	INTEGER*4	Converts and returns the date and time formats to GM (Greenwich) time as month, day, and so on.
hostnm	INTEGER*4	Sets or Gets the name of the current host. If setting the hostname, returns 0 if successful, errno if not.
iargc	INTEGER*4	Returns an integer representing the number of arguments for the last program entered on the command line.
idate	INTEGER*4	Returns the date in numerical form, day, month, year.
ierrno	INTEGER*4	Returns the system error number for the last error.
inmax	INTEGER*4	Returns the maximum positive integer value.
ioinit	None	Establishes the properties of file I/O for files opened after the call to ioinit, such as whether to recognize carriage control, how to treat blanks and zeros, and whether to open files at the beginning or end of the file.
irand1	INTEGER*4	Generates pseudo-random integer in the range of 0 through (2**31)-1, or (2**15)-1 if called with no argument.
irand2	INTEGER*4	Generates pseudo-random integer in the range of 0 through (2**31)-1, or (2**15)-1 if called with no argument.
irandm	INTEGER*4	Generates pseudo-random integer in the range of 0 through (2**31)-1, or (2**15)-1 if called with no argument.
isatty	LOGICAL	Finds the name of a terminal port. Returns TRUE if the specified unit is a terminal.
itime	numerical form of time	Fills and returns TARRAY with numerical values at the current local time, with elements 1,2,and 3 of TARRY being the hour (1-24), minute (1-60) and seconds (1-60).
kill	INTEGER*4	Sends the specified signal to the specified process or group of processes. Returns 0 if successful, -1 otherwise
link	INTEGER*4	Creates an additional directory entry for the specified existing file.

Routine	Result	Description
lnblnk	INTEGER*4	Returns the position of the last non-blank string character in the specified string.
loc	INTEGER*4	Returns the address of an object.
long	INTEGER*4	Converts INTEGER*2 to INTEGER*4
lstat	INTEGER*4	Obtains information about the referenced open file or shared memory object in a large-file enables programming environment.
lstat64	INTEGER*4	Obtains information in a 64-bit structure about the referenced open file or shared memory object in a large-file enables programming environment.
ltime	Array of INTEGER*4	Converts the system time from seconds into TARRAY, which contains GMT for the current local time zone.
malloc	INTEGER*8	Allocates SIZE byes of dynamic memory, returning the address of the allocated memory. Intended for users compiling legacy code. Use ALLOCATE for newer code.
mclock	INTEGER*4	Returns time accounting information about the current process and its child processes in 1/100 or second units of measure. The returned value is the sum of the current process's user time and system time of all child processes.
outstr	INTEGER*4	Outputs the value of the specified character to the standard output file.
perror	None	Writes a message to standard error output the describes the last error encountered by a system call or library subroutine.
putc	INTEGER*4	Puts the specified character at the end of the character list.
putenv	INTEGER*4	Sets the value of the specified environment variable or creates a new environment variable.
qsort	INTEGER*4	Uses quick-sort algorithm to sort a table of data.
rand1	REAL*4	Provides a method for generating a random number that can be used as the starting point for the rand procedure.
rand2	REAL*4	Provides a random value between 0 and 1, which is generated using the specified seed value, and computed for each returned row when used in the select list.
random	REAL*4	Uses a non-linear additive feedback random-number generator to return pseudo-random numbers in the range of 0 to $(2^{31}-1)$
rename	INTEGER*4	Renames the specified directory or file

Routine	Result	Description
rindex	INTEGER*4	Returns the index of the last occurrence of a specific string of characters in a specified string.
rtc	REAL*8	Returns the real-time clock value expressed as a number of clock ticks.
secnds	REAL*4	Gets the time in seconds from the real-time system clock. If the value is zero, the time in seconds from midnight is used.
short	INTEGER*2	Converts INTEGER*4 to INTEGER*2.
signal	INTEGER*4	Specifies the action to take upon delivery of a signal.
sleep	None	Puts the calling kernel thread to sleep, requiring it to wait for a wakeup to be issued to continue to run. Provided for compatibility with older code and should not be used with new code.
srand1	None	Sets the seed for the pseudo-random number generation that rand1 provides.
srand2	None	Sets the seed for the pseudo-random number generation that rand2 provides.
stat	INTEGER*4	Obtains information about the specified file.
stat64	INTEGER*4	Obtains information in a 64-bit structure about the specified file.
stime	INTEGER*4	Sets the current value of the specified parameter for the system-wide timer.
symlnk	INTEGER*4	Creates a symbolic link with the specified name to the specified file.
system	INTEGER*4	Runs a shell command.
time	INTEGER*4	Returns the time in seconds since January 1, 1970.
timef	REAL*8	Returns the elapsed time in milliseconds since the first call to timef.
times	INTEGER*4	Fills the specified structure with time-accounting information.
ttynam	STRING(100)	Either gets the path name of the terminal or determines if the device is a terminal.
unlink	INTEGER*4	Removes the specified directory entry, and decreases the link count of the file referenced by the link.
wait	INTEGER*4	Suspends the calling thread until the process receives a signal that is not blocked or ignored, or until the calling process' child processes stop or terminate.

Using the DFWIN module

The DFWIN module includes all the modules needed to access the Win32 API. You can use modules supporting specific portions of the Win32 API separately. DFWIN is the only module you need to access the Fortran interfaces to the Win32 API. To use this module, add the following line to your Fortran code.

```
use dfwin
```

To utilize any of the Win32 API interfaces, you can add a Fortran use statement for the specific library or module that includes it. For example, to use user32.lib, add the following Fortran use statement:

```
use user32
```

Function calls made through the module interfaces ultimately resolve to C Language interfaces, so some accommodation for inter-language calling conventions must be made in the Fortran application. These accommodations include:

- On x64 platforms, pointers and pointer types such as HANDLE, HINSTANCE, WPARAM, and HWND must be treated as 8-byte quantities (INTEGER(8)). On x86 (32-bit) platforms, these are 4-byte quantities (INTEGER(4)).
- In general, C makes calls by value while Fortran makes calls by reference.
- When doing Windows development one must sometimes provide callback functions for message processing, dialog processing, etc. These routines are called by the Windows system when events are processed. To provide the expected function signature for a callback function, the user may need to use the STDCALL attribute directive (!DEC$ ATTRIBUTE::STDCALL) in the declaration.

For information on the arguments and functionality of a given routine, refer to The Microsoft Windows API documentation.

Supported Libraries and Modules

The following tables provide lists of the functions in each library or module that PGI supports in DFWIN.

Note

For information on the interfaces associated with these functions, refer to the files located here:

```
C:\Program Files\PGI\win32\10.0-0\src
```

or

```
C:\Program Files\PGI\win64\10.0-0\src
```

advapi32

The following table lists the functions that advapi32 includes:

Table 9.5. DFWIN advapi32 Functions

AccessCheckAndAuditAlarm	AccessCheckByType
AccessCheckByTypeAndAuditAlarm	AccessCheckByTypeResultList
AccessCheckByTypeResultListAndAuditAlarm	AccessCheckByTypeResultListAndAuditAlarmByHandle
AddAccessAllowedAce	AddAccessAllowedAceEx

AddAccessAllowedObjectAce
AddAccessDeniedAce
AddAccessDeniedAceEx
AddAccessDeniedObjectAce
AddAce
AddAuditAccessAce
AddAuditAccessAceEx
AddAuditAccessObjectAce
AdjustTokenGroups
AdjustTokenPrivileges
AllocateAndInitializeSid
AllocateLocallyUniqueId
AreAllAccessesGranted
AreAnyAccessesGranted
BackupEventLog
CheckTokenMembership
ClearEventLog
CloseEncryptedFileRaw
CloseEventLog
ConvertToAutoInheritPrivateObjectSecurity
CopySid
CreatePrivateObjectSecurity
CreatePrivateObjectSecurityEx
CreatePrivateObjectSecurityWithMultipleInheritance
CreateProcessAsUser
CreateProcessWithLogonW
CreateProcessWithTokenW
CreateRestrictedToken
CreateWellKnownSid
DecryptFile
DeleteAce
DeregisterEventSource
DestroyPrivateObjectSecurity
DuplicateToken
DuplicateTokenEx
EncryptFile
EqualDomainSid
EqualPrefixSid
EqualSid
FileEncryptionStatus
FindFirstFreeAce
FreeSid
GetAce
GetAclInformation
GetCurrentHwProfile
GetEventLogInformation
GetFileSecurity
GetKernelObjectSecurity
GetLengthSid
GetNumberOfEventLogRecords
GetOldestEventLogRecord
GetPrivateObjectSecurity
GetSecurityDescriptorControl
GetSecurityDescriptorDacl
GetSecurityDescriptorGroup
GetSecurityDescriptorLength
GetSecurityDescriptorOwner
GetSecurityDescriptorRMControl
GetSecurityDescriptorSacl
GetSidIdentifierAuthority
GetSidLengthRequired
GetSidSubAuthority
GetSidSubAuthorityCount
GetTokenInformation
GetUserName
GetWindowsAccountDomainSid
ImpersonateAnonymousToken
ImpersonateLoggedOnUser
ImpersonateNamedPipeClient
ImpersonateSelf

InitializeAcl
InitializeSecurityDescriptor
InitializeSid
IsTextUnicode
IsTokenRestricted
IsTokenUntrusted
IsValidAcl
IsValidSecurityDescriptor
IsValidSid
IsWellKnownSid
LogonUser
LogonUserEx
LookupAccountName
LookupAccountSid
LookupPrivilegeDisplayName
LookupPrivilegeName
LookupPrivilegeValue
MakeAbsoluteSD
MakeAbsoluteSD2
MakeSelfRelativeSD
MapGenericMask
NotifyChangeEventLog
ObjectCloseAuditAlarm
ObjectDeleteAuditAlarm
ObjectOpenAuditAlarm
ObjectPrivilegeAuditAlarm
OpenBackupEventLog
OpenEncryptedFileRaw
OpenEventLog
OpenProcessToken
OpenThreadToken
PrivilegeCheck
PrivilegedServiceAuditAlarm
ReadEncryptedFileRaw
ReadEventLog
RegisterEventSource
ReportEvent
RevertToSelf
SetAclInformation
SetFileSecurity
SetKernelObjectSecurity
SetPrivateObjectSecurity
SetPrivateObjectSecurityEx
SetSecurityDescriptorControl
SetSecurityDescriptorDacl
SetSecurityDescriptorGroup
SetSecurityDescriptorOwner
SetSecurityDescriptorRMControl
SetSecurityDescriptorSacl
SetThreadToken
SetTokenInformation
WriteEncryptedFileRaw

comdlg32

The following table lists the functions that `comdlg32` includes:

AfxReplaceText
ChooseColor
ChooseFont
CommDlgExtendedError
FindText
GetFileTitle
GetOpenFileName
GetSaveFileName
PageSetupDlg
PrintDlg
PrintDlgEx
ReplaceText

dfwbase

These are the functions that `dfwbase` includes:

chartoint	LoByte	MakeWord
chartoreal	LoWord	MakeWparam
CopyMemory	LoWord64	PaletteIndex
GetBlueValue	MakeIntAtom	PaletteRGB
GetGreenValue	MakeIntResource	PrimaryLangID
GetRedValue	MakeLangID	RGB
HiByte	MakeLCID	RtlCopyMemory
HiWord	MakeLong	SortIDFromLCID
HiWord64	MakeLParam	SubLangID
inttochar	MakeLResult	

dfwinty

These are the functions that `dfwinty` includes:

dwNumberOfFunctionKeys	rdFunction

gdi32

These are the functions that `gdi32` includes:

AbortDoc	AbortPath	AddFontMemResourceEx
AddFontResource	AddFontResourceEx	AlphaBlend
AngleArc	AnimatePalette	Arc
ArcTo	BeginPath	BitBlt
CancelDC	CheckColorsInGamut	ChoosePixelFormat
Chord	CloseEnhMetaFile	CloseFigure
CloseMetaFile	ColorCorrectPalette	ColorMatchToTarget
CombineRgn	CombineTransform	CopyEnhMetaFile
CopyMetaFile	CreateBitmap	CreateBitmapIndirect
CreateBrushIndirect	CreateColorSpace	CreateCompatibleBitmap
CreateCompatibleDC	CreateDC	CreateDIBitmap
CreateDIBPatternBrush	CreateDIBPatternBrushPt	CreateDIBSection
CreateDiscardableBitmap	CreateEllipticRgn	CreateEllipticRgnIndirect
CreateEnhMetaFile	CreateFont	CreateFontIndirect
CreateFontIndirectEx	CreateHalftonePalette	CreateHatchBrush
CreateIC	CreateMetaFile	CreatePalette

CreatePatternBrush
CreatePen
CreatePenIndirect
CreatePolygonRgn
CreatePolyPolygonRgn
CreateRectRgn
CreateRectRgnIndirect
CreateRoundRectRgn
CreateScalableFontResource
CreateSolidBrush
DeleteColorSpace
DeleteDC
DeleteEnhMetaFile
DeleteMetaFile
DeleteObject
DescribePixelFormat
DeviceCapabilities
DPtoLP
DrawEscape
Ellipse
EndDoc
EndPage
EndPath
EnumEnhMetaFile
EnumFontFamilies
EnumFontFamiliesEx
EnumFonts
EnumICMProfiles
EnumMetaFile
EnumObjects
EqualRgn
Escape
ExcludeClipRect
ExtCreatePen
ExtCreateRegion
ExtEscape
ExtFloodFill
ExtSelectClipRgn
ExtTextOut
FillPath
FillRgn
FixBrushOrgEx
FlattenPath
FloodFill
FrameRgn
GdiComment
GdiFlush
GdiGetBatchLimit
GdiSetBatchLimit
GetArcDirection
GetAspectRatioFilterEx
GetBitmapBits
GetBitmapDimensionEx
GetBkColor
GetBkMode
GetBoundsRect
GetBrushOrgEx
GetCharABCWidthsA
GetCharABCWidthsFloat
GetCharABCWidthsI
GetCharABCWidthsW
GetCharacterPlacement
GetCharWidth
GetCharWidth32
GetCharWidthFloat
GetCharWidthI
GetClipBox
GetClipRgn
GetColorAdjustment
GetColorSpace
GetCurrentObject
GetCurrentPositionEx
GetDCBrushColor
GetDCOrgEx
GetDCPenColor
GetDeviceCaps
GetDeviceGammaRamp
GetDIBColorTable
GetDIBits
GetEnhMetaFile
GetEnhMetaFileBits
GetEnhMetaFileDescriptionA
GetEnhMetaFileDescriptionW
GetEnhMetaFileHeader
GetEnhMetaFilePaletteEntries
GetEnhMetaFilePixelFormat
GetFontData
GetFontLanguageInfo
GetFontUnicodeRanges
GetGlyphIndices
GetGlyphOutline
GetGraphicsMode
GetICMProfileA
GetICMProfileW
GetKerningPairs
GetLayout
GetLogColorSpace
GetMapMode
GetMetaFile
GetMetaFileBitsEx
GetMetaRgn
GetMiterLimit
GetNearestColor
GetNearestPaletteIndex
GetObject

GetObjectType
GetOutlineTextMetrics
GetPaletteEntries
GetPath
GetPixel
GetPixelFormat
GetPolyFillMode
GetRandomRgn
GetRasterizerCaps
GetRegionData
GetRgnBox
GetROP2
GetStockObject
GetStretchBltMode
GetSystemPaletteEntries
GetSystemPaletteUse
GetTextAlign
GetTextCharacterExtra
GetTextCharset
GetTextCharsetInfo
GetTextColor
GetTextExtentExPoint
GetTextExtentExPointI
GetTextExtentPoint
GetTextExtentPoint32
GetTextExtentPointI
GetTextFace
GetTextMetrics
GetViewportExtEx
GetViewportOrgEx
GetWindowExtEx
GetWindowOrgEx
GetWinMetaFileBits
GetWorldTransform
GradientFill
IntersectClipRect
InvertRgn
LineDD
LineTo
LPtoDP
MaskBlt
ModifyWorldTransform
MoveToEx
OffsetClipRgn
OffsetRgn
OffsetViewportOrgEx
OffsetWindowOrgEx
PaintRgn
PatBlt
PathToRegion
Pie
PlayEnhMetaFile
PlayEnhMetaFileRecord
PlayMetaFile
PlayMetaFileRecord
PlgBlt
PolyBezier
PolyBezierTo
PolyDraw
Polygon
Polyline
PolylineTo
PolyPolygon
PolyPolyline
PolyTextOut
PtInRegion
PtVisible
RealizePalette
Rectangle
RectInRegion
RectVisible
RemoveFontMemResourceEx
RemoveFontResource
RemoveFontResourceEx
ResetDC
ResizePalette
RestoreDC
RoundRect
SaveDC
ScaleViewportExtEx
ScaleWindowExtEx
SelectClipPath
SelectClipRgn
SelectObject
SelectPalette
SetAbortProc
SetArcDirection
SetBitmapBits
SetBitmapDimensionEx
SetBkColor
SetBkMode
SetBoundsRect
SetBrushOrgEx
SetColorAdjustment
SetColorSpace
SetDCBrushColor
SetDCPenColor
SetDeviceGammaRamp
SetDIBColorTable
SetDIBits
SetDIBitsToDevice
SetEnhMetaFileBits
SetGraphicsMode
SetICMMode
SetICMProfile

SetLayout
SetMapMode
SetMapperFlags
SetMetaFileBitsEx
SetMetaRgn
SetMiterLimit
SetPaletteEntries
SetPixel
SetPixelFormat
SetPixelV
SetPolyFillMode
SetRectRgn
SetROP2
SetStretchBltMode
SetSystemPaletteUse
SetTextAlign
SetTextCharacterExtra
SetTextColor
SetTextJustification
SetViewportExtEx
SetViewportOrgEx
SetWindowExtEx
SetWindowOrgEx
SetWinMetaFileBits
SetWorldTransform
StartDoc
StartPage
StretchBlt
StretchDIBits
StrokeAndFillPath
StrokePath
SwapBuffers
TextOut
TranslateCharsetInfo
TransparentBlt
UnrealizeObject
UpdateColors
UpdateICMRegKey
wglCopyContext
wglCreateContext
wglCreateLayerContext
wglDeleteContext
wglDescribeLayerPlane
wglGetCurrentContext
wglGetCurrentDC
wglGetLayerPaletteEntries
wglGetProcAddress
wglMakeCurrent
wglRealizeLayerPalette
wglSetLayerPaletteEntries
wglShareLists
wglSwapLayerBuffers
wglSwapMultipleBuffers
wglUseFontBitmaps
wglUseFontOutlines
WidenPath

kernel32

These are the functions that `kernel32` includes:

ActivateActCtx
AddAtom
AddConsoleAlias
AddRefActCtx
AddVectoredContinueHandler
AddVectoredExceptionHandler
AllocateUserPhysicalPages
AllocConsole
AreFileApisANSI
AssignProcessToJobObject
AttachConsole
BackupRead
BackupSeek
BackupWrite
Beep
BeginUpdateResource
BindIoCompletionCallback
BuildCommDCB
BuildCommDCBAndTimeouts
CallNamedPipe
CancelDeviceWakeupRequest
CancelIo
CancelTimerQueueTimer
CancelWaitableTimer
CheckNameLegalDOS8Dot3
CheckRemoteDebuggerPresent

ClearCommBreak
ClearCommError
CloseHandle
CommConfigDialog
CompareFileTime
ConnectNamedPipe
ContinueDebugEvent
ConvertFiberToThread
ConvertThreadToFiber
ConvertThreadToFiberEx
CopyFile
CopyFileEx
CreateActCtx
CreateConsoleScreenBuffer
CreateDirectory
CreateDirectoryEx
CreateEvent
CreateFiber
CreateFiberEx
CreateFile
CreateFileMapping
CreateHardLink
CreateIoCompletionPort
CreateJobObject
CreateJobSet
CreateMailslot
CreateMemoryResourceNotification
CreateMutex
CreateNamedPipe
CreatePipe
CreateProcess
CreateRemoteThread
CreateSemaphore
CreateTapePartition
CreateThread
CreateTimerQueue
CreateTimerQueueTimer
CreateWaitableTimer
DeactivateActCtx
DebugActiveProcess
DebugActiveProcessStop
DebugBreak
DebugBreakProcess
DebugSetProcessKillOnExit
DecodePointer
DecodeSystemPointer
DefineDosDevice
DeleteAtom
DeleteCriticalSection
DeleteFiber
DeleteFile
DeleteTimerQueue
DeleteTimerQueueEx
DeleteTimerQueueTimer
DeleteVolumeMountPoint
DeviceIoControl
DisableThreadLibraryCalls
DisconnectNamedPipe
DnsHostnameToComputerName
DosDateTimeToFileTime
DuplicateHandle
EncodePointer
EncodeSystemPointer
EndUpdateResource
EnterCriticalSection
EnumResourceLanguages
EnumResourceNames
EnumResourceTypes
EnumSystemFirmwareTables
EraseTape

EscapeCommFunction
ExitProcess
ExitThread
ExpandEnvironmentStrings
FatalAppExit
FatalExit
FileTimeToDosDateTime
FileTimeToLocalFileTime
FileTimeToSystemTime
FillConsoleOutputAttribute
FillConsoleOutputCharacter
FindActCtxSectionGuid
FindActCtxSectionString
FindAtom
FindClose
FindCloseChangeNotification
FindFirstChangeNotification
FindFirstFile
FindFirstFileEx
FindFirstVolume
FindFirstVolumeMountPoint
FindNextChangeNotification
FindNextFile
FindNextVolume
FindNextVolumeMountPoint
FindResource
FindResourceEx
FindVolumeClose
FindVolumeMountPointClose
FlsAlloc
FlsFree
FlsGetValue
FlsSetValue
FlushConsoleInputBuffer
FlushFileBuffers
FlushInstructionCache
FlushViewOfFile
FormatMessage
FreeConsole
FreeEnvironmentStrings
FreeLibrary
FreeLibraryAndExitThread
FreeResource
FreeUserPhysicalPages
GenerateConsoleCtrlEvent
GetAtomName
GetBinaryType
GetCommandLine
GetCommConfig
GetCommMask
GetCommModemStatus
GetCommProperties
GetCommState
GetCommTimeouts
GetCompressedFileSize
GetComputerName
GetConsoleAlias
GetConsoleAliases
GetConsoleAliasesLength
GetConsoleAliasExes
GetConsoleAliasExesLength
GetConsoleCP
GetConsoleCursorInfo
GetConsoleDisplayMode
GetConsoleFontSize
GetConsoleMode
GetConsoleOutputCP
GetConsoleProcessList
GetConsoleScreenBufferInfo
GetConsoleSelectionInfo

GetConsoleTitle
GetCurrentActCtx
GetCurrentDirectory
GetCurrentProcessId
GetCurrentThread
GetDefaultCommConfig
GetDiskFreeSpace
GetDllDirectory
GetEnvironmentStrings
GetExitCodeProcess
GetFileAttributes
GetFileInformationByHandle
GetFileSizeEx
GetFileType
GetFullPathName
GetLargePageMinimum
GetLastError
GetLogicalDrives
GetLogicalProcessorInformation
GetMailslotInfo
GetModuleHandle
GetNamedPipeHandleState
GetNativeSystemInfo
GetNumaHighestNodeNumber
GetNumaProcessorNode
GetNumberOfConsoleMouseButtons
GetPriorityClass
GetPrivateProfileSection
GetPrivateProfileString
GetProcAddress
GetProcessHandleCount
GetProcessHeaps
GetProcessIdOfThread
GetProcessPriorityBoost
GetProcessTimes
GetConsoleWindow
GetCurrentConsoleFont
GetCurrentProcess
GetCurrentProcessorNumber
GetCurrentThreadId
GetDevicePowerState
GetDiskFreeSpaceEx
GetDriveType
GetEnvironmentVariable
GetExitCodeThread
GetFileAttributesEx
GetFileSize
GetFileTime
GetFirmwareEnvironmentVariable
GetHandleInformation
GetLargestConsoleWindowSize
GetLocalTime
GetLogicalDriveStrings
GetLongPathName
GetModuleFileName
GetModuleHandleEx
GetNamedPipeInfo
GetNumaAvailableMemoryNode
GetNumaNodeProcessorMask
GetNumberOfConsoleInputEvents
GetOverlappedResult
GetPrivateProfileInt
GetPrivateProfileSectionNames
GetPrivateProfileStruct
GetProcessAffinityMask
GetProcessHeap
GetProcessId
GetProcessIoCounters
GetProcessShutdownParameters
GetProcessVersion

GetProcessWorkingSetSize
GetProcessWorkingSetSizeEx
GetProfileInt
GetProfileSection
GetProfileString
GetQueuedCompletionStatus
GetShortPathName
GetStartupInfo
GetStdHandle
GetSystemDirectory
GetSystemFirmwareTable
GetSystemInfo
GetSystemRegistryQuota
GetSystemTime
GetSystemTimeAdjustment
GetSystemTimeAsFileTime
GetSystemWindowsDirectory
GetSystemWow64Directory
GetTapeParameters
GetTapePosition
GetTapeStatus
GetTempFileName
GetTempPath
GetThreadContext
GetThreadId
GetThreadIOPendingFlag
GetThreadPriority
GetThreadPriorityBoost
GetThreadSelectorEntry
GetThreadTimes
GetTickCount
GetTimeZoneInformation
GetVersion
GetVersionEx
GetVolumeInformation
GetVolumeNameForVolumeMountPoint
GetVolumePathName
GetVolumePathNamesForVolumeName
GetWindowsDirectory
GetWriteWatch
GlobalAddAtom
GlobalAlloc
GlobalCompact
GlobalDeleteAtom
GlobalFindAtom
GlobalFix
GlobalFlags
GlobalFree
GlobalGetAtomName
GlobalHandle
GlobalLock
GlobalMemoryStatus
GlobalMemoryStatusEx
GlobalReAlloc
GlobalSize
GlobalUnfix
GlobalUnlock
GlobalUnWire
GlobalWire
HeapAlloc
HeapCompact
HeapCreate
HeapDestroy
HeapFree
HeapLock
HeapQueryInformation
HeapReAlloc
HeapSetInformation
HeapSize
HeapUnlock

HeapValidate
HeapWalk
InitAtomTable
InitializeCriticalSection
InitializeCriticalSectionAndSpinCount
InitializeSListHead
InterlockedCompareExchange
InterlockedCompareExchange64
InterlockedDecrement
InterlockedExchange
InterlockedExchangeAdd
InterlockedFlushSList
InterlockedIncrement
InterlockedPopEntrySList
InterlockedPushEntrySList
IsBadCodePtr
IsBadHugeReadPtr
IsBadHugeWritePtr
IsBadReadPtr
IsBadStringPtr
IsBadWritePtr
IsDebuggerPresent
IsProcessInJob
IsProcessorFeaturePresent
IsSystemResumeAutomatic
LeaveCriticalSection
LoadLibrary
LoadLibraryEx
LoadModule
LoadResource
LocalAlloc
LocalCompact
LocalFileTimeToFileTime
LocalFlags
LocalFree
LocalHandle
LocalLock
LocalReAlloc
LocalShrink
LocalSize
LocalUnlock
LockFile
LockFileEx
LockResource
lstrcat
lstrcmp
lstrcmpi
lstrcpy
lstrcpyn
lstrlen
MapUserPhysicalPages
MapUserPhysicalPagesScatter
MapViewOfFile
MapViewOfFileEx
MoveFile
MoveFileEx
MoveFileWithProgress
MulDiv
NeedCurrentDirectoryForExePath
OpenEvent
OpenFile
OpenFileMapping
OpenJobObject
OpenMutex
OpenProcess
OpenSemaphore
OpenThread
OpenWaitableTimer
OutputDebugString
PeekConsoleInput

PeekNamedPipe
PrepareTape
PulseEvent
QueryActCtxW
QueryDosDevice
QueryMemoryResourceNotification
QueryPerformanceFrequency
QueueUserWorkItem
ReadConsole
ReadConsoleOutput
ReadConsoleOutputCharacter
ReadFile
ReadFileScatter
RegisterWaitForSingleObject
ReleaseActCtx
ReleaseSemaphore
RemoveVectoredContinueHandler
ReOpenFile
RequestDeviceWakeup
ResetEvent
RestoreLastError
ScrollConsoleScreenBuffer
SetCommBreak
SetCommMask
SetCommTimeouts
SetComputerNameEx
SetConsoleCP
SetConsoleCursorInfo
SetConsoleMode
SetConsoleScreenBufferSize
SetConsoleTitle
SetCriticalSectionSpinCount
SetDefaultCommConfig
SetEndOfFile
SetEnvironmentVariable
PostQueuedCompletionStatus
ProcessIdToSessionId
PurgeComm
QueryDepthSList
QueryInformationJobObject
QueryPerformanceCounter
QueueUserAPC
RaiseException
ReadConsoleInput
ReadConsoleOutputAttribute
ReadDirectoryChangesW
ReadFileEx
ReadProcessMemory
RegisterWaitForSingleObjectEx
ReleaseMutex
RemoveDirectory
RemoveVectoredExceptionHandler
ReplaceFile
RequestWakeupLatency
ResetWriteWatch
ResumeThread
SearchPath
SetCommConfig
SetCommState
SetComputerName
SetConsoleActiveScreenBuffer
SetConsoleCtrlHandler
SetConsoleCursorPosition
SetConsoleOutputCP
SetConsoleTextAttribute
SetConsoleWindowInfo
SetCurrentDirectory
SetDllDirectory
SetEnvironmentStrings
SetErrorMode

SetEvent
SetFileApisToOEM
SetFilePointer
SetFileShortName
SetFileValidData
SetHandleCount
SetInformationJobObject
SetLocalTime
SetMessageWaitingIndicator
SetPriorityClass
SetProcessPriorityBoost
SetProcessWorkingSetSize
SetStdHandle
SetSystemTimeAdjustment
SetTapePosition
SetThreadContext
SetThreadIdealProcessor
SetThreadPriorityBoost
SetTimerQueueTimer
SetUnhandledExceptionFilter
SetVolumeLabel
SetWaitableTimer
SizeofResource
SleepEx
SwitchToFiber
SystemTimeToFileTime
TerminateJobObject
TerminateThread
TlsFree
TlsSetValue
TransmitCommChar
TzSpecificLocalTimeToSystemTime
UnlockFile
UnmapViewOfFile
UnregisterWaitEx
SetFileApisToANSI
SetFileAttributes
SetFilePointerEx
SetFileTime
SetFirmwareEnvironmentVariable
SetHandleInformation
SetLastError
SetMailslotInfo
SetNamedPipeHandleState
SetProcessAffinityMask
SetProcessShutdownParameters
SetProcessWorkingSetSizeEx
SetSystemTime
SetTapeParameters
SetThreadAffinityMask
SetThreadExecutionState
SetThreadPriority
SetThreadStackGuarantee
SetTimeZoneInformation
SetupComm
SetVolumeMountPoint
SignalObjectAndWait
Sleep
SuspendThread
SwitchToThread
SystemTimeToTzSpecificLocalTime
TerminateProcess
TlsAlloc
TlsGetValue
TransactNamedPipe
TryEnterCriticalSection
UnhandledExceptionFilter
UnlockFileEx
UnregisterWait
UpdateResource

VerifyVersionInfo
VirtualAlloc
VirtualAllocEx
VirtualFree
VirtualFreeEx
VirtualLock
VirtualProtect
VirtualProtectEx
VirtualQuery
VirtualQueryEx
VirtualUnlock
WaitCommEvent
WaitForDebugEvent
WaitForMultipleObjects
WaitForMultipleObjectsEx
WaitForSingleObject
WaitForSingleObjectEx
WaitNamedPipe
WinExec
Wow64DisableWow64FsRedirection
Wow64EnableWow64FsRedirection
Wow64RevertWow64FsRedirection
WriteConsole
WriteConsoleInput
WriteConsoleOutput
WriteConsoleOutputAttribute
WriteConsoleOutputCharacter
WriteFile
WriteFileEx
WriteFileGather
WritePrivateProfileSection
WritePrivateProfileString
WritePrivateProfileStruct
WriteProcessMemory
WriteProfileSection
WriteProfileString
WriteTapemark
WTSGetActiveConsoleSessionId
ZombifyActCtx
_hread
_hwrite
_lclose
_lcreat
_llseek
_lopen
_lread
_lwrite

shell32

These are the functions that `shell32` includes:

DoEnvironmentSubst
ShellExecuteEx
DragAcceptFiles
Shell_NotifyIcon
DragFinish
SHEmptyRecycleBin
DragQueryFile
SHFileOperation
DragQueryPoint
SHFreeNameMappings
DuplicateIcon
SHGetDiskFreeSpaceEx
ExtractAssociatedIcon
SHGetFileInfo
ExtractIcon
SHGetNewLinkInfo
ExtractIconEx
SHInvokePrinterCommand

FindExecutable
IsLFNDrive
SHAppBarMessage
SHCreateProcessAsUserW
ShellAbout
ShellExecute
SHIsFileAvailableOffline
SHLoadNonloadedIconOverlayIdentifiers
SHQueryRecycleBin
SHSetLocalizedName
WinExecError

user32

These are the functions that `user32` includes:

ActivateKeyboardLayout
AdjustWindowRect
AdjustWindowRectEx
AllowSetForegroundWindow
AnimateWindow
AnyPopup
AppendMenu
ArrangeIconicWindows
AttachThreadInput
BeginDeferWindowPos
BeginPaint
BringWindowToTop
BroadcastSystemMessage
BroadcastSystemMessageEx
CallMsgFilter
CallNextHookEx
CallWindowProc
CascadeWindows
ChangeClipboardChain
ChangeDisplaySettings
ChangeDisplaySettingsEx
ChangeMenu
CharLower
CharLowerBuff
CharNext
CharNextEx
CharPrev
CharPrevEx
CharToOem
CharToOemBuff
CharUpper
CharUpperBuff
CheckDlgButton
CheckMenuItem
CheckMenuRadioItem
CheckRadioButton
ChildWindowFromPoint
ChildWindowFromPointEx
ClientToScreen
ClipCursor
CloseClipboard
CloseDesktop
CloseWindow
CloseWindowStation
CopyAcceleratorTable
CopyCursor
CopyIcon
CopyImage
CopyRect
CountClipboardFormats
CreateAcceleratorTable
CreateCaret
CreateCursor
CreateDesktop
CreateDialogIndirectParam
CreateDialogParam
CreateIcon
CreateIconFromResource
CreateIconFromResourceEx
CreateIconIndirect
CreateMDIWindow
CreateMenu
CreatePopupMenu
CreateWindow
CreateWindowEx
CreateWindowStation
DeferWindowPos
DefFrameProc
DefMDIChildProc
DefRawInputProc
DefWindowProc
DeleteMenu
DeregisterShellHookWindow
DestroyAcceleratorTable
DestroyCaret
DestroyCursor
DestroyIcon
DestroyMenu

DestroyWindow
DialogBoxIndirectParam
DialogBoxParam1
DialogBoxParam2
DisableProcessWindowsGhosting
DispatchMessage
DlgDirList
DlgDirListComboBox
DlgDirSelectComboBoxEx
DlgDirSelectEx
DragDetect
DragObject
DrawAnimatedRects
DrawCaption
DrawEdge
DrawFocusRect
DrawFrameControl
DrawIcon
DrawIconIndirect
DrawMenuBar
DrawState
DrawText
DrawTextEx
EmptyClipboard
EnableMenuItem
EnableScrollBar
EnableWindow
EndDeferWindowPos
EndDialog
EndMenu
EndPaint
EndTask
EnumChildWindows
EnumClipboardFormats
EnumDesktops
EnumDesktopWindows
EnumDisplayDevices
EnumDisplayMonitors
EnumDisplaySettings
EnumDisplaySettingsEx
EnumProps
EnumPropsEx
EnumThreadWindows
EnumWindows
EnumWindowStations
EqualRect
ExcludeUpdateRgn
ExitWindowsEx
FillRect
FindWindow
FindWindowEx
FlashWindow
FlashWindowEx
FrameRect
GetActiveWindow
GetAltTabInfo
GetAncestor
GetAsyncKeyState
GetCapture
GetCaretBlinkTime
GetCaretPos
GetClassInfo
GetClassInfoEx
GetClassLong
GetClassLongPtr
GetClassName
GetClassWord
GetClientRect
GetClipboardData
GetClipboardFormatName
GetClipboardOwner
GetClipboardSequenceNumber
GetClipboardViewer
GetClipCursor
GetComboBoxInfo
GetCursor
GetCursorInfo
GetCursorPos
GetDC
GetDCEx
GetDesktopWindow
GetDialogBaseUnits
GetDlgCtrlID
GetDlgItem
GetDlgItemInt
GetDlgItemText
GetDoubleClickTime
GetFocus
GetForegroundWindow
GetGuiResources
GetGUIThreadInfo
GetIconInfo
GetInputState
GetKBCodePage
GetKeyboardLayout
GetKeyboardLayoutList
GetKeyboardLayoutName
GetKeyboardState
GetKeyboardType
GetKeyNameText
GetKeyState
GetLastActivePopup
GetLastInputInfo
GetLayeredWindowAttributes
GetListBoxInfo

GetMenu
GetMenuBarInfo
GetMenuCheckMarkDimensions
GetMenuContextHelpId
GetMenuDefaultItem
GetMenuInfo
GetMenuItemCount
GetMenuItemID
GetMenuItemInfo
GetMenuItemRect
GetMenuState
GetMenuString
GetMessage
GetMessageExtraInfo
GetMessagePos
GetMessageTime
GetMonitorInfo
GetMouseMovePointsEx
GetNextDlgGroupItem
GetNextDlgTabItem
GetOpenClipboardWindow
GetParent
GetPriorityClipboardFormat
GetProcessDefaultLayout
GetProcessWindowStation
GetProp
GetQueueStatus
GetRawInputBuffer
GetRawInputData
GetRawInputDeviceInfo
GetRawInputDeviceList
GetRegisteredRawInputDevices
GetScrollBarInfo
GetScrollInfo
GetScrollPos
GetScrollRange
GetShellWindow
GetSubMenu
GetSysColor
GetSysColorBrush
GetSystemMenu
GetSystemMetrics
GetTabbedTextExtent
GetThreadDesktop
GetTitleBarInfo
GetTopWindow
GetUpdateRect
GetUpdateRgn
GetUserObjectInformation
GetUserObjectSecurity
GetWindow
GetWindowContextHelpId
GetWindowDC
GetWindowInfo
GetWindowLong
GetWindowLongPtr
GetWindowModuleFileName
GetWindowPlacement
GetWindowRect
GetWindowRgn
GetWindowRgnBox
GetWindowText
GetWindowTextLength
GetWindowThreadProcessId
GetWindowWord
GrayString
HideCaret
HiliteMenuItem
InflateRect
InSendMessage
InSendMessageEx
InsertMenu
InsertMenuItem
InternalGetWindowText
IntersectRect
InvalidateRect
InvalidateRgn
InvertRect
IsCharAlpha
IsCharAlphaNumeric
IsCharLower
IsCharUpper
IsChild
IsClipboardFormatAvailable
IsDialogMessage
IsDlgButtonChecked
IsGUIThread
IsHungAppWindow
IsIconic
IsMenu
IsRectEmpty
IsWindow
IsWindowEnabled
IsWindowUnicode
IsWindowVisible
IsWinEventHookInstalled
IsWow64Message
IsZoomed
keybd_event
KillTimer
LoadAccelerators
LoadBitmap
LoadCursor1
LoadCursor2
LoadCursorFromFile

LoadIcon1
LoadIcon2
LoadImage
LoadKeyboardLayout
LoadMenu1
LoadMenu2
LoadMenuIndirect
LoadString
LockSetForegroundWindow
LockWindowUpdate
LockWorkStation
LookupIconIdFromDirectory
LookupIconIdFromDirectoryEx
LRESULT
MapDialogRect
MapVirtualKey
MapVirtualKeyEx
MapWindowPoints
MenuItemFromPoint
MessageBeep
MessageBox
MessageBoxEx
MessageBoxIndirect
ModifyMenu1
ModifyMenu2
MonitorFromPoint
MonitorFromRect
MonitorFromWindow
mouse_event
MoveWindow
MsgWaitForMultipleObjects
MsgWaitForMultipleObjectsEx
NotifyWinEvent
OemKeyScan
OemToChar
OemToCharBuff
OffsetRect
OpenClipboard
OpenDesktop
OpenIcon
OpenInputDesktop
OpenWindowStation
PaintDesktop
PeekMessage
PostMessage
PostQuitMessage
PostThreadMessage
PrintWindow
PrivateExtractIcons
PtInRect
RealChildWindowFromPoint
RealGetWindowClass
RedrawWindow
RegisterClass
RegisterClassEx
RegisterClipboardFormat
RegisterDeviceNotification
RegisterHotKey
RegisterRawInputDevices
RegisterShellHookWindow
RegisterWindowMessage
ReleaseCapture
ReleaseDC
RemoveMenu
RemoveProp
ReplyMessage
ScreenToClient
ScrollDC
ScrollWindow
ScrollWindowEx
SendDlgItemMessage
SendInput
SendMessage
SendMessageCallback
SendMessageTimeout
SendNotifyMessage
SetActiveWindow
SetCapture
SetCaretBlinkTime
SetCaretPos
SetClassLong
SetClassLongPtr
SetClassWord
SetClipboardData
SetClipboardViewer
SetCursor
SetCursorPos
SetDebugErrorLevel
SetDlgItemInt
SetDlgItemText
SetDoubleClickTime
SetFocus
SetForegroundWindow
SetKeyboardState
SetLastErrorEx
SetLayeredWindowAttributes
SetMenu
SetMenuContextHelpId
SetMenuDefaultItem
SetMenuInfo
SetMenuItemBitmaps
SetMenuItemInfo
SetMessageExtraInfo
SetMessageQueue
SetParent

SetProcessDefaultLayout
SetProcessWindowStation
SetProp
SetRect
SetRectEmpty
SetScrollInfo
SetScrollPos
SetScrollRange
SetSysColors
SetSystemCursor
SetThreadDesktop
SetTimer
SetUserObjectInformation
SetUserObjectSecurity
SetWindowContextHelpId
SetWindowLong
SetWindowLongPtr
SetWindowPlacement
SetWindowPos
SetWindowRgn
SetWindowsHook
SetWindowsHookEx
SetWindowText
SetWindowWord
SetWinEventHook
ShowCaret
ShowCursor
ShowOwnedPopups
ShowScrollBar
ShowWindow
ShowWindowAsync
SubtractRect
SwapMouseButton
SwitchDesktop
SwitchToThisWindow
SystemParametersInfo
TabbedTextOut
TileWindows
ToAscii
ToAsciiEx
ToUnicode
ToUnicodeEx
TrackMouseEvent
TrackPopupMenu
TrackPopupMenuEx
TranslateAccelerator
TranslateMDISysAccel
TranslateMessage
UnhookWindowsHook
UnhookWindowsHookEx
UnhookWinEvent
UnionRect
UnloadKeyboardLayout
UnregisterClass
UnregisterDeviceNotification
UnregisterHotKey
UpdateLayeredWindow
UpdateLayeredWindowIndirect
UpdateWindow
UserHandleGrantAccess
ValidateRect
ValidateRgn
VkKeyScan
VkKeyScanEx
WaitForInputIdle
WaitMessage
WindowFromDC
WindowFromPoint
WinHelp
wsprintf
wvsprintf

winver

These are the functions that `winver` includes:

GetFileVersionInfo
GetFileVersionInfoSize
VerFindFile
VerInstallFile
VerLanguageName
VerQueryValue

wsock32

These are the functions that wsock32 includes:

accept	AcceptEx	bind
closesocket	connect	GetAcceptExSockaddrs
getpeername	gethostname	getprotobyname
getprotobynumber	getservbyname	getservbyport
getsockname	getsockopt	htonl
htons	inet_addr	inet_ntoa
ioctlsocket	listen	ntohl
ntohs	recv	select
send	sendto	setsockopt
shutdown	socket	TransmitFile
WSAAsyncGetHostByName	WSAAsyncGetProtoByName	WSAAsyncGetProtoByNumber
WSAAsyncGetServByName	WSAAsyncGetServByPort	WSAAsyncSelect
WSACancelAsyncRequest	WSACancelBlockingCall	WSACleanup
WSAGetLastError	WSAIsBlocking	WSARecvEx
WSASetBlockingHook	WSASetLastError	WSAStartup

Chapter 10. C/C++ MMX/SSE Inline Intrinsics

An intrinsic is a function available in a given language whose implementation is handled specifically by the compiler. Typically, an intrinsic substitutes a sequence of automatically-generated instructions for the original function call. Since the compiler has an intimate knowledge of the intrinsic function, it can better integrate it and optimize it for the situation.

PGI provides support for MMX (and SSE/SSE2/SSE3/SSSE3/SSE4a/ABM intrinsics in C/C++ programs.

Intrinsics make the use of processor-specific enhancements easier because they provide a C/C++ language interface to assembly instructions. In doing so, the compiler manages things that the user would normally have to be concerned with, such as register names, register allocations, and memory locations of data.

This chapter contains these seven tables associated with inline intrinsics:

- A table of MMX inline intrinsics (mmintrin.h)
- A table of SSE inline intrinsics (xmmintrin.h)
- A table of SSE2 inline intrinsics (emmintrin.h)
- A table of SSE3 inline intrinsics (pmmintrin.h)
- A table of SSSE3 inline intrinsics (tmmintrin.h)
- A table of SSE4a inline intrinsics (ammintrin.h)
- A table of ABM inline intrinsics (intrin.h)

Using Intrinsic functions

The definitions of the intrinsics are provided in the inline library `libintrinsics.il`, which is automatically included when you compile.

Required Header File

To call these intrinsic functions from a C/C++ source, you must include the corresponding header file - one of the following:

- For MMX, use `mmintrin.h`
- For SSE, use `xmmintrin.h`
- For SSE2, use `emmintrin.h`
- For SSE3, use `pmmintrin.h`
- For SSSE3 use `tmmintrin.h`
- For SSE4a use `ammintrin.h`
- For ABM use `intrin.h`

Intrinsic Data Types

The following table describes the data types that are defined for intrinsics:

Data Types	Defined in	Description
__m64	`mmintrin.h`	For use with MMX intrinsics, this 64-bit data type stores one 64-bit or two 32-bit integer values.
__m128	`xmmintrin.h`	For use with SSE intrinsics, this 128-bit data type, aligned on 16-byte boundaries, stores four single-precision floating point values.
__m128d	`emmintrin.h`	For use with SSE2/SSE3 intrinsics, this 128-bit data type, aligned on 16-byte boundaries, stores two double-precision floating point values.
__m128i	`emmintrin.h`	For use with SSE2/SSE3 intrinsics, this 128-bit data type, aligned on 16-byte boundaries, stores two 64-bit integer values.

Intrinsic Example

The MMX/SSE intrinsics include functions for initializing variables of the types defined in the preceding table. The following sample program, `example.c`, illustrates the use of the SSE intrinsics _mm_add_ps and _mm_set_ps.

```
#include<xmmintrin.h>
 int main(){
 __m128 A, B, result;
 A = _mm_set_ps(23.3, 43.7, 234.234, 98.746); /* initialize A */
 B = _mm_set_ps(15.4, 34.3, 4.1, 8.6); /* initialize B */
 result = _mm_add_ps(A, B);
 return 0;
 }
```

To compile this program, use the following command:

```
$ pgcc example.c -o myprog
```

The inline library `libintrinsics.il` is automatically inlined.

MMX Intrinsics

PGI supports a set of MMX Intrinsics which allow the use of the MMX instructions directly from C/C++ code, without writing the assembly instructions. The following table lists the MMX intrinsics that PGI supports.

Note

Intrinsics with a * are only available on 64-bit systems.

Table 10.1. MMX Intrinsics (mmintrin.h)

_mm_empty	_m_paddd	_m_psllw	_m_pand
_m_empty	_mm_add_si64	_mm_slli_pi16	_mm_andnot_si64
_mm_cvtsi32_si64	_mm_adds_pi8	_m_psllwi	_m_pandn
_m_from_int	_m_paddsb	_mm_sll_pi32	_mm_or_si64
_mm_cvtsi64x_si64*	_mm_adds_pi16	_m_pslld	_m_por
_mm_set_pi64x*	_m_paddsw	_mm_slli_pi32	_mm_xor_si64
_mm_cvtsi64_si32	_mm_adds_pu8	_m_pslldi	_m_pxor
_m_to_int	_m_paddusb	_mm_sll_si64	_mm_cmpeq_pi8
_mm_cvtsi64_si64x*	_mm_adds_pu16	_m_psllq	_m_pcmpeqb
_mm_packs_pi16*	_m_paddusw	_mm_slli_si64	_mm_cmpgt_pi8
_m_packsswb	_mm_sub_pi8	_m_psllqi	_m_pcmpgtb
_mm_packs_pi32	_m_psubb	_mm_sra_pi16	_mm_cmpeq_pi16
_m_packssdw	_mm_sub_pi16	_m_psraw	_m_pcmpeqw
_mm_packs_pu16	_m_psubw	_mm_srai_pi16	_mm_cmpgt_pi16
_m_packuswb	_mm_sub_pi32	_m_psrawi	_m_pcmpgtw
_mm_unpackhi_pi8	_m_psubd	_mm_sra_pi32	_mm_cmpeq_pi32
_m_punpckhbw	_mm_sub_si64	_m_psrad	_m_pcmpeqd
_mm_unpackhi_pi16	_mm_subs_pi8	_mm_srai_pi32	_mm_cmpgt_pi32
_m_punpckhwd	_m_psubsb	_m_psradi	_m_pcmpgtd
_mm_unpackhi_pi32	_mm_subs_pi16	_mm_srl_pi16	_mm_setzero_si64
_m_punpckhdq	_m_psubsw	_m_psrlw	_mm_set_pi32
_mm_unpacklo_pi8	_mm_subs_pu8	_mm_srli_pi16	_mm_set_pi16
_m_punpcklbw	_m_psubusb	_m_psrlwi	_mm_set_pi8
_mm_unpacklo_pi16	_mm_subs_pu16	_mm_srl_pi32	_mm_setr_pi32
_m_punpcklwd	_m_psubusw	_m_psrld	_mm_setr_pi16
_mm_unpacklo_pi32	_mm_madd_pi16	_mm_srli_pi32	_mm_setr_pi8
_m_punpckldq	_m_pmaddwd	_m_psrldi	_mm_set1_pi32
_mm_add_pi8	_mm_mulhi_pi16	_mm_srl_si64	_mm_set1_pi16
_m_paddb	_m_pmulhw	_m_psrlq	_mm_set1_pi8
_mm_add_pi16	_mm_mullo_pi16	_mm_srli_si64	
_m_paddw	_m_pmullw	_m_psrlqi	
_mm_add_pi32	_mm_sll_pi16	_mm_and_si64	

SSE Intrinsics

PGI supports a set of SSE Intrinsics which allow the use of the SSE instructions directly from C/C++ code, without writing the assembly instructions. The following tables list the SSE intrinsics that PGI supports.

Note

Intrinsics with a * are only available on 64-bit systems.

Table 10.2. SSE Intrinsics (xmmintrin.h)

_mm_add_ss	_mm_comige_ss	_mm_load_ss
_mm_sub_ss	_mm_comineq_ss	_mm_load1_ps
_mm_mul_ss	_mm_ucomieq_ss	_mm_load_ps1
_mm_div_ss	_mm_ucomilt_ss	_mm_load_ps
_mm_sqrt_ss	_mm_ucomile_ss	_mm_loadu_ps
_mm_rcp_ss	_mm_ucomigt_ss	_mm_loadr_ps
_mm_rsqrt_ss	_mm_ucomige_ss	_mm_set_ss
_mm_min_ss	_mm_ucomineq_ss	_mm_set1_ps
_mm_max_ss	_mm_cvtss_si32	_mm_set_ps1
_mm_add_ps	_mm_cvt_ss2si	_mm_set_ps
_mm_sub_ps	_mm_cvtss_si64x*	_mm_setr_ps
_mm_mul_ps	_mm_cvtps_pi32	_mm_store_ss
_mm_div_ps	_mm_cvt_ps2pi	_mm_store_ps
_mm_sqrt_ps	_mm_cvttss_si32	_mm_store1_ps
_mm_rcp_ps	_mm_cvtt_ss2si	_mm_store_ps1
_mm_rsqrt_ps	_mm_cvttss_si64x*	_mm_storeu_ps
_mm_min_ps	_mm_cvttps_pi32	_mm_storer_ps
_mm_max_ps	_mm_cvtt_ps2pi	_mm_move_ss
_mm_and_ps	_mm_cvtsi32_ss	_mm_extract_pi16
_mm_andnot_ps	_mm_cvt_si2ss	_m_pextrw
_mm_or_ps	_mm_cvtsi64x_ss*	_mm_insert_pi16
_mm_xor_ps	_mm_cvtpi32_ps	_m_pinsrw
_mm_cmpeq_ss	_mm_cvt_pi2ps	_mm_max_pi16
_mm_cmplt_ss	_mm_movelh_ps	_m_pmaxsw
_mm_cmple_ss	_mm_setzero_ps	_mm_max_pu8
_mm_cmpgt_ss	_mm_cvtpi16_ps	_m_pmaxub
_mm_cmpge_ss	_mm_cvtpu16_ps	_mm_min_pi16
_mm_cmpneq_ss	_mm_cvtpi8_ps	_m_pminsw

_mm_cmpnlt_ss	_mm_cvtpu8_ps	_mm_min_pu8
_mm_cmpnle_ss	_mm_cvtpi32x2_ps	_m_pminub
_mm_cmpngt_ss	_mm_movehl_ps	_mm_movemask_pi8
_mm_cmpnge_ss	_mm_cvtps_pi16	_m_pmovmskb
_mm_cmpord_ss	_mm_cvtps_pi8	_mm_mulhi_pu16
_mm_cmpunord_ss	_mm_shuffle_ps	_m_pmulhuw
_mm_cmpeq_ps	_mm_unpackhi_ps	_mm_shuffle_pi16
_mm_cmplt_ps	_mm_unpacklo_ps	_m_pshufw
_mm_cmple_ps	_mm_loadh_pi	_mm_maskmove_si64
_mm_cmpgt_ps	_mm_storeh_pi	_m_maskmovq
_mm_cmpge_ps	_mm_loadl_pi	_mm_avg_pu8
_mm_cmpneq_ps	_mm_storel_pi	_m_pavgb
_mm_cmpnlt_ps	_mm_movemask_ps	_mm_avg_pu16
_mm_cmpnle_ps	_mm_getcsr	_m_pavgw
_mm_cmpngt_ps	_MM_GET_EXCEPTION_STATE	_mm_sad_pu8
_mm_cmpnge_ps	_MM_GET_EXCEPTION_MASK	_m_psadbw
_mm_cmpord_ps	_MM_GET_ROUNDING_MODE	_mm_prefetch
_mm_cmpunord_ps	_MM_GET_FLUSH_ZERO_MODE	_mm_stream_pi
_mm_comieq_ss	_mm_setcsr	_mm_stream_ps
_mm_comilt_ss	_MM_SET_EXCEPTION_STATE	_mm_sfence
_mm_comile_ss	_MM_SET_EXCEPTION_MASK	_mm_pause
_mm_comigt_ss	_MM_SET_ROUNDING_MODE	_MM_TRANSPOSE4_PS
	_MM_SET_FLUSH_ZERO_MODE	

Table 10.3 lists the SSE2 intrinsics that PGI supports and that are available in emmintrin.h.

Table 10.3. SSE2 Intrinsics (emmintrin.h)

_mm_load_sd	_mm_cmpge_sd	_mm_cvtps_pd	_mm_srl_epi32
_mm_load1_pd	_mm_cmpneq_sd	_mm_cvtsd_si32	_mm_srl_epi64
_mm_load_pd1	_mm_cmpnlt_sd	_mm_cvtsd_si64x*	_mm_slli_epi16
_mm_load_pd	_mm_cmpnle_sd	_mm_cvttsd_si32	_mm_slli_epi32
_mm_loadu_pd	_mm_cmpngt_sd	_mm_cvttsd_si64x*	_mm_slli_epi64
_mm_loadr_pd	_mm_cmpnge_sd	_mm_cvtsd_ss	_mm_srai_epi16
_mm_set_sd	_mm_cmpord_sd	_mm_cvtsi32_sd	_mm_srai_epi32
_mm_set1_pd	_mm_cmpunord_sd	_mm_cvtsi64x_sd*	_mm_srli_epi16
_mm_set_pd1	_mm_comieq_sd	_mm_cvtss_sd	_mm_srli_epi32
_mm_set_pd	_mm_comilt_sd	_mm_unpackhi_pd	_mm_srli_epi64

_mm_setr_pd
_mm_setzero_pd
_mm_store_sd
_mm_store_pd
_mm_store1_pd
_mm_store_pd1
_mm_storeu_pd
_mm_storer_pd
_mm_move_sd
_mm_add_pd
_mm_add_sd
_mm_sub_pd
_mm_sub_sd
_mm_mul_pd
_mm_mul_sd
_mm_div_pd
_mm_div_sd
_mm_sqrt_pd
_mm_sqrt_sd
_mm_min_pd
_mm_min_sd
_mm_max_pd
_mm_max_sd
_mm_and_pd
_mm_andnot_pd
_mm_or_pd
_mm_xor_pd
_mm_cmpeq_pd
_mm_cmplt_pd
_mm_cmple_pd
_mm_cmpgt_pd
_mm_cmpge_pd
_mm_cmpneq_pd
_mm_cmpnlt_pd
_mm_cmpnle_pd

_mm_comile_sd
_mm_comigt_sd
_mm_comige_sd
_mm_comineq_sd
_mm_ucomieq_sd
_mm_ucomilt_sd
_mm_ucomile_sd
_mm_ucomigt_sd
_mm_ucomige_sd
_mm_ucomineq_sd
_mm_load_si128
_mm_loadu_si128
_mm_loadl_epi64
_mm_store_si128
_mm_storeu_si128
_mm_storel_epi64
_mm_movepi64_pi64
_mm_move_epi64
_mm_setzero_si128
_mm_set_epi64
_mm_set_epi32
_mm_set_epi64x*
_mm_set_epi16
_mm_set_epi8
_mm_set1_epi64
_mm_set1_epi32
_mm_set1_epi64x*
_mm_set1_epi16
_mm_set1_epi8
_mm_setr_epi64
_mm_setr_epi32
_mm_setr_epi16
_mm_setr_epi8
_mm_cvtepi32_pd
_mm_cvtepi32_ps

_mm_unpacklo_pd
_mm_loadh_pd
_mm_storeh_pd
_mm_loadl_pd
_mm_storel_pd
_mm_movemask_pd
_mm_packs_epi16
_mm_packs_epi32
_mm_packus_epi16
_mm_unpackhi_epi8
_mm_unpackhi_epi16
_mm_unpackhi_epi32
_mm_unpackhi_epi64
_mm_unpacklo_epi8
_mm_unpacklo_epi16
_mm_unpacklo_epi32
_mm_unpacklo_epi64
_mm_add_epi8
_mm_add_epi16
_mm_add_epi32
_mm_add_epi64
_mm_adds_epi8
_mm_adds_epi16
_mm_adds_epu8
_mm_adds_epu16
_mm_sub_epi8
_mm_sub_epi16
_mm_sub_epi32
_mm_sub_epi64
_mm_subs_epi8
_mm_subs_epi16
_mm_subs_epu8
_mm_subs_epu16
_mm_madd_epi16
_mm_mulhi_epi16

_mm_and_si128
_mm_andnot_si128
_mm_or_si128
_mm_xor_si128
_mm_cmpeq_epi8
_mm_cmpeq_epi16
_mm_cmpeq_epi32
_mm_cmplt_epi8
_mm_cmplt_epi16
_mm_cmplt_epi32
_mm_cmpgt_epi8
_mm_cmpgt_epi16
_mm_srl_epi16
_mm_cmpgt_epi32
_mm_max_epi16
_mm_max_epu8
_mm_min_epi16
_mm_min_epu8
_mm_movemask_epi8
_mm_mulhi_epu16
_mm_maskmoveu_si128
_mm_avg_epu8
_mm_avg_epu16
_mm_sad_epu8
_mm_stream_si32
_mm_stream_si128
_mm_stream_pd
_mm_movpi64_epi64
_mm_lfence
_mm_mfence
_mm_cvtsi32_si128
_mm_cvtsi64x_si128*
_mm_cvtsi128_si32
_mm_cvtsi128_si64x*
_mm_srli_si128

_mm_cmpngt_pd	_mm_cvtpd_epi32	_mm_mullo_epi16	_mm_slli_si128
_mm_cmpnge_pd	_mm_cvtpd_pi32	_mm_mul_su32	_mm_shuffle_pd
_mm_cmpord_pd	_mm_cvtpd_ps	_mm_mul_epu32	_mm_shufflehi_epi16
_mm_cmpunord_pd	_mm_cvttpd_epi32	_mm_sll_epi16	_mm_shufflelo_epi16
_mm_cmpeq_sd	_mm_cvttpd_pi32	_mm_sll_epi32	_mm_shuffle_epi32
_mm_cmplt_sd	_mm_cvtpi32_pd	_mm_sll_epi64	_mm_extract_epi16
_mm_cmple_sd	_mm_cvtps_epi32	_mm_sra_epi16	_mm_insert_epi16
_mm_cmpgt_sd	_mm_cvttps_epi32	_mm_sra_epi32	

Table 10.4 lists the SSE3 intrinsics that PGI supports and that are available in pmmintrin.h.

Table 10.4. SSE3 Intrinsics (pmmintrin.h)

_mm_addsub_ps	_mm_moveldup_ps	_mm_loaddup_pd	_mm_mwait
_mm_hadd_ps	_mm_addsub_pd	_mm_movedup_pd	
_mm_hsub_ps	_mm_hadd_pd	_mm_lddqu_si128	
_mm_movehdup_ps	_mm_hsub_pd	_mm_monitor	

Table 10.5 lists the SSSE3 intrinsics that PGI supports and that are available in tmmintrin.h.

Table 10.5. SSSE3 Intrinsics (tmmintrin.h)

_mm_hadd_epi16	_mm_hsubs_pi16	_mm_sign_pi16
_mm_hadd_epi32	_mm_maddubs_epi16	_mm_sign_pi32
_mm_hadds_epi16	_mm_maddubs_pi16	_mm_alignr_epi8
_mm_hadd_pi16	_mm_mulhrs_epi16	_mm_alignr_pi8
_mm_hadd_pi32	_mm_mulhrs_pi16	_mm_abs_epi8
_mm_hadds_pi16	_mm_shuffle_epi8	_mm_abs_epi16
_mm_hsub_epi16	_mm_shuffle_pi8	_mm_abs_epi32
_mm_hsub_epi32	_mm_sign_epi8	_mm_abs_pi8
_mm_hsubs_epi16	_mm_sign_epi16	_mm_abs_pi16
_mm_hsub_pi16	_mm_sign_epi32	_mm_abs_pi32
_mm_hsub_pi32	_mm_sign_pi8	

Table 10.6 lists the SSE4a intrinsics that PGI supports and that are available in ammintrin.h.

Table 10.6. SSE4a Intrinsics (ammintrin.h)

_mm_stream_sd	_mm_extract_si64	_mm_insert_si64
_mm_stream_ss	_mm_extracti_si64	_mm_inserti_si64

ABM Intrinsics

PGI supports a set of ABM Intrinsics which allow the use of the ABM instructions directly from C/C++ code, without writing the assembly instructions. The following table lists the ABM intrinsics that PGI supports.

Table 10.7. ABM Intrinsics (intrin.h)

__lzcnt16	__lzcnt64	__popcnt	__rdtscp
__lzcnt	__popcnt16	__popcnt64	

Chapter 11. Messages

This chapter describes the various messages that the compiler produces. These messages include the sign-on message and diagnostic messages for remarks, warnings, and errors. The compiler always displays any error messages, along with the erroneous source line, on the screen. If you specify the –Mlist option, the compiler places any error messages in the listing file. You can also use the `-v` option to display more information about the compiler, assembler, and linker invocations and about the host system. For more information on the `-Mlist` and `-v` options, refer to "using Command Line Options" in the PGI Compiler User's Guide.

Diagnostic Messages

Diagnostic messages provide syntactic and semantic information about your source text. Syntactic information includes information such as syntax errors. Semantic information includes information such as unreachable code.

You can specify that the compiler displays error messages at a certain level with the *-Minform* option.

The compiler messages refer to a severity level, a message number, and the line number where the error occurs.

The compiler can also display internal error messages on standard error. If your compilation produces any internal errors, contact The Portland Group's technical reporting service by sending e-mail to *trs@pgroup.com*.

If you use the listing file option `-Mlist`, the compiler places diagnostic messages after the source lines in the listing file, in the following format:

```
PGFTN-etype-enum-message (filename: line)
```

Where:

etype
: is a character signifying the severity level

enum
: is the error number

message
is the error message

filename
is the source filename

line
is the line number where the compiler detected an error.

Phase Invocation Messages

You can display compiler, assembler, and linker phase invocations by using the `-v` command line option. For further information about this option, refer to "Using Command Line Options" in the PGI Compiler User's Guide.

Fortran Compiler Error Messages

This section presents the error messages generated by the PGF77, PGF95, and PGFORTRAN compilers. The compilers display error messages in the program listing and on standard output. They can also display internal error messages on standard error.

Message Format

Each message is numbered. Each message also lists the line and column number where the error occurs. A dollar sign ($) in a message represents information that is specific to each occurrence of the message.

Message List

Error message severities:

I
informative

W
warning

S
severe error

F
fatal error

V
variable

`V000 Internal compiler error. $ $`
This message indicates an error in the compiler, rather than a user error – although it may be possible for a user error to cause an internal error. The severity may vary; if it is informative or warning, correct object code was probably generated, but it is not safe to rely on this. Regardless of the severity or cause, internal errors should be reported to trs@pgroup.com.

F001 Source input file name not specified
On the command line, source file name should be specified either before all the switches, or after them.

F002 Unable to open source input file: $
Source file name is misspelled, file is not in current working directory, or file is read protected.

F003 Unable to open listing file
Probably, user does not have write permission for the current working directory.

F004 $ $
Generic message for file errors.

F005 Unable to open temporary file
Compiler uses directory "/usr/tmp" or "/tmp" in which to create temporary files. If neither of these directories is available on the node on which the compiler is being used, this error will occur.

S006 Input file empty
Source input file does not contain any Fortran statements other than comments or compiler directives.

F007 Subprogram too large to compile at this optimization level $
Internal compiler data structure overflow, working storage exhausted, or some other non-recoverable problem related to the size of the subprogram. If this error occurs at opt 2, reducing the opt level to 1 may work around the problem. Moving the subprogram being compiled to its own source file may eliminate the problem. If this error occurs while compiling a subprogram of fewer than 2000 statements it should be reported to the compiler maintenance group as a possible compiler problem.

F008 Error limit exceeded
The compiler gives up because too many severe errors were issued; the error limit can be reset on the command line.

F009 Unable to open assembly file
Probably, user does not have write permission for the current working directory.

F010 File write error occurred $
Probably, file system is full.

S011 Unrecognized command line switch: $
Refer to PDS reference document for list of allowed compiler switches.

S012 Value required for command line switch: $
Certain switches require an immediately following value, such as "-opt 2".

S013 Unrecognized value specified for command line switch: $

S014 Ambiguous command line switch: $
Too short an abbreviation was used for one of the switches.

W015 Hexadecimal or octal constant truncated to fit data type

I016 Identifier, $, truncated to 31 chars
An identifier may be at most 31 characters in length; characters after the 31st are ignored.

S017 Unable to open include file: $
File is missing, read protected, or maximum include depth (10) exceeded. Remember that the file name should be enclosed in quotes.

S018 Illegal label $ $
Used for label 'field' errors or illegal values. E.g., in fixed source form, the label field (first five characters) of the indicated line contains a non-numeric character.

S019 Illegally placed continuation line
A continuation line does not follow an initial line, or more than 99 continuation lines were specified.

S020 Unrecognized compiler directive
Refer to user's manual for list of allowed compiler directives.

S021 Label field of continuation line is not blank
The first five characters of a continuation line must be blank.

S022 Unexpected end of file - missing END statement

S023 Syntax error - unbalanced $
Unbalanced parentheses or brackets.

W024 CHARACTER or Hollerith constant truncated to fit data type
A character or hollerith constant was converted to a data type that was not large enough to contain all of the characters in the constant. This type conversion occurs when the constant is used in an arithmetic expression or is assigned to a non-character variable. The character or hollerith constant is truncated on the right, that is, if 4 characters are needed then the first 4 are used and the remaining characters are discarded.

W025 Illegal character ($) - ignored
The current line contains a character, possibly non-printing, which is not a legal Fortran character (characters inside of character or Hollerith constants cannot cause this error). As a general rule, all non-printing characters are treated as white space characters (blanks and tabs); no error message is generated when this occurs. If for some reason, a non-printing character is not treated as a white space character, its hex representation is printed in the form dd where each d is a hex digit.

S026 Unmatched quote

S027 Illegal integer constant: $
Integer constant is too large for 32 bit word.

S028 Illegal real or double precision constant: $

S029 Illegal $ constant: $
Illegal hexadecimal, octal, or binary constant. A hexadecimal constant consists of digits 0..9 and letters A..F or a..f; any other character in a hexadecimal constant is illegal. An octal constant consists of digits 0..7; any other digit or character in an octal constant is illegal. A binary constant consists of digits 0 or 1; any other digit or character in a binary constant is illegal.

S030 Explicit shape must be specified for $

S031 Illegal data type length specifier for $
The data type length specifier (e.g. 4 in INTEGER*4) is not a constant expression that is a member of the set of allowed values for this particular data type.

W032 Data type length specifier not allowed for $

The data type length specifier (e.g. 4 in INTEGER*4) is not allowed in the given syntax (e.g. DIMENSION A(10)*4).

S033 Illegal use of constant $

A constant was used in an illegal context, such as on the left side of an assignment statement or as the target of a data initialization statement.

S034 Syntax error at or near $

I035 Predefined intrinsic $ loses intrinsic property

An intrinsic name was used in a manner inconsistent with the language definition for that intrinsic. The compiler, based on the context, will treat the name as a variable or an external function.

S036 Illegal implicit character range

First character must alphabetically precede second.

S037 Contradictory data type specified for $

The indicated identifier appears in more than one type specification statement and different data types are specified for it.

S038 Symbol, $, has not been explicitly declared

The indicated identifier must be declared in a type statement; this is required when the IMPLICIT NONE statement occurs in the subprogram.

W039 Symbol, $, appears illegally in a SAVE statement $

An identifier appearing in a SAVE statement must be a local variable or array.

S040 Illegal common variable $

Indicated identifier is a dummy variable, is already in a common block, or has previously been defined to be something other than a variable or array.

W041 Illegal use of dummy argument $

This error can occur in several situations. It can occur if dummy arguments were specified on a PROGRAM statement. It can also occur if a dummy argument name occurs in a DATA, COMMON, SAVE, or EQUIVALENCE statement. A program statement must have an empty argument list.

S042 $ is a duplicate dummy argument

S043 Illegal attempt to redefine $ $

An attempt was made to define a symbol in a manner inconsistent with an earlier definition of the same symbol. This can happen for a number of reasons. The message attempts to indicate the situation that occurred.

intrinsic - An attempt was made to redefine an intrinsic function. A symbol that represents an intrinsic function may be redefined if that symbol has not been previously verified to be an intrinsic function. For example, the intrinsic `sin` can be defined to be an integer array. If a symbol is verified to be an intrinsic function via the INTRINSIC statement or via an intrinsic function reference then it must be referred to as an intrinsic function for the remainder of the program unit.

symbol - An attempt was made to redefine a symbol that was previously defined. An example of this is to declare a symbol to be a PARAMETER which was previously declared to be a subprogram argument.

S044 Multiple declaration for symbol $
A redundant declaration of a symbol has occurred. For example, an attempt was made to declare a symbol as an ENTRY when that symbol was previously declared as an ENTRY.

S045 Data type of entry point $ disagrees with function $
The current function has entry points with data types inconsistent with the data type of the current function. For example, the function returns type character and an entry point returns type complex.

S046 Data type length specifier in wrong position
The CHARACTER data type specifier has a different position for the length specifier from the other data types. Suppose, we want to declare arrays ARRAYA and ARRAYB to have 8 elements each having an element length of 4 bytes. The difference is that ARRAYA is character and ARRAYB is integer. The declarations would be CHARACTER ARRAYA(8)*4 and INTEGER ARRAYB*4(8).

S047 More than seven dimensions specified for array

S048 Illegal use of '*' in declaration of array $
An asterisk may be used only as the upper bound of the last dimension.

S049 Illegal use of '*' in non-subroutine subprogram
The alternate return specifier '*' is legal only in the subroutine statement. Programs, functions, and block data are not allowed to have alternate return specifiers.

S050 Assumed size array, $, is not a dummy argument

S051 Unrecognized built-in % function
The allowable built-in functions are %VAL, %REF, %LOC, and %FILL. One was encountered that did not match one of these allowed forms.

S052 Illegal argument to %VAL or %LOC

S053 %REF or %VAL not legal in this context
The built-in functions %REF and %VAL can only be used as actual parameters in procedure calls.

W054 Implicit character $ used in a previous implicit statement
An implicit character has been given an implied data type more than once. The implied data type for the implicit character is changed anyway.

W055 Multiple implicit none statements
The IMPLICIT NONE statement can occur only once in a subprogram.

W056 Implicit type declaration
The -Mdclchk switch and an implicit declaration following an IMPLICIT NONE statement will produce a warning message for IMPLICIT statements.

S057 Illegal equivalence of dummy variable, $
Dummy arguments may not appear in EQUIVALENCE statements.

S058 Equivalenced variables $ and $ not in same common block
A common block variable must not be equivalenced with a variable in another common block.

S059 Conflicting equivalence between $ and $
The indicated equivalence implies a storage layout inconsistent with other equivalences.

S060 Illegal equivalence of structure variable, $

STRUCTURE and UNION variables may not appear in EQUIVALENCE statements.

S061 Equivalence of $ and $ extends common block backwards

W062 Equivalence forces $ to be unaligned

EQUIVALENCE statements have defined an address for the variable which has an alignment not optimal for variables of its data type. This can occur when INTEGER and CHARACTER data are equivalenced, for instance.

I063 Gap in common block $ before $

S064 Illegal use of $ in DATA statement implied DO loop

The indicated variable is referenced where it is not an active implied DO index variable.

S065 Repeat factor less than zero

S066 Too few data constants in initialization statement

S067 Too many data constants in initialization statement

S068 Numeric initializer for CHARACTER $ out of range 0 through 255

A CHARACTER*1 variable or character array element can be initialized to an integer, octal, or hexadecimal constant if that constant is in the range 0 through 255.

S069 Illegal implied DO expression

The only operations allowed within an implied DO expression are integer +, -, *, and /.

S070 Incorrect sequence of statements $

The statement order is incorrect. For instance, an IMPLICIT NONE statement must precede a specification statement which in turn must precede an executable statement.

S071 Executable statements not allowed in block data

S072 Assignment operation illegal to $ $

The destination of an assignment operation must be a variable, array reference, or vector reference. The assignment operation may be by way of an assignment statement, a data statement, or the index variable of an implied DO-loop. The compiler has determined that the identifier used as the destination is not a storage location. The error message attempts to indicate the type of entity used.

entry point - An assignment to an entry point that was not a function procedure was attempted.

external procedure - An assignment to an external procedure or a Fortran intrinsic name was attempted. If the identifier is the name of an entry point that is not a function, an external procedure.

S073 Intrinsic or predeclared, $, cannot be passed as an argument

S074 Illegal number or type of arguments to $ $

The indicated symbol is an intrinsic or generic function, or a predeclared subroutine or function, requiring a certain number of arguments of a fixed data type.

S075 Subscript, substring, or argument illegal in this context for $
This can happen if you try to doubly index an array such as ra(2)(3). This also applies to substring and function references.

S076 Subscripts specified for non-array variable $

S077 Subscripts omitted from array $

S078 Wrong number of subscripts specified for $

S079 Keyword form of argument illegal in this context for $$

S080 Subscript for array $ is out of bounds

S081 Illegal selector $ $

S082 Illegal substring expression for variable $
Substring expressions must be of type integer and if constant must be greater than zero.

S083 Vector expression used where scalar expression required
A vector expression was used in an illegal context. For example, `iscalar = iarray`, where a scalar is assigned the value of an array. Also, character and record references are not vectorizable.

S084 Illegal use of symbol $ $
This message is used for many different errors.

S085 Incorrect number of arguments to statement function $

S086 Dummy argument to statement function must be a variable

S087 Non-constant expression where constant expression required

S088 Recursive subroutine or function call of $
A function may not call itself.

S089 Illegal use of symbol, $, with character length = *
Symbols of type CHARACTER*(*) must be dummy variables and must not be used as statement function dummy parameters and statement function names. Also, a dummy variable of type CHARACTER*(*) cannot be used as a function.

S090 Hollerith constant more than 4 characters
In certain contexts, Hollerith constants may not be more than 4 characters long.

S091 Constant expression of wrong data type

S092 Illegal use of variable length character expression
A character expression used as an actual argument, or in certain contexts within I/O statements, must not consist of a concatenation involving a passed length character variable.

W093 Type conversion of expression performed
An expression of some data type appears in a context which requires an expression of some other data type. The compiler generates code to convert the expression into the required type.

S094 Variable $ is of wrong data type $

The indicated variable is used in a context which requires a variable of some other data type.

S095 Expression has wrong data type

An expression of some data type appears in a context which requires an expression of some other data type.

S096 Illegal complex comparison

The relations .LT., .GT., .GE., and .LE. are not allowed for complex values.

S097 Statement label $ has been defined more than once

More than one statement with the indicated statement number occurs in the subprogram.

S098 Divide by zero

S099 Illegal use of $

Aggregate record references may only appear in aggregate assignment statements, unformatted I/O statements, and as parameters to subprograms. They may not appear, for example, in expressions. Also, records with differing structure types may not be assigned to one another.

S100 Expression cannot be promoted to a vector

An expression was used that required a scalar quantity to be promoted to a vector illegally. For example, the assignment of a character constant string to a character array. Records, too, cannot be promoted to vectors.

S101 Vector operation not allowed on $

Record and character typed entities may only be referenced as scalar quantities.

S102 Arithmetic IF expression has wrong data type

The parenthetical expression of an arithmetic if statement must be an integer, real, or double precision scalar expression.

S103 Type conversion of subscript expression for $

The data type of a subscript expression must be integer. If it is not, it is converted.

S104 Illegal control structure $

This message is issued for a number of errors involving IF-THEN statements and DO loops. If the line number specified is the last line (END statement) of the subprogram, the error is probably an unterminated DO loop or IF-THEN statement.

S105 Unmatched ELSEIF, ELSE or ENDIF statement

An ELSEIF, ELSE, or ENDIF statement cannot be matched with a preceding IF-THEN statement.

S106 DO index variable must be a scalar variable

The DO index variable cannot be an array name, a subscripted variable, a PARAMETER name, a function name, a structure name, etc.

S107 Illegal assigned goto variable $

S108 Illegal variable, $, in NAMELIST group $

A NAMELIST group can only consist of arrays and scalars which are not dummy arguments and pointer-based variables.

I109 Overflow in $ constant $, constant truncated at left
A non-decimal (hexadecimal, octal, or binary) constant requiring more than 64-bits produces an overflow. The constant is truncated at left (e.g. '1234567890abcdef1'x will be '234567890abcdef1'x).

I110 <reserved message number>

I111 Underflow of real or double precision constant

I112 Overflow of real or double precision constant

S113 Label $ is referenced but never defined

S114 Cannot initialize $

W115 Assignment to DO variable $ in loop

S116 Illegal use of pointer-based variable $ $

S117 Statement not allowed within a $ definition
The statement may not appear in a STRUCTURE or derived type definition.

S118 Statement not allowed in DO, IF, or WHERE block

I119 Redundant specification for $
Data type of indicated symbol specified more than once.

I120 Label $ is defined but never referenced

I121 Operation requires logical or integer data types
An operation in an expression was attempted on data having a data type incompatible with the operation. For example, a logical expression can consist of only logical elements of type integer or logical. Real data would be invalid.

I122 Character string truncated
Character string or Hollerith constant appearing in a DATA statement or PARAMETER statement has been truncated to fit the declared size of the corresponding identifier.

W123 Hollerith length specification too big, reduced
The length specifier field of a hollerith constant specified more characters than were present in the character field of the hollerith constant. The length specifier was reduced to agree with the number of characters present.

S124 Relational expression mixes character with numeric data
A relational expression is used to compare two arithmetic expressions or two character expressions. A character expression cannot be compared to an arithmetic expression.

I125 Dummy procedure $ not declared EXTERNAL
A dummy argument which is not declared in an EXTERNAL statement is used as the subprogram name in a CALL statement, or is called as a function, and is therefore assumed to be a dummy procedure. This message can result from a failure to declare a dummy array.

I126 Name $ is not an intrinsic function

I127 Optimization level for $ changed to opt 1 $

W128 Integer constant truncated to fit data type: $
An integer constant will be truncated when assigned to data types smaller than 32-bits, such as a BYTE.

I129 Floating point overflow. Check constants and constant expressions

I130 Floating point underflow. Check constants and constant expressions

I131 Integer overflow. Check floating point expressions cast to integer

I132 Floating pt. invalid oprnd. Check constants and constant expressions

I133 Divide by 0.0. Check constants and constant expressions

S134 Illegal attribute $ $

W135 Missing STRUCTURE name field
A STRUCTURE name field is required on the outermost structure.

W136 Field-namelist not allowed
The field-namelist field of the STRUCTURE statement is disallowed on the outermost structure.

W137 Field-namelist is required in nested structures

W138 Multiply defined STRUCTURE member name $
A member name was used more than once within a structure.

W139 Structure $ in RECORD statement not defined
A RECORD statement contains a reference to a STRUCTURE that has not yet been defined.

S140 Variable $ is not a RECORD

S141 RECORD required on left of $

S142 $ is not a member of this RECORD

S143 $ requires initializer

W144 NEED ERROR MESSAGE $ $
This is used as a temporary message for compiler development.

W145 %FILL only valid within STRUCTURE block
The %FILL special name was used outside of a STRUCTURE multiline statement. It is only valid when used within a STRUCTURE multiline statement even though it is ignored.

S146 Expression must be character type

S147 Character expression not allowed in this context

S148 Reference to $ required
An aggregate reference to a record was expected during statement compilation but another data type was found instead.

S149 Record where arithmetic value required
An aggregate record reference was encountered when an arithmetic expression was expected.

S150 Structure, Record, derived type, or member $ not allowed in this context
A structure, record, or member reference was found in a context which is not supported. For example, the use of structures, records, or members within a data statement is disallowed.

S151 Empty TYPE, STRUCTURE, UNION, or MAP
TYPE - ENDTYPE, STRUCTURE - ENDSTRUCTURE, UNION - ENDUNION MAP - ENDMAP declaration contains no members.

S152 All dimension specifiers must be ':'

S153 Array objects are not conformable $

S154 DISTRIBUTE target, $, must be a processor

S155 $ $

S156 Number of colons and triplets must be equal in ALIGN $ with $

S157 Illegal subscript use of ALIGN dummy $ - $

S158 Alternate return not specified in SUBROUTINE or ENTRY
An alternate return can only be used if alternate return specifiers appeared in the SUBROUTINE or ENTRY statements.

S159 Alternate return illegal in FUNCTION subprogram
An alternate return cannot be used in a FUNCTION.

S160 ENDSTRUCTURE, ENDUNION, or ENDMAP does not match top

S161 Vector subscript must be rank-one array

W162 Not equal test of loop control variable $ replaced with < or > test.

S163 <reserved message number>

S164 Overlapping data initializations of $
An attempt was made to data initialize a variable or array element already initialized.

S165 $ appeared more than once as a subprogram
A subprogram name appeared more than once in the source file. The message is applicable only when an assembly file is the output of the compiler.

S166 $ cannot be a common block and a subprogram
A name appeared as a common block name and a subprogram name. The message is applicable only when an assembly file is the output of the compiler.

I167 Inconsistent size of common block $
A common block occurs in more than one subprogram of a source file and its size is not identical. The maximum size is chosen. The message is applicable only when an assembly file is the output of the compiler.

S168 Incompatible size of common block $
A common block occurs in more than one subprogram of a source file and is initialized in one subprogram. Its initialized size was found to be less than its size in the other subprogram(s). The message is applicable only when an assembly file is the output of the compiler.

W169 Multiple data initializations of common block $
A common block is initialized in more than one subprogram of a source file. Only the first set of initializations apply. The message is applicable only when an assembly file is the output of the compiler.

W170 PGI Fortran extension: $ $
Use of a nonstandard feature. A description of the feature is provided.

W171 PGI Fortran extension: nonstandard statement type $

W172 PGI Fortran extension: numeric initialization of CHARACTER $
A CHARACTER*1 variable or array element was initialized with a numeric value.

W173 PGI Fortran extension: nonstandard use of data type length specifier

W174 PGI Fortran extension: type declaration contains data initialization

W175 PGI Fortran extension: IMPLICIT range contains nonalpha characters

W176 PGI Fortran extension: nonstandard operator $

W177 PGI Fortran extension: nonstandard use of keyword argument $

W178 <reserved message number>

W179 PGI Fortran extension: use of structure field reference $

W180 PGI Fortran extension: nonstandard form of constant

W181 PGI Fortran extension: & alternate return

W182 PGI Fortran extension: mixed non-character and character elements in COMMON $

W183 PGI Fortran extension: mixed non-character and character EQUIVALENCE ($,$)

W184 Mixed type elements (numeric and/or character types) in COMMON $

W185 Mixed numeric and/or character type EQUIVALENCE ($,$)

S186 Argument missing for formal argument $

S187 Too many arguments specified for $

S188 Argument number $ to $: type mismatch

S189 Argument number $ to $: association of scalar actual argument to array dummy argument

S190 Argument number $ to $: non-conformable arrays

S191 Argument number $ to $ cannot be an assumed-size array

S192 Argument number $ to $ must be a label

W193 Argument number $ to $ does not match INTENT (OUT)

W194 INTENT(IN) argument cannot be defined - $

S195 Statement may not appear in an INTERFACE block $

S196 Deferred-shape specifiers are required for $

S197 Invalid qualifier or qualifier value (/$) in OPTIONS statement

An illegal qualifier was found or a value was specified for a qualifier which does not expect a value. In either case, the qualifier for which the error occurred is indicated in the error message.

S198 $ $ in ALLOCATE/DEALLOCATE

W199 Unaligned memory reference

A memory reference occurred whose address does not meet its data alignment requirement.

S200 Missing UNIT/FILE specifier

S201 Illegal I/O specifier - $

S202 Repeated I/O specifier - $

S203 FORMAT statement has no label

S204 $ $

Miscellaneous I/O error.

S205 Illegal specification of scale factor

The integer following + or - has been omitted, or P does not follow the integer value.

S206 Repeat count is zero

S207 Integer constant expected in edit descriptor

S208 Period expected in edit descriptor

S209 Illegal edit descriptor

S210 Exponent width not used in the Ew.dEe or Gw.dEe edit descriptors

S211 Internal I/O not allowed in this I/O statement

S212 Illegal NAMELIST I/O

Namelist I/O cannot be performed with internal, unformatted, formatted, and list-directed I/O. Also, I/O lists must not be present.

S213 $ is not a NAMELIST group name

S214 Input item is not a variable reference

S215 Assumed sized array name cannot be used as an I/O item or specifier
An assumed size array was used as an item to be read or written or as an I/O specifier (i.e., FMT = array-name). In these contexts the size of the array must be known.

S216 STRUCTURE/UNION cannot be used as an I/O item

S217 ENCODE/DECODE buffer must be a variable, array, or array element

S218 Statement labeled $ $

S219 <reserved message number>

S220 Redefining predefined macro $

S221 #elif after #else
A preprocessor #elif directive was found after a #else directive; only #endif is allowed in this context.

S222 #else after #else
A preprocessor #else directive was found after a #else directive; only #endif is allowed in this context.

S223 #if-directives too deeply nested
Preprocessor #if directive nesting exceeded the maximum allowed (currently 10).

S224 Actual parameters too long for $
The total length of the parameters in a macro call to the indicated macro exceeded the maximum allowed (currently 2048).

W225 Argument mismatch for $
The number of arguments supplied in the call to the indicated macro did not agree with the number of parameters in the macro's definition.

F226 Can't find include file $
The indicated include file could not be opened.

S227 Definition too long for $
The length of the macro definition of the indicated macro exceeded the maximum allowed (currently 2048).

S228 EOF in comment
The end of a file was encountered while processing a comment.

S229 EOF in macro call to $
The end of a file was encountered while processing a call to the indicated macro.

S230 EOF in string
The end of a file was encountered while processing a quoted string.

S231 Formal parameters too long for $
The total length of the parameters in the definition of the indicated macro exceeded the maximum allowed (currently 2048).

S232 Identifier too long
The length of an identifier exceeded the maximum allowed (currently 2048).

S233 <reserved message number>

W234 Illegal directive name
The sequence of characters following a # sign was not an identifier.

W235 Illegal macro name
A macro name was not an identifier.

S236 Illegal number $
The indicated number contained a syntax error.

F237 Line too long
The input source line length exceeded the maximum allowed (currently 2048).

W238 Missing #endif
End of file was encountered before a required #endif directive was found.

W239 Missing argument list for $
A call of the indicated macro had no argument list.

S240 Number too long
The length of a number exceeded the maximum allowed (currently 2048).

W241 Redefinition of symbol $
The indicated macro name was redefined.

I242 Redundant definition for symbol $
A definition for the indicated macro name was found that was the same as a previous definition.

F243 String too long
The length of a quoted string exceeded the maximum allowed (currently 2048).

S244 Syntax error in #define, formal $ not identifier
A formal parameter that was not an identifier was used in a macro definition.

W245 Syntax error in #define, missing blank after name or arglist
There was no space or tab between a macro name or argument list and the macro's definition.

S246 Syntax error in #if
A syntax error was found while parsing the expression following a #if or #elif directive.

S247 Syntax error in #include
The #include directive was not correctly formed.

W248 Syntax error in #line
A #line directive was not correctly formed.

W249 Syntax error in #module
A #module directive was not correctly formed.

W250 Syntax error in #undef
A #undef directive was not correctly formed.

W251 Token after #ifdef must be identifier
The #ifdef directive was not followed by an identifier.

W252 Token after #ifndef must be identifier
The #ifndef directive was not followed by an identifier.

S253 Too many actual parameters to $
The number of actual arguments to the indicated macro exceeded the maximum allowed (currently 31).

S254 Too many formal parameters to $
The number of formal arguments to the indicated macro exceeded the maximum allowed (currently 31).

F255 Too much pushback
The preprocessor ran out of space while processing a macro expansion. The macro may be recursive.

W256 Undefined directive $
The identifier following a # was not a directive name.

S257 EOF in #include directive
End of file was encountered while processing a #include directive.

S258 Unmatched #elif
A #elif directive was encountered with no preceding #if or #elif directive.

S259 Unmatched #else
A #else directive was encountered with no preceding #if or #elif directive.

S260 Unmatched #endif
A #endif directive was encountered with no preceding #if, #ifdef, or #ifndef directive.

S261 Include files nested too deeply
The nesting depth of #include directives exceeded the maximum (currently 20).

S262 Unterminated macro definition for $
A newline was encountered in the formal parameter list for the indicated macro.

S263 Unterminated string or character constant
A newline with no preceding backslash was found in a quoted string.

I264 Possible nested comment
The characters /* were found within a comment.

S265 <reserved message number>

S266 <reserved message number>

S267 <reserved message number>

W268 Cannot inline subprogram; common block mismatch

W269 Cannot inline subprogram; argument type mismatch
This message may be Severe if the compilation has gone too far to undo the inlining process.

F270 Missing -exlib option

W271 Can’t inline $ - wrong number of arguments

I272 Argument of inlined function not used

S273 Inline library not specified on command line (-inlib switch)

F274 Unable to access file $/TOC

S275 Unable to open file $ while extracting or inlining

F276 Assignment to constant actual parameter in inlined subprogram

I277 Inlining of function $ may result in recursion

S278 <reserved message number>

W279 Possible use of $ before definition in $
The optimizer has detected the possibility that a variable is used before it has been assigned a value. The names of the variable and the function in which the use occurred are listed. The line number, if specified, is the line number of the basic block containing the use of the variable.

W280 Syntax error in directive $
Messages 280-300 reserved for directives. handling

W281 Directive ignored - $ $

S300 Too few data constants in initialization of derived type $

S301 $ must be TEMPLATE or PROCESSOR

S302 Unmatched END$ statement

S303 END statement for $ required in an interface block

S304 EXIT/CYCLE statement must appear in a DO/DOWHILE loop$$

S305 $ cannot be named, $

S306 $ names more than one construct

S307 $ must have the construct name $

S308 DO may not terminate at an EXIT, CYCLE, RETURN, STOP, GOTO, or arithmetic IF

S309 Incorrect name, $, specified in END statement

S310 $ $
Generic message for MODULE errors.

W311 Non-replicated mapping for $ array, $, ignored

W312 Array $ should be declared SEQUENCE

W313 Subprogram $ called within INDEPENDENT loop not PURE

E314 IPA: actual argument $ is a label, but dummy argument $ is not an asterisk
The call passes a label to the subprogram; the corresponding dummy argument in the subprogram should be an asterisk to declare this as the alternate return.

I315 IPA: routine $, $ constant dummy arguments
This many dummy arguments are being replaced by constants due to interprocedural analysis.

I316 IPA: routine $, $ INTENT(IN) dummy arguments
This many dummy arguments are being marked as INTENT(IN) due to interprocedural analysis.

I317 IPA: routine $, $ array alignments propagated
This many array alignments were propagated by interprocedural analysis.

I318 IPA: routine $, $ distribution formats propagated
This many array distribution formats were propagated by interprocedural analysis.

I319 IPA: routine $, $ distribution targets propagated
This many array distribution targets were propagated by interprocedural analysis.

I320 IPA: routine $, $ common blocks optimized
This many mapped common blocks were optimized by interprocedural analysis.

I321 IPA: routine $, $ common blocks not optimized
This many mapped common blocks were not optimized by interprocedural analysis, either because they were declared differently in different routines, or they did not appear in the main program.

I322 IPA: analyzing main program $
Interprocedural analysis is building the call graph and propagating information with the named main program.

I323 IPA: collecting information for $
Interprocedural analysis is saving information for the current subprogram for subsequent analysis and propagation.

W324 IPA file $ appears to be out of date

W325 IPA file $ is for wrong subprogram: $

W326 Unable to open file $ to propagate IPA information to $

I327 IPA: $ subprograms analyzed

I328 IPA: $ dummy arguments replaced by constants

I329 IPA: $ INTENT(IN) dummy arguments should be INTENT(INOUT)

I330 IPA: $ dummy arguments changed to INTENT(IN)

I331 IPA: $ inherited array alignments replaced

I332 IPA: $ transcriptive distribution formats replaced

I333 IPA: $ transcriptive distribution targets replaced

I334 IPA: $ descriptive/prescriptive array alignments verified

I335 IPA: $ descriptive/prescriptive distribution formats verified

I336 IPA: $ descriptive/prescriptive distribution targets verified

I337 IPA: $ common blocks optimized

I338 IPA: $ common blocks not optimized

S339 Bad IPA contents file: $

S340 Bad IPA file format: $

S341 Unable to create file $ while analyzing IPA information

S342 Unable to open file $ while analyzing IPA information

S343 Unable to open IPA contents file $

S344 Unable to create file $ while collecting IPA information

F345 Internal error in $: table overflow
Analysis failed due to a table overflowing its maximum size.

W346 Subprogram $ appears twice
The subprogram appears twice in the same source file; IPA will ignore the first appearance.

F347 Missing -ipalib option
Interprocedural analysis, enabled with the `–ipacollect`, `–ipaanalyze`, or `–ipapropagate` options, requires the `-ipalib` option to specify the library directory.

W348 Common /$/ $ has different distribution target
The array was declared in a common block with a different distribution target in another subprogram.

W349 Common /$/ $ has different distribution format
The array was declared in a common block with a different distribution format in another subprogram.

W350 Common /$/ $ has different alignment
The array was declared in a common block with a different alignment in another subprogram.

W351 Wrong number of arguments passed to $
The subroutine or function statement for the given subprogram has a different number of dummy arguments than appear in the call.

W352 Wrong number of arguments passed to $ when bound to $
The subroutine or function statement for the given subprogram has a different number of dummy arguments than appear in the call to the EXTERNAL name given.

W353 Subprogram $ is missing
A call to a subroutine or function with this name appears, but it could not be found or analyzed.

I354 Subprogram $ is not called
No calls to the given subroutine or function appear anywhere in the program.

W355 Missing argument in call to $

A nonoptional argument is missing in a call to the given subprogram.

I356 Array section analysis incomplete

Interprocedural analysis for array section arguments is incomplete; some information may not be available for optimization.

I357 Expression analysis incomplete

Interprocedural analysis for expression arguments is incomplete; some information may not be available for optimization.

W358 Dummy argument $ is EXTERNAL, but actual is not subprogram

The call statement passes a scalar or array to a dummy argument that is declared EXTERNAL.

W359 SUBROUTINE $ passed to FUNCTION dummy argument $

The call statement passes a subroutine name to a dummy argument that is used as a function.

W360 FUNCTION $ passed to FUNCTION dummy argument $ with different result type

The call statement passes a function argument to a function dummy argument, but the dummy has a different result type.

W361 FUNCTION $ passed to SUBROUTINE dummy argument $

The call statement passes a function name to a dummy argument that is used as a subroutine.

W362 Argument $ has a different type than dummy argument $

The type of the actual argument is different than the type of the corresponding dummy argument.

W363 Dummy argument $ is a POINTER but actual argument $ is not

The dummy argument is a pointer, so the actual argument must be also.

W364 Array or array expression passed to scalar dummy argument $

The actual argument is an array, but the dummy argument is a scalar variable.

W365 Scalar or scalar expression passed to array dummy argument $

The actual argument is a scalar variable, but the dummy argument is an array.

F366 Internal error: interprocedural analysis fails

An internal error occurred during interprocedural analysis; please report this to the compiler maintenance group. If user errors were reported when collecting IPA information or during IPA analysis, correcting them may avoid this error.

I367 Array $ bounds cannot be matched to formal argument

Passing a nonsequential array to a sequential dummy argument may require copying the array to sequential storage. The most common cause is passing an ALLOCATABLE array or array expression to a dummy argument that is declared with explicit bounds. Declaring the dummy argument as assumed shape, with bounds (:,:,:), will remove this warning.

W368 Array-valued expression passed to scalar dummy argument $

The actual argument is an array-valued expression, but the dummy argument is a scalar variable.

W369 Dummy argument $ has different rank than actual argument
The actual argument is an array or array-valued expression with a different rank than the dummy argument.

W370 Dummy argument $ has different shape than actual argument
The actual argument is an array or array-valued expression with a different shape than the dummy argument; this may require copying the actual argument into sequential storage.

W371 Dummy argument $ is INTENT(IN) but may be modified
The dummy argument was declared as INTENT(IN), but analysis has found that the argument may be modified; the INTENT(IN) declaration should be changed.

W372 Cannot propagate alignment from $ to $
The most common cause is when passing an array with an inherited alignment to a dummy argument with non-inherited alignment.

I373 Cannot propagate distribution format from $ to $
The most common cause is when passing an array with a transcriptive distribution format to a dummy argument with prescriptive or descriptive distribution format.

I374 Cannot propagate distribution target from $ to $
The most common cause is when passing an array with a transcriptive distribution target to a dummy argument with prescriptive or descriptive distribution target.

I375 Distribution format mismatch between $ and $
Usually this arises when the actual and dummy arguments are distributed in different dimensions.

I376 Alignment stride mismatch between $ and $
This may arise when the actual argument has a different stride in its alignment to its template than does the dummy argument.

I377 Alignment offset mismatch between $ and $
This may arise when the actual argument has a different offset in its alignment to its template than does the dummy argument.

I378 Distribution target mismatch between $ and $
This may arise when the actual and dummy arguments have different distribution target sizes.

I379 Alignment of $ is too complex
The alignment specification of the array is too complex for interprocedural analysis to verify or propagate; the program will work correctly, but without the benefit of IPA.

I380 Distribution format of $ is too complex
The distribution format specification of the array is too complex for interprocedural analysis to verify or propagate; the program will work correctly, but without the benefit of IPA.

I381 Distribution target of $ is too complex
The distribution target specification of the array is too complex for interprocedural analysis to verify or propagate; the program will work correctly, but without the benefit of IPA.

I382 IPA: $ subprograms analyzed
Interprocedural analysis succeeded in finding and analyzing this many subprograms in the whole program.

I383 IPA: $ dummy arguments replaced by constants

Interprocedural analysis has found this many dummy arguments in the whole program that can be replaced by constants.

I384 IPA: $ dummy arguments changed to INTENT(IN)

Interprocedural analysis has found this many dummy arguments in the whole program that are not modified and can be declared as INTENT(IN).

W385 IPA: $ INTENT(IN) dummy arguments should be INTENT(INOUT)

Interprocedural analysis has found this many dummy arguments in the whole program that were declared as INTENT(IN) but should be INTENT(INOUT).

I386 IPA: $ array alignments propagated

Interprocedural analysis has found this many array dummy arguments that could have the inherited array alignment replaced by a descriptive alignment.

I387 IPA: $ array alignments verified

Interprocedural analysis has verified that the prescriptive or descriptive alignments of this many array dummy arguments match the alignments of the actual argument.

I388 IPA: $ array distribution formats propagated

Interprocedural analysis has found this many array dummy arguments that could have the transcriptive distribution format replaced by a descriptive format.

I389 IPA: $ array distribution formats verified

Interprocedural analysis has verified that the prescriptive or descriptive distribution formats of this many array dummy arguments match the formats of the actual argument.

I390 IPA: $ array distribution targets propagated

Interprocedural analysis has found this many array dummy arguments that could have the transcriptive distribution target replaced by a descriptive target.

I391 IPA: $ array distribution targets verified

Interprocedural analysis has verified that the prescriptive or descriptive distribution targets of this many array dummy arguments match the targets of the actual argument.

I392 IPA: $ common blocks optimized

Interprocedural analysis has found this many common blocks that could be optimized.

I393 IPA: $ common blocks not optimized

Interprocedural analysis has found this many common blocks that could not be optimized, either because the common block was not declared in the main program, or because it was declared differently in different subprograms.

I394 IPA: $ replaced by constant value

The dummy argument was replaced by a constant as per interprocedural analysis.

I395 IPA: $ changed to INTENT(IN)

The dummy argument was changed to INTENT(IN) as per interprocedural analysis.

I396 IPA: array alignment propagated to $

The template alignment for the dummy argument was changed as per interprocedural analysis.

I397 IPA: distribution format propagated to $
The distribution format for the dummy argument was changed as per interprocedural analysis.

I398 IPA: distribution target propagated to $
The distribution target for the dummy argument was changed as per interprocedural analysis.

I399 IPA: common block $ not optimized
The given common block was not optimized by interprocedural analysis either because it was not declared in the main program, or because it was declared differently in different subprograms.

E400 IPA: dummy argument $ is an asterisk, but actual argument is not a label
The subprogram expects an alternate return label for this argument.

E401 Actual argument $ is a subprogram, but Dummy argument $ is not declared EXTERNAL
The call statement passes a function or subroutine name to a dummy argument that is a scalar variable or array.

E402 Actual argument $ is illegal

E403 Actual argument $ and formal argument $ have different ranks
The actual and formal array arguments differ in rank, which is allowed only if both arrays are declared with the HPF SEQUENCE attribute.

E404 Sequential array section of $ in argument $ is not contiguous
When passing an array section to a formal argument that has the HPF SEQUENCE attribute, the actual argument must be a whole array with the HPF SEQUENCE attribute, or an array section of such an array where the section is a contiguous sequence of elements.

E405 Array expression argument $ may not be passed to sequential dummy argument $
When the dummy argument has the HPF SEQUENCE attribute, the actual argument must be a whole array with the HPF SEQUENCE attribute or a contiguous array section of such an array, unless an INTERFACE block is used.

E406 Actual argument $ and formal argument $ have different character lengths
The actual and formal array character arguments have different character lengths, which is allowed only if both character arrays are declared with the HPF SEQUENCE attribute, unless an INTERFACE block is used.

W407 Argument $ has a different character length than dummy argument $
The character length of the actual argument is different than the length specified for the corresponding dummy argument.

W408 Specified main program $ is not a PROGRAM
The main program specified on the command line is a subroutine, function, or block data subprogram.

W409 More than one main program in IPA directory: $ and $
There is more than one main program analyzed in the IPA directory shown. The first one found is used.

W410 No main program found; IPA analysis fails.
The main program must appear in the IPA directory for analysis to proceed.

W411 Formal argument $ is DYNAMIC but actual argument is an expression

W412 Formal argument $ is DYNAMIC but actual argument $ is not

I413 Formal argument $ has two reaching distributions and may be a candidate for cloning

I414 $ and $ may be aliased and one of them is assigned
Interprocedural analysis has determined that two formal arguments may be aliased because the same variable is passed in both argument positions; or one formal argument and a global or COMMON variable may be aliased, because the global or COMMON variable is passed as an actual argument. If either alias is assigned in the subroutine, unexpected results may occur; this message alerts the user that this situation is disallowed by the Fortran standard.

F415 IPA fails: incorrect IPA file
Interprocedural analysis saves its information in special IPA files in the specified IPA directory. One of these files has been renamed or corrupted. This can arise when there are two files with the same prefix, such as `a.hpf` and `a.f90`.

E416 Argument $ has the SEQUENCE attribute, but the dummy parameter $ does not
When an actual argument is an array with the SEQUENCE attribute, the dummy parameter must have the SEQUENCE attribute or an INTERFACE block must be used.

E417 Interface block for $ is a SUBROUTINE but should be a FUNCTION

E418 Interface block for $ is a FUNCTION but should be a SUBROUTINE

E419 Interface block for $ is a FUNCTION has wrong result type

W420 Earlier $ directive overrides $ directive

W421 $ directive can only appear in a function or subroutine

E422 Nonconstant DIM= argument is not supported

E423 Constant DIM= argument is out of range

E424 Equivalence using substring or vector triplets is not allowed

E425 A record is not allowed in this context

E426 WORD type cannot be converted

E427 Interface block for $ has wrong number of arguments

E428 Interface block for $ should have $

E429 Interface block for $ should not have $

E430 Interface block for $ has wrong $

W431 Program is too large for Interprocedural Analysis to complete

W432 Illegal type conversion $

E433 Subprogram $ called within INDEPENDENT loop not LOCAL

W434 Incorrect home array specification ignored

S435 Array declared with zero size
An array was declared with a zero or negative dimension bound, as 'real a(-1)', or an upper bound less than the lower bound, as 'real a(4:2)'.

W436 Independent loop not parallelized$

W437 Type $ will be mapped to $
Where DOUBLE PRECISION is not supported, it is mapped to REAL, and similarly for COMPLEX(16) or COMPLEX*32.

E438 $ $ not supported on this platform
This construct is not supported by the compiler for this target.

S439 An internal subprogram cannot be passed as argument - $

S440 Defined assignment statements may not appear in WHERE statement or WHERE block

S441 $ may not appear in a FORALL block

E442 Adjustable-length character type not supported on this host - $ $

S443 EQUIVALENCE of derived types not supported on this host - $

S444 Derived type in EQUIVALENCE statement must have SEQUENCE attribute - $
A variable or array with derived type appears in an EQUIVALENCE statement. The derived type must have the SEQUENCE attribute, but does not.

E445 Array bounds must be integer $ $
The expressions in the array bounds must be integer.

S446 Argument number $ to $: rank mismatch
The number of dimensions in the array or array expression does not match the number of dimensions in the dummy argument.

S447 Argument number $ to $ must be a subroutine or function name

S448 Argument number $ to $ must be a subroutine name

S449 Argument number $ to $ must be a function name

S450 Argument number $ to $: kind mismatch

S451 Arrays of derived type with a distributed member are not supported

S452 Assumed length character, $, is not a dummy argument

S453 Derived type variable with pointer member not allowed in IO - $ $

S454 Subprogram $ is not a module procedure
Only names of module procedures declared in this module or accessed through USE association can appear in a MODULE PROCEDURE statement.

S455 A derived type array section cannot appear with a member array section - $
A reference like A(:)%B(:), where 'A' is a derived type array and 'B' is a member array, is not allowed; a section subscript may appear after 'A' or after 'B', but not both.

S456 Unimplemented for data type for MATMUL

S457 Illegal expression in initialization

S458 Argument to NULL() must be a pointer

S459 Target of NULL() assignment must be a pointer

S460 ELEMENTAL procedures cannot be RECURSIVE

S461 Dummy arguements of ELEMENATAL procedures must be scalar

S462 Arguments and return values of ELEMENATAL procedures cannot have the POINTER attribute

S463 Arguments of ELEMENATAL procedures cannot be procedures

S464 An ELEMENTAL procedure cannot be passed as argument - $

Fortran Run-time Error Messages

This section presents the error messages generated by the run-time system. The run-time system displays error messages on standard output.

Message Format

The messages are numbered but have no severity indicators because they all terminate program execution.

Message List

Here are the run-time error messages:

201 illegal value for specifier
An improper specifier value has been passed to an I/O run-time routine. Example: within an OPEN statement, form='unknown'.

202 conflicting specifiers
Conflicting specifiers have been passed to an I/O run-time routine. Example: within an OPEN statement, form='unformatted', blank='null'.

203 record length must be specified
A recl specifier required for an I/O run-time routine has not been passed. Example: within an OPEN statement, access='direct' has been passed, but the record length has not been specified (recl=specifier).

204 illegal use of a readonly file
Self explanatory. Check file and directory modes for readonly status.

205 'SCRATCH' and 'SAVE'/'KEEP' both specified
In an OPEN statement, a file disposition conflict has occurred. Example: within an OPEN statement, status='scratch' and dispose='keep' have been passed.

206 attempt to open a named file as 'SCRATCH'

207 file is already connected to another unit

208 'NEW' specified for file that already exists

209 'OLD' specified for file that does not exist

210 dynamic memory allocation failed
Memory allocation operations occur only in conjunction with namelist I/O. The most probable cause of fixed buffer overflow is exceeding the maximum number of simultaneously open file units.

211 invalid file name

212 invalid unit number
A file unit number less than or equal to zero has been specified.

215 formatted/unformatted file conflict
Formatted/unformatted file operation conflict.

217 attempt to read past end of file

219 attempt to read/write past end of record
For direct access, the record to be read/written exceeds the specified record length.

220 write after last internal record

221 syntax error in format string
A run-time encoded format contains a lexical or syntax error.

222 unbalanced parentheses in format string

223 illegal P or T edit descriptor - value missing

224 illegal Hollerith or character string in format
An unknown token type has been found in a format encoded at run-time.

225 lexical error -- unknown token type

226 unrecognized edit descriptor letter in format
An unexpected Fortran edit descriptor (FED) was found in a run-time format item.

228 end of file reached without finding group

229 end of file reached while processing group

230 scale factor out of range -128 to 127
Fortran P edit descriptor scale factor not within range of -128 to 127.

231 error on data conversion

233 too many constants to initialize group item

234 invalid edit descriptor
An invalid edit descriptor has been found in a format statement.

235 edit descriptor does not match item type
Data types specified by I/O list item and corresponding edit descriptor conflict.

236 formatted record longer than 2000 characters

237 quad precision type unsupported

238 tab value out of range
A tab value of less than one has been specified.

239 entity name is not member of group

240 no initial left parenthesis in format string

241 unexpected end of format string

242 illegal operation on direct access file

243 format parentheses nesting depth too great

244 syntax error - entity name expected

245 syntax error within group definition

246 infinite format scan for edit descriptor

248 illegal subscript or substring specification

249 error in format - illegal E, F, G or D descriptor

250 error in format - number missing after '.', '-', or '+'

251 illegal character in format string

252 operation attempted after end of file

253 attempt to read non-existent record (direct access)

254 illegal repeat count in format

Index

Symbols

A

B

C

D

E

F

I

K

L

M

N

O

P

R

S

T

U

V

W