PVF® User's Guide

Parallel Fortran for Scientists and Engineers

Release 2010

The Portland Group®

PVF® User's Guide

Printed in the United States of America
First Printing: Release 6.1, December 2005
Second Printing: Release 6.2, August 2006
Third Printing: Release 7.0, December 2006
Fourth Printing: Release 7.1, October 2007
Fifth Printing: Release 7.2, May 2008
Sixth Printing: Release 8.0, November 2008
Seventh Printing: Release 9.0, June 2009
Eighth Printing: Release 2010, November 2009
Ninth Printing: Release 2010, 10.2, February 2010
Tenth Printing: Release 2010, 10.3, March 2010
Eleventh Printing: Release 2010, 10.4, April 2010
Twelfth Printing: Release 2010, 10.5, May 2010
Thirteenth Printing: Release 2010, 10.6, June 2010

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

ID: 101531642

Contents

Tables

Examples

Preface

This guide is part of a set of manuals that describe how to use The Portland Group (PGI) Fortran compilers and program development tools integrated with Microsoft Visual Studio. These tools, combined with Visual Studio and assorted libraries, are collectively known as PGI Visual Fortran®, or PVF®. You can use PVF to edit, compile, debug, optimize, and profile serial and parallel applications for x86 (Intel Pentium II/III/4/M, Intel Centrino, Intel Xeon, AMD Athlon XP/MP) or x64 (AMD Athlon64/Opteron/Turion, Intel EM64T/Core 2) processor-based systems.

The *PVF User's Guide* provides operating instructions for both the Visual Studio integrated development environment as well as command-level compilation and general information about PGI's implementation of the Fortran language. This guide does not teach the Fortran programming language.

Audience Description

This manual is intended for scientists and engineers using PGI Visual Fortran. To fully understand this guide, you should be aware of the role of high-level languages, such as Fortran, in the software development process; and you should have some level of understanding of programming. PGI Visual Fortran is available on a variety of x86 or x64 hardware platforms and variants of the Windows operating system. 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 PVF. For information on installing PVF, 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.
- *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).

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 is divided into the following two parts:

- Part I, Compiler Usage, contains the essential information on how to use the compiler.
- Part II, Reference Information, contains more detailed reference information about specific aspects of the compiler, such as the details of compiler options, directives, and more.

Part I, Compiler Usage, contains these chapters:

Chapter 1, "*Getting Started with PVF*" gives an overview of the Visual Studio environment and how to use PGI Visual Fortran in that environment.

Chapter 2, "*Build with PVF*" gives an overview of how to use PGI Visual Fortran (PVF) within the Microsoft Visual Studio IDE (Integrated Development Environment) to create and build a PVF project.

Chapter 3, "*Debug with PVF*" gives an overview of how to use the custom debug engine that provides the language-specific debugging capability required for Fortran.

Chapter 4, "*Using MPI in PVF*" describes how to use MPI with PGI Visual Fortran.

Chapter 5, "*Getting Started with the Command Line Compilers*" provides an introduction to the PGI compilers and describes their use and overall features.

Chapter 6, "*Using Command Line Options*" provides an overview of the command-line options as well as task-related lists of options.

Chapter 7, "*Optimizing & Parallelizing*" describes standard optimization techniques that, with little effort, allow users to significantly improve the performance of programs.

Chapter 8, "*Using Function Inlining*" describes how to use function inlining and shows how to create an inline library.

Chapter 9, "*Using OpenMP*" provides a description of the OpenMP Fortran parallelization directives and shows examples of their use.

Chapter 10, "*Using an Accelerator*" describes how to use the PGI Accelerator compilers.

Chapter 11, "*Using Directives*" provides a description of each Fortran optimization directive, and shows examples of their use.

Chapter 12, "*Creating and Using Libraries*" discusses PGI support libraries, shared object files, and environment variables that affect the behavior of the PGI compilers.

Chapter 13, " *Using Environment Variables*" describes the environment variables that affect the behavior of the PGI compilers.

Chapter 14, "*Distributing Files - Deployment*" describes the deployment of your files once you have built, debugged and compiled them successfully.

Chapter 15, "*Inter-language Calling*" provides examples showing how to place C Language calls in a Fortran program and Fortran Language calls in a C program.

Chapter 16, "*Programming Considerations for 64-Bit Environments*" discusses issues of which programmers should be aware when targeting 64-bit processors.

Part II, Reference Information, contains these chapters:

Chapter 17, "*Fortran Data Types*" describes the data types that are supported by the PGI Fortran compilers.

Chapter 18, "*Command-Line Options Reference*" provides a detailed description of each command-line option.

Chapter 19, "*OpenMP Reference Information*" contains detailed descriptions of each of the OpenMP directives that PGI supports.

Chapter 21, "*Directives Reference*" contains detailed descriptions of PGI's proprietary directives.

Chapter 22, "*Run-time Environment*" describes the assembly language calling conventions and examples of assembly language calls.

Chapter 23, "*PVF Properties*" provides a description of Property Pages that PGI supports.

Chapter 24, "*PVF Build Macros*" provides a description of the build macros that PVF supports.

Chapter 25, "*Fortran Module/Library Interfaces for Windows*" provides a description of the Fortran module library interfaces that PVF supports, describing each property available.

Chapter 26, "*Messages*" provides a list of compiler error messages.

Hardware and Software Constraints

This guide describes versions of PGI Visual Fortran that are intended for use on 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 PGI Visual Fortran.

Conventions

The *PVF User's Guide* 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 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 *PVF User's Guide* 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	Large arrays	SSE	Win32
barcelona	-mcmodel=small	SSE1	Win64
DLL	-mcmodel=medium	SSE2	Windows
driver	MPI	SSE3	x64
dynamic library	MPICH	SSE4A and ABM	x86
EM64T	multi-core	SSSE3	x87
hyperthreading (HT)	NUMA	static linking	
IA32	shared library		

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	pgf95

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 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.

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, 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.
- *OpenMP Application Program Interface*, Version 2.5 May 2005 (OpenMP Architecture Review Board, 1997-2005).

Part I. Compiler Usage

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. In the chapters in this part of the guide you learn how to:

- Get started using the PGI compilers, as described in Chapter 1, "Getting Started" on page 1.
- Use the most common command line options and learn why specific ones are especially beneficial for you to use, as described in Chapter 2, "Using Command Line Options" on page 17.
- Use optimization and parallelization to increase the performance of your program, as described in Chapter 3, "Optimizing & Parallelizing" on page 23.
- Invoke function inlining and create an inline library, as described in Chapter 4, "Using Function Inlining" on page 49.
- Use OpenMP directives, pragmas, run-time libraries, and environment variables, as described in Chapter 5, "Using OpenMP" on page 55.
- Use MPI, including compiling, linking and generating MPI profile data, as described in Chapter 6, "Using MPI" on page 73.
- Using PGI Accelerator compilers, as described in Chapter 7, "Using an Accelerator" on page 83.
- Use PGI directives and pragmas, as described in Chapter 8, "Using Directives and Pragmas" on page 107.
- Create and use libraries, as described in Chapter 9, "Creating and Using Libraries" on page 117.
- Create and use environment variables to control the behavior of PGI software, as described in Chapter 10, "Using Environment Variables" on page 131.
- Distribute files and deploy your applications, as described in Chapter 11, "Distributing Files - Deployment" on page 145.
- Make inter-language calls, as described in Chapter 12, "Inter-language Calling" on page 151.
- Incorporate programming considerations for 64-bit environments, as described in Chapter 13, "Programming Considerations for 64-bit Environments" on page 167.
- Properly use C/C++ inline assembly instructions and intrinsics, as described in Chapter 14, "C/C++ Inline Assembly and Intrinsics" on page 175.

Chapter 1. Getting Started with PVF

This chapter describes how to use PGI Visual Fortran (PVF) within the Microsoft Visual Studio IDE (Integrated Development Environment). For information on general use of Visual Studio, refer to Microsoft's documentation.

There are three products in the PVF product family. Each product is integrated with a particular version of Microsoft Visual Studio. PGI Visual Fortran 2010 is integrated with Microsoft Visual Studio 2010 (VS 2010), PGI Visual Fortran 2008 is integrated with Microsoft Visual Studio 2008 (VS 2008), and PGI Visual Fortran 2005 is integrated with Microsoft Visual Studio 2005 (VS 2005). Throughout this document, "PGI Visual Fortran" or "PVF" refers to all PVF products collectively. Further, "Microsoft Visual Studio" refers to the compatible version of Visual Studio that you are using with PVF.

The following sections provide a general overview of the many features and capabilities available to you once you have installed PVF. Exploring the menus and trying the sample program in this section provide a quick way to get started using PVF.

PVF on the Start Menu

PVF provides a Start menu entry that provides access to different versions of PVF as well as easy access to the PGI Profiler, Documentation and Licensing. This section provides a quick overview of the PVF menu selections.

To access the PGI Visual Fortran menu, from the Start menu, select *Start* | *All Programs* | *PGI Visual Fortran*.

Shortcuts to Launch PVF

From the PGI Visual Fortran menu, you have access to each version of PGI Visual Fortran you have installed. For example, if you have all three PVF products installed at the same time, then you have a shortcut for PVF 2010, PVF 2008 and PVF 2005.

PVF runs within Visual Studio, so to invoke PVF you must invoke Visual Studio. If other languages such as Visual C++ or Visual Basic are installed, they are available in the same instance of Visual Studio as PVF.

The PVF shortcuts include the following:

PGI Visual Fortran 2010 – Select this option to invoke PGI Visual Fortran 2010.
PGI Visual Fortran 2008 – Select this option to invoke PGI Visual Fortran 2008.
PGI Visual Fortran 2005 – Select this option to invoke PGI Visual Fortran 2005.

Commands Submenu

From the Commands menu, you have access to PVF command shells for each version of PVF installed on your system. For example, if you have PVF 2005 and PVF 2008 installed, then you have a selection for each of these versions.

These shortcuts invoke a command shell with the environment configured for the PGI compilers and tools. The command line compilers and graphical tools may be invoked from any of these command shells without any further configuration.

Important

If you invoke a generic Command Prompt using *Start* | *All Programs* | *Accessories* | *Command Prompt*, then the environment is not pre-configured for PGI products.

- **PVF 2010 Cmd (64) –** Select this option to launch a command shell in which the environment is pre-initialized to use PVF 2010 and the 64-bit PGI compilers and tools. The default environment variables are already set and available. (Available only on x64 systems.)
- **PVF 2010 Cmd –** Select this option to launch a command shell in which the environment is pre-initialized to use PVF 2010 and the 32-bit PGI compilers and tools. The default environment variables are already set and available.
- **PVF 2008 Cmd (64) –** Select this option to launch a command shell in which the environment is pre-initialized to use PVF 2008 and the 64-bit PGI compilers and tools. The default environment variables are already set and available. (Available only on x64 systems.)
- **PVF 2008 Cmd –** Select this option to launch a command shell in which the environment is pre-initialized to use PVF 2008 and the 32-bit PGI compilers and tools. The default environment variables are already set and available.
- **PVF 2005 Cmd (64) –** Select this option to launch a command shell in which the environment is pre-initialized to use PVF 2005 and the 64-bit PGI compilers and tools. The default environment variables are already set and available. (Available only on x64 systems.)
- **PVF 2005 Cmd –** Select this option to launch a command shell in which the environment is pre-initialized to use PVF 2005 and the 32-bit PGI compilers and tools. The default environment variables are already set and available.

Profiler Submenu

Use the profiler menu to launch the PGPROF Performance Profiler. PGPROF provides a way to visualize and diagnose the performance of the components of your program and provides features for helping you to understand why certain parts of your program have high execution times.

Documentation Submenu

From the Documentation menu, you have access to all PGI documentation that is useful for PVF users. The documentation that is available includes the following:

- **AMD Core Math Library**– Select this option to display documentation that describes elements of the AMD Core Math Library, a software development library released by AMD that includes a set of useful mathematical routines optimized for AMD processors.
- **CUDA Fortran Reference**– Select this option to display the CUDA Fortran Programming Guide and Reference. This document describes CUDA Fortran, a small set of extensions to Fortran that support and build upon the CUDA computing architecture.
- **Fortran Language Reference**– Select this option to display the *PGI Fortran Reference* for PGI Visual Fortran. This document describes The Portland Group's implementation of the FORTRAN 77 and Fortran 90/95 languages and presents the Fortran language statements, intrinsics, and extension directives.
- **Installation Guide**– Select this option to display the *PVF Installation Guide*. This document provides an overview of the steps required to successfully install and license PGI Visual Fortran.
- **Release Notes**– Select this option to display the latest *PVF Release Notes*. This document describes the new features of the PVF IDE interface, differences in the compilers and tools from previous releases, and late-breaking information not included in the standard product documentation.
- **User's Guide**– Select this option to display the *PVF User's Guide*. This document provides operating instructions for both the Visual Studio integrated development environment as well as command-level compilation and general information about PGI's implementation of the Fortran language.

Licensing Submenu

From the Licensing menu, you have access to the PGI License Agreement and an automated license generating tool:

- **Generate License**– Select this option to display the PGI License Setup dialog that walks you through the steps required to download and install a license for PVF. To complete this process you need an internet connection.
- **License Agreement**– Select this option to display the license agreement that is associated with use of PGI software.

Introduction to PVF

This section provides an introduction to PGI Visual Fortran as well as basic information about how things work in Visual Studio. It contains an example of how to create a PVF project that builds a simple application, along with the information on how to run and debug this application from within PVF. If you're already familiar with PVF or are comfortable with project creation in VS, you may want to skip ahead to the next section.

Visual Studio Settings

PVF projects and settings are available as with any other language. The first time Visual Studio is started it may display a list of default settings from which to choose; select *General Development Settings*. If Visual Studio was installed prior to the PVF install, it will start as usual after PVF is installed, except PVF projects and settings will be available.

Solutions and Projects

The Visual Studio IDE frequently uses the terms solution and project. For consistency of terminology, it is useful to discuss these here.

solution
: All the things you need to build your application, including source code, configuration settings, and build rules. You can see the graphical representation of your solution in the Solution Explorer window in the VS IDE.

project
: Every solution contains one or more projects. Each project produces one output, where an output is an executable, a static library, or a dynamic-link library (DLL). Each project is specific to a single programming language, like PGI Visual Fortran or Microsoft Visual C++, but you can have projects of different languages in the same solution.

We examine the relationship between a solution and its projects in more detail by using an example. But first let's look at an overview of the process. Typically there are these steps:

1. Create a new PVF project.
2. Modify the source.
3. Build the solution.
4. Run the application.
5. Debug the application.

Creating a Hello World Project

Let's walk through how to create a PVF solution for a simple program that prints `"Hello World"`.

Step 1: Create Hello World Project

Follow these steps to create a PVF project to run "Hello World".

1. Select *File* | *New* | *Project* from the Visual Studio main menu.

   The *New Project* dialog appears.

2. In the *Project types* window located in the left pane of the dialog box, expand *PGI Visual Fortran*, and then select *Win32*.
3. In the *Templates* window located in the right pane of the dialog box, select *Console Application (32-bit)*.
4. In the *Name* field located at the bottom of the dialog box, type: `HelloWorld`.
5. Click OK.

You should see the Solution Explorer window in PVF. If not, you can open it now using *View* | *Solution Explorer* from the main menu. In this window you should see a solution named HelloWorld that contains a PVF project, which is also named HelloWorld.

Step 2: Modify the Hello World Source

The project contains a single source file called `ConsoleApp.f90`. If the source file is not already opened in the editor, open it by double-clicking the file name in the Solution Explorer. The source code in this file should look similar to this:

```
program main
implicit none
! Variables
! Body
end program main
```

Now add a print statement to the body of the main program so this application produces output. For example, the new program may look similar to this:

```
program main
implicit none
! Variables
! Body
print *, "Hello World"
end program main
```

Step 3: Build the Solution

You are now ready to build a solution. To do this, from the main menu, select *Build* | *Build Solution*.

The *View* | *Output* window shows the results of the build.

Step 4: Run the Application

To run the application, select *Debug* | *Start Without Debugging*.

This action launches a command window in which you see the output of the program. It looks similar to this:

```
Hello World
Press any key to continue . . .
```

Step 5: View the Solution, Project, and Source File Properties

The solution, projects, and source files that make up your application have properties associated with them.

Note

The set of property pages and properties may vary depending on whether you are looking at a solution, a project, or a file. For a description of the property pages that PVF supports, refer to Chapter 23, "*PVF Properties*," on page 313.

To see a solution's properties:

1. Select the solution in the Solution Explorer.
2. Right-click to bring up a context menu.
3. Select the *Properties* option.

   This action brings up the Property Pages dialog.

To see the properties for a project or file:

1. Select a project or a file in the Solution Explorer.
2. Right-click to bring up a context menu.
3. Select the *Properties* option.

   This action brings up the Property Pages dialog.

At the top of the Property Pages dialog there is a box labeled *Configuration*. In a PVF project, two configurations are created by default:

- The **Debug** configuration has properties set to build a version of your application that can be easily examined and controlled using the PVF debugger.
- The **Release** configuration has properties set so a version of your application is built with some general optimizations.

When a project is initially created, the Debug configuration is the active configuration. When you built the HelloWorld solution in "Creating a Hello World Project," on page 4, you built and ran the Debug configuration of your project. Let's look now at how to debug this application.

Step 6: Run the Application Using the Debugger

To debug an application in PVF:

1. Set a breakpoint on the print statement in `ConsoleApp.f90`.

   To set a breakpoint, left-click in the far left side of the editor on the line where you want the breakpoint. A red circle appears to indicate that the breakpoint is set.

2. Select *Debug* | *Start Debugging* from the main menu to start the PGI Visual Fortran debug engine.

   The debug engine stops execution at the breakpoint set in Step 1.

3. Select *Debug* | *Step Over* to step over the print statement. Notice that the program output appears in a PGI Visual Fortran console window.
4. Select *Debug* | *Continue* to continue execution.

   The program should exit.

For more information about building and debugging your application, refer to Chapter 2, "*Build with PVF*" and Chapter 3, "*Debug with PVF*". Now that you have seen a complete example, let's take a look at more of the functionality available in several areas of PVF.

Using PVF Help

The PVF User's Guide and PGI Fortran Reference are accessible in PDF form from the Visual Studio Help menu:

Help | PGI Visual Fortran Help

Help | PGI Fortran Reference

Both of these documents, and all other PGI documentation installed with PVF, are also available from the *PGI Visual Fortran | Documentation* folder off the Start Menu.

Context-sensitive (<F1>) help is not currently supported in PVF.

PVF Sample Projects

The PVF installation includes several sample solutions, available from the PVF installation directory, typically in a directory called `Samples`:

```
$(VSInstallDir)\PGI Visual Fortran\Samples\
```

These samples provide simple demonstrations of specific PVF project and solution types.

In the `dlls` subdirectory of the `Samples` directory, you find this sample program:

`pvf_dll`
: Creates a DLL that exports routines written in Fortran.

In the `gpu` subdirectory of the `Samples` directory, you find these sample programs which require a PGI Accelerator License to compile and a GPU to run.

`AccelPM_Matmul`
: Uses directives from the PGI Accelerator Programming Model to offload a matmul computation to a GPU.

`CUDAFor_Matmul`
: Uses CUDA Fortran to offload a matmul computation to a GPU.

In the `interlanguage` subdirectory of the `Samples` directory, you find these sample programs which require that Visual C++ be installed to build and run:

`pvf_calling_vc`
: Creates a solution containing a Visual C++ static library, where the source is compiled as C, and a PVF main program that calls it.

`vcmain_calling_pvfdll`
: Calls a routine in a PVF DLL from a main program compiled by VC++.

In the `win32api` subdirectory of the `Samples` directory, you find this sample program:

`menu_dialog`
: Uses a resource file and Win32 API calls to create and control a menu and a dialog box.

Compatibility

PGI Visual Fortran provides features that are compatible with those supported by older Windows Fortran products, such as Compaq® Visual Fortran. These include:

- Win32 API Support (`dfwin`)
- Unix/Linux Portability Support (`dflib, dfport`)
- Graphical User Interface Support

PVF provides access to a number of libraries that export C interfaces by using Fortran modules. This is the mechanism used by PVF to support the Win32 Application Programming Interface (API) and Unix/Linux portability libraries. If `C:` is your system drive, and <target> is your target system, such as win64, then source code containing the interfaces in these modules is located here:

```
C:\Program Files\PGI\<target>\<release_number>\src\
```

For more information about the specific functions in `dfwin`, `dflib`, and `dfport`, refer to Chapter 25, *"Fortran Module/Library Interfaces for Windows"*.

Win32 API Support (dfwin)

The Microsoft Windows operating system interface (the system call and library interface) is known collectively as the Win32 API. This is true for both the 32-bit and 64-bit versions of Windows; there is no "Win64 API" for 64-bit Windows. The only difference on 64-bit systems is that pointers are 64-bits rather than the 32-bit pointers found on 32-bit Windows.

PGI Visual Fortran provides access to the Win32 API using Fortran modules. For details on specific Win32 API routines, refer to the Microsoft MSDN website.

For ease of use, the only module you need to use to access the Fortran interfaces to the Win32 API is `dfwin`. To use this module, simply add the following line to your Fortran code.

```
use dfwin
```

Table 1.1 lists all of the Win32 API modules and the Win32 libraries to which they correspond.

Table 1.1. PVF Win32 API Module Mappings

PVF Fortran Module	C Win32 API Lib	C Header File
advapi32	advapi32.lib	WinBase.h
comdlg32	comdlg32.lib	ComDlg.h
gdi32	gdi32.lib	WinGDI.h
kernel32	kernel32.lib	WinBase.h
shell32	shell32.lib	ShellAPI.h
user32	user32.lib	WinUser.h
winver	winver.lib	WinVer.h
wsock32	wsock32.lib	WinSock.h

Unix/Linux Portability Interfaces (dflib, dfport)

PVF also includes Fortran module interfaces to libraries supporting some standard C library and Unix/Linux system call functionality. These functions are provided by the `dflib` and `dfport` modules. To utilize these modules add the appropriate `use` statement:

```
use dflib
```

```
use dfport
```

For more information about the specific functions in `dflib` and `dfport`, refer to Chapter 25, *"Fortran Module/Library Interfaces for Windows"*.

Windows Applications and Graphical User Interfaces

Programs that manage graphical user interface components using Fortran code are referred to as Windows Applications within PVF.

PVF Windows Applications are characterized by the lack of a `PROGRAM` statement. Instead, Windows Applications must provide a `WinMain` function like the following:

Example 1.1. PVF WinMain for Win32

```
integer(4) function WinMain (hInstance,&
                             hPrevInstance, lpszCmdLine, nCmdShow)
integer(4) hInstance
integer(4) hPrevInstance
integer(4) lpszCmdLine
integer(4) nCmdShow
```

Example 1.2. PVF WinMain for x64

```
integer(4) function WinMain (hInstance,&
                             hPrevInstance, lpszCmdLine, nCmdShow)
integer(8) hInstance
integer(8) hPrevInstance
integer(8) lpszCmdLine
integer(4) nCmdShow
```

`nCmdShow` is an integer specifying how the window is to be shown. Since `hInstance`, `hPrevInstance`, and `lpszCmdLine` are all pointers, in a 32-bit program they must be 4-byte integers; in a 64-bit program, they must be 8-byte integers. For more details you can look up `WinMain` in the Microsoft Platform SDK documentation.

You can create a PVF Windows Application template by selecting Windows Application in the PVF New Project dialog. The project type of this name provides a default implementation of `WinMain`, and the project's properties are configured appropriately. You can also change the Configuration Type property of another project type to Windows Application using the General property page, described in "General Property Page," on page 327. If you do this, the configuration settings change to expect `WinMain` instead of `PROGRAM`, but a `WinMain` implementation is not provided.

For an illustration of how to build a small application that uses `WinMain`, see the `menu_dialog` sample program available in the sample programs area:

```
$(VSInstallDir)\PGI Visual Fortran\Samples\win32api\menu_dialog
```

Building Windows Applications from the Command Line

Windows Applications can also be built using a command line version of pgfortran. To enable this feature, add the `-winapp` option to the compiler driver command line when linking the application. This option causes the linker to include the correct libraries and object files needed to support a Windows Application. However,

it does not add any additional system libraries to the link line. Add any required system libraries by adding the option `-defaultlib:<library name>` to the link command line for each library. For this option, <library name> can be any of the following: advapi32, comdlg32, gdi32, kernel32, shell32, user32, winver, or wsock32.

For more information about the specific functions in each of these libraries, refer to Chapter 25, "*Fortran Module/Library Interfaces for Windows*".

Menus, Dialog Boxes, and Resources

The use of resources in PVF is similar to their use in Visual C++. The resource files that control menus and dialog boxes have the file extension `.rc`. These files are processed with the Microsoft Resource Compiler to produce binary `.res` files. A `.res` file is then directly passed to the linker which incorporates the resources into the output file. See the PVF sample project `menu_dialog` for details on how resources are used within a windows application.

Note

The complete Visual C++ Resource Editor is not available in PVF. Although you can edit files like icons (`.ico`) and bitmaps (`.bmp`) directly, the `.rc` file is not updated automatically by the environment. You must either install Visual C++, in which case the resource editor is fully functional, or you must edit `.rc` files using the source code (text) editor.

Chapter 2. Build with PVF

This chapter describes how to use PGI Visual Fortran (PVF) within the Microsoft Visual Studio IDE (Integrated Development Environment) to create and build a PVF project.

For information on general use of Visual Studio, see the Visual Studio integrated help. PVF runs within Visual Studio, so to invoke PVF you must invoke Visual Studio. If other languages such as Visual C++ or Visual Basic are installed, they will be available in the same instance of Visual Studio as PVF.

Creating a PVF Project

PVF Project Types

Once Visual Studio is running, you can use it to create a PGI Visual Fortran project. PVF supports a variety of project types:

- **Console Application** - An application (`.exe`) that runs in a console window, using text input and output.
- **Dynamic Library** - A dynamically-linked library file (`.dll`) that provides routines that can be loaded on-demand when called by the program that needs them.
- **Static Library** - An archive file (`.lib`) containing one or more object files that can be linked to create an executable.
- **Windows Application** - An application (`.exe`) that supports a graphical user interface that makes use of components like windows, dialog boxes, menus, and so on. The name of the program entry point for such applications is `WinMain`.
- **Empty Project** - A skeletal project intended to allow migration of existing applications to PVF. This project type does not include any source files. By default, an empty project is set to build an application (`.exe`).

Creating a New Project

To create a new project, follow these steps:

1. Select *File* | *New* | *Project* from the File menu.

   The *New Project* dialog appears.

2. In the left-hand pane of the dialog, select *PGI Visual Fortran*.

The right-hand pane displays the icons that correspond to the project types listed in "PVF Project Types," on page 11.

Note

On x64 systems, 32-bit and 64-bit project types are clearly labeled. These types may be filtered using the 32-bit and 64-bit folders in the left-hand navigation pane of the dialog.

3. Select the project type icon corresponding to the project type you want to create.
4. Name the project in the edit box labeled *Name*.

Tip

The name of the first project in a solution is also used as the name of the solution itself.

5. Select where to create the project in the edit box labeled *Location*.
6. Click OK and the project is created.

Now look in the Solution Explorer to see the newly created project files and folders.

PVF Solution Explorer

PVF uses the standard Visual Studio Solution Explorer to organize files in PVF projects.

Tip

If the Solution Explorer is not already visible in the VS IDE, open it by selecting *View* | *Solution Explorer*.

Visual Studio uses the term project to refer to a set of files, build rules, and so on that are used to create an output like an executable, DLL, or static library. Projects are collected into a solution, which is composed of one or more projects that are usually related in some way.

PVF projects are reference-based projects, which means that although there can be folders in the representation of the project in the Solution Explorer, there are not necessarily any corresponding folders in the file system. Similarly, files added to the project can be located anywhere in the file system; adding them to the project does not copy them or move them to a project folder in the file system. The PVF project system keeps an internal record of the location of all the files added to a project.

Adding Files to a PVF Project

This section describes how to add a new file to a project and how to add an existing file to a project.

Add a New File

To add a new file to a PVF project, follow these steps:

1. Use the Solution Explorer to select the PVF project to which you want to add the new file.

2. Right-click on this PVF project to bring up a context menu.
3. Select *Add => New Item...*
4. In the *Add New Item* dialog box, select a file type from the available templates.
5. A default name for this new file will be in the *Name* box. Type in a new name if you do not want to use the default.
6. Click Add.

Add an Existing File

To add an existing file to a PVF project, follow these steps:

1. Use the Solution Explorer to select the PVF project to which you want to add the new file.
2. Right-click on this PVF project to bring up a context menu.
3. Select *Add => Existing Item…*
4. In the Browse window that appears, navigate to the location of the file you want to add.
5. Select the file and click Add.

Tip

You can add more than one file at a time by selecting multiple files.

Adding a New Project to a Solution

Each project produces one output, where an output is an executable, a static library, or a dynamic-link library (DLL). For example, if you want one solution to build both a library and an application that links against that library, you need two projects in the solution.

To add a project to a solution, follow these steps:

1. Use the Solution Explorer to select the solution.
2. Right-click on the solution to bring up a context menu.
3. Select *Add => New Project...*

   The Add New Project dialog appears. To learn how to use this dialog, refer to "Creating a PVF Project ," on page 11.
4. In the *Add New Project* dialog box, select a project type from the available templates.
5. When you have selected and named the new project, click OK.

Note

Each project is specific to a single programming language, like PGI Visual Fortran or Microsoft Visual C++, but you can have projects that use different languages in the same solution.

Project Dependencies and Build Order

If your solution contains more than one project, set up the dependencies for each project to ensure that projects are built in the correct order.

To set project dependencies:

1. Right-click a project in the Solution Explorer.
2. From the resulting context menu select *Project Dependencies*.

   The dialog box that opens has two tabs: Dependencies and Build Order.

   a. Use the Dependencies tab to put a check next to the projects on which the current project depends.

   b. Use the Build Order tab to verify the order in which projects will be built.

Configurations

Visual Studio projects are generally created with two default configurations: Debug and Release. The Debug configuration is set up to build a version of your application that can be easily debugged. The Release configuration is set up to build a generally-optimized version of your application. Other configurations may be created as desired using the Configuration Manager.

Platforms

In Visual Studio, the platform refers to the operating system for which you are building your application. In a PVF project on a system running a 32-bit Windows OS, only the Win32 platform is available. In a PVF project on a system running a 64-bit Windows OS, both the Win32 and x64 platforms are available.

When you create a new project, you select its default platform. When more than one platform is available, you can add additional platforms to your project once it exists. To do this, you use the Configuration Manager.

Setting Global User Options

Global user options are settings that affect all Visual Studio sessions for a particular user, regardless of which project they have open. PVF supports several global user settings which affect the directories that are searched for executables, include files, and library files. To access these:

1. From the main menu, select *Tools* | *Options...*
2. From the Options dialog, expand *Projects and Solutions*.
3. Select *PVF Directories* in the dialog's navigation pane.

The PVF Directories page has two combo boxes at the top:

- **Platform** allows selection of the platform (i.e., x64).
- **Show directories for** allows selection of the search path to edit.

  Search paths that can be edited include the *Executable files* path, the *Include and module files* path, and the *Library files* path.

Tip

It is good practice to ensure that all three paths contain directories from the same release of the PGI compilers; mixing and matching different releases of the compiler executables, include files, and libraries can have undefined results.

Setting Configuration Options using Property Pages

Visual Studio makes extensive use of property pages to specify configuration options. Property pages are used to set options for compilation, optimization and linking, as well as how and where other tools like the debugger operate in the Visual Studio environment. Some property pages apply to the whole project, while others apply to a single file and can override the project-wide properties.

You can invoke the *Property Page* dialog in several ways:

- Select *Project* | *Properties* to invoke the property pages for the currently selected item in the Solution Explorer. This item may be a project, a file, a folder, or the solution itself.
- Right-click a project node in the Solution Explorer and select *Properties* from the resulting context menu to invoke that project's property pages.
- Right-click a file node in the Solution Explorer and select *Properties* from the context menu to invoke that file's property pages.

The Property Page dialog has two combo boxes at the top: **Configuration** and **Platform**. You can change the configuration box to *All Configurations* so the property is changed for all configurations.

Tip

A common error is to change a property like 'Additional Include Directories' for the Debug configuration but not the Release configuration, thereby breaking the build of the Release configuration.

Chapter 18, "*Command-Line Options Reference*," on page 173 contains descriptions of compiler options in terms of the corresponding command-line switches. For compiler options that can be set using the PVF property pages, the description of the option includes instructions on how to do so.

Property Pages

Properties, or configuration options, are grouped into property pages. Further, property pages are grouped into categories. Depending on the type of project, the set of available categories and property pages vary. The property pages in a PVF project are organized into the following categories:

- General
- Debugging
- Fortran
- Linker
- Librarian
- Resources
- Build Events
- Custom Build Step

Tip

The Fortran, Linker and Librarian categories contain a Command Line property page where the command line derived from the properties can be seen. Options that are not supported by the PVF property pages can be added to the command line from this property page by entering them in the Additional Options field.

Table 2.1 shows the properties associated with each property page, listing them in the order in which you see them in the Properties dialog box. For a complete description of each property, refer to Chapter 23, "*PVF Properties*," on page 313.

Table 2.1. Property Summary by Property Page

This Property Page...	Contains these properties...
General Property Page	Output Directory Intermediate Directory Extensions to Delete on Clean Configuration Type Build Log File Build Log Level
Debugging Property Page	Application Command Application Arguments Environment Merge Environment MPI Debugging Working Directory [Serial] Number of Processes [Local MPI] Working Directory [Local MPI] Additional Arguments: mpiexec [Local MPI] Location of mpiexec [Local MPI] Number of Cores [Cluster MPI] Working Directory [Cluster MPI] Standard Input [Cluster MPI] Standard Output [Cluster MPI] Standard Error [Cluster MPI] Additional Arguments: job submit [Cluster MPI] Additional Arguments: mpiexec [Cluster MPI] Location of job.exe [Cluster MPI]
Fortran | General	Display Startup Banner Additional Include Directories Module Path Object File Name Debug Information Format Optimization

This Property Page...	Contains these properties...
Fortran | Optimization	Optimization Global Optimizations Vectorization Inlining Use Frame Pointer Loop Unroll Count Auto-Parallelization
Fortran | Preprocessing	Preprocess Source File Additional Include Directories Ignore Standard Include Path Preprocessor Definitions Undefine Preprocessor Definitions
Fortran | Code Generation	Runtime Library
Fortran | Language	Fortran Dialect Treat Backslash as Character Extend Line Length Process OpenMP Directives MPI Enable CUDA Fortran CUDA Fortran Register Limit CUDA Fortran Use Fused Multiply-Adds CUDA Fortran Use Fast Math Library CUDA Fortran Toolkit CUDA Fortran Compute Capability CUDA Fortran CC 1.0 CUDA Fortran CC 1.1 CUDA Fortran CC 1.2 CUDA Fortran CC 1.3 CUDA Fortran CC 2.0 CUDA Fortran Keep Binary CUDA Fortran Keep Kernel Source CUDA Fortran Keep PTX CUDA Fortran Emulation
Fortran | Floating Point Options	Floating Point Exception Handling Floating Point Consistency Flush Denormalized Results to Zero Treat Denormalized Values as Zero IEEE Arithmetic
Fortran | External Procedures	Calling Convention String Length Arguments Case of External Names

This Property Page...	Contains these properties...
Fortran | Target Processors	AMD Athlon AMD Barcelona AMD Istanbul AMD Shanghai Intel Core 2 Intel Core i7 Intel Penryn Intel Pentium 4 Generic x86 [Win32 only] Generic x86-64 [x64 only]
Fortran | Target Accelerators	Target NVIDIA Accelerator NVIDIA: Register Limit NVIDIA: Use Fused Multiply-Adds NVIDIA: Use Fast Math Library NVIDIA: Use 24-bit Subscript Multiplication NVIDIA: Synchronous Kernel Launch NVIDIA: CUDA Toolkit NVIDIA: Compute Capability NVIDIA: CC 1.0 NVIDIA: CC 1.1 NVIDIA: CC 1.2 NVIDIA: CC 1.3 NVIDIA: CC 2.0 NVIDIA: Keep Kernel Binary NVIDIA: Keep Kernel Source NVIDIA: Keep Kernel PTX NVIDIA: Enable Profiling NVIDIA: Analysis Only Target Host

This Property Page...	Contains these properties...
Fortran | Diagnostics	Warning Level Generate Assembly Annotate Assembly Accelerator Information CCFF Information Fortran Language Information Inlining Information IPA Information Loop Intensity Information Loop Optimization Information LRE Information OpenMP Information Optimization Information Parallelization Information Unified Binary Information Vectorization Information
Fortran | Profiling	Function-Level Profiling Line-Level Profiling MPI Suppress CCFF Information Enable Limited DWARF
Fortran | Command Line	All options (read-only contents box) Additional options (contents box you can modify)
Linker | General	Output File Additional Library Directories Export Symbols
Linker | Input	Additional Dependencies
Linker | Command Line	All options (read-only contents box) Additional options (contents box you can modify)
Librarian | General	Output File Additional Library Directories Additional Dependencies
Librarian | Command Line	All options (read-only contents box) Additional options (contents box you can modify)
Resources | Command Line	All options (read-only contents box) Additional options (contents box you can modify)
Build Events | Pre-Build Event	Command Line Description Excluded From Build

This Property Page...	Contains these properties...
Build Events | Pre-Link Event	Command Line Description Excluded From Build
Build Events | Post-Build Event	Command Line Description Excluded From Build
Custom Build Step | General	Command Line Description Outputs Additional Dependencies

Setting File Properties Using the Properties Window

Properties accessed from the Property Pages dialog allow you to change the configuration options for a project or file. The term property, however, has another meaning in the context of the Properties Window. In the Properties Window *property* means attribute or characteristic.

To see a file's properties, do this:

1. Select the file in the Solution Explorer.
2. From the *View* menu, open the *Properties Window*.

Some file properties can be modified, while others are read-only.

The values of the properties in the Properties Window remain constant regardless of the Configuration (Debug, Release) or Platform (Win32, x64) selected.

Table 2.2 lists the file properties that are available in a PVF project.

Table 2.2. PVF Project File Properties

This property...	Does this...
Name	Shows the name of the selected file.
Filename	Shows the name of the selected file.
FilePath	Shows the absolute path to the file on disk. (Read-only)
FileType	Shows the registered type of the file, which is determined by the file's extension. (Read-only)

This property...	Does this...
IsFixedFormat	Determines whether the Fortran file is fixed format. `True` indicates fixed format and `False` indicates free format. To change whether a source file is compiled as fixed or free format source, set this property appropriately. PVF initially uses file extensions to determine format style: the `.f` and `.for` extensions imply fixed format, while other extensions such as `.f90` or `.f95` imply free format. **Note** The 'C' and '*' comment characters are only valid for fixed format compilation.
IsIncludeFile	A boolean value that indicates if the file is an include file. When `True`, PVF considers the file to be an include file and it does not attempt to compile it. When `False`, if the filename has a supported Fortran or Resource file extension, PVF compiles the file as part of the build. **Tip** You can use this property to exclude a source file from a build.
IsOutput	Indicates whether a file is produced by the build. (Read-only)
ModifiedDate	Contains the date and time that the file was last saved to disk. (Read-only)
ReadOnly	Indicates the status of the Read-Only attribute of the file on disk.
Size	Describes the size of the file on disk.

Setting Fixed Format

Some Fortran source is written in fixed-format style. If your fixed-format code does not compile, check that it is designated as fixed-format in PVF.

Procedure 2.1. To check fixed-format in PVF, follow these steps:

1. Use the Solution Explorer to select a file: View | Solution Explorer.
2. Open the Properties Window: View | Other Windows | Properties Window.
3. From the dropdown list for the file property *IsFixedFormat*, select *True*.

Building a Project with PVF

Once a PVF project has been created, populated with source files, and any necessary configuration settings have been made, the project can be built. The easiest way to start a build is to use the *Build | Build Solution* menu selection; all projects in the solution will be built.

If there are compile-time errors, the *Error List* window is displayed, showing a summary of the errors that were encountered. If the error message shows a line number, then double-clicking the error record in the *Error List* window will navigate to the location of the error in the editor.

When a project is built for the first time, PVF must determine the build dependencies. Build dependencies are the result of `USE` or `INCLUDE` statements or `#include` preprocessor directives in the source. In particular, if file A contains a `USE` statement referring to a Fortran module defined in file B, file B must be compiled successfully before file A will compile.

To determine the build dependencies, PVF begins compiling files in alphabetical order. If a compile fails due to an unsatisfied module dependency, the offending file is placed back on the build queue and a message is printed to the *Output Window*, but not to the *Error List*. In a correct Fortran program, all dependencies will eventually be met, and the project will be built successfully. Otherwise, errors will be printed to the *Error List* as usual.

Unless the build dependencies change, subsequent builds use the build dependency information generated during the course of the initial build.

Order of PVF Build Operations

In the default PVF project build, the build operations are executed in the following order:

1. Pre-Build Event
2. Custom Build Steps for Files
3. Build Resources
4. Compile Fortran Files to Objects (using the PGI Fortran compiler)
5. Pre-Link Event
6. Build Output Files (using linker or librarian)
7. Custom Build Step for Project
8. Post-Link Event

Build Events and Custom Build Steps

PVF provides default build rules for Fortran files and Resource files. Other files are ignored unless a build action is specified using a Build Event or a Custom Build Step.

Build Events

Build events allow definition of a specific command to be executed at a predetermined point during the project build. You define build events using the property pages for the project. Build events can be specified as Pre-

Build, Pre-Link, or Post-Build. For specific information about where build events are run in the PVF build, refer to "Order of PVF Build Operations," on page 22. Build events are always run unless the project is up to date. There is no dependency checking for build events.

Custom Build Steps

Custom build steps are defined using the Custom Build Step Property Page. You can specify a custom build step for an entire project or for an individual file, provided the file is not a Fortran or Resource file.

When a custom build step is defined for a project, dependencies are not checked during a build. As a result, the custom build step only runs when the project itself is out of date. Under these conditions, the custom build step is very similar to the post-build event.

When a custom build step is defined for an individual file, dependencies may be specified. In this case, the dependencies must be out of date for the custom build step to run.

Note

The 'Outputs' property for a file-level custom build step must be defined or the custom build step is skipped.

PVF Build Macros

PVF implements a subset of the build macros supported by Visual C++ along with a few PVF-specific macros. The macro names are not case-sensitive, and they should be usable in any string field in a property page. Unless otherwise noted, macros that evaluate to directory names end with a trailing backslash ('\').

In general these items can only be changed if there is an associated PVF project or file property. For example, $(VCInstallDir) cannot be changed, while $(IntDir) can be changed by modifying the General | Intermediate Directory property.

For the names and descriptions of the build macros that PVF supports, refer to Chapter 24, "*PVF Build Macros*".

Static and Dynamic Linking

PVF supports both static and dynamic linking to the PGI and Microsoft runtime. In versions prior to release 7.1, only dynamic linking was supported.

The *Fortran* | *Code Generation* | *Runtime Library* property in a project's property pages determines which runtime library the project targets.

- For executable and static library projects, the default value of this property is **static linking** (-Bstatic). A statically-linked executable can be run on any system for which it is built; neither the PGI nor the Microsoft redistributable libraries need be installed on the target system.
- For dynamically linked library projects, the default value of this property is **dynamic linking** (-Bdynamic). A dynamically-linked executable can only be run on a system on which the PGI and Microsoft runtime redistributables have been installed.

For more information on deploying PGI-compiled applications to other systems, refer to Chapter 14, "*Distributing Files - Deployment*".

VC++ Interoperability

If Visual C++ is installed along with PVF, Visual Studio solutions containing both PVF and VC++ projects can be created. Each project, though, must be purely PVF or VC++; Fortran and C/C++ code cannot be mixed in a single project. This constraint is purely an organizational issue. Fortran subprograms may call C functions and C functions may call Fortran subprograms as outlined in Chapter 15, "*Inter-language Calling*".

For an example of how to create a solution containing a VC++ static library, where the source is compiled as C, and a PVF main program that calls into it, refer to the PVF sample project `pvf_calling_vc`.

Note

Because calling Visual C++ code (as opposed to C code) from Fortran is very complicated, it is only recommended for the advanced programmer. Further, to make interfaces easy to call from Fortran, Visual C++ code should export the interfaces using `extern "C"`.

Linking PVF and VC++ Projects

If you have multiple projects in a solution, be certain to use the same type of runtime library for all the projects. Further, if you have Microsoft VC++ projects in your solution, you need to be certain to match the runtime library types in the PVF projects to those of the VC++ projects.

PVF's property *Fortran* | *Code Generation* | *Runtime Library* corresponds to the Microsoft VC++ property named *C/C++* | *Code Generation* | *Runtime Library*. Table 2.3 lists the appropriate combinations of Runtime Library property values when mixing PVF and VC++ projects.

Table 2.3. Runtime Library Values for PVF and VC++ Projects

If PVF uses ...	VC++ should use...
Multi-threaded (-Bstatic)	Multi-threaded (/MT)
Multi-threaded DLL (-Bdynamic)	Multi-threaded DLL (/MD)
Multi-threaded DLL (-Bdynamic)	Multi-threaded debug DLL (/MDd)

Common Link-time Errors

The runtime libraries specified for all projects in a solution should be the same. If both PVF and VC++ projects exist in the same solution, the runtime libraries targeted should be compatible.

Keep in mind the following guidelines:

- Projects that produce DLLs should use the Multi-threaded DLL (-Bdynamic) runtime.
- Projects that produce executables or static libraries can use either type of linking.

The following examples provide a look at some of the link-time errors you might see when the runtime library targeted by a PVF project is not compatible with the runtime library targeted by a VC++ project. To resolve these errors, refer to Table 2.3 and set the Runtime Library properties for the PVF and VC++ projects accordingly.

Errors seen when linking a PVF project using -Bstatic and a VC++ library project using /MDd:

```
MSVCRTD.lib(MSVCR80D.dll) : error LNK2005: _printf already defined in
libcmt.lib(printf.obj) LINK : warning LNK4098: defaultlib 'MSVCRTD'
conflicts with use of other libs; use /NODEFAULTLIB:library test.exe :
fatal error LNK1169: one or more multiply defined symbols found
```

Errors seen when linking a PVF project using -Bstatic and a VC++ project using /MTd:

```
LIBCMTD.lib(dbgheap.obj) : error LNK2005: _malloc already defined in
libcmt.lib(malloc.obj) … LINK : warning LNK4098: defaultlib 'LIBCMTD'
conflicts with use of other libs; use /NODEFAULTLIB:library test.exe :
fatal error LNK1169: one or more multiply defined
```

Migrating an Existing Application to PVF

An existing non-PVF Fortran application or library project can be migrated to PVF. This section provides a rough outline of how one might go about such a migration.

Tip

Depending on your level of experience with Visual Studio and the complexity of your existing application, you might want to experiment with a practice project first to become familiar with the project directory structure and the process of adding existing files.

Start your project migration by creating a new Empty Project. Add the existing source and include files associated with your application to the project. If some of your source files build a library, while other files build the application itself, you will need to create a separate project within your solution for the files that build the library.

Set the configuration options using the property pages. You may need to add include paths, module paths, library dependency paths and library dependency files. If your solution contains more than one project, you will want to set up the dependencies between projects to ensure that the projects are built in the correct order.

When you are ready to try a build, select *Build* | *Build Solution* from the main menu. This action starts a full build. If there are compiler or linker errors, you will probably have a bit more build or configuration work to do.

Fortran Editing Features

PVF provides several Fortran-aware features to ease the task of entering and examining Fortran code in the Visual Studio Editor.

Source Colorization – Fortran source is colorized, so keywords, comments, and strings are distinguished from other language elements. You can use the *Tools* | *Options* | *Environment* | *Fonts and Colors* dialog to assign colors for identifiers and numeric constants, and to modify the default colors for strings, keywords and comments.

Method Tips – Fortran intrinsic functions are supported with method tips. When an opening parenthesis is entered in the source editor following an intrinsic name, a method tip pop-up is displayed that shows the data types of the arguments to the intrinsic function. If the intrinsic is a generic function supporting more than one set of arguments, the method tip window supports scrolling through the supported argument lists.

Keyword Completion – Fortran keywords are supported with keyword completion. When entering a keyword into the source editor, typing <CTRL>+<SPACE> will open a pop-up list displaying the possible completions for the portion of the keyword entered so far. Use the up or down arrow keys or the mouse to select one of the displayed items; type <ENTER> or double-click to enter the remainder of the highlighted keyword into the source. Type additional characters to narrow the keyword list or use <BACKSPACE> to expand it.

Chapter 3. Debug with PVF

PVF utilizes the Visual Studio debugger for debugging Fortran programs. PGI has implemented a custom debug engine that provides the language-specific debugging capability required for Fortran. This debug engine also supports Visual C++.

The Debug configuration is usually used for debugging. By default, this configuration will build the application so that debugging information is provided.

The debugger can be started by clicking on the green arrow in the toolbar (looks like the 'play' button on a CD or DVD player) or by selecting *Debug* | *Start Debugging*. Then use the Visual Studio debugger controls as usual.

Windows Used in Debugging

Visual Studio uses a number of different windows to provide debugging information. Only a subset of these is opened by default in your initial debugging session. Use the *Debug* | *Windows* menu option to see a list of all the windows available and to select the one you want to open.

This section provides an overview of most of the debugging windows you can use to get information about your debug session, along with a few tips about working with some of these windows.

Autos Window

The autos window provides information about a changing set of variables as determined by the current debugging location. This window is supported for VC++ code but will not contain any information when debugging in a Fortran source file.

Breakpoints Window

The breakpoints window contains all the breakpoints that have been set in the current application. You use the breakpoints window to manage the application's breakpoints.

Note

This window is available even when the application is not being debugged.

You can disable, enable or delete any or all breakpoints from within this window.

- Double-clicking on a breakpoint opens the editor to the place in the source where the breakpoint is set.
- Right-clicking on a breakpoint brings up a context menu display that shows the conditions that are set for the breakpoint. You can update these conditions via this display.
- During debugging, each breakpoint's status is shown in this window.

Breakpoint States

A breakpoint can be enabled, disabled, or in an error state. A breakpoint in an error state indicates that it failed to bind to a code location when the program was loaded. An error breakpoint can be caused by a variety of things. Two of the most common reasons a breakpoint fails to bind are these:

- The code containing the breakpoint may be in a module (DLL) that has not yet been loaded.
- A breakpoint condition may contain a syntax error.

Breakpoints in Multi-Process Programs

When debugging a multi-process program, each user-specified breakpoint is bound on a per-process basis. When this situation occurs, the breakpoints in the breakpoints window can be expanded to reveal each bound breakpoint.

Call Stack Window

The call stack window shows the call stack based on the current debugging location. Call frames are listed from the top down, with the innermost function on the top. Double-click on a call frame to select it.

- The yellow arrow is the *instruction pointer*, which indicates the current location.
- A green arrow beside a frame indicates the frame is selected but is not the current frame.

Disassembly Window

The disassembly window shows the assembly code corresponding to the source code under debug.

Using *Step* and *Step Into* in the disassembly window moves the instruction pointer one assembly instruction instead of one source line. Whenever possible, source lines are interleaved with disassembly.

Immediate Window

The immediate window provides direct communication with the debug engine. You can type `help` in this window to get a list of supported commands.

Variable Values in Multi-Process Programs

When debugging a multi-process program, use the `print` command in the immediate window with a process/thread set to display the values of a variable across all processes at once. For example, the following command prints the value of `iVar` for all processes and their threads.

```
[*.*] print iVar
```

Locals Window

The locals window lists all variables in the current scope, providing the variable's name, value, and type. You can expand variables of type array, record, structure, union and derived type variables to view all members. The variables listed include any Fortran module variables that are referenced in the current scope.

Memory Window

The memory window lists the contents of memory at a specified address. Type an address in memory into the memory window's Address box to display the contents of memory at that address.

Modules Window

In Visual Studio, the term *module* means a program unit such as a DLL. It is unrelated to the Fortran concept of module.

The modules window displays the DLLs that were loaded when the application itself was loaded. You can view information such as whether or not symbol information is provided for a given module.

Output Window

The output window displays a variety of status messages. When building an application, this window displays build information. When debugging, the output window displays information about loading and unloading modules, and exiting processes and threads.

The output window does not receive application output such as standard out or standard error. In serial and local MPI debugging, such output is directed to a console window. In cluster MPI debugging, such output is directed to user-specified files. For more information, refer to the properties: "Standard Output ," on page 330 and "Standard Output ," on page 330.

Processes Window

The processes window displays each process that is currently being debugged. For serial debugging, there is only one process displayed. For MPI debugging, the number of processes specified in the Debugging property page determines the number of processes that display in this window. The Title column of the processes window contains the rank of each process, as well as the name of the system on which the process is running and the process id.

Switching Processes in Multi-Process Programs

Many of the debugging windows display information for one process at a time. During multi-process debugging, the information in these windows pertains to the process with focus in the processes window. The process with focus has a yellow arrow next to it.

You can change the focus from one process to another by selecting the desired process in one of these ways:

- Double-click on the process.
- Highlight the process and press <Enter>.

Registers Window

The registers window is available during debugging so you can see the value of the OS registers. Registers are shown in functional groups. The first time you use the registers window, the CPU registers are shown by default.

- To show other register sets, follow these steps:
  1. Right-click in the registers window to bring up a context menu.
  2. From the context menu, select the group of registers to add to the registers window display.
- To remove a group from the display, follow these steps:
  1. Right-click in the registers window to bring up a context menu.
  2. From the context menu, deselect the group of registers to remove from the registers window display.

Threads Window

The threads window lists the active threads in the current process. Threads are named by process and thread rank using the form "process.thread".

Note

Not all threads may be executing in user code at any given time.

Watch Window

You use the watch window during debugging to display a user-selected set of variables.

Note

If a watched variable is no longer in scope, its value is no longer valid in the watch window, although the variable itself remains listed until you remove it.

Variable Rollover

Visual Studio provides a debugging feature called *variable rollover*. This feature is available when an application in debug mode stops at a breakpoint or is otherwise suspended. To activate variable rollover, use the mouse pointer to hover over a variable in the source code editor. After a moment, the value of the variable appears as a data tip next to the mouse pointer.

The first data tip that you see is often upper level information, such as an array address or possibly the members of a user-defined type. If additional information is available for a variable, you see a plus sign in the data tip. Hovering over the plus sign expands the information. Once the expansion reaches the maximum number of lines available for display, about fifteen lines, the data tip has up and down triangles which allow you to scroll to see additional information.

You can use variable rollover to obtain information about scalars, arrays, array elements, as well as user-defined type variables and their members.

Scalar Variables

If you roll over a scalar variable, such as an integer or a real, the data tip displays the scalar's value.

Array Variables

If you roll over an array, the data tip displays the array's address.

To see the elements of an array, either roll over the specific array element's subscript operator (parenthesis), or roll over the array and then expand the data tip by moving the mouse over the plus sign in the data tip. The expanded view displays the individual array elements.

The data tip can display up to about fifteen array elements at a time. For arrays with more than fifteen elements, use the up and down arrows on the top and bottom of the expanded data tip to scroll through the other elements.

Fortran character arrays work slightly differently.

- When rolling over a single element character array, the data tip displays the value of the string. To see the individual character elements, expand the string.
- When rolling over a multi-element character array, the initial data tip contains the array's address. To see the elements of the array, expand the array. Each expanded element appears as a string, which is also expandable.

User-Defined Type Variables

User-defined types include derived types, records, structs, and unions. When rolling over a user-defined type, the initial data tip displays a condensed form of the value of the user-defined type variable, which is also expandable.

To see a member of a user-defined type, you can either roll over the specific user-defined variable directly, or roll over the user-defined type and then expand the data tip by moving the mouse over the plus sign in the data tip. The expanded view displays the individual members of the variable and their values.

The data tip can display up to about fifteen user-defined type members at a time. For user-defined types with more than fifteen members, use the up and down arrows on the top and bottom of the expanded data tip to scroll through the other members.

Debugging an MPI Application in PVF

PVF has full debugging support for MPI applications running locally and on a cluster. For specific information on how to do this, refer to "Debug an MPI Application ," on page 37.

Attaching the PVF Debugger to a Running Application

PGI Visual Fortran can debug a running application using the PVF "Attach to Process" option. PVF supports attaching to Fortran applications built for 32-bit and 64-bit native Windows systems.

PVF includes PGI compilers that build 32-bit and, on Win64, 64-bit native Windows applications. A PVF installation is all that is required to use PVF to attach to PGI-compiled native Windows applications.

The following instructions describe how to use PVF to attach to a running native Windows application. As is often true, the richest debugging experience is obtained if the application being debugged has been compiled with debug information enabled.

Attach to a Native Windows Application

To attach to a native Windows application, follow these steps:

1. Open PVF from the Start menu, invoke PVF as described in "PVF on the Start Menu," on page 1.
2. From the main *Tools* menu, select *Attach to Process…*
3. In the *Attach to:* box of the *Attach to Process* dialog, verify that **PGI Debug Engine** is selected.

   If it is not selected, follow these steps to select it:

   a. Click the *Select* button.

   b. In the *Select Code* dialog box that appears, choose *Debug these code type*s.

   c. Deselect any options that are selected and select *PGI Debug Engine*.

   d. Click OK.
4. Select the application to which you want to attach PVF from the *Available Processes* box in the *Attach to Process* dialog.

   This area of the dialog box contains the system's running processes. If the application to which you want to attach PVF is missing from this list, try this procedure to locate it:

   a. Depending on where the process may be located, select *Show processes in all sessions* or *Show processes from all users*. You can select both.

   b. Click Refresh.
5. With the application to attach to selected, click *Attach*.

   PVF should now be attached to the application.

To debug, there are two ways to stop the application:

- Set a breakpoint using *Debug* | *New Breakpoint* | *Break at Function…* and let execution stop when the breakpoint is hit.

  ### Tip

  Be certain to set the breakpoint at a line in the function that has yet to be executed.

- Use *Debug* | *Break All* to stop execution.

  With this method, if you see a message box appear that reads `There is no source code available for the current location.`, click OK. Use *Step Over (F10)* to advance to a line for which source is available.

Note

To detach PVF from the application and stop debugging, select *Debug* | *Stop Debugging*.

Using PVF to Debug a Standalone Executable

You can invoke the PVF debug engine to debug an executable that was not created by a PVF project. To do this, you invoke Visual Studio from a command shell with special arguments implemented by PVF. You can use this method in any Native Windows command prompt environment.

PGI Visual Fortran includes PGI compilers that build both 32-bit and, on Win64, 64-bit native Windows applications. A PVF installation is all that is required to use the PVF standalone executable debugging feature with PGI-compiled native Windows applications. The following instructions describe how to invoke the PGI Visual Fortran debug engine from a native Windows prompt.

Tip

The richest debugging experience is obtained when the application being debugged has been compiled and linked with debug information enabled.

Launch PGI Visual Fortran from a Native Windows Command Prompt

To launch PGI Visual Fortran from a native Windows Command Prompt, follow these steps:

1. Set the environment by opening a PVF Command Prompt window using the PVF Start menu, as described in "Shortcuts to Launch PVF," on page 1.
   - To debug a 32-bit executable, choose the 32-bit command prompt: *PVF Cmd*.
   - To debug a 64-bit executable, choose the 64-bit command prompt: *PVF Cmd (64)*.

     The environment in the option you choose is automatically set to debug a native Windows application.
2. Start PGI Visual Fortran using the executable `devenv.exe`.

   If you followed Step 1 to open the PVF Command Prompt, this executable should already be on your path.

In the PVF Command Prompt window, you must supply the switch `/PVF:DebugExe`, your executable, and any arguments that your executable requires. The following examples illustrate this requirement.

Example 3.1. Use PVF to Debug an Application

This example uses PVF to debug an application, `MyApp1.exe`, that requires no arguments.

```
CMD> devenv /PVF:DebugExe MyApp1
```

Example 3.2. Use PVF to Debug an Application with Arguments

This example uses PVF to debug an application, `MyApp2.exe`, and pass it two arguments: arg1, arg2.

```
CMD> devenv /PVF:DebugExe MyApp2 arg1 arg2
```

Once PVF starts, you should see a Solution and Project with the same name as the name of the executable you passed in on the command line, such as `MyApp2` in the previous example.

You are now ready to use PGI Visual Fortran after a command line launch, as described in the next section.

Using PGI Visual Fortran After a Command Line Launch

Once you have started PVF from the command line, it does not matter how you started it, you are now ready to run and debug your application from within PVF.

To run your application from within PVF, from the main menu, select *Debug* | *Start Without Debugging*.

To debug your application using PVF:

1. Set a breakpoint using the *Debug* | *New Breakpoint* | *Break at Function* dialog box.
2. Enter either a function or a function and line that you know will be executed.

   ### Tip

   You can always use the routine name `MAIN` for the program's entry point (i.e. main program) in a Fortran program compiled by PGI compilers.
3. Start the application using *Debug* | *Start Debugging*.

When the debugger hits the breakpoint, execution stops and, if available, the source file containing the breakpoint is opened in the PVF editor.

Tips on Launching PVF from the Command Line

If you choose to launch PVF from a command line, here are a few tips to help you be successful:

- The path to the executable you want to debug must be specified using a full or relative path. Further, paths containing spaces must be quoted using double quotes (").
- If you specify an executable that does not exist, PVF starts up with a warning message and no solution is created.
- If you specify a file to debug that exists but is not in an executable format, PVF starts up with a warning message and no solution is created.

Chapter 4. Using MPI in PVF

Message Passing Interface (MPI) is an industry-standard application programming interface designed for rapid data exchange between processors in a cluster application. MPI is software used in computer clusters that allows many computers to communicate with one another.

PGI provides MPI support with PGI compilers and tools. You can build, run, debug, and profile MPI applications on Windows using PVF and Microsoft's implementation of MPI, MSMPI. This chapter describes how to use these capabilities and indicates some of their limitations, provides the requirements for using MPI in PVF, explains how to compile and enable MPI execution, and describes how to launch, debug, and profile your MPI application. In addition, there are tips on how to get the most out of PVF's MPI capabilities.

MPI Overview

MPI is a set of function calls and libraries that are used to send messages between multiple processes. These processes can be located on the same system or on a collection of distributed servers. Unlike OpenMP, the distributed nature of MPI allows it to work in almost any parallel environment. Further, distributed execution of a program does not necessarily mean that an MPI job must run on multiple machines.

PVF has built-in support for Microsoft's version of MPI: MSMPI.

System and Software Requirements

To use PVF's MPI features, additional Microsoft software which you can download from Microsoft must be installed on your system.

There are two types of MPI support in PVF: local MPI and cluster MPI. The type of support you need determines your system requirements.

Local MPI

To use PVF's local MPI capabilities the Microsoft HPC Pack 2008 SDK must be installed on your system. The HPC Pack 2008 SDK contains the MSMPI headers and libraries. It also contains `mpiexec`, which PVF uses to launch MPI applications.

Cluster MPI

To use PVF's cluster MPI capabilities:

- Both the Microsoft HPC Pack 2008 SDK and the HPC Pack must be installed on your system.
  - The HPC Pack 2008 SDK contains the MSMPI headers and libraries. It also contains `mpiexec`, which PVF uses to launch MPI applications.
  - The HPC Pack contains, among other things, the HPC Job Manager, which is used to launch MPI applications on a cluster.
- A Windows HPC Server 2008 cluster must be running.
- Your PGI license must be enabled to run and debug on remote nodes.

  For information on upgrading your current license to a cluster license, please contact sales@pgroup.com.

Compile using MSMPI

The PVF Fortran | Language | MPI property enables MPI compilation and linking with the Microsoft MPI headers and libraries. Set this property to *Microsoft MPI* to enable an MPI build.

Enable MPI Execution

Once your MPI application is built, you can run and debug it. The PVF Debugging property page is the key to both running and debugging an MPI application. For simplicity, in this section we use the term *execution* to mean either running or debugging the application.

Use the MPI Debugging property to determine the type of execution you need, provided you have the appropriate system configuration and license.

MPI Debugging Property Options

The MPI Debugging property can be set to one of three options: Disabled, Local, Cluster.

Disabled
: When *Disabled* is selected, execution is performed serially.

Local
: When *Local* is selected, MPI execution is performed locally. That is, multiple processes are used but all of them run on the local host.

Cluster
: When *Cluster* is selected, MPI execution is performed using multiple processes – some or all of which can be running on a remote cluster node.

Additional MPI properties become available when you select either the Local or Cluster MPI Debugging option. For more information about these properties, refer to "Debugging Property Page," on page 328.

Launch an MPI Application

As soon as you have built your MPI application, and selected Local or Cluster MPI Debugging, you can launch your executable using the *Debug* | *Start Without Debugging* menu option.

PVF uses Microsoft's version of `mpiexec` to support Local MPI execution.

PVF uses Microsoft's HPC Job Manager to support Cluster MPI execution. PVF submits jobs to the HPC Job Manager when you execute an application. To view the status of jobs you have submitted, open the HPC Job Manager interface from *Start* | *All Programs* | *Microsoft HPC Pack* | *HPC Job Manager* and use it.

Note

It is not necessary to have the HPC Job Manager interface open to use MPI in PVF.

Debug an MPI Application

To debug your MPI application, select *Debug* | *Start Debugging* or hit F5. As with running your MPI application, PVF uses `mpiexec` for Local MPI and the HPC Job Manager for Cluster MPI jobs.

PVF supports two styles of multi-process debugging: Run Altogether and Run One At a Time.

Run Altogether
: With this style of debugging, execution of all processes occurs at the same time. When you select *Continue*, all processes are continued. When one process hits a breakpoint, it stops. The other processes do not stop, however, until they hit a breakpoint or some other type of barrier. When you select *Step*, all processes are stepped. Control returns to you as soon as one or more processes finishes its step. If some process does not finish its step when the other processes are finished, it continues execution until it completes.

Run One At a Time
: With this style of debugging, processes are shown as advancing one at a time. For example, on start-up all processes run to a breakpoint. In the source pane, you see the first process that reaches the breakpoint. When you select *Continue*, execution switches and you see the next process at that same breakpoint. When you select *Continue* again, execution switches to show you the next process at that same breakpoint, and so on. Once all processes have been shown at the first breakpoint, selecting *Continue* advances the processes one at a time to the next breakpoint.

  In this style of debugging, *Step* works in a manner similar to *Continue*, stepping each process one at a time.

  Note

  This style of debugging is not available for use with Microsoft Visual Studio 2005 or Microsoft Visual Studio 2008.

To select the debugging style, open the Tools | Options menu and navigate to the Debugging | General page. To enable the Run Altogether style, check the box labeled Break All Processes When One Process Breaks. To enable the Run One At a Time style, uncheck that box.

Note

Changing styles during a debug session is not supported. Checking or unchecking the Break All Processes option has no effect until the next time you debug.

Additionally, PVF 2005 supports Run Altogether only; the status of the Break All Processes option has no effect with PVF 2005.

Tips and Tricks

To get the most out of PVF's MPI capabilities, here are a few techniques to keep in mind.

- Cluster Debugging can only be launched from a working directory that is designated as shared among all cluster nodes. If the working directory is not shared, your application may fail to launch. Specify the working directory using the PVF Debugging property page. In you need assistance in designating your working directory as shared, contact your system or cluster administrator.
- Standard I/O in Cluster Debugging comes from and goes to files specified in PVF's Debugging property pages. If you do not specify a file for standard output, you can view the output of your application using the Job Manager interface.
- Use the HPC Job Manager to view the status of jobs launched from PVF.

  Note

  Jobs that queue at the start of a debug session prevent PVF from completing its launch, that is, if you are starting to debug and see no activity, it may seem like PVF is not working. Use the HPC Job Manager to determine why the job has queued, and to unqueue or cancel it.

- To run a dynamically-linked application on a cluster from within PVF, the application must be able to find the dynamically-linked libraries upon which it depends. To ensure this is true, do the following:

  1. Copy the PGI runtime DLLs, located in the PGI compilers' REDIST directory, to *all* the nodes. In the following paths, <rel> is the release version, such as 10.0-1.

     32-bit Windows:
     `C:\Program Files\PGI\win32\<rel>\REDIST`
     64-bit Windows:
     `C:\Program Files\PGI\win64\<rel>\REDIST`
     `C:\Program Files (x86)\PGI\win32\<rel>\REDIST`

  2. Make certain that the system PATH environment variable on *every* node, including the head node, includes the path to the PGI runtime DLLs.

Profile an MPI Application

The PGI profiling tool PGPROF is included with PVF. The process of profiling involves these basic steps:

1. Build the application with profiling options enabled.
2. Run the application to generate profiling output.
3. Analyze the profiling output with a profiler.

To build a profiling-enabled application in PVF, use the Profiling property page. Enable MPICH-style profiling for Microsoft MPI by setting the MPI property to `Microsoft MPI`. Doing this adds the `-Mprof=msmpi`

option to compilation and linking. You must also enable either Function-Level Profiling (`-Mprof=func`) or Line-Level Profiling (`-Mprof=lines`).

The profile data generated by running an application built with the option `-Mprof=msmpi` contains information about the number of sends and receives, as well as the number of bytes sent and received, correlated with the source location associated with the sends and receives.

Once you've built your application with the profiling properties enabled, run your application to generate the profiling information files. These files, named pgprof*.out, contain the profiled output of the application's run.

Launch PGPROF from the PVF start menu via *Start* | *All Programs* | *PGI Visual Fortran* | *Profiler* | *PGPROF Performance Profiler*. Once PGPROF is running, use the File | New Profiling Session menu option to specify the location of the `pgprof.out` files and your application. For details on using `pgprof`, refer to the online Help available within the application.

MSMPI Environment

Tip

This section provides information about the MSMPI environment that is set up for you "behind-the-scenes". It is not necessary for you to define or set default values for these variables. Knowing they exist may be useful if your programs have difficulty locating include files or libraries.

When the Microsoft HPC Pack and HPC Pack 2008 SDK are installed, some system environment variables are set. Further, there are two environment variables available to help you specify directory locations associated with using MSMPI on Windows: `CCP_HOME` and `CCP_SDK`.

- `CCP_HOME` specifies the root directory of the Microsoft cluster management software for systems on which the Microsoft HPC Pack 2008 is installed.
- `CCP_SDK` specifies the root directory of the MSMPI software for systems on which Microsoft's HPC Pack 2008 SDK is installed.

If the appropriate environment variable is set for the version of MSMPI that you are using, then the PVF MPI property page use of `-Mmpi=msmpi` automatically brings in the appropriate include files and libraries.

Chapter 5. Getting Started with the Command Line Compilers

This chapter describes how to use the command-line PGI compilers. The PGI Visual Fortran IDE invokes the PGI compilers when you build a PVF project. You can invoke the compilers directly from the Start menu using the appropriate command prompt for your system, as described in "Shortcuts to Launch PVF," on page 1.

The command used to invoke a compiler, such as the pgfortran command, is called a *compiler driver*. The compiler driver controls the following phases of compilation: preprocessing, compiling, assembling, and linking. Once a file is compiled and an executable file is produced, you can execute, debug, or profile the program on your system. Executables produced by the PGI compilers are unconstrained, meaning they can be executed on any compatible x86 or x64 processor-based system, regardless of whether the PGI compilers are installed on that system.

Overview

In general, using a PGI compiler involves three steps:

1. Produce a program source code in a file containing a `.f` extension or another appropriate extension, as described in "Input Files," on page 44. This program may be one that you have written or one that you are modifying.
2. Compile the program using the appropriate compiler command.
3. Execute, debug, or profile the executable file on your system.

You might also want to deploy your application, though this is not a required step.

The PGI compilers allow many variations on these general program development steps. These variations include the following:

- Stop the compilation after preprocessing, compiling or assembling to save and examine intermediate results.
- Provide options to the driver that control compiler optimization or that specify various features or limitations.

- Include as input intermediate files such as preprocessor output, compiler output, or assembler output.

Invoking the Command-level PGI Compilers

To translate and link a Fortran language program, the pgf77, pgf95, and pgfortran commands do the following:

1. Preprocess the source text file.
2. Check the syntax of the source text.
3. Generate an assembly language file.
4. Pass control to the subsequent assembly and linking steps.

Example 5.1. Hello program

Let's look at a simple example of using the PGI compiler to create, compile, and execute a program that prints *hello*.

Step 1: Create your program.

For this example, suppose you enter the following simple Fortran program in the file `hello.f`:

```
print *, "hello"
 end
```

Step 2: Compile the program.

When you created your program, you called it `hello.f`. In this example, we compile it from a shell command prompt using the default pgfortran driver option. Use the following syntax:

```
PGI$ pgfortran hello.f
PGI$
```

By default, the executable output is placed in a filename based on the name of the first source or object file on the command line. However, you can specify an output file name by using the `-o` option.

To place the executable output in the file `hello`, use this command:

```
PGI$ pgfortran -o hello hello.f
PGI$
```

Step 3: Execute the program.

To execute the resulting hello program, simply type the filename at the command prompt and press the Return or Enter key on your keyboard:

```
PGI$ hello
 hello
PGI$
```

Command-line Syntax

The compiler command-line syntax, using pgfortran as an example, is:

```
pgfortran [options] [path]filename [...]
```

Where:

options
: is one or more command-line options, all of which are described in detail in Chapter 6, "*Using Command Line Options*".

path
: is the pathname to the directory containing the file named by filename. If you do not specify the path for a filename, the compiler uses the current directory. You must specify the path separately for each filename not in the current directory.

filename
: is the name of a source file, preprocessed source file, assembly-language file, object file, or library to be processed by the compilation system. You can specify more than one [path]filename.

Command-line Options

The command-line options control various aspects of the compilation process. For a complete alphabetical listing and a description of all the command-line options, refer to Chapter 6, "*Using Command Line Options*".

The following list provides important information about proper use of command-line options.

- Case is significant for options and their arguments.
- The compiler drivers recognize characters preceded by a hyphen (-) as command-line options. For example, the `-Mlist` option specifies that the compiler creates a listing file.

  ### Note

  The convention for the text of this manual is to show command-line options using a dash instead of a hyphen; for example, you see `-Mlist`.

- The order of options and the filename is flexible. That is, you can place options before and after the filename argument on the command line. However, the placement of some options is significant, such as the `-l` option, in which the order of the filenames determines the search order.

  ### Note

  If two or more options contradict each other, the *last* one in the command line takes precedence.

Fortran Directives

You can insert Fortran directives in program source code to alter the effects of certain command-line options and to control various aspects of the compilation process for a specific routine or a specific program loop. For more information on Fortran directives, refer to Chapter 9, "*Using OpenMP*" and Chapter 11, "*Using Directives*".

Filename Conventions

The PGI compilers use the filenames that you specify on the command line to find and to create input and output files. This section describes the input and output filename conventions for the phases of the compilation process.

Input Files

You can specify assembly-language files, preprocessed source files, Fortran source files, object files, and libraries as inputs on the command line. The compiler driver determines the type of each input file by examining the filename extensions.

Note

For systems with a case-insensitive file system, use the `-M`preprocess option, described in Chapter 18, "*Command-Line Options Reference*", under the commands for Fortran preprocessing.

The drivers use the following conventions:

filename.f
: indicates a Fortran source file.

filename.F
: indicates a Fortran source file that can contain macros and preprocessor directives (to be preprocessed).

filename.FOR
: indicates a Fortran source file that can contain macros and preprocessor directives (to be preprocessed).

filename.F95
: indicates a Fortran 90/95 source file that can contain macros and preprocessor directives (to be preprocessed).

filename.f90
: indicates a Fortran 90/95 source file that is in freeform format.

filename.f95
: indicates a Fortran 90/95 source file that is in freeform format.

filename.cuf
: indicates a Fortran 90/95 source file in free format with CUDA Fortran extensions.

filename.CUF
: indicates a Fortran 90/95 source file in free format with CUDA Fortran extensions and that can contain macros and preprocessor directives (to be preprocessed).

filename.s
: indicates an assembly-language file.

filename.obj
: (Windows systems only) indicates an object file.

filename.lib
: (Windows systems only) indicates a statically-linked library of object files or an import library.

filename.dll
(Windows systems only) indicates a dynamically-linked library.

The driver passes files with `.s` extensions to the assembler and files with .obj, .dll, and .lib extensions to the linker. Input files with unrecognized extensions, or no extension, are also passed to the linker.

Files with a `.fpp` suffix are first preprocessed by the Fortran compilers and the output is passed to the compilation phase. The Fortran preprocessor is built in to the Fortran compilers. This design ensures consistency in the preprocessing step regardless of the type or revision of operating system under which you're compiling.

Any input files not needed for a particular phase of processing are not processed. For example, if on the command line you specify an assembly-language file (filename.s) and the `-S` option to stop before the assembly phase, the compiler takes no action on the assembly language file. Processing stops after compilation and the assembler does not run. In this scenario, the compilation must have been completed in a previous pass which created the `.s` file. For a complete description of the `-S` option, refer to the following section:"Output Files".

In addition to specifying primary input files on the command line, code within other files can be compiled as part of include files using the `INCLUDE` statement in a Fortran source file or the `preprocessor #include` directive in Fortran source files that use a `.F` extension.

When linking a program with a library, the linker extracts only those library components that the program needs. The compiler drivers link in several libraries by default. For more information about libraries, refer to Chapter 12, "*Creating and Using Libraries*".

Output Files

By default, an executable output file produced by one of the PGI compilers is placed in the file a.out, or, on Windows, in a filename based on the name of the first source or object file on the command line. As the example in the preceding section shows, you can use the `-o` option to specify the output file name.

If you use one of the options: `-F` (Fortran only), `-S` or `-c`, the compiler produces a file containing the output of the last completed phase for each input file, as specified by the option supplied. The output file will be a preprocessed source file, an assembly-language file, or an unlinked object file respectively. Similarly, the `-E` option does not produce a file, but displays the preprocessed source file on the standard output. Using any of these options, the `-o` option is valid only if you specify a single input file. If no errors occur during processing, you can use the files created by these options as input to a future invocation of any of the PGI compiler drivers. The following table lists the stop-after options and the output files that the compilers create when you use these options. It also describes the accepted input files.

Table 5.1. Stop-after Options, Inputs and Outputs

Option	Stop after	Input	Output
–E	preprocessing	Source files.	preprocessed file to standard out
–F	preprocessing	Source files.	preprocessed file (.f)
–S	compilation	Source files or preprocessed files.	assembly-language file (.s)

Option	Stop after	Input	Output
–c	assembly	Source files, preprocessed files or assembly-language files.	unlinked object file (.obj)
none	linking	Source files, preprocessed files, assembly-language files, object files or libraries.	executable file (.exe)

If you specify multiple input files or do not specify an object filename, the compiler uses the input filenames to derive corresponding default output filenames of the following form, where filename is the input filename without its extension:

filename.f
: indicates a preprocessed file, if you compiled a Fortran file using the `–F` option.

filename.i
: indicates a preprocessed file, if you compiled using the `–P` option.

filename.lst
: indicates a listing file from the `–Mlist` option.

filename.obj
: indicates an object file from the `–c` option.

filename.s
: indicates an assembly-language file from the `–S` option.

Note

Unless you specify otherwise, the destination directory for any output file is the current working directory. If the file exists in the destination directory, the compiler overwrites it.

The following example demonstrates the use of output filename extensions.

```
$ pgfortran -c proto.f proto1.F
```

This produces the output files proto.obj and proto1.obj all of which are binary object files. Prior to compilation, the file proto1.F is preprocessed because it has a .F filename extension.

Fortran Data Types

The PGI Fortran compilers recognize scalar and aggregate data types. A scalar data type holds a single value, such as the integer value 42 or the real value 112.6. An aggregate data type consists of one or more scalar data type objects, such as an array of integer values.

For information about the format and alignment of each data 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, refer to Chapter 17, "*Fortran Data Types*".

For more information on x86-specific data representation, refer to the *System V Application Binary Interface Processor Supplement* by AT&T UNIX System Laboratories, Inc. (Prentice Hall, Inc.).

This manual 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 this ABI, see *www.x86-64.org/abi.pdf*.

Parallel Programming Using the PGI Compilers

The PGI Visual Fortran compilers support two styles of parallel programming:

- Automatic shared-memory parallel programs compiled using the `-Mconcur` option to pgf77, pgf95, or pgfortran — parallel programs of this variety can be run on shared-memory parallel (SMP) systems such as dual-core or multi-processor workstations.
- OpenMP shared-memory parallel programs compiled using the `-mp` option to pgf77, pgf95, or pgfortran — parallel programs of this variety can be run on SMP systems. Carefully coded user-directed parallel programs using OpenMP directives can often achieve significant speed-ups on dual-core workstations or large numbers of processors on SMP server systems. contains complete descriptions of user-directed parallel programming.

On a single silicon die, some newer CPUs incorporate two or more complete processor cores - functional units, registers, level 1 cache, level 2 cache, and so on. These CPUs are known as multi-core processors. For purposes of threads, or OpenMP parallelism, these cores function as two or more distinct processors. However, the processing cores are on a single chip occupying a single socket on a system motherboard. For purposes of PGI software licensing, a multi-core processor is treated as a single CPU.

Running SMP Parallel Programs

When you execute an SMP parallel program, by default it uses only one processor. To run on more than one processor, set the `NCPUS` environment variable to the desired number of processors, subject to a maximum of four for PGI's workstation-class products. For information on how to set environment variables, refer to"Setting Environment Variables," on page 137

Note

If you set NCPUS to a number larger than the number of physical processors, your program may execute very slowly.

Site-specific Customization of the Compilers

If you are using the PGI compilers and want all your users to have access to specific libraries or other files, there are special files that allow you to customize the compilers for your site.

Using siterc Files

The PGI compiler drivers utilize a file named `siterc` to enable site-specific customization of the behavior of the PGI compilers. The `siterc` file is located in the `bin` subdirectory of the PGI installation directory. Using `siterc`, you can control how the compiler drivers invoke the various components in the compilation tool chain.

Using User rc Files

In addition to the siterc file, user `rc` files can reside in a given user's home directory, as specified by the user's HOME environment variable. You can use these files to control the respective PGI compilers. All of these files are optional.

On Windows, these files are named `mypgf77rc`, `mypgf90rc`, `mypgf95rc`, and `mypgfortranrc`.

Common Development Tasks

Now that you have a brief introduction to the compiler, let's look at some common development tasks that you might wish to perform.

- When you compile code you can specify a number of options on the command line that define specific characteristics related to how the program is compiled and linked, typically enhancing or overriding the default behavior of the compiler. For a list of the most common command line options and information on all the command line options, refer to Chapter 6, "*Using Command Line Options*".
- Code optimization and parallelization allows the compiler to organize your code for efficient execution. While possibly increasing compilation time and making the code more difficult to debug, these techniques typically produce code that runs significantly faster than code that does not use them. For more information on optimization and parallelization, refer to Chapter 7, "*Optimizing & Parallelizing*".
- Function inlining, a special type of optimization, replaces a call to a function or a subroutine with the body of the function or subroutine. This process can speed up execution by eliminating parameter passing and the function or subroutine call and return overhead. In addition, function inlining allows the compiler to optimize the function with the rest of the code. However, function inlining may also result in much larger code size with no increase in execution speed. For more information on function inlining, refer to Chapter 8, "*Using Function Inlining*".
- Directives allow users to place hints in the source code to help the compiler generate better assembly code. You typically use directives to control the actions of the compiler in a particular portion of a program without affecting the program as a whole. You place them in your source code where you want them to take effect. A directive typically stays in effect from the point where included until the end of the compilation unit or until another directive changes its status. For more information on directives, refer to Chapter 9, "*Using OpenMP*" and Chapter 11, "*Using Directives*".
- A library is a collection of functions or subprograms used to develop software. Libraries contain "helper" code and data, which provide services to independent programs, allowing code and data to be shared and changed in a modular fashion. The functions and programs in a library are grouped for ease of use and linking. When creating your programs, it is often useful to incorporate standard libraries or proprietary ones. For more information on this topic, refer to Chapter 12, "*Creating and Using Libraries*".
- Environment variables define a set of dynamic values that can affect the way running processes behave on a computer. It is often useful to use these variables to set and pass information that alters the default behavior of the PGI compilers and the executables which they generate. For more information on these variables, refer to Chapter 13, " *Using Environment Variables*".
- Deployment, though possibly an infrequent task, can present some unique issues related to concerns of porting the code to other systems. Deployment, in this context, involves distribution of a specific file

or set of files that are already compiled and configured. The distribution must occur in such a way that the application executes accurately on another system which may not be configured exactly the same as the system on which the code was created. For more information on what you might need to know to successfully deploy your code, refer to Chapter 14, "*Distributing Files - Deployment*".

Chapter 6. Using Command Line Options

A command line option allows you to control specific behavior when a program is compiled and linked. This chapter describes the syntax for properly using command-line options and provides a brief overview of a few of the more common options.

Note

For a complete list of command-line options, their descriptions and use, refer to Chapter 18, "*Command-Line Options Reference*," on page 173.

Command Line Option Overview

Before looking at all the command-line options, first become familiar with the syntax for these options. There are a large number of options available to you, yet most users only use a few of them. So, start simple and progress into using the more advanced options.

By default, the PGI compilers generate code that is optimized for the type of processor on which compilation is performed, the compilation host. Before adding options to your command-line, review the sections "Help with Command-line Options," on page 52 and "Frequently-used Options," on page 55.

Command-line Options Syntax

On a command-line, options need to be preceded by a hyphen (-). If the compiler does not recognize an option, it passes the option to the linker.

This document uses the following notation when describing options:

[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.

NOTE

Some options do not allow a space between the option and its argument or within an argument. When applicable, the syntax section of the option description in Chapter 18, "*Command-Line Options Reference*," on page 173 contains this information.

Command-line Suboptions

Some options accept several suboptions. You can specify these suboptions either by using the full option statement multiple times or by using a comma-separated list for the suboptions.

The following two command lines are equivalent:

```
pgfortran -Mvect=sse -Mvect=noaltcode
```

```
pgfortran -Mvect=sse,noaltcode
```

Command-line Conflicting Options

Some options have an opposite or negated counterpart. For example, both `-Mvect` and `-Mnovect` are available. `-Mvect` enables vectorization and `-Mnovect` disables it. If you used both of these commands on a command line, they would conflict.

Note

Rule: When you use conflicting options on a command line, the last encountered option takes precedence over any previous one.

This rule is important for a number of reasons.

- Some options, such as `-fast`, include other options. Therefore, it is possible for you to be unaware that you have conflicting options.
- You can use this rule to create makefiles that apply specific flags to a set of files, as shown in Example 6.1.

Example 6.1. Makefiles with Options

In this makefile fragment, CCFLAGS uses vectorization. CCNOVECTFLAGS uses the flags defined for CCFLAGS but disables vectorization.

```
CCFLAGS=c -Mvect=sse
CCNOVECTFLAGS=$(CCFLAGS) -Mnovect
```

Help with Command-line Options

If you are just getting started with the PGI compilers and tools, it is helpful to know which options are available, when to use them, and which options most users find effective.

Using –help

The `–help` option is useful because it provides information about all options supported by a given compiler. 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>
```

Suppose you use the following command to restrict the output to information about the `-fast` option:

```
$ pgfortran -help -fast
```

The output you see is similar to this:

```
-fast Common optimizations; includes -O2 -Munroll=c:1 -Mnoframe -Mlre
```

In the following example, we add the `–help` parameter to restrict the output to information about the help command. The usage information for `–help` shows how groups of options can be listed or examined according to function.

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

- 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>
```

By using the command `pgfortran -help -help`, as previously shown, we can see output that shows the available subgroups. You can use the following command to restrict the output on the `–help` command to information about only the options related to only one group, such as debug information generation.

```
$ pgfortran -help=debug
```

The output you see is similar to this:

```
Debugging switches:
-M[no]bounds Generate code to check array bounds
-Mchkfpstk Check consistency of floating point stack at subprogram calls
 (32-bit only)
-Mchkstk Check for sufficient stack space upon subprogram entry
-Mcoff Generate COFF format object
-Mdwarf1 Generate DWARF1 debug information with -g
-Mdwarf2 Generate DWARF2 debug information with -g
-Mdwarf3 Generate DWARF3 debug information with -g
-Melf Generate ELF format object
-g Generate information for debugger
-gopt Generate information for debugger without disabling
 optimizations
```

For a complete description of subgroups, refer to “–help ,” on page 184.

Getting Started with Performance

One of the top priorities of most users is performance and optimization. This section provides a quick overview of a few of the command-line options that are useful in improving performance.

Using –fast and –fastsse Options

PGI compilers implement a wide range of options that allow users a fine degree of control on each optimization phase. When it comes to optimization of code, the quickest way to start is to use the options `-fast` and `-fastsse`. These options create a generally optimal set of flags for targets that support SSE/SSE2 capability. They incorporate optimization options to enable use of vector streaming SIMD (SSE/SSE2) instructions for 64-bit targets. They enable vectorization with SSE instructions, cache alignment, and SSE arithmetic to flush to zero mode.

Note

The contents of the `-fast` and `-fastsse` options are host-dependent. Further, you should use these options on both compile and link command lines.

- `-fast` and `-fastsse` typically include these options:

–O2	Specifies a code optimization level of 2.
–Munroll=c:1	Unrolls loops, executing multiple instances of the loop during each iteration.
–Mnoframe	Indicates to not generate code to set up a stack frame.
–Mlre	Indicates loop-carried redundancy elimination.
–Mpre	Indicates partial redundancy elimination.

- These additional options are also typically available when using `-fast` for 64-bit targets or `-fastsse` for both 32- and 64-bit targets:

–Mvect=sse	Generates SSE instructions.
–Mscalarsse	Generates scalar SSE code with xmm registers; implies –Mflushz.
–Mcache_align	Aligns long objects on cache-line boundaries.
–Mflushz	Sets SSE to flush-to-zero mode.
–M[no]vect	Controls automatic vector pipelining.

Note

For best performance on processors that support SSE instructions, use the PGFORTRAN compiler, even for FORTRAN 77 code, and the `-fast` option.

To see the specific behavior of `-fast` for your target, use the following command:

```
$ pgfortran -help -fast
```

Other Performance-related Options

While `-fast` and `-fastsse` are options designed to be the quickest route to best performance, they are limited to routine boundaries. Depending on the nature and writing style of the source code, the compiler often can perform further optimization by knowing the global context of usage of a given routine. For instance, determining the possible value range of actual parameters of a routine could enable a loop to be vectorized; similarly, determining static occurrence of calls helps to decide which routine is beneficial to inline.

These types of global optimizations are under control of Interprocedural Analysis (IPA) in PGI compilers. Option `-Mipa` enables Interprocedural Analysis. `-Mpi=fast` is the recommended option to get best performances for global optimization. You can also add the suboption `inline` to enable automatic global inlining across files. You might consider using `-Mipa=fast,inline`. This option for interprocedural analysis and global optimization can improve performance.

You may also obtain further performance improvements by experimenting with the `-M<pgflag>` options described in the section "–M Options by Category," on page 213. These options include, but are not limited to, `-Mconcur`, `-Mvect`, `-Munroll`, `-Minline`, and `-Mpfi/-Mpfo`. However, performance improvements using these options are typically application- and system-dependent. It is important to time your application carefully when using these options to ensure no performance degradations occur.

For more information on optimization, refer to Chapter 7, "*Optimizing & Parallelizing*," on page 57. For specific information about these options, refer to "Optimization Controls," on page 223.

Targeting Multiple Systems - Using the -tp Option

The `-tp` option allows you 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 sub-options 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

For more information about the `-tp` option, refer to "–tp <target> [,target...]," on page 206.

Frequently-used Options

In addition to overall performance, there are a number of other options that many users find useful when getting started. The following table provides a brief summary of these options.

For more information on these options, refer to the complete description of each option available in Chapter 18, "*Command-Line Options Reference*," on page 173. Also, there are a number of suboptions available with each of the `-M` options listed. For more information on those options, refer to "–M Options by Category," on page 213.

Table 6.1. Commonly Used Command Line Options

Option	Description
–fast –fastsse	These options create a generally optimal set of flags for targets that support SSE/SSE2 capability. They incorporate optimization options to enable use of vector streaming SIMD instructions (64-bit targets) and enable vectorization with SEE instructions, cache aligned and flushz.
–g	Instructs the compiler to include symbolic debugging information in the object module.
–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.
–Mconcur	Instructs the compiler to enable auto-concurrentization of loops. If specified, the compiler uses multiple processors to execute loops that it determines to be parallelizable; thus, loop iterations are split to execute optimally in a multithreaded execution context.
–Minfo	Instructs the compiler to produce information on standard error.
–Minline	Enables function inlining.
–Mipa=fast,inline	Enables interprocedural analysis and optimization. Also enables automatic procedure inlining.
–Mpfi or –Mpfo	Enable profile feedback driven optimizations.
–Mkeepasm	Keeps the generated assembly files.
–Munroll	Invokes the loop unroller to unroll loops, executing 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.
–M[no]vect	Enables/Disables the code vectorizer.
–o	Names the output file.
–O<level>	Specifies code optimization level where <level> is 0, 1, 2, 3, or 4.

Chapter 7. Optimizing & Parallelizing

Source code that is readable, maintainable, and produces correct results is not always organized for efficient execution from the command line. Normally, the first step in the program development process involves producing code that executes and produces the correct results. This first step usually involves compiling without much worry about optimization. After code is compiled and debugged, code optimization and parallelization become an issue.

Invoking one of the PGI compiler commands with certain options instructs the compiler to generate optimized code. Optimization is not always performed since it increases compilation time and may make debugging difficult. However, optimization produces more efficient code that usually runs significantly faster than code that is not optimized.

Note

PGI provides a profiler, PGPROF, that provides a way to visualize the performance of the components of your program. Using tables and graphs, PGPROF associates execution time and resource utilization data with the source code and instructions of your program, allowing you to see where execution time is spent. Through resource utilization data and compiler analysis information, PGPROF helps you to understand why certain parts of your program have high execution times.

To launch PGPROF, use the shortcut on the PVF Start menu: Start | All Programs | PGI Visual Fortran | Profiler | PGPROF Performance Profiler.

The compilers optimize code according to the specified optimization level. In PVF, you use the Fortran | Optimization property page to specify optimization levels; on the command line, the options you commonly use are `-O`, `-Mvect`, `-Mipa`, and `-Mconcur`. You can also use several `-M<pgflag>` switches to control specific types of optimization and parallelization. You can set the options not supported by the Fortran | Optimization property page by using the *Additional Options* field of the Fortran | Command Line property page. For more information on these property pages, refer to "Fortran Property Pages".

This chapter describes the optimization options displayed in the following list.

`-fast`	`-Minline`	`-Mpfi`	`-Mvect`
`-Mconcur`	`-Mipa=fast`	`-Mpfo`	`-O`
`-Minfo`	`-Mneginfo`	`-Munroll`	

This chapter also describes how to choose optimization options to use with the PGI compilers. This overview will help if you are just getting started with one of the PGI compilers, or wish to experiment with individual optimizations. Complete specifications of each of these options is available in Chapter 18, "*Command-Line Options Reference*".

Overview of Optimization

In general, optimization involves using transformations and replacements that generate more efficient code. This is done by the compiler and involves replacements that are independent of the particular target processor's architecture as well as replacements that take advantage of the x86 or x64 architecture, instruction set and registers. For the discussion in this and the following chapters, optimization is divided into the following categories:

Local Optimization

This optimization is performed on a block-by-block basis within a program's basic blocks. A basic block is a sequence of statements in which the flow of control enters at the beginning and leaves at the end without the possibility of branching, except at the end. The PGI compilers perform many types of local optimization including: algebraic identity removal, constant folding, common sub-expression elimination, redundant load and store elimination, scheduling, strength reduction, and peephole optimizations.

Global Optimization

This optimization is performed on a program unit over all its basic blocks. The optimizer performs control-flow and data-flow analysis for an entire program unit. All loops, including those formed by IFs and GOTOs, are detected and optimized. Global optimization includes: constant propagation, copy propagation, dead store elimination, global register allocation, invariant code motion, and induction variable elimination.

Loop Optimization: Unrolling, Vectorization, and Parallelization

The performance of certain classes of loops may be improved through vectorization or unrolling options. Vectorization transforms loops to improve memory access performance and make use of packed SSE instructions which perform the same operation on multiple data items concurrently. Unrolling replicates the body of loops to reduce loop branching overhead and provide better opportunities for local optimization, vectorization and scheduling of instructions. Performance for loops on systems with multiple processors may also improve using the parallelization features of the PGI compilers.

Interprocedural Analysis (IPA) and Optimization

Interprocedural analysis (IPA) allows use of information across function call boundaries to perform optimizations that would otherwise be unavailable. For example, if the actual argument to a function is in fact a constant in the caller, it may be possible to propagate that constant into the callee and perform optimizations that are not valid if the dummy argument is treated as a variable. A wide range of optimizations are enabled or improved by using IPA, including but not limited to data alignment optimizations, argument removal, constant propagation, pointer disambiguation, pure function detection, F90/F95 array shape propagation, data placement, vestigial function removal, automatic function inlining, inlining of functions from pre-compiled libraries, and interprocedural optimization of functions from pre-compiled libraries.

Function Inlining

This optimization allows a call to a function to be replaced by a copy of the body of that function. This optimization will sometimes speed up execution by eliminating the function call and return overhead. Function inlining may also create opportunities for other types of optimization. Function inlining is not always beneficial. When used improperly it may increase code size and generate less efficient code.

Profile-Feedback Optimization (PFO)

Profile-feedback optimization (PFO) makes use of information from a trace file produced by specially instrumented executables which capture and save information on branch frequency, function and subroutine call frequency, semi-invariant values, loop index ranges, and other input data dependent information that can only be collected dynamically during execution of a program. By definition, use of profile-feedback optimization is a two-phase process: compilation and execution of a specially-instrumented executable, followed by a subsequent compilation which reads a trace file generated during the first phase and uses the information in that trace file to guide compiler optimizations.

Getting Started with Optimizations

Your first concern should be getting your program to execute and produce correct results. To get your program running, start by compiling and linking without optimization. Use the optimization level `-O0` or select `-g` to perform minimal optimization. At this level, you will be able to debug your program easily and isolate any coding errors exposed during porting to x86 or x64 platforms.

If you want to get started quickly with optimization, a good set of options to use with any of the PGI compilers is `-fast -Mipa=fast`. For example:

```
$ pgfortran -fast -Mipa=fast prog.f
```

In PVF, similar options may be accessed using the Optimization property in the Fortran | Optimization property page. For more information on these property pages, refer to "Optimization," on page 334.

For all of the PGI Fortran compilers, the `-fast -Mipa=fast` options generally produce code that is well-optimized without the possibility of significant slowdowns due to pathological cases.

- The `-fast` option is an aggregate option that includes a number of individual PGI compiler options; which PGI compiler options are included depends on the target for which compilation is performed.
- The `-Mipa=fast` option invokes interprocedural analysis including several IPA suboptions.

By experimenting with individual compiler options on a file-by-file basis, further significant performance gains can sometimes be realized. However, depending on the coding style, individual optimizations can sometimes cause slowdowns, and must be used carefully to ensure performance improvements. In addition to `-fast`, the optimization flags most likely to further improve performance are `-O3`, `-Mpfi`, `-Mpfo`, `-Minline`; and on targets with multiple processors, you can use `-Mconcur`.

In PVF, you may access the `-O3`, `-Minline`, and `-Mconcur` options by using the Global Optimizations, Inlining, and Auto-Parallelization properties on the Fortran | Optimization property page, respectively. For more information on these property pages, refer to "Optimization," on page 334.

Three other extremely useful options are `-help`, `-Minfo`, and `-dryrun`.

–help

As described in "Help with Command-line Options," on page 52, you can see a specification of any command-line option by invoking any of the PGI compilers with `-help` in combination with the option in question, without specifying any input files.

For example, you might want information on `-O`:

```
$ pgfortran -help -O
```

The resulting output is similar to this:

```
Reading rcfile /usr/pgi/linux86-64/7.0/bin/.pgfortranrc
-O[<n>] Set optimization level, -O0 to -O4, default -O2
```

Or you can see the full functionality of `-help` itself, which can return information on either an individual option or groups of options:

```
$ pgfortran -help -help
```

The resulting output is similar to this:

```
Reading rcfile /usr/pgi_rel/linux86-64/7.0/bin/.pgfortranrc
-help[=groups|asm|debug|language|linker|opt|other|overall|
  phase|prepro|suffix|switch|target|variable]
```

In PVF these options may be accessed via the Fortran | Command Line property page, or perhaps more appropriately for the `–help` option via a Build Event or Custom Build Step. For more information on these property pages, refer to "Command Line," on page 352

–Minfo

You can use the `-Minfo` option to display compile-time optimization listings. When this option is used, the PGI compilers issue informational messages to stderr as compilation proceeds. From these messages, you can determine which loops are optimized using unrolling, SSE instructions, vectorization, parallelization, interprocedural optimizations and various miscellaneous optimizations. You can also see where and whether functions are inlined.

For more information on `–Minfo`, refer to "Optimization Controls," on page 223.

–Mneginfo

You can use the `-Mneginfo` option to display informational messages listing why certain optimizations are inhibited.

In PVF, you can use the Warning Level property available in the Fortran | General property page to specify the option `-Mneginfo`.

For more information on `–Mneginfo`, refer to "Optimization Controls," on page 223.

–dryrun

The `–dryrun` option can be useful as a diagnostic tool if you need to see the steps used by the compiler driver to preprocess, compile, assemble and link in the presence of a given set of command line inputs. When you

specify the `-dryrun` option, these steps will be printed to stderr but are not actually performed. For example, you can use this option to inspect the default and user-specified libraries that are searched during the link phase, and the order in which they are searched by the linker.

The remainder of this chapter describes the `-O` options, the loop unroller option `-Munroll`, the vectorizer option `-Mvect`, the auto-parallelization option `-Mconcur`, the interprocedural analysis optimization `-Mipa`, and the profile-feedback instrumentation (`-Mpfi`) and optimization (`-Mpfo`) options. You should be able to get very near optimal compiled performance using some combination of these switches.

Common Compiler Feedback Format (CCFF)

Using the Common Compiler Feedback Format (CCFF), PGI compilers save information about how your program was optimized, or why a particular optimization was not made, in the executable file. To append this information to the object file, use the compiler option`-Minfo=ccff`.

If you choose to use PGPROF to aid with your optimization, PGPROF can extract this information and associate it with source code and other performance data, allowing you to view all of this information simultaneously in one of the available profiler panels.

Local and Global Optimization using -O

Using the PGI compiler commands with the –Olevel option (the capital O is for Optimize), you can specify any of the following optimization levels:

–O0
: Level zero specifies no optimization. A basic block is generated for each language statement.

–O1
: Level one specifies local optimization. Scheduling of basic blocks is performed. Register allocation is performed.

–O2
: Level two specifies global optimization. This level performs all level-one local optimization as well as level-two global optimization. If optimization is specified on the command line without a level, level 2 is the default.

–O3
: 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.

–O4
: Level four performs all level-one, level-two, and level-three optimizations and enables hoisting of guarded invariant floating point expressions.

Note

If you use the `-O` option to specify optimization and do not specify a level, then level-two optimization (`-O2`) is the default.

Level-zero optimization specifies no optimization (`-O0`). At this level, the compiler generates a basic block for each statement. Performance will almost always be slowest using this optimization level. This level is useful for the initial execution of a program. It is also useful for debugging, since there is a direct correlation between the program text and the code generated.

Level-one optimization specifies local optimization (`-O1`). The compiler performs scheduling of basic blocks as well as register allocation. Local optimization is a good choice when the code is very irregular, such as code that contains many short statements containing IF statements and does not contain loops (DO or DO WHILE statements). Although this case rarely occurs, for certain types of code, this optimization level may perform better than level-two (`-O2`).

The PGI compilers perform many different types of local optimizations, including but not limited to:

- Algebraic identity removal
- Constant folding
- Common subexpression elimination
- Local register optimization
- Peephole optimizations
- Redundant load and store elimination
- Strength reductions

Level-two optimization (`-O2` or `-O`) specifies global optimization. The `-fast` option generally will specify global optimization; however, the `-fast` switch varies from release to release, depending on a reasonable selection of switches for any one particular release. The –O or –O2 level performs all level-one local optimizations as well as global optimizations. Control flow analysis is applied and global registers are allocated for all functions and subroutines. Loop regions are given special consideration. This optimization level is a good choice when the program contains loops, the loops are short, and the structure of the code is regular.

The PGI compilers perform many different types of global optimizations, including but not limited to:

- Branch to branch elimination
- Constant propagation
- Copy propagation
- Dead store elimination
- Global register allocation
- Invariant code motion
- Induction variable elimination

You can explicitly select the optimization level on the command line. For example, the following command line specifies level-two optimization which results in global optimization:

```
$ pgfortran -O2 prog.f
```

Specifying `-O` on the command-line without a level designation is equivalent to `-O2`. The default optimization level changes depending on which options you select on the command line. For example, when you select the `-g` debugging option, the default optimization level is set to level-zero (`-O0`). However, if you need to debug optimized code, you can use the `-gopt` option to generate debug information without perturbing optimization. Refer to "Default Optimization Levels," on page 79 for a description of the default levels.

As noted previously, the `-fast` option includes `-O2` on all x86 and x64 targets. If you want to override the default for`-fast` with `-O3` while maintaining all other elements of `-fast`, simply compile as follows:

```
$ pgfortran -fast -O3 prog.f
```

Scalar SSE Code Generation

For all processors prior to Intel Pentium 4 and AMD Opteron/Athlon64, for example Intel Pentium III and AMD AthlonXP/MP processors, scalar floating-point arithmetic as generated by the PGI Workstation compilers is performed using x87 floating-point stack instructions. With the advent of SSE/SSE2 instructions on Intel Pentium 4/Xeon and AMD Opteron/Athlon64, it is possible to perform all scalar floating-point arithmetic using SSE/SSE2 instructions. In most cases, this is beneficial from a performance standpoint.

The default on 32-bit Intel Pentium II/III (options `-tp p6`, `-tp piii`, and so on) or on AMD AthlonXP/MP (`option -tp k7`) is to use x87 instructions for scalar floating-point arithmetic. The default on Intel Pentium 4/Xeon or Intel EM64T running a 32-bit operating system (`-tp p7`), AMD Opteron/Athlon64 running a 32-bit operating system (`-tp k8-32`), or AMD Opteron/Athlon64 or Intel EM64T processors running a 64-bit operating system (using `-tp k8-64` and `-tp p7-64` respectively) is to use SSE/SSE2 instructions for scalar floating-point arithmetic. The only way to override this default on AMD Opteron/Athlon64 or Intel EM64T processors running a 64-bit operating system is to specify an older 32-bit target. For example, you can use `-tp k7` or `-tp piii`.

In PVF, the `-tp` option is accessed using the Fortran | Target Processors and Fortran | Target Accelerators property pages. For more information on these property pages, refer to "Fortran | Target Processors," on page 344 and to "Fortran | Target Accelerators," on page 345.

Note

There can be significant arithmetic differences between calculations performed using x87 instructions versus SSE/SSE2.

By default, all floating-point data is promoted to IEEE 80-bit format when stored on the x87 floating-point stack, and all x87 operations are performed register-to-register in this same format. Values are converted back to IEEE 32-bit or IEEE 64-bit when stored back to memory (for REAL/float and DOUBLE PRECISION/double data respectively). The default precision of the x87 floating-point stack can be reduced to IEEE 32-bit or IEEE 64-bit globally by compiling the main program with the `-pc {32|64}` option to the PGI compilers, which is described in detail in Chapter 6, "*Using Command Line Options*". However, there is no way to ensure that operations performed in mixed precision will match those produced on a traditional load-store RISC/UNIX system which implements IEEE 64-bit and IEEE 32-bit registers and associated floating-point arithmetic instructions.

In contrast, arithmetic results produced on Intel Pentium 4/Xeon, AMD Opteron/Athlon64 or Intel EM64T processors will usually closely match or be identical to those produced on a traditional RISC/UNIX system if all scalar arithmetic is performed using SSE/SSE2 instructions. You should keep this in mind when porting applications to and from systems which support both x87 and full SSE/SSE2 floating-point arithmetic. Many subtle issues can arise which affect your numerical results, sometimes to several digits of accuracy.

Loop Unrolling using –Munroll

This optimization unrolls loops, executing multiple instances of the loop during each iteration. This reduces branch overhead, and can improve execution speed by creating better opportunities for instruction scheduling. A loop with a constant count may be completely unrolled or partially unrolled. A loop with a non-constant count may also be unrolled. A candidate loop must be an innermost loop containing one to four blocks of code.

The following example shows the use of the -Munroll option:

```
$ pgfortran -Munroll prog.f
```

The -Munroll option is included as part of -fast on all x86 and x64 targets. The loop unroller expands the contents of a loop and reduces the number of times a loop is executed. Branching overhead is reduced when a loop is unrolled two or more times, since each iteration of the unrolled loop corresponds to two or more iterations of the original loop; the number of branch instructions executed is proportionately reduced. When a loop is unrolled completely, the loop's branch overhead is eliminated altogether.

In PVF, this option is accessed using the Loop Unroll Count property in the Fortran | Optimization property page. For more information on these property pages, refer to "Fortran | Optimization," on page 334.

Loop unrolling may be beneficial for the instruction scheduler. When a loop is completely unrolled or unrolled two or more times, opportunities for improved scheduling may be presented. The code generator can take advantage of more possibilities for instruction grouping or filling instruction delays found within the loop.

Example 7.1, "Dot Product Code"and Example 7.2, "Unrolled Dot Product Code" show the effect of code unrolling on a segment that computes a dot product.

Example 7.1. Dot Product Code

```
REAL*4 A(100), B(100), Z
INTEGER I
DO I=1, 100
  Z = Z + A(i) * B(i)
END DO
END
```

Example 7.2. Unrolled Dot Product Code

```
REAL*4 A(100), B(100), Z
 INTEGER I
 DO I=1, 100, 2
   Z = Z + A(i) * B(i)
   Z = Z + A(i+1) * B(i+1)
 END DO
END
```

Using the -Minfo option, the compiler informs you when a loop is being unrolled. For example, a message similar to the following, indicating the line number, and the number of times the code is unrolled, displays when a loop is unrolled:

```
dot:
 5, Loop unrolled 5 times
```

Using the c:<m> and n:<m> sub-options to -Munroll, or using -Mnounroll, you can control whether and how loops are unrolled on a file-by-file basis. Using directives or pragmas as specified in Chapter 11, "*Using Directives*", you can precisely control whether and how a given loop is unrolled. For a detailed description of the -Munroll option, refer to Chapter 6, "*Using Command Line Options*".

Vectorization using –Mvect

The -Mvect option is included as part of -fast on all x86 and x64 targets. If your program contains computationally-intensive loops, the -Mvect option may be helpful. If in addition you specify -Minfo, and your code contains loops that can be vectorized, the compiler reports relevant information on the optimizations applied.

When a PGI compiler command is invoked with the -Mvect option, the vectorizer scans code searching for loops that are candidates for high-level transformations such as loop distribution, loop interchange, cache tiling, and idiom recognition (replacement of a recognizable code sequence, such as a reduction loop, with

optimized code sequences or function calls). When the vectorizer finds vectorization opportunities, it internally rearranges or replaces sections of loops (the vectorizer changes the code generated; your source code's loops are not altered). In addition to performing these loop transformations, the vectorizer produces extensive data dependence information for use by other phases of compilation and detects opportunities to use vector or packed Streaming SIMD Extensions (SSE) instructions on processors where these are supported.

The `-Mvect` option can speed up code which contains well-behaved countable loops which operate on large REAL, REAL*4, REAL*8, INTEGER*4, COMPLEX or COMPLEX DOUBLE arrays in Fortran. However, it is possible that some codes will show a decrease in performance when compiled with the `-Mvect` option due to the generation of conditionally executed code segments, inability to determine data alignment, and other code generation factors. For this reason, it is recommended that you check carefully whether particular program units or loops show improved performance when compiled with this option enabled.

In PVF, you access the basic forms of this option using the Vectorization property in the Fortran | Optimization property page. For more advanced use of this option, use the Fortran | Command Line property page. For more information on these property pages, refer to "Fortran Property Pages," on page 331.

Vectorization Sub-options

The vectorizer performs high-level loop transformations on countable loops. A loop is countable if the number of iterations is set only before loop execution and cannot be modified during loop execution. Some of the vectorizer transformations can be controlled by arguments to the `-Mvect` command line option. The following sections describe the arguments that affect the operation of the vectorizer. In addition, some of these vectorizer operations can be controlled from within code using directives and pragmas. For details on the use of directives and pragmas, refer to Chapter 11, "*Using Directives*," on page 123.

The vectorizer performs the following operations:

- Loop interchange
- Loop splitting
- Loop fusion
- Memory-hierarchy (cache tiling) optimizations
- Generation of SSE instructions on processors where these are supported
- Generation of prefetch instructions on processors where these are supported
- Loop iteration peeling to maximize vector alignment
- Alternate code generation

By default, `-Mvect` without any sub-options is equivalent to:

```
-Mvect=assoc,cachesize=c
```

where c is the actual cache size of the machine.

This enables the options for nested loop transformation and various other vectorizer options. These defaults may vary depending on the target system.

Assoc Option

The option `–Mvect=assoc` instructs the vectorizer to perform associativity conversions that can change the results of a computation due to a round-off error (`–Mvect=noassoc` disables this option). For example, a typical optimization is to change one arithmetic operation to another arithmetic operation that is mathematically correct, but can be computationally different and generate faster code. This option is provided to enable or disable this transformation, since a round-off error for such associativity conversions may produce unacceptable results.

Cachesize Option

The option `–Mvect=cachesize:n` instructs the vectorizer to tile nested loop operations assuming a data cache size of n bytes. By default, the vectorizer attempts to tile nested loop operations, such as matrix multiply, using multi-dimensional strip-mining techniques to maximize re-use of items in the data cache.

SSE Option

The option `–Mvect=sse` instructs the vectorizer to automatically generate packed SSE (Streaming SIMD Extensions), SSE2, and prefetch instructions when vectorizable loops are encountered. SSE instructions, first introduced on Pentium III and AthlonXP processors, operate on single-precision floating-point data, and hence apply only to vectorizable loops that operate on single-precision floating-point data. SSE2 instructions, first introduced on Pentium 4, Xeon and Opteron processors, operate on double-precision floating-point data. Prefetch instructions, first introduced on Pentium III and AthlonXP processors, can be used to improve the performance of vectorizable loops that operate on either 32-bit or 64-bit floating-point data. Refer to the PGI Release Notes for a concise list of processors that support SSE, SSE2 and prefetch instructions.

Note

Program units compiled with `–Mvect=sse` will not execute on Pentium, Pentium Pro, Pentium II or first generation AMD Athlon processors. They will only execute correctly on Pentium III, Pentium 4, Xeon, EM64T, AthlonXP, Athlon64 and Opteron systems running an SSE-enabled operating system.

Prefetch Option

The option `–Mvect=prefetch` instructs the vectorizer to automatically generate prefetch instructions when vectorizable loops are encountered, even in cases where SSE or SSE2 instructions are not generated. Usually, explicit prefetching is not necessary on Pentium 4, Xeon and Opteron because these processors support hardware prefetching; nonetheless, it sometimes can be worthwhile to experiment with explicit prefetching. Prefetching can be controlled on a loop-by-loop level using prefetch directives, which are described in detail in “Prefetch Directives ,” on page 126.

Note

Program units compiled with `–Mvect=prefetch` will not execute correctly on Pentium, Pentium Pro, or Pentium II processors. They will execute correctly only on Pentium III, Pentium 4, Xeon, EM64T, AthlonXP, Athlon64 or Opteron systems. In addition, the `prefetch` instruction is only supported on AthlonXP, Athlon64 or Opteron systems and can cause instruction faults on non-AMD processors. For this reason, the PGI compilers do not generate `prefetch` instructions by default on any target.

In addition to these sub-options to `-Mvect`, several other sub-options are supported. Refer to the description of `-M[no]vect` in Chapter 18, "*Command-Line Options Reference*" for a detailed description of all available sub-options.

Vectorization Example Using SSE/SSE2 Instructions

One of the most important vectorization options is `-Mvect=sse`. When you use this option, the compiler automatically generates SSE and SSE2 instructions, where possible, when targeting processors on which these instructions are supported. This process can improve performance by up to a factor of two compared with the equivalent scalar code. All of the PGI Fortran, C and C++ compilers support this capability. The PGI Release Notes show which x86 and x64 processors support these instructions.

Prior to release 7.0, `-Mvect=sse` was omitted from the compiler switch `-fast` but was included in the switch `-fastsse`. Since release 7.0 , `-fast` is synonymous with `-fastsse`; therefore, both options include `-Mvect=sse`.

In the program in Example 7.3, "Vector operation using SSE instructions", the vectorizer recognizes the vector operation in subroutine 'loop' when either the compiler switch `-Mvect=sse` or `-fast` is used. This example shows the compilation, informational messages, and run-time results using the SSE instructions on an AMD Opteron processor-based system, along with issues that affect SSE performance.

First note that the arrays in Example 7.3 are single-precision and that the vector operation is done using a unit stride loop. Thus, this loop can potentially be vectorized using SSE instructions on any processor that supports SSE or SSE2 instructions. SSE operations can be used to operate on pairs of single-precision floating-point numbers, and do not apply to double-precision floating-point numbers. SSE2 instructions can be used to operate on quads of single-precision floating-point numbers or on pairs of double-precision floating-point numbers.

Loops vectorized using SSE or SSE2 instructions operate much more efficiently when processing vectors that are aligned to a cache-line boundary. You can cause unconstrained data objects of size 16 bytes or greater to be cache-aligned by compiling with the `-Mcache_align` switch. An unconstrained data object is a data object that is not a common block member and not a member of an aggregate data structure.

Note

For stack-based local variables to be properly aligned, the main program or function must be compiled with `-Mcache_align`.

The `-Mcache_align` switch has no effect on the alignment of Fortran allocatable or automatic arrays. If you have arrays that are constrained, such as vectors that are members of Fortran common blocks, you must specifically pad your data structures to ensure proper cache alignment. You can use `-Mcache_align` for only the beginning address of each common block to be cache-aligned.

The following examples show the results of compiling the sample code in Example 7.3 both with and without the option `-Mvect=sse`.

Example 7.3. Vector operation using SSE instructions

```
program vector_op
 parameter (N = 9999)
 real*4 x(N), y(N), z(N), W(N)
```

```
 do i = 1, n
   y(i) = i
   z(i) = 2*i
   w(i) = 4*i
 enddo
 do j = 1, 200000
   call loop(x,y,z,w,1.0e0,N)
 enddo
 print *, x(1),x(771),x(3618),x(6498),x(9999)
end
```

```
subroutine loop(a,b,c,d,s,n)
 integer i, n
 real*4 a(n), b(n), c(n), d(n),s
 do i = 1, n
   a(i) = b(i) + c(i) - s * d(i)
 enddo
end
```

Assume the preceding program is compiled as follows, where `-Mvect=nosse` disables SSE vectorization:

```
 % pgfortran -fast -Mvect=nosse -Minfo vadd.f
 vector_op:
 4, Loop unrolled 4 times
 loop:
 18, Loop unrolled 4 times
```

The following output shows a sample result if the generated executable is run and timed on a standalone AMD Opteron 2.2 Ghz system:

```
 % /bin/time vadd
 -1.000000 -771.000 -3618.000 -6498.00 -9999.00
 5.39user 0.00system 0:05.40elapsed 99%CP
```

Now, recompile with SSE vectorization enabled, and you see results similar to these:

```
 % pgfortran -fast -Minfo vadd.f -o vadd
 vector_op:
 4, Unrolled inner loop 8 times
 Loop unrolled 7 times (completely unrolled)
 loop:
 18, Generated 4 alternate loops for the inner loop
 Generated vector sse code for inner loop
 Generated 3 prefetch instructions for this loop
```

Notice the informational message for the loop at line 18.

- The first two lines of the message indicate that the loop was vectorized, SSE instructions were generated, and four alternate versions of the loop were also generated. The loop count and alignments of the arrays determine which of these versions is executed.
- The last line of the informational message indicates that prefetch instructions have been generated for three loads to minimize latency of data transfers from main memory.

Executing again, you should see results similar to the following:

```
 % /bin/time vadd
 -1.000000 -771.000 -3618.00 -6498.00
 -9999.0
 3.59user 0.00system 0:03.59elapsed 100%CPU
```

The result is a 50% speed-up over the equivalent scalar, that is, the non-SSE, version of the program.

Speed-up realized by a given loop or program can vary widely based on a number of factors:

- When the vectors of data are resident in the data cache, performance improvement using vector SSE or SSE2 instructions is most effective.
- If data is aligned properly, performance will be better in general than when using vector SSE operations on unaligned data.
- If the compiler can guarantee that data is aligned properly, even more efficient sequences of SSE instructions can be generated.
- The efficiency of loops that operate on single-precision data can be higher. SSE2 vector instructions can operate on four single-precision elements concurrently, but only two double-precision elements.

Note

Compiling with `-Mvect=sse` can result in numerical differences from the executables generated with less optimization. Certain vectorizable operations, for example dot products, are sensitive to order of operations and the associative transformations necessary to enable vectorization (or parallelization).

Auto-Parallelization using -Mconcur

With the `-Mconcur` option the compiler scans code searching for loops that are candidates for auto-parallelization. `-Mconcur` must be used at both compile-time and link-time. When the parallelizer finds opportunities for auto-parallelization, it parallelizes loops and you are informed of the line or loop being parallelized if the `-Minfo` option is present on the compile line. See "Optimization Controls," on page 223, for a complete specification of `-Mconcur`.

In PVF, the basic form of this option is accessed using the Auto-Parallelization property of the Fortran | Optimization property page. For more advanced auto-parallelization, use the Fortran | Command Line property page. For more information on these property pages, refer to "Fortran Property Pages," on page 331

A loop is considered parallelizable if doesn't contain any cross-iteration data dependencies. Cross-iteration dependencies from reductions and expandable scalars are excluded from consideration, enabling more loops to be parallelizable. In general, loops with calls are not parallelized due to unknown side effects. Also, loops with low trip counts are not parallelized since the overhead in setting up and starting a parallel loop will likely outweigh the potential benefits. In addition, the default is to not parallelize innermost loops, since these often by definition are vectorizable using SSE instructions and it is seldom profitable to both vectorize and parallelize the same loop, especially on multi-core processors. Compiler switches and directives are available to let you override most of these restrictions on auto-parallelization.

Auto-parallelization Sub-options

The parallelizer performs various operations that can be controlled by arguments to the `-Mconcur` command line option. The following sections describe these arguments that affect the operation of the vectorizer. In addition, these vectorizer operations can be controlled from within code using directives and pragmas.

For details on the use of directives and pragmas, refer to Chapter 11, "*Using Directives*".

By default, `-Mconcur` without any sub-options is equivalent to:

```
-Mconcur=dist:block
```

This enables parallelization of loops with blocked iteration allocation across the available threads of execution. These defaults may vary depending on the target system.

Altcode Option

The option `–Mconcur=altcode` 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, no alternate serial code is generated.

Dist Option

The option `–Mconcur=dist:{block|cyclic}` option specifies whether to assign loop iterations to the available threads in blocks or in a cyclic (round-robin) fashion. Block distribution is the default. If cyclic is specified, iterations are allocated to processors cyclically. That is, 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.

Cncall Option

The option `–Mconcur=cncall` specifies that it is safe to parallelize loops that contain subroutine or function calls. By default, such loops are excluded from consideration for auto-parallelization. Also, no minimum loop count threshold must be satisfied before parallelization will occur, and last values of scalars are assumed to be safe.

The environment variable `NCPUS` is checked at run-time for a parallel program. If `NCPUS` is set to 1, a parallel program runs serially, but will use the parallel routines generated during compilation. If `NCPUS` is set to a value greater than 1, the specified number of processors will be used to execute the program. Setting `NCPUS` to a value exceeding the number of physical processors can produce inefficient execution. Executing a program on multiple processors in an environment where some of the processors are being time-shared with another executing job can also result in inefficient execution.

As with the vectorizer, the `-Mconcur` option can speed up code if it contains well-behaved countable loops and/or computationally intensive nested loops that operate on arrays. However, it is possible that some codes will show a decrease in performance on multi-processor systems when compiled with `-Mconcur` due to parallelization overheads, memory bandwidth limitations in the target system, false-sharing of cache lines, or other architectural or code-generation factors. For this reason, it is recommended that you check carefully whether particular program units or loops show improved performance when compiled using this option.

If the compiler is not able to successfully auto-parallelize your application, you should refer to Chapter 9, "*Using OpenMP*". It is possible that insertion of explicit parallelization directives or pragmas, and use of the compiler option `–mp` might enable the application to run in parallel.

Loops That Fail to Parallelize

In spite of the sophisticated analysis and transformations performed by the compiler, programmers may notice loops that are seemingly parallel, but are not parallelized. In this subsection, we look at some examples of common situations where parallelization does not occur.

Innermost Loops

As noted earlier in this chapter, the PGI compilers will not parallelize innermost loops by default, because it is usually not profitable. However, you can override this default using the `-Mconcur=innermost` command-line option.

Timing Loops

Often, loops occur in programs that are similar to timing loops. The outer loop in the following example is one such loop.

```
do j = 1, 2
 do i = 1, n
 a(i) = b(i) + c(i)
1 enddo
enddo
```

The outer loop in the preceding example is not parallelized because the compiler detects a cross-iteration dependence in the assignment to `a(i)`. Suppose the outer loop were parallelized. Then both processors would simultaneously attempt to make assignments into `a(1:n)`. Now in general the values computed by each processor for `a(1:n)` will differ, so that simultaneous assignment into `a(1:n)` will produce values different from sequential execution of the loops.

In this example, values computed for `a(1:n)` don't depend on `j`, so that simultaneous assignment by both processors will not yield incorrect results. However, it is beyond the scope of the compilers' dependence analysis to determine that values computed in one iteration of a loop don't differ from values computed in another iteration. So the worst case is assumed, and different iterations of the outer loop are assumed to compute different values for `a(1:n)`. Is this assumption too pessimistic? If `j` doesn't occur anywhere within a loop, the loop exists only to cause some delay, most probably to improve timing resolution. It is not usually valid to parallelize timing loops; to do so would distort the timing information for the inner loops.

Scalars

Quite often, scalars will inhibit parallelization of non-innermost loops. There are two separate cases that present problems. In the first case, scalars appear to be expandable, but appear in non-innermost loops, as in the following example.

```
do j = 1, n
 x = b(j)
 do i = 1, n
 a(i,j) = x + c(i,j)
 enddo
enddo
```

There are a number of technical problems to be resolved in order to recognize expandable scalars in non-innermost loops. Until this generalization occurs, scalars like `x` in the preceding code segment inhibit parallelization of loops in which they are assigned. In the following example, scalar `k` is not expandable, and it is not an accumulator for a reduction.

```
    k = 1
     do i = 1, n
       do j = 1, n
1       a(j,i) = b(k) * x
       enddo
```

```
       k = i
2      if (i .gt. n/2) k = n - (i - n/2)
     enddo
```

If the outer loop is parallelized, conflicting values are stored into k by the various processors. The variable k cannot be made local to each processor because its value must remain coherent among the processors. It is possible the loop could be parallelized if all assignments to k are placed in critical sections. However, it is not clear where critical sections should be introduced because in general the value for k could depend on another scalar (or on k itself), and code to obtain the value of other scalars must reside in the same critical section.

In the previous example, the assignment to k within a conditional at label 2 prevents k from being recognized as an induction variable. If the conditional statement at label 2 is removed, k would be an induction variable whose value varies linearly with j, and the loop could be parallelized.

Scalar Last Values

During parallelization, scalars within loops often need to be privatized; that is, each execution thread has its own independent copy of the scalar. Problems can arise if a privatized scalar is accessed outside the loop. For example, consider the following loop:

```
    do I = 1,N
      if (x(I) > 5.0 ) then
        t = I
      endif
    enddo
    v = t
```

The value of t may not be computed on the last iteration of the loop. Normally, if a scalar is assigned within a loop and used following the loop, the PGI compilers save the last value of the scalar. However, if the loop is parallelized and the scalar is not assigned on every iteration, it may be difficult, without resorting to costly critical sections, to determine on what iteration t is last assigned. Analysis allows the compiler to determine that a scalar is assigned on each iteration and hence that the loop is safe to parallelize if the scalar is used later, as illustrated in the following example.

```
do I = 1,N
 if (x(I) > 0.0 ) then
 t = 2.0
 else
 t = 3.0
 y(i) = t
 endif
enddo
v = t
```

where t is assigned on every iteration of the loop. However, there are cases where a scalar may be privatizable, but if it is used after the loop, it is unsafe to parallelize. Examine the following loop in which each use of t within the loop is reached by a definition from the same iteration.

```
do I = 1,N
 if (x(I) > 0.0 ) then
 t = x(I)
 ...
 y(i) = ... t
 endif
enddo
v = t
```

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 last assigned. The compiler detects the previous cases. When a scalar is used after the loop but is not defined on every iteration of the loop, parallelization does not occur.

When the programmer knows that the scalar is assigned on the last iteration of the loop, the programmer may use a directive to let the compiler know the loop is safe to parallelize. The Fortran directive `safe_lastval` informs the compiler that, for a given loop, all scalars are assigned in the last iteration of the loop; thus, it is safe to parallelize the loop. We could add the following line to any of our previous examples.

```
cpgi$l safe_lastval
```

The resulting code looks similar to this:

```
cpgi$l safe_lastv
...
do I = 1,N
 if (x(I) > 5.0 ) then
 t = I
 endif
 enddo
 v = t
```

In addition, a command-line option `-M`safe_lastval, provides this information for all loops within the routines being compiled, which essentially provides global scope.

Processor-Specific Optimization & the 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 things such as instruction selection, instruction scheduling, and vectorization. By default, the PGI compilers produce code specifically targeted to the type of processor on which the compilation is performed. That is, the default is to use 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.

All PGI compilers have the capability of generating *unified binaries*, which provide a low-overhead means for generating a single executable that is compatible with and has good performance on more than one hardware platform.

You can use the `-tp` option to control compilation behavior by specifying the processor or processors with which the generated code is compatible. The compilers generate and combine into one executable multiple binary code streams, each optimized for a specific platform. At run-time, the one executable senses the environment and dynamically selects the appropriate code stream. For specific information on the `-tp` option, refer to –tp <target> [,target...].

Executable size is automatically controlled via unified binary culling. Only those functions and subroutines where the target affects the generated code have unique binary images, resulting in a code-size savings of from 10% to 90% compared to generating full copies of code for each target.

Programs can use the PGI Unified Binary even if all of the object files and libraries are not compiled as unified binaries. Like any other object file, you can use PGI Unified Binary object files to create programs or libraries. No special start up code is needed; support is linked in from the PGI libraries.

The `-Mipa` option disables generation of PGI Unified Binary. Instead, the default target auto-detect rules for the host are used to select the target processor.

Interprocedural Analysis and Optimization using –Mipa

The PGI Fortran compilers use interprocedural analysis (IPA) that results in minimal changes to makefiles and the standard edit-build-run application development cycle. Other than adding `-Mipa` to the command line or selecting the appropriate value for the PVF Optimization property from the property page Fortran | Optimization, no other changes are required. For reference and background, the process of building a program without IPA is described later in this section, followed by the minor modifications required to use IPA with the PGI compilers.

Note

PVF's internal build engine uses the method described in "Building a Program with IPA - Several Steps," on page 76.

Building a Program Without IPA – Single Step

Using the pgfortran command-level compiler driver, multiple source files can be compiled and linked into a single executable with one command. The following example compiles and links three source files:

```
% pgfortran -o file1.exe file1.f95 file2.f95 file3.f95
```

In actuality, the pgfortran driver executes several steps to produce the assembly code and object files corresponding to each source file, and subsequently to link the object files together into a single executable file. This command is roughly equivalent to the following commands performed individually:

```
% pgfortran -S -o file1.s file1.f95
% as -o file1.obj file1.s
% pgfortran -S -o file2.s file2.f95
% as -o file2.obj file2.s
% pgfortran -S -o file3.s file3.f95
% as -o file3.obj file3.s
% pgfortran -o file1.exe file1.obj file2.obj file3.obj
```

If any of the three source files is edited, the executable can be rebuilt with the same command line:

```
% pgfortran -o file1.exe file1.f95 file2.f95 file3.f95
```

Note

This always works as intended, but has the side-effect of recompiling all of the source files, even if only one has changed. For applications with a large number of source files, this can be time-consuming and inefficient.

Building a Program Without IPA - Several Steps

It is also possible to use individual pgfortran commands to compile each source file into a corresponding object file, and one to link the resulting object files into an executable:

```
% pgfortran -c file1.f95
% pgfortran -c file2.f95
% pgfortran -c file3.f95
```

```
% pgfortran -o file1.exe file1.obj file2.obj file3.obj
```

The pgfortran driver invokes the compiler and assembler as required to process each source file, and invokes the linker for the final link command. If you modify one of the source files, the executable can be rebuilt by compiling just that file and then relinking:

```
% pgfortran -c file1.f95
% pgfortran -o file1.exe file1.obj file2.obj file3.obj
```

Building a Program Without IPA Using Make

The program compilation and linking process can be simplified greatly using the `make` utility on systems where it is supported. Suppose you create a `makefile` containing the following lines:

```
file1.exe: file1.obj file2.obj file3.obj
 pgfortran $(OPT) -o file1.exe file1.obj file2.obj
file3.obj file1.obj: file1.c
 pgfortran $(OPT) -c file1.f95
file2.obj: file2.c
 pgfortran $(OPT) -c file2.f95
file3.obj: file3.c
 pgfortran $(OPT) -c file3.f95
```

It is then possible to type a single make command:

```
% make
```

The `make` utility determines which object files are out of date with respect to their corresponding source files, and invokes the compiler to recompile only those source files and to relink the executable. If you subsequently edit one or more source files, the executable can be rebuilt with the minimum number of recompilations using the same single `make` command.

Building a Program with IPA

Interprocedural analysis and optimization (IPA) by the PGI compilers alters the standard and `make` utility command-level interfaces as little as possible. IPA occurs in three phases:

- **Collection**: Create a summary of each function or procedure, collecting the useful information for interprocedural optimizations. This is done during the compile step if the `-Mipa` switch is present on the command line; summary information is collected and stored in the object file.
- **Propagation**: Process all the object files to propagate the interprocedural summary information across function and file boundaries. This is done during the link step, when all the object files are combined, if the `-Mipa` switch is present on the link command line.
- **Recompile/Optimization**: Recompile each of the object files with the propagated interprocedural information, producing a specialized object file. This process is also performed during the link step when the `-Mipa` switch is present on the link command line.

When linking with `-Mipa`, the PGI compilers automatically regenerate IPA-optimized versions of each object file, essentially recompiling each file. If there are IPA-optimized objects from a previous build, the compilers will minimize the recompile time by reusing those objects if they are still valid. They will still be valid if the IPA-optimized object is newer than the original object file, and the propagated IPA information for that file has not changed since it was optimized.

After each object file has been recompiled, the regular linker is invoked to build the application with the IPA-optimized object files. The IPA-optimized object files are saved in the same directory as the original object files, for use in subsequent program builds.

Building a Program with IPA - Single Step

By adding the `–Mipa` command line switch, several source files can be compiled and linked with interprocedural optimizations with one command:

```
% pgfortran -Mipa=fast -o file1.exe file1.f95 file2.f95 file3.f95
```

Just like compiling without `–Mipa`, the driver executes several steps to produce the assembly and object files to create the executable:

```
% pgfortran -Mipa=fast -S -o file1.s file1.f95
% as -o file1.obj file1.s
% pgfortran -Mipa=fast -S -o file2.s file2.f95
% as -o file2.obj file2.s
% pgfortran -Mipa=fast -S -o file3.s file3.f95
% as -o file3.obj file3.s
% pgfortran -Mipa=fast -o file1.exe file1.obj file2.obj file3.obj
```

In the last step, an IPA linker is invoked to read all the IPA summary information and perform the interprocedural propagation. The IPA linker reinvokes the compiler on each of the object files to recompile them with interprocedural information. This creates three new objects with mangled names:

```
file1_ipa5_file1.exe.obj, file2_ipa5_file1.exe.obj, file2_ipa5_file1.exe.obj
```

The system linker is then invoked to link these IPA-optimized objects into the final executable. Later, if one of the three source files is edited, the executable can be rebuilt with the same command line:

```
% pgfortran -Mipa=fast -o file1.exe file1.f95 file2.f95 file3.f95
```

This will work, but again has the side-effect of compiling each source file, and recompiling each object file at link time.

Building a Program with IPA - Several Steps

Just by adding the `–Mipa` command-line switch, it is possible to use individual pgfortran commands to compile each source file, followed by a command to link the resulting object files into an executable:

```
% pgfortran -Mipa=fast -c file1.f95
% pgfortran -Mipa=fast -c file2.f95
% pgfortran -Mipa=fast -c file3.f95
% pgfortran -Mipa=fast -o file1.exe file1.obj file2.obj file3.obj
```

The pgfortran driver invokes the compiler and assembler as required to process each source file, and invokes the IPA linker for the final link command. If you modify one of the source files, the executable can be rebuilt by compiling just that file and then relinking:

```
% pgfortran -Mipa=fast -c file1.f95
% pgfortran -Mipa=fast -o file1.exe file1.obj file2.obj file3.obj
```

When the IPA linker is invoked, it determines that the IPA-optimized object for `file1.obj` (`file1_ipa5_a.out.obj`) is stale, since it is older than the object `file1.obj`, and hence will need to be rebuilt, and will reinvoke the compiler to generate it. In addition, depending on the nature of the

changes to the source file `file1.f95`, the interprocedural optimizations previously performed for `file2` and `file3` may now be inaccurate. For instance, IPA may have propagated a constant argument value in a call from a function in `file1.f95` to a function in `file2.f95`; if the value of the argument has changed, any optimizations based on that constant value are invalid. The IPA linker determines which of any previously created IPA-optimized objects need to be regenerated, and reinvokes the compiler as appropriate to regenerate them. Only those objects that are stale or which have new or different IPA information will be regenerated, which saves on compile time.

Building a Program with IPA Using Make

As in the previous two sections, programs can be built with IPA using the make utility. Just add the command-line switch `-Mipa`, as shown here:

```
OPT=-Mipa=fast
file1.exe
 pgfortran $(OPT) -o file1 file1.obj file2.obj file3.obj
file1.obj: file1.f95
 pgfortran $(OPT) -c file1.f95
file2.obj: file2.f95
 pgfortran $(OPT) -c file2.f95
file3.obj: file3.f95
 pgfortran $(OPT) -c file3.f95
```

Using the single `make` command invokes the compiler to generate any object files that are out-of-date, then invokes pgfortran to link the objects into the executable; at link time, pgfortran calls the IPA linker to regenerate any stale or invalid IPA-optimized objects.

```
% make
```

Questions about IPA

1. Why is the object file so large?

An object file created with `-Mipa` contains several additional sections. One is the summary information used to drive the interprocedural analysis. In addition, the object file contains the compiler internal representation of the source file, so the file can be recompiled at link time with interprocedural optimizations. There may be additional information when inlining is enabled. The total size of the object file may be 5-10 times its original size. The extra sections are not added to the final executable.

2. What if I compile with `-Mipa` and link without `-Mipa`?

The PGI compilers generate a legal object file, even when the source file is compiled with `-Mipa`. If you compile with `-Mipa` and link without `-Mipa`, the linker is invoked on the original object files. A legal executable will be generated; while this will not have the benefit of interprocedural optimizations, any other optimizations will apply.

3. What if I compile without `-Mipa` and link with `-Mipa`?

At link time, the IPA linker must have summary information about all the functions or routines used in the program. This information is created only when a file is compiled with `-Mipa`. If you compile a file without `-Mipa` and then try to get interprocedural optimizations by linking with `-Mipa`, the IPA linker will issue a message that some routines have no IPA summary information, and will proceed to run the system linker using the original object files. If some files were compiled with `-Mipa` and others

were not, it will determine the safest approximation of the IPA summary information for those files not compiled with `-Mipa`, and use that to recompile the other files using interprocedural optimizations.

4. Can I build multiple applications in the same directory with `-Mipa`?

   Yes. Suppose you have three source files: `main1.f95`, `main2.f95`, and `sub.f95` where `sub.f95` is shared between the two applications. Suppose you build the first application with `-Mipa`, using this command:

   ```
   % pgfortran -Mipa=fast -o app1 main1.f95 sub.f95
   ```

   Then the IPA linker creates two IPA-optimized object files and uses them to build the first application.

   ```
   main1_ipa4_app1.exe.oobj sub_ipa4_app1.exe.oobj
   ```

   Now suppose you build the second application using this command:

   ```
   % pgfortran -Mipa=fast -o app2 main2.f95 sub.f95
   ```

   The IPA linker creates two more IPA-optimized object files:

   ```
   main2_ipa4_app2.exe.oobj sub_ipa4_app2.exe.oobj
   ```

   Note

   There are now three object files for `sub.f95`: the original `sub.obj`, and two IPA-optimized objects, one for each application in which it appears.

5. How is the mangled name for the IPA-optimized object files generated?

   The mangled name has '_ipa' appended, followed by the decimal number of the length of the executable file name, followed by an underscore and the executable file name itself. The suffix is changed to .oobj so linking *.obj does not pull in the IPA-optimized objects. If the IPA linker determines that the file would not benefit from any interprocedural optimizations, it does not have to recompile the file at link time and uses the original object.

Profile-Feedback Optimization using –Mpfi/–Mpfo

The PGI compilers support many common profile-feedback optimizations, including semi-invariant value optimizations and block placement. These are performed under control of the `-Mpfi`/`-Mpfo` command-line options.

When invoked with the `-Mpfi` option, the PGI compilers instrument the generated executable for collection of profile and data feedback information. This information can be used in subsequent compilations that include the `-Mpfo` optimization option. `-Mpfi` must be used at both compile-time and link-time. Programs compiled with `-Mpfi` include extra code to collect run-time statistics and write them out to a trace file. When the resulting program is executed, a profile feedback trace file `pgfi.out` is generated in the current working directory.

Note

Programs compiled and linked with `-Mpfi` execute more slowly due to the instrumentation and data collection overhead. You should use executables compiled with `-Mpfi` only for execution of training runs.

When invoked with the `-Mpfo` option, the PGI compilers use data from a `pgfi.out` profile feedback tracefile to enable or enhance certain performance optimizations. Use of this option requires the presence of a `pgfi.out` trace file in the current working directory.

Default Optimization Levels

The following table shows the interaction between the –O<level> , –g, and `-M<opt>` options. In the table, level can be 0, 1, 2, 3 or 4, and <opt> can be `vect`, `concur`, `unroll` or `ipa`. The default optimization level is dependent upon these command-line options.

Table 7.1. Optimization and –O, –g and –M<opt> Options

Optimize Option	Debug Option	–M<opt> Option	Optimization Level
none	none	none	1
none	none	–M<opt>	2
none	–g	none	0
–O	none or –g	none	2
–Olevel	none or –g	none	level
–Olevel <= 2	none or –g	–M<opt>	2

Code that is not optimized yet compiled using the option `-O0` can be significantly slower than code generated at other optimization levels. The `-M<opt>` option, where <opt> is `vect`, `concur`, `unroll` or `ipa`, sets the optimization level to 2 if no `-O` options are supplied. Both the `-fast` and the `-fastsse` options set the optimization level to a target-dependent optimization level if no `-O` options are supplied.

Local Optimization Using Directives

Command-line options let you specify optimizations for an entire source file. Directives supplied within a Fortran source file provide information to the compiler and alter the effects of certain command-line options or the default behavior of the compiler. (Many directives have a corresponding command-line option.)

While a command line option affects the entire source file that is being compiled, directives let you do the following:

- Apply, or disable, the effects of a particular command-line option to selected subprograms or to selected loops in the source file (for example, an optimization).
- Globally override command-line options.
- Tune selected routines or loops based on your knowledge or on information obtained through profiling.

Chapter 11, "*Using Directives*" provides details on how to add directives and pragmas to your source files.

Execution Timing and Instruction Counting

As this chapter describes, once you have a program that compiles, executes and gives correct results, you may optimize your code for execution efficiency.

Selecting the correct optimization level requires some thought and may require that you compare several optimization levels before arriving at the best solution. To compare optimization levels, you need to measure the execution time for your program. There are several approaches you can take for timing execution.

- You can use shell commands that provide execution time statistics.
- You can include function calls in your code that provide timing information.
- You can profile sections of code.

Timing functions available with the PGI compilers include these:

- 3F timing routines
- The SECNDS pre-declared function in PGF77, PGF95, or PGFORTRAN
- The SYSTEM_CLOCK or CPU_CLOCK intrinsics in PGF95 .

In general, when timing a program, you should try to eliminate or reduce the amount of system level activities such as I/O, program loading, and task switching.

The following example shows a fragment that indicates how to use SYSTEM_CLOCK effectively within an F90, F95 program unit.

Example 7.4. Using SYSTEM_CLOCK code fragment

```
 . . .
 integer :: nprocs, hz, clock0, clock1
 real :: time
 integer, allocatable :: t(:)
#if defined (F95)
 allocate (t(number_of_processors()))
#elif defined (_OPENMP)
 allocate (t(OMP_GET_NUM_THREADS()))
#else
 allocate (t(1))
#endif
 call system_clock (count_rate=hz)
!
 call system_clock(count=clock0)
 < do work>
 call system_clock(count=clock1)
!
 t = (clock1 - clock0)
 time = real (sum(t)) / (real(hz) * size(t))
 . . .
```

Chapter 8. Using Function Inlining

Function inlining replaces a call to a function or a subroutine with the body of the function or subroutine. This can speed up execution by eliminating parameter passing and function/subroutine call and return overhead. It also allows the compiler to optimize the function with the rest of the code. Note that using function inlining indiscriminately can result in much larger code size and no increase in execution speed.

The PGI compilers provide two categories of inlining:

- **Automatic inlining** - During the compilation process, a hidden pass precedes the compilation pass. This hidden pass extracts functions that are candidates for inlining. The inlining of functions occurs as the source files are compiled.
- **Inline libraries** - You create inline libraries, for example using the pgfortran compiler driver and the `-o` and `-Mextract` options. There is no hidden extract pass but you must ensure that any files that depend on the inline library use the latest version of the inline library.

There are important restrictions on inlining. Inlining only applies to certain types of functions. Refer to "Restrictions on Inlining," on page 85 for more details on function inlining limitations.

This chapter describes how to use the following options related to function inlining:

```
-Mextract
-Minline
-Mrecursive
```

Invoking Function Inlining

To invoke the function inliner, use the `-Minline` option. If you do not specify an inline library, the compiler performs a special prepass on all source files named on the compiler command line before it compiles any of them. This pass extracts functions that meet the requirements for inlining and puts them in a temporary inline library for use by the compilation pass.

In PVF, inlining can be turned on using the Inlining property in the Fortran | Optimization property page. For more advanced configuration of inlining, use the Fortran | Command Line property page. For more information on these property pages, refer to "Fortran Property Pages," on page 331.

Several `-Minline` suboptions let you determine the selection criteria for functions to be inlined. These suboptions include:

except:`func`
: Inlines all eligible functions except `func`, a function in the source text. you can use a comma-separated list to specify multiple functions.

[name:]`func`
: Inlines all functions in the source text whose name matches `func`. you can use a comma-separated list to specify multiple functions.

[size:]`n`
: Inlines functions with a statement count less than or equal to `n`, the specified size.

> **Note**
>
> The size `n` may not exactly equal the number of statements in a selected function; the size parameter is merely a rough gauge.

levels:`n`
: Inlines `n` level of function calling levels. The default number is one (1). Using a level greater than one indicates that function calls within inlined functions may be replaced with inlined code. This approach allows the function inliner to automatically perform a sequence of inline and extract processes.

[lib:]`file.ext`
: Instructs the inliner to inline the functions within the library file `file.ext`. If no inline library is specified, functions are extracted from a temporary library created during an extract prepass.

> **Tip**
>
> Create the library file using the `-Mextract` option.

If you specify both a function name and a size n, the compiler inlines functions that match the function name *or* have n or fewer statements.

If a name is used without a keyword, then a name with a period is assumed to be an inline library and a name without a period is assumed to be a function name. If a number is used without a keyword, the number is assumed to be a size.

In the following example, the compiler inlines functions with fewer than approximately 100 statements in the source file `myprog.f` and writes the executable code in the output file `myprog.exe`.

```
$ pgfortran -Minline=size:100 myprog.f -o myprog
```

Refer to "–M Options by Category," on page 213 for more information on the `-Minline` options.

Using an Inline Library

If you specify one or more inline libraries on the command line with the `-Minline` option, the compiler does not perform an initial extract pass. The compiler selects functions to inline from the specified inline library. If you also specify a size or function name, all functions in the inline library meeting the selection criteria are selected for inline expansion at points in the source text where they are called.

If you do not specify a function name or a size limitation for the `-Minline` option, the compiler inlines every function in the inline library that matches a function in the source text.

In the following example, the compiler inlines the function `proc` from the inline library `lib.il` and writes the executable code in the output file `myprog.exe`.

```
$ pgfortran -Minline=name:proc,lib:lib.il myprog.f -o myprog
```

The following command line is equivalent to the preceding line, with the exception that in the following example does not use the keywords `name:` and `lib:`. You typically use keywords to avoid name conflicts when you use an inline library name that does not contain a period. Otherwise, without the keywords, a period informs the compiler that the file on the command line is an inline library.

```
$ pgfortran -Minline=proc,lib.il myprog.f -o myprog
```

Creating an Inline Library

You can create or update an inline library using the `-Mextract` command-line option. If you do not specify selection criteria with the `-Mextract` option, the compiler attempts to extract all subprograms.

Several `-Mextract` options let you determine the selection criteria for creating or updating an inline library. These selection criteria include:

`func`
: Extracts the function `func`. you can use a comma-separated list to specify multiple functions.

[name:]`func`
: Extracts the functions whose name matches `func`, a function in the source text.

[size:]`n`
: Limits the size of the extracted functions to functions with a statement count less than or equal to `n`, the specified size.

> **Note**
>
> The size n may not exactly equal the number of statements in a selected function; the size parameter is merely a rough gauge.

[lib:]`ext.lib`
: Stores the extracted information in the library directory `ext.lib`.

  If no inline library is specified, functions are extracted to a temporary library created during an extract prepass for use during the compilation stage.

When you use the `-Mextract` option, only the extract phase is performed; the compile and link phases are not performed. The output of an extract pass is a library of functions available for inlining. This output is placed in the inline library file specified on the command line with the `–o` filename specification. If the library file exists, new information is appended to it. If the file does not exist, it is created. You can use a command similar to the following:

```
$ pgfortran -Mextract=lib:lib.il myfunc.f
```

You can use the `-Minline` option with the `-Mextract` option. In this case, the extracted library of functions can have other functions inlined into the library. Using both options enables you to obtain more than one

level of inlining. In this situation, if you do not specify a library with the `-Minline` option, the inline process consists of two extract passes. The first pass is a hidden pass implied by the `-Minline` option, during which the compiler extracts functions and places them into a temporary library. The second pass uses the results of the first pass but puts its results into the library that you specify with the `-o` option.

Working with Inline Libraries

An inline library is implemented as a directory with each inline function in the library stored as a file using an encoded form of the inlinable function.

A special file named `TOC` in the inline library directory serves as a table of contents for the inline library. This is a printable, ASCII file which you can examine to locate information about the library contents, such as names and sizes of functions, the source file from which they were extracted, the version number of the extractor which created the entry, and so on.

Libraries and their elements can be manipulated using ordinary system commands.

- Inline libraries can be copied or renamed.
- Elements of libraries can be deleted or copied from one library to another.
- The `ls` or `dir` command can be used to determine the last-change date of a library entry.

Dependencies

When a library is created or updated using one of the PGI compilers, the last-change date of the library directory is updated. This allows a library to be listed as a dependence in a makefile or a PVF property and ensures that the necessary compilations are performed when a library is changed.

Updating Inline Libraries - Makefiles

If you use inline libraries you must be certain that they remain up-to-date with the source files into which they are inlined. One way to assure inline libraries are updated is to include them in a makefile.

The makefile fragment in the following example assumes the file `utils.f` contains a number of small functions used in the files `parser.f` and `alloc.f`.

This portion of the makefile:

- Maintains the inline library `utils.il`.
- Updates the library whenever you change `utils.f` or one of the include files it uses.
- Compiles `parser.f` and `alloc.f` whenever you update the library.

Example 8.1. Sample Makefile

```
SRC = mydir
FC = pgfortran
FFLAGS = -O2
main.o: $(SRC)/main.f $(SRC)/global.h
 $(FC) $(FFLAGS) -c $(SRC)/main.f
utils.o: $(SRC)/utils.f $(SRC)/global.h $(SRC)/utils.h
```

```
 $(FC) $(FFLAGS) -c $(SRC)/utils.f
utils.il: $(SRC)/utils.f $(SRC)/global.h $(SRC)/utils.h
 $(FC) $(FFLAGS) -Mextract=15 -o utils.il utils.f
parser.o: $(SRC)/parser.f $(SRC)/global.h utils.il
 $(FC) $(FFLAGS) -Minline=utils.il -c $(SRC)/parser.f
alloc.o: $(SRC)/alloc.f $(SRC)/global.h utils.il
 $(FC) $(FFLAGS) -Minline=utils.il -c $(SRC)/alloc.f
myprog: main.o utils.o parser.o alloc.o
 $(FC) -o myprog main.o utils.o parser.o alloc.o
```

Error Detection during Inlining

You can specify the `–Minfo=inline` option to request inlining information from the compiler when you invoke the inliner. For example:

```
$ pgfortran -Minline=mylib.il -Minfo=inline myext.f
```

Examples

Assume the program `dhry` consists of a single source file `dhry.f`. The following command line builds an executable file for `dhry` in which proc7 is inlined wherever it is called:

```
$ pgfortran dhry.f -Minline=proc7
```

The following command lines build an executable file for `dhry` in which proc7 plus any functions of approximately 10 or fewer statements are inlined (one level only).

> **Note**
>
> The specified functions are inlined only if they are previously placed in the inline library, `temp.il`, during the extract phase.

```
$ pgfortran dhry.f -Mextract=lib:temp.il
$ pgfortran dhry.f -Minline=10,proc7,temp.il
```

Using the same source file `dhry.f`, the following example builds an executable for `dhry` in which all functions of roughly ten or fewer statements are inlined. Two levels of inlining are performed. This means that if function A calls function B, and B calls C, and both B and C are inlinable, then the version of B which is inlined into A will have had C inlined into it.

```
$ pgfortran dhry.f -Minline=size:10,levels:2
```

Restrictions on Inlining

The following Fortran subprograms cannot be extracted:

- Main or BLOCK DATA programs.
- Subprograms containing alternate return, assigned GO TO, DATA, SAVE, or EQUIVALENCE statements.
- Subprograms containing FORMAT statements.
- Subprograms containing multiple entries.

A Fortran subprogram is not inlined if any of the following applies:

- It is referenced in a statement function.
- A common block mismatch exists; in other words, the caller must contain all common blocks specified in the callee, and elements of the common blocks must agree in name, order, and type (except that the caller's common block can have additional members appended to the end of the common block).
- An argument mismatch exists; in other words, the number and type (size) of actual and formal parameters must be equal.
- A name clash exists, such as a call to subroutine `xyz` in the extracted subprogram and a variable named `xyz` in the caller.

Chapter 9. Using OpenMP

The PGF77, PGF95, and PGFORTRAN Fortran compilers support the OpenMP Fortran Application Program Interface.

OpenMP is a specification for a set of compiler directives, an applications programming interface (API), and a set of environment variables that can be used to specify shared memory parallelism in FORTRAN programs. OpenMP may be used to obtain most of the parallel performance you can expect from your code, or it may serve as a stepping stone to parallelizing an entire application with MPI.

This chapter provides information on OpenMP as it is supported by PGI compilers.

Use the `-mp` compiler switch to enable processing of the OMP pragmas listed in this chapter. Users must link with the `-mp` switch to link the OpenMP runtime library.

This chapter describes how to use the following options related to using OpenMP:

`-mp`

OpenMP Overview

Let's look at the OpenMP shared-memory parallel programming model and some common OpenMP terminology.

OpenMP Shared-Memory Parallel Programming Model

The OpenMP shared-memory programming model is a collection of compiler directives, library routines, and environment variables that can be used to specify shared-memory parallelism in Fortran programs.

Fortran directives
: Allow users to mark sections of code that can be executed in parallel when the code is compiled using the `-mp` switch. When this switch is not present, the compiler ignores these directives.

  OpenMP Fortran directives begin with `!$OMP` , `C$OMP`, or `*$OMP`, beginning in column 1. This format allows the user to have a single source for use with or without the `-mp` switch, as these lines are then merely viewed as comments when `-mp` is not present or the compilers are not capable of handling directives.

These directives allow the user to create task, loop, and parallel section work-sharing constructs and synchronization constructs. They also allow the user to define how data is shared or copied between parallel threads of execution.

Fortran directives include a parallel region construct for writing coarse grain SPMD programs, work-sharing constructs which specify that DO loop iterations should be split among the available threads of execution, and synchronization constructs.

Note

The data environment is controlled either by using clauses on the directives or with additional directives.

Run-time library routines
: Are available to query the parallel run-time environment, for example to determine how many threads are participating in execution of a parallel region.

Environment variables
: Are available to control the execution behavior of parallel programs. For more information on OpenMP, see *www.openmp.org*.

Macro substitution
: C and C++ omp pragmas are subject to macro replacement after `#pragma omp`.

Terminology

For OpenMP 3.0 there are a number of terms for which it is useful to have common definitions.

Thread
: An execution entity with a stack and associated static memory, called *threadprivate memory*.

  - An OpenMP thread is a thread that is managed by the OpenMP runtime system.
  - A thread-safe routine is a routine that performs the intended function even when executed concurrently, that is, by more than one thread.

Region
: All code encountered during a specific instance of the execution of a given construct or of an OpenMP library routine. A region includes any code in called routines as well as any implicit code introduced by the OpenMP implementation.

  Regions are *nested* if one region is (dynamically) enclosed by another region, that is, a region is encountered during the execution of another region. PGI currently does not support nested parallel regions.

Parallel region
: In OpenMP 3.0 there is a distinction between a parallel region and an active parallel region. A parallel region can be either inactive or active.

  - An inactive parallel region is executed by a single thread.
  - An active parallel region is a parallel region that is executed by a team consisting of more than one thread.

Note

The definition of an active parallel region changed between OpenMP 2.5 and OpenMP 3.0. In OpenMP 2.5, the definition was a parallel region whose IF clause evaluates to true. To examine the significance of this change, look at the following example:

```
     program test
     logical omp_in_parallel

!$omp parallel
     print *, omp_in_parallel()
!$omp end parallel

     stop
     end
```

Suppose we run this program with OMP_NUM_THREADS set to one. In OpenMP 2.5, this program yields T while in OpenMP 3.0, the program yields F. In OpenMP 3.0, execution is not occurring by more than one thread. Therefore, change in this definition may mean previous programs require modification.

PGI currently does not support nested parallel regions so currently has only one level of active parallel regions.

Task
: A specific instance of executable code and its data environment, generated when a thread encounters a task construct or a parallel construct.

OpenMP Example

Look at the following simple OpenMP example involving loops.

Example 9.1. OpenMP Loop Example

```
      PROGRAM MAIN
      INTEGER I, N, OMP_GET_THREAD_NUM
      REAL*8 V(1000), GSUM, LSUM

      GSUM = 0.0D0
      N = 1000

      DO I = 1, N
         V(I) = DBLE(I)
      ENDDO

!$OMP PARALLEL PRIVATE(I,LSUM) SHARED(V,GSUM,N)
      LSUM = 0.0D0
!$OMP DO
      DO I = 1, N
         LSUM = LSUM + V(I)
      ENDDO
!$OMP END DO
!$OMP CRITICAL
      print *, "Thread ",OMP_GET_THREAD_NUM()," local sum: ",LSUM
      GSUM = GSUM + LSUM
!$OMP END CRITICAL
!$OMP END PARALLEL
```

```
      PRINT *, "Global Sum: ",GSUM

      STOP
      END
```

If you execute this example with the environment variable OMP_NUM_THREADS set to 4, then the output looks similar to this:

```
Thread            0 local sum:    31375.00000000000
Thread            1 local sum:    93875.00000000000
Thread            2 local sum:    156375.0000000000
Thread            3 local sum:    218875.0000000000
Global Sum:    500500.0000000000
FORTRAN STOP
```

Task Overview

Every part of an OpenMP program is part of a task. 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.

In the following sections, we use this terminology:

Task
: The package of code and instructions for allocating data created when a thread encounters a task construct. A task can be implicit or explicit.

  - An explicit task is a task generated when a task construct is encountered during execution.
  - An implicit task is a task generated by the implicit parallel region or generated when a parallel construct is encountered during execution.

Task construct
: A task directive plus a structured block

Task region
: The dynamic sequence of instructions produced by the execution of a task by a thread.

Fortran Parallelization Directives

Parallelization directives 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]
```

With the exception of the SGI-compatible DOACROSS directive, the *sentinel* must comply with these rules:

- Be one of these: !OMP, COMP, or *$OMP.
- Must start in column 1 (one).
- Must appear as a single word without embedded white space.
- The sentinel marking a DOACROSS directive is C$.

The *directive_name* can be any of the directives listed in Table 9.1, "Directive Summary Table," on page 92. The valid clauses depend on the directive. Chapter 19, "*OpenMP Reference Information*" provides a list of directives and their clauses, their usage, and examples.

In addition to the sentinel rules, the directive must also comply with these rules:

- Standard Fortran syntax restrictions, such as line length, case insensitivity, and so on, apply to the directive line.
- Initial directive lines must have a space or zero in column six.
- Continuation directive lines must have a character other than a space or a zero in column six. Continuation lines for C$DOACROSS directives are specified using the C$& sentinel.
- Directives which are presented in pairs must be used in pairs.

Clauses associated with directives have these characteristics:

- The order in which clauses appear in the parallelization directives is not significant.
- Commas separate clauses within the directives, but commas are not allowed between the directive name and the first clause.
- Clauses on directives may be repeated as needed, subject to the restrictions listed in the description of each clause.

Directive Recognition

The compiler option `-mp` enables recognition of the parallelization directives. The use of this option also implies:

`-Mreentrant`
: Local variables are placed on the stack and optimizations, such as `-Mnoframe`, that may result in non-reentrant code are disabled.

`-Miomutex`
: For directives, critical sections are generated around Fortran I/O statements.

In PVF, you set the `-mp` option by using the Process OpenMP Directives property in the Fortran | Language property page. For more information on these property pages, refer to "Fortran Property Pages," on page 331.

Directive Summary Table

The following table provides a brief summary of the directives and pragmas that PGI supports. For complete information on these statements and examples, refer to Chapter 19, "*OpenMP Reference Information*".

Table 9.1. Directive Summary Table

Fortran Directive	Description
ATOMIC	Semantically equivalent to enclosing a single statement in the CRITCIAL...END CRITICAL directive. Note: Only certain statements are allowed.
BARRIER	Synchronizes all threads at a specific point in a program so that all threads complete work to that point before any thread continues.
CRITICAL ... END CRITICAL	Defines a subsection of code within a parallel region, a critical section, which is executed one thread at a time.
DO...END DO	Provides a mechanism for distribution of loop iterations across the available threads in a parallel region.
C$DOACROSS	Specifies that the compiler should parallelize the loop to which it applies, even though that loop is not contained within a parallel region.
FLUSH	When this appears, all processor-visible data items, or, when a list is present (FLUSH [list]), only those specified in the list, are written to memory, thus ensuring that all the threads in a team have a consistent view of certain objects in memory.
MASTER ... END MASTER	Designates code that executes on the master thread and that is skipped by the other threads.
ORDERED	Defines a code block that is executed by only one thread at a time, and in the order of the loop iterations; this makes the ordered code block sequential, while allowing parallel execution of statements outside the code block.
PARALLEL DO	Enables you to specify which loops the compiler should parallelize.
PARALLEL ... END PARALLEL	Supports a fork/join execution model in which a single thread executes all statements until a parallel region is encountered.
PARALLEL SECTIONS	Defines a non-iterative work-sharing construct without the need to define an enclosing parallel region.
PARALLEL WORKSHARE ... *END PARALLEL WORKSHARE*	Provides a short form method for including a WORKSHARE directive inside a PARALLEL construct.
SECTIONS ... END SECTIONS	Defines a non-iterative work-sharing construct within a parallel region.
SINGLE ... END SINGLE	Designates code that executes on a single thread and that is skipped by the other threads.
TASK	Defines an explicit task.
TASKWAIT	Specifies a wait on the completion of child tasks generated since the beginning of the current task.

Fortran Directive	Description
THREADPRIVATE	When a common block or variable that is initialized appears in this directive, each thread's copy is initialized once prior to its first use.
WORKSHARE ... END WORKSHARE	Provides a mechanism to effect parallel execution of non-iterative but implicitly data parallel constructs.

Directive Clauses

Some directives and 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, so the clauses that are valid are included with the description of the directive and pragma.

The following table provides a brief summary of the clauses associated with OPENMP directives and pragmas that PGI supports.

Table 9.2. Directive Clauses Summary Table

This clause	Applies to this directive	Has this functionality
"COLLAPSE (n)"	DO...END DO PARALLEL DO ... END PARALLEL DO PARALLEL WORKSHARE	Specifies how many loops are associated with the loop construct.
"COPYIN (list)"	PARALLEL PARALLEL DO ... END PARALLEL DO PARALLEL SECTIONS ... END PARALLEL SECTIONS PARALLEL WORKSHARE	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. Then, 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.
"COPYPRIVATE(list)"	END SINGLE	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.
"DEFAULT"	PARALLEL PARALLEL DO ... END PARALLEL DO PARALLEL SECTIONS ... END PARALLEL SECTIONS PARALLEL WORKSHARE	Specifies the behavior of unscoped variables in a parallel region, such as the data-sharing attributes of variables.

This clause	Applies to this directive	Has this functionality
"FIRSTPRIVATE(list)"	DO PARALLEL PARALLEL DO ... END PARALLEL DO PARALLEL SECTIONS ... END PARALLEL SECTIONS PARALLEL WORKSHARE SECTIONS SINGLE	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.
"IF()"	PARALLEL ... END PARALLEL PARALLEL DO ... END PARALLEL DO PARALLEL SECTIONS ... END PARALLEL SECTIONS PARALLEL WORKSHARE	Specifies whether a loop should be executed in parallel or in serial.
"LASTPRIVATE(list)"	DO PARALLEL DO ... END PARALLEL DO PARALLEL SECTIONS ... END PARALLEL SECTIONS SECTIONS	Specifies that the enclosing context's version of the variable is set equal to the *private* version of whichever thread executes the final iteration of a for-loop construct.
"NOWAIT"	DO ... END DO SECTIONS SINGLE WORKSHARE . . . END WORKSHARE	Overrides the barrier implicit in a directive.
"NUM_THREADS"	PARALLEL PARALLEL DO ... END PARALLEL DO PARALLEL SECTIONS ... END PARALLEL SECTIONS PARALLEL WORKSHARE	Sets the number of threads in a thread team.
"ORDERED"	DO...END DO PARALLEL DO ... END PARALLEL DO	Required on a parallel FOR statement if an ordered directive is used in the loop.

This clause	Applies to this directive	Has this functionality
"PRIVATE"	DO PARALLEL PARALLEL DO ... END PARALLEL DO PARALLEL SECTIONS ... END PARALLEL SECTIONS PARALLEL WORKSHARE SECTIONS SINGLE	Specifies that each thread should have its own instance of a variable.
"REDUCTION"({operator | intrinsic } : list)	DO PARALLEL PARALLEL DO ... END PARALLEL DO PARALLEL SECTIONS ... END PARALLEL SECTIONS PARALLEL WORKSHARE SECTIONS	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.
"SCHEDULE"(type [,chunk])	DO ... END DO PARALLEL DO... END PARALLEL DO	Applies to the FOR directive, allowing the user to specify the chunking method for parallelization. Work is assigned to threads in different manners depending on the scheduling type or chunk size used.
"SHARED"	PARALLEL PARALLEL DO ... END PARALLEL DO PARALLEL SECTIONS ... END PARALLEL SECTIONS PARALLEL WORKSHARE	Specifies that one or more variables should be shared among all threads. All threads within a team access the same storage area for shared variables
"UNTIED"	TASK TASKWAIT	Specifies that any thread in the team can resume the task region after a suspension.

For complete information on these clauses, refer to the OpenMP documentation available on the WorldWide Web.

Run-time Library Routines

User-callable functions are available to the programmer to query and alter the parallel execution environment.

Note

Unlimited OpenMP thread counts are available in all PGI configurations. The number of threads is unlicensed in the OpenMP run-time libraries - up to the hard limit of 64 threads.

The following table summarizes the run-time library calls.

Table 9.3. Run-time Library Routines Summary

Run-time Library Routines with Examples	
omp_get_num_threads Returns the number of threads in the team executing the parallel region from which it is called. When called from a serial region, this function returns 1. A nested parallel region is the same as a single parallel region. By default, the value returned by this function is equal to the value of the environment variable `OMP_NUM_THREADS` or to the value set by the last previous call to **omp_set_num_threads()**.	
Fortran	`integer function omp_get_num_threads()`
omp_set_num_threads Sets the number of threads to use for the next parallel region. This subroutine can only be called from a serial region of code. If it is called from within a parallel region, or from within a subroutine that is called from within a parallel region, the results are undefined. Further, this subroutine has precedence over the `OMP_NUM_THREADS` environment variable.	
Fortran	`subroutine omp_set_num_threads(scalar_integer_exp)`
omp_get_thread_num Returns the thread number within the team. The thread number lies between 0 and **omp_get_num_threads()-1**. When called from a serial region, this function returns 0. A nested parallel region is the same as a single parallel region.	
Fortran	`integer function omp_get_thread_num()`
omp_get_ancestor_thread_num Returns, for a given nested level of the current thread, the thread number of the ancestor.	
Fortran	`integer function omp_get_ancestor_thread_num(level)` `integer level`
omp_get_active_level Returns the number of enclosing active parallel regions enclosing the task that contains the call. PGI currently supports only one level of active parallel regions, so the return value currently is 1.	
Fortran	`integer function omp_get_active_level()`
omp_get_level Returns the number of parallel regions enclosing the task that contains the call.	
Fortran	`integer function omp_get_level()`
omp_get_max_threads Returns the maximum value that can be returned by calls to **omp_get_num_threads()**. If **omp_set_num_threads()** is used to change the number of processors, subsequent calls to **omp_get_max_threads()** return the new value. Further, this function returns the maximum value whether executing from a parallel or serial region of code.	

Run-time Library Routines with Examples	
Fortran	`integer function omp_get_max_threads()`

omp_get_num_procs

Returns the number of processors that are available to the program

Fortran	`integer function omp_get_num_procs()`

omp_get_stack_size

Returns the value of the OpenMP internal control variable that specifies the size that is used to create a stack for a newly created thread.

This value may *not* be the size of the stack of the current thread.

Fortran	`!omp_get_stack_size interface`
	`function omp_get_stack_size ()`
	`use omp_lib_kinds`
	`integer ( kind=OMP_STACK_SIZE_KIND )`
	`:: omp_get_stack_size`
	`end function omp_get_stack_size`
	`end interface`

omp_set_stack_size

Changes the value of the OpenMP internal control variable that specifies the size to be used to create a stack for a newly created thread.

The integer argument specifies the stack size in kilobytes. The size of the stack of the current thread cannot be changed. In the PGI implementation, all OpenMP or auto-parallelization threads are created just prior to the first parallel region; therefore, only calls to `omp_set_stack_size()` that occur prior to the first region have an effect.

Fortran	`subroutine omp_set_stack_size(integer(KIND=OMP_STACK_SIZE_KIND))`

omp_get_team_size

Returns, for a given nested level of the current thread, the size of the thread team to which the ancestor belongs.

Fortran	`integer function omp_get_team_size (level)` `integer level`

omp_in_parallel

Returns whether or not the call is within a parallel region.

Returns `.TRUE.` if called from within a parallel region and `.FALSE.` if called outside of a parallel region. When called from within a parallel region that is serialized, for example in the presence of an IF clause evaluating `.FALSE.`, the function returns `.FALSE.`.

Fortran	`logical function omp_in_parallel()`

omp_set_dynamic

Allows automatic dynamic adjustment of the number of threads used for execution of parallel regions.

This function is recognized, but currently has no effect.

Fortran	`subroutine omp_set_dynamic(scalar_logical_exp)`

Run-time Library Routines with Examples	
omp_get_dynamic	
Allows the user to query whether automatic dynamic adjustment of the number of threads used for execution of parallel regions is enabled. This function is recognized, but currently always returns .FALSE..	
Fortran	`logical function omp_get_dynamic()`
omp_set_nested	
Allows enabling/disabling of nested parallel regions. This function is recognized, but currently has no effect.	
Fortran	`subroutine omp_set_nested(nested)` `  logical nested`
omp_get_nested	
Allows the user to query whether dynamic adjustment of the number of threads available for execution of parallel regions is enabled. This function is recognized, but currently always returns .FALSE..	
Fortran	`logical function omp_get_nested()`
omp_set_schedule	
Set the value of the run_sched_var.	

Fortran	`subroutine omp_set_schedule(kind, modifier)`
	`  include 'omp_lib_kinds.h'`
	`  integer (kind=omp_sched_kind) kind`
	`  integer modifier`

omp_get_schedule	
Retrieve the value of the run_sched_var.	

Fortran	`subroutine omp_get_schedule(kind, modifier)`
	`  include 'omp_lib_kinds.h'`
	`  integer (kind=omp_sched_kind) kind`
	`  integer modifier`

omp_get_wtime	
Returns the elapsed wall clock time, in seconds, as a DOUBLE PRECISION value. Times returned are per-thread times, and are not necessarily globally consistent across all threads.	
Fortran	`double precision function omp_get_wtime()`
omp_get_wtick	
Returns the resolution of omp_get_wtime(), in seconds, as a DOUBLE PRECISION value.	
Fortran	`double precision function omp_get_wtick()`

Run-time Library Routines with Examples	
omp_init_lock Initializes a lock associated with the variable lock for use in subsequent calls to lock routines. The initial state of the lock is unlocked. If the variable is already associated with a lock, it is illegal to make a call to this routine.	

Fortran	`subroutine omp_init_lock(lock)`
	`  include 'omp_lib_kinds.h'`
	`  integer(kind=omp_lock_kind) lock`

omp_destroy_lock Disassociates a lock associated with the variable.	

Fortran	`subroutine omp_destroy_lock(lock)`
	`  include 'omp_lib_kinds.h'`
	`  integer(kind=omp_lock_kind) lock`

omp_set_lock Causes the calling thread to wait until the specified lock is available. The thread gains ownership of the lock when it is available. If the variable is not already associated with a lock, it is illegal to make a call to this routine.	

Fortran	`subroutine omp_set_lock(lock)`
	`  include 'omp_lib_kinds.h'`
	`  integer(kind=omp_lock_kind) lock`

omp_unset_lock Causes the calling thread to release ownership of the lock associated with `integer_var`. If the variable is not already associated with a lock, it is illegal to make a call to this routine.	

Fortran	`subroutine omp_unset_lock(lock)`
	`  include 'omp_lib_kinds.h'`
	`  integer(kind=omp_lock_kind) lock`

omp_test_lock Causes the calling thread to try to gain ownership of the lock associated with the variable. The function returns `.TRUE.` if the thread gains ownership of the lock; otherwise it returns `.FALSE.`. If the variable is not already associated with a lock, it is illegal to make a call to this routine.	

Fortran	`logical function omp_test_lock(lock)`
	`  include 'omp_lib_kinds.h'`
	`  integer(kind=omp_lock_kind) lock`

Environment Variables

You can use OpenMP environment variables to control the behavior of OpenMP programs. These environment variables allow you to set and pass information that can alter the behavior of directives.

Note

To set the environment for programs run from within PVF, whether or not they are run in the debugger, use the environment properties available in the "Debugging Property Page," on page 328.

The following summary table is a quick reference for the OpenMP environment variables that PGI uses. For detailed descriptions of each of these variables, refer to "OpenMP Environment Variables," on page 262.

Table 9.4. OpenMP-related Environment Variable Summary Table

Environment Variable	Default	Description
OMP_DYNAMIC	FALSE	Currently has no effect. Typically enables (TRUE) or disables (FALSE) the dynamic adjustment of the number of threads.
OMP_NESTED		Currently has no effect. Typically specifies the maximum number of nested parallel regions.
OMP_MAX_ACTIVE_LEVELS	FALSE	Currently has no effect. Typically enables (TRUE) or disables (FALSE) nested parallelism.
OMP_NUM_THREADS	1	Specifies the number of threads to use during execution of parallel regions.
OMP_SCHEDULE	STATIC with chunk size of 1	Specifies the type of iteration scheduling and optionally the chunk size to use for *omp for* and *omp parallel for* loops that include the run-time schedule clause. The supported schedule types, which can be specified in upper- or lower-case are static, dynamic, guided, and auto.
OMP_STACKSIZE		Overrides the default stack size for a newly created thread.
OMP_THREAD_LIMIT	64	Specifies the absolute maximum number of threads that can be used in a program.
OMP_WAIT_POLICY	ACTIVE	Sets the behavior of idle threads, defining whether they spin or sleep when idle. The values are ACTIVE and PASSIVE.

Chapter 10. Using an Accelerator

An accelerator is a special-purpose co-processor attached to a CPU and to which the CPU can offload data and executable kernels to perform compute-intensive calculations. This chapter describes a collection of compiler directives used to specify regions of code in Fortran that can be offloaded from a *host* CPU to an attached *accelerator*.

Overview

The programming model and directives described in this chapter allow programmers to create high-level *host+accelerator* programs without the need to explicitly initialize the accelerator, manage data or program transfers between the host and accelerator, or initiate accelerator startup and shutdown. Rather, all of these details are implicit in the programming model and are managed by the PGI Fortran accelerator compilers.

The method described provides a model for accelerator programming that is portable across operating systems and various host CPUs and accelerators. The directives allow a programmer to migrate applications incrementally to accelerator targets using standards-compliant Fortran.

This programming model allows the programmer to augment information available to the compilers, including specification of data local to an accelerator region, guidance on mapping of loops onto an accelerator, and similar performance-related details.

Components

The PGI Accelerator compiler technology includes the following components:

- PGF95 auto-parallelizing accelerator-enabled Fortran 90/95 compiler
- NVIDIA CUDA Toolkit components
- PVF Target Accelerators property page
- A simple command-line tool to detect whether the system has an appropriate GPU or accelerator card

No accelerator-enabled debugger is included with this release

Availability

The PGI 10.0 Fortran Accelerator compilersare available only on x86 processor-based workstations and servers with an attached NVIDIA CUDA-enabled GPU or Tesla card. These compilers target all platforms that

PGI supports except 64-bit Mac OS X. All examples included in this chapter are developed and presented on such a platform. For a list of supported GPUs, refer to the Accelerator Installation and Supported Platforms list in the latest PVF Release Notes.

User-directed Accelerator Programming

In user-directed accelerator programming the user specifies the regions of a host program to be targeted for offloading to an accelerator device. The bulk of a user's program, as well as regions containing constructs that are not supported on the targeted accelerator, are executed on the host. This chapter concentrates on specification of loops and regions of code to be offloaded to an accelerator.

Features Not Covered or Implemented

This chapter does not describe features or limitations of the host programming environment as a whole. Further, it does not cover automatic detection and offloading of regions of code to an accelerator by a compiler or other tool. While future versions of the PGI compilers may allow for automatic offloading or multiple accelerators of different types, these features are not currently supported.

Terminology

Clear and consistent terminology is important in describing any programming model. This section provides definitions of the terms required for you to effectively use this chapter and the associated programming model.

Accelerator
: a special-purpose co-processor attached to a CPU and to which the CPU can offload data and executable kernels to perform compute-intensive calculations.

Compute intensity
: for a given loop, region, or program unit, the ratio of the number of arithmetic operations performed on computed data divided by the number of memory transfers required to move that data between two levels of a memory hierarchy.

Compute region
: a region defined by an Accelerator compute region directive. A compute region is a structured block containing loops which are compiled for the accelerator. A compute region may require device memory to be allocated and data to be copied from host to device upon region entry, and data to be copied from device to host memory and device memory deallocated upon exit. Compute regions may not contain other compute regions or data regions.

CUDA
: stands for Compute Unified Device Architecture; the CUDA environment from NVIDIA is a C-like programming environment used to explicitly control and program an NVIDIA GPU.

Data region
: a region defined by an Accelerator data region directive, or an implicit data region for a function or subroutine containing Accelerator directives. Data regions typically require device memory to be allocated and data to be copied from host to device memory upon entry, and data to be copied from device to host memory and device memory deallocated upon exit. Data regions may contain other data regions and compute regions.

Device
: a general reference to any type of accelerator.

Device memory
: memory attached to an accelerator which is physically separate from the host memory.

Directive
: a specially formatted comment statement that is interpreted by a compiler to augment information about or specify the behavior of the program.

DMA
: Direct Memory Access, a method to move data between physically separate memories; this is typically performed by a DMA engine, separate from the host CPU, that can access the host physical memory as well as an IO device or GPU physical memory.

GPU
: a Graphics Processing Unit; one type of accelerator device.

GPGPU
: General Purpose computation on Graphics Processing Units.

Host
: the main CPU that in this context has an attached accelerator device. The host CPU controls the program regions and data loaded into and executed on the device.

Loop trip count
: the number of times a particular loop executes.

OpenCL - Open Compute Language
: a proposed standard C-like programming environment similar to CUDA that enables portable low-level general-purpose programming on GPUs and other accelerators.

Private data
: with respect to an iterative loop, data which is used only during a particular loop iteration. With respect to a more general region of code, data which is used within the region but is not initialized prior to the region and is re-initialized prior to any use after the region.

Region
: a structured block identified by the programmer or implicitly defined by the language. Certain actions may occur when program execution reaches the start and end of a region, such as device memory allocation or data movement between the host and device memory. Loops in a compute region are targeted for execution on the accelerator.

Structured block
: a block of executable statements with a single entry at the top and a single exit at the bottom.

Vector operation
: a single operation or sequence of operations applied uniformly to each element of an array.

Visible device copy
: a copy of a variable, array, or subarray allocated in device memory, that is visible to the program unit being compiled.

System Requirements

To use the PGI Accelerator compiler features, you must install the NVIDIA drivers. You may download these components from the NVIDIA website at

```
www.nvidia.com/cuda
```

These are not PGI products, and are licensed and supported by NVIDIA.

Note

You must be using an operating system that is supported by both the current PGI release and by the CUDA software and drivers.

Supported Processors and GPUs

This PGI Accelerator compiler release supports all AMD64 and Intel 64 host processors supported by Release 9.0 or higher of the PGI compilers and tools. You can use the `-tp <target>` flag as documented in the release to specify the target processor.

Use the `-ta=nvidia` flag to enable the accelerator directives and target the NVIDIA GPU. You can then use the generated code on any system with CUDA installed that has a CUDA-enabled GeForce, Quadro, or Tesla card.

In PVF you can use the PVF Target Accelerators property page to enable accelerator compilation. For more information on the properties, refer to "Fortran | Target Accelerators," on page 345.

For more information on these flags as they relate to accelerator technology, refer to "Applicable Command Line Options," on page 115.

For a complete list of supported GPUs, refer to the NVIDIA website at:

```
www.nvidia.com/object/cuda_learn_products.html
```

You can detect whether the system has CUDA properly installed and has an attached graphics card by running the **pgaccelinfo** command, which is delivered as part of the PGI Accelerator compilers software package.

Installation and Licensing

Note

The PGI Accelerator compilers require a separate license key in addition to a normal PGI Visual Fortran license.

Enable Accelerator Compilation

Once you have installed PVF Release 2010, you can enable accelerator compilation by using the properties available on the Fortran | Target Accelerators property page. For more information about these properties, refer to "Fortran | Target Accelerators".

Execution Model

The execution model targeted by the PGI Accelerator compilers is host-directed execution with an attached accelerator device, such as a GPU. The bulk of a user application executes on the host. Compute intensive regions are offloaded to the accelerator device under control of the host. The accelerator device executes kernels, which may be as simple as a tightly-nested loop, or as complex as a subroutine, depending on the accelerator hardware.

Host Functions

Even in accelerator-targeted regions, the host must orchestrate the execution; it

- allocates memory on the accelerator device
- initiates data transfer
- sends the kernel code to the accelerator
- passes kernel arguments
- queues the kernel
- waits for completion
- transfers results back to the host
- deallocates memory

Note

In most cases, the host can queue a sequence of kernels to be executed on the device, one after the other.

Levels of Parallelism

Most current GPUs support two levels of parallelism:

- an outer *doall* (fully parallel) loop level
- an inner *synchronous* (SIMD or vector) loop level

Each level can be multidimensional with 2 or 3 dimensions, but the domain must be strictly rectangular. The *synchronous* level may not be fully implemented with SIMD or vector operations, so explicit synchronization is supported and required across this level. No synchronization is supported between parallel threads across the *doall* level.

The execution model on the device side exposes these two levels of parallelism and the programmer is required to understand the difference between, for example, a fully parallel loop and a loop that is vectorizable but requires synchronization across iterations. All fully parallel loops can be scheduled for either doall or synchronous parallel execution, but by definition SIMD vector loops that require synchronization can only be scheduled for synchronous parallel execution.

Memory Model

The most significant difference between a *host-only* program and a *host+accelerator* program is that the memory on the accelerator can be completely separate from host memory, which is the case on most current GPUs. For example:

- The host cannot read or write accelerator memory by reference because it is not mapped into the virtual memory space of the host.
- All data movement between host memory and accelerator memory must be performed by the host through runtime library calls that explicitly move data between the separate memories.
- It is not valid to assume the accelerator can read or write host memory, though this may be supported by accelerators in the future.

Separate Host and Accelerator Memory Considerations

The concept of separate host and accelerator memories is very apparent in low-level accelerator programming models such as CUDA or OpenCL, in which data movement between the memories dominates user code. In the PGI Accelerator programming model, data movement between the memories is implicit and managed by the compiler.

The programmer must be aware of the potentially separate memories for many reasons, including but not limited to:

- Memory bandwidth between host memory and accelerator memory determines the compute intensity required to effectively accelerate a given region of code.
- Limited size of accelerator memory may prohibit offloading of regions of code that operate on very large amounts of data.

Accelerator Memory

On the accelerator side, current GPUs implement a weak memory model. In particular, they do not support memory coherence between threads unless those threads are parallel only at the synchronous level and the memory operations are separated by an explicit barrier. Otherwise, if one thread updates a memory location and another reads the same location, or two threads store a value to the same location, the hardware does not guarantee the results. While the results of running such a program might be inconsistent, it is not accurate to say that the results are incorrect. By definition, such programs are defined as being in error. While a compiler can detect some potential errors of this nature, it is nonetheless possible to write an accelerator region that produces inconsistent numerical results.

Cache Management

Some current GPUs have a software-managed cache, some have hardware-managed caches, and most have hardware caches that can be used only in certain situations and are limited to read-only data. In low-level programming models such as CUDA or OpenCL, it is up to the programmer to manage these caches. However, in the PGI Accelerator programming model, the compiler manages these caches using hints from the programmer in the form of directives.

Running an Accelerator Program

In PVF you can use the PVF Target Accelerators property page to enable accelerator compilation. For more information on the properties, refer to"Fortran | Target Accelerators," on page 345.

Running a program that has accelerator directives and was compiled and linked with the `-ta=nvidia` flag is the same as running the program compiled without the `-ta=nvidia` flag.

- The program looks for and dynamically loads the CUDA libraries. If the libraries are not available, or if they are in a different directory than they were when the program was compiled, you may need to append the CUDA library directory to your PATH environment variable on Windows.
- On Linux, if you have no server running on your NVIDIA GPU, when your program reaches its first accelerator region, there may be a 0.5 to 1.5 second pause to warm up the GPU from a power-off condition. You can avoid this delay by running the `pgcudainit` program in the background, which keeps the GPU powered on.
- If you run an accelerated program on a system without a CUDA-enabled NVIDIA GPU, or without the CUDA software installed in a directory where the runtime library can find it, the program fails at runtime with an error message.
- If you set the environment variable `ACC_NOTIFY` to a nonzero integer value, the runtime library prints a line to standard error every time it launches a kernel.

Accelerator Directives

This section provides an overview of the Fortran directives used to delineate accelerator regions and to augment information available to the compiler for scheduling of loops and classification of data. For complete descriptions of each accelerator directive, refer to "PGI Accelerator Directives," on page 265.

Enable Accelerator Directives

PGI Accelerator compilers enable accelerator directives with the `-ta` command line option.In PVF, use the"Fortran | Target Accelerators" page to enable the `-ta` option. For more information on this option as it relates to the Accelerator, refer to "Applicable Command Line Options," on page 115.

Note

The syntax used to define directives allows compilers to ignore accelerator directives if support is disabled or not provided.

_ACCEL macro

The `_ACCEL` macro name is defined to have a value `yyyymm` where `yyyy` is the year and `mm` is the month designation of the version of the Accelerator directives supported by the implementation. For example, the version for May, 2009 is 200905. This macro must be defined by a compiler when accelerator directives are enabled.

Format

The specific format of the directive depends on the language and the format or form of the source.

Directives include a name and clauses, and the format of the directive depends on the type:

- Free-form Fortran directives, described in "Free-Form Fortran Directives"
- Fixed-form Fortran directives, described in "Fixed-Form Fortran Directives"

Note

This document uses free form for all PGI Accelerator compiler Fortran directive examples.

Rules

The following rules apply to all PGI Accelerator compiler directives:

- Only one directive-name can be specified per directive.
- The order in which clauses appear is not significant.
- Clauses may be repeated unless otherwise specified.
- For clauses that have a *list* argument, a list is a comma-separated list of variable names, array names, or, in some cases, subarrays with subscript ranges.

Free-Form Fortran Directives

PGI Accelerator compiler Fortran directives can be either Free-Form or Fixed-Form directives. Free-Form Accelerator directives are specified with the `!$acc` mechanism.

Syntax

The syntax of directives in free-form source files is:

```
!$acc directive-name [clause [,clause]…]
```

Rules

In addition to the general directive rules, the following rules apply to PGI Accelerator compiler Free-Form Fortran directives:

- The comment prefix (!) may appear in any column, but may only be preceded by white space (spaces and tabs).
- The sentinel (!$acc) must appear as a single word, with no intervening white space.
- Line length, white space, and continuation rules apply to the directive line.
- Initial directive lines must have a space after the sentinel.
- Continued directive lines must have an ampersand (&) as the last nonblank character on the line, prior to any comment placed in the directive.
- Comments may appear on the same line as the directive, starting with an exclamation point and extending to the end of the line.

- If the first nonblank character after the sentinel is an exclamation point, the line is ignored.
- Directives are case-insensitive.
- Directives cannot be embedded within continued statements.
- Statements must not be embedded within continued directives.

Fixed-Form Fortran Directives

Fixed-Form Accelerator directives are specified using one of three formats.

Syntax

The syntax of directives in fixed-form source files is one these three formats:

```
!$acc directive-name [clause [,clause]…]
c$acc directive-name [clause [,clause]…]
*$acc directive-name [clause [,clause]…]
```

Rules

In addition to the general directive rules, the following rules apply to Accelerator Fixed-Form Fortran directives:

- The sentinel (!acc, cacc, or *$acc) must occupy columns 1-5.
- Fixed form line length, white space, continuation, and column rules apply to the directive line.
- Initial directive lines must have a space or zero in column 6, and continuation directive lines must have a character other than a space or zero in column 6.
- Comments may appear on the same line as a directive, starting with an exclamation point on or after column 7 and continuing to the end of the line.
- Directives are case-insensitive.
- Directives cannot be embedded within continued statements.
- Statements must not be embedded within continued directives.

Accelerator Directive Summary

PGI currently supports these types of accelerator directives, which are defined in more detail in "PGI Accelerator Directives," on page 265:

Accelerator Compute Region Directive
Accelerator Loop Mapping Directive
Combined Directive
Accelerator Declarative Data Directive
Accelerator Update Directive

Table 10.1 lists and briefly describes each of the accelerator directives that PGI currently supports. For a complete description of each directive, refer to "PGI Accelerator Directives," on page 265.

Table 10.1. PGI Accelerator Directive Summary Table

This directive...	Accepts these clauses...	Has this functionality...
Accelerator Compute Region Directive	if(condition) copy (*list*) copyin(*list*) copyout(*list*) local(*list*) updatein(*list*) updateout(*list*)	Defines the region of the program that should be compiled for execution on the accelerator device.

Fortran Syntax
`!$acc region [clause [, clause]…]`
`  *structured block*`
`!$acc end region`		

Accelerator Data Region Directive	copy (*list*) copyin(*list*) copyout(*list*) local(*list*) mirror(*list*) updatein(*list*) updateout(*list*)	Defines data, typically arrays, that should be allocated in the device memory for the duration of the data region, 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.

Fortran Syntax
`!$acc data region [clause [, clause]…]`
`  *structured block*`
`!$acc end data region`		

Accelerator Loop Mapping Directive	cache(*list*) host [(*width*)] independent kernel parallel [(*width*)] private(*list*) seq [(*width*)] unroll [(*width*)] vector [(*width*)]	Describes what type of parallelism to use to execute the loop and declare loop-private variables and arrays. Applies to a loop which must appear on the following line.

Fortran Syntax
`!$acc do [clause [,clause]…]`
`  *do loop*`		

This directive...	Accepts these clauses...	Has this functionality...
Combined Directive	Any clause that is allowed on a region directive or a loop directive is allowed on a combined directive.	Is a shortcut for specifying a loop directive nested immediately inside an accelerator compute region directive. The meaning is identical to explicitly specifying a region construct containing a loop directive.

Fortran Syntax
`!$acc region do [clause [, clause]...]`
`do loop`		

Accelerator Declarative Data Directive	copy (*list*) copyin(*list*) copyout(*list*) local(*list*) mirror (*list*) reflected(*list*)	Specifies 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. 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. Creates a visible device copy of the variable or array.
Fortran Syntax `!$acc declclause [,declclause]...`		
Accelerator Update Directive	host (*list*) device(*list*)	Used within an explicit or implicit data region to update all or part of a host memory array with values from the corresponding array in device memory, or to update all or part of a device memory array with values from the corresponding array in host memory.
Fortran Syntax `!$acc update updateclause [,updateclause]...`		

Accelerator Directive Clauses

Table 10.2 provides an alphabetical listing and brief description of each clause that is applicable for the various Accelerator directives. The table also indicates for which directives the clause is applicable.

For more information on the restrictions and use of each clause, refer to "PGI Accelerator Directive Clauses," on page 271.

Table 10.2. Directive Clauses Summary

Use this clause...	In these directives...	To do this...
cache (*list*)	Accelerator Loop Mapping	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.

Use this clause...	In these directives...	To do this...
copy (*list*)	Accelerator Data Region Declarative Data	Declares 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.
copyin (*list*)	Accelerator Data Region Declarative Data	Declares that the variables, arrays or subarrays in the list have values in the host memory that need to be copied to the accelerator memory.
copyout (*list*)	Accelerator Data Region Declarative Data	Declares 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 at the end of the accelerator region.
device (*list*)	Update	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. Copy occurs before beginning execution of the compute or data region.
host (*list*)	Update	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.
host [(*width*)]	Accelerator Loop Mapping	Tells the compiler to execute the loop sequentially on the host processor.
if (condition)	Accelerator Compute Data Region	When present, tells the compiler to generate two copies of the region - one for the accelerator, one for the host - and to generate code to decide which copy to execute.
independent	Accelerator Loop Mapping	Tells the compiler that the iterations of this loop are data-independent of each other, thus allowing the compiler to generate code to examine the iterations in parallel, without synchronization.
kernel	Accelerator Loop Mapping	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.
local (*list*)	Accelerator Data Region Declarative Data	Declares 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*)	Accelerator Data Region Declarative Data	Declares that the arrays in the *list* need to mirror the allocation state of the host array within the region. Valid only in Fortran on Accelerator data region directive.

Use this clause...	In these directives...	To do this...
parallel [(*width*)]	Accelerator Loop Mapping	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
private (*list*)	Accelerator Loop Mapping	Declares 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.
reflected (*list*)	Declarative Data	Declares 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.
seq [(*width*)]	Accelerator Loop Mapping	Tells the compiler to execute this loop sequentially on the accelerator. There is no maximum number of iterations for a seq schedule.
unroll [(*width*)]	Accelerator Loop Mapping	Tells the compiler to unroll *width* iterations for sequential execution on the accelerator. The *width* argument must be a compile time positive constant integer.
updatein (*list*)	Accelerator Data Region	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*)	Accelerator Data Region	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.
vector [(*width*)]	Accelerator Loop Mapping	Tells the compiler to execute this loop in vector mode on the accelerator.

PGI Accelerator Compilers Runtime Libraries

This section provides an overview of the 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.

Note

In Fortran, none of the PGI Accelerator compilers runtime library routines may be called from a PURE or ELEMENTAL procedure.

Runtime Library Definitions

There are separate runtime library files for Fortran.

Fortran Runtime Library Files

In Fortran, interface declarations are provided in a Fortran include file named `accel_lib.h` and in a Fortran module named `accel_lib`. These files define:

- Interfaces for all routines in this section.
- Integer parameters to define integer kinds for arguments to those routines.
- Integer parameters to describe types of accelerators.
- The integer parameter `accel_version` with a value `yyyymm` where `yyyy` and `mm` are the year and month designations of the version of the Accelerator programming model supported. This value matches the value of the preprocessor variable `_ACCEL`.

Runtime Library Routines

Table 10.3 lists and briefly describes the supported PGI Accelerator compilers runtime library routines. For a complete description of these routines, refer to "PGI Accelerator Runtime Routines," on page 279.

Table 10.3. Accelerator Runtime Library Routines

This Runtime Library Routine...	Does this...
acc_get_device	Returns the type of accelerator device used to run the next accelerator region, if one is selected.
acc_get_device_num	Returns the number of the device being used to execute an accelerator region.
acc_get_num_devices	Returns the number of accelerator devices of the given type attached to the host.
acc_init	Connects to and initializes the accelerator device and allocates control structures in the accelerator library.
acc_on_device	Tells the program whether it is executing on a particular device.
acc_set_device	Tells the runtime which type of device to use when executing an accelerator compute region.
acc_set_device_num	Tells the runtime which device of the given type to use among those that are attached.
acc_shutdown	Tells the runtime to shutdown the connection to the given accelerator device, and free up any runtime resources.

Environment Variables

PGI supports environment variables that modify the behavior of accelerator regions. This section defines the user-setable environment variables used to control behavior of accelerator-enabled programs at execution. These environment variables must comply with these rules:

- 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.

Table 10.4 lists and briefly describes the Accelerator environment variables that PGI supports.

Table 10.4. Accelerator Environment Variables

This environment variable...	Does this...
ACC_DEVICE	Controls which accelerator device 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, and currently may be the string NVIDIA or HOST.
ACC_DEVICE_NUM	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.
ACC_NOTIFY	When set to a non-negative integer, indicates to print a message to standard output when a kernel is executed on an accelerator.

Applicable PVF Property Pages

The following property pages are applicable specifically when working with accelerators.

"Fortran | Target Accelerators"
: Use the `-ta` option to enable recognition of Accelerator directives.

"Fortran | Target Processors"
: Use the `-tp` option to specify the target host processor architecture.

"Fortran | Diagnostics"
: Use the `-Minfo` option to see messages about the success or failure of the compiler in translating the accelerator region into GPU kernels.

For more information about the many suboptions available with these options, refer to the respective sections in "Fortran Property Pages," on page 331

Applicable Command Line Options

The following command line options are applicable specifically when working with accelerators. Each of these command line options are available through the property pages described in the previous section: "Applicable PVF Property Pages".

`-ta`
: Use this option to enable recognition of the **!$ACC** directives in Fortran.

`-tp`
Use this option to specify the target host processor architecture.

`-Minfo` or `-Minfo=accel`
Use this option to see messages about the success or failure of the compiler in translating the accelerator region into GPU kernels.

The `-ta` flag has the following accelerator-related suboptions:

nvidia
Select NVIDIA accelerator target. This option has a number of suboptions:

analysis	Perform loop analysis only; do not generate GPU code.
cc10, cc11, cc12, cc13, cc20	Generate code for compute capability 1.0, 1.1, 1.2, 1.3, or 2.0 respectively; multiple selections are valid.
cuda2.3 or 2.3	Specify the CUDA 2.3 version of the toolkit.
cuda3.0 or 3.0	Specify the CUDA 3.0 version of the toolkit.
fastmath	Use routines from the fast math library.
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," on page 118.
[no]wait	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 Binary for Accelerators," on page 116.

The compiler automatically invokes the necessary CUDA software tools to create the kernel code and embeds the kernels in the object file.

PGI Unified Binary for Accelerators

Note

The information and capabilities described in this section are only supported for 64-bit systems.

PGI compilers support the PGI Unified Binary feature to generate executables with functions optimized for different host processors, all packed into a single binary. This release extends the PGI Unified Binary technology for accelerators. Specifically, you can generate a single binary that includes two versions of functions:

- one is optimized for the accelerator
- one runs on the host processor when the accelerator is not available or when you want to compare host to accelerator execution.

To enable this feature, use the properties available on the "Fortran | Target Accelerators" property page.

This flag tells the compiler to generate two versions of functions that have valid accelerator regions.

- A compiled version that targets the accelerator.
- A compiled version that ignores the accelerator directives and targets the host processor.

If you use the `-Minfo` flag, which you enable in PVF through the "Fortran | Diagnostics" property page, you get messages similar to the following:

```
s1:
     12, PGI Unified Binary version for -tp=barcelona-64 -ta=host
     18, Generated an alternate loop for the inner loop
         Generated vector sse code for inner loop
         Generated 1 prefetch instructions for this loop
s1:
     12, PGI Unified Binary version for -tp=barcelona-64 -ta=nvidia
     15, Generating copy(b(:,2:90))
         Generating copyin(a(:,2:90))
     16, Loop is parallelizable
     18, Loop is parallelizable
         Parallelization requires privatization of array t(2:90)
         Accelerator kernel generated
         16, !$acc do parallel
         18, !$acc do parallel, vector(256)
             Using register for t
```

The PGI Unified Binary message shows that two versions of the subprogram `s1` were generated:

- one for no accelerator (`-ta=host`)
- one for the NVIDIA GPU (`-ta=nvidia`)

At run time, the program tries to load the NVIDIA CUDA dynamic libraries and test for the presence of a GPU. If the libraries are not available or no GPU is found, the program runs the host version.

You can also set an environment variable to tell the program to run on the NVIDIA GPU. To do this, set ACC_DEVICE to the value `NVIDIA` or `nvidia`. Any other value of the environment variable causes the program to use the host version.

Note

The only supported `-ta` targets for this release are `nvidia` and `host`.

Multiple Processor Targets

With 64-bit processors, you can use the `-tp` flag with multiple processor targets along with the `-ta` flag. You see the following behavior:

- If you specify one `-tp` value and one `-ta` value:

You see one version of each subprogram generated for that specific target processor and target accelerator.

- If you specify one `-tp` value and multiple `-ta` values:

  The compiler generates two versions of subprograms that contain accelerator regions for the specified target processor and each target accelerator.

- If you specify multiple `-tp` values and one `-ta` value:

  If 2 or more `-tp` values are given, the compiler generates up to that many versions of each subprogram, for each target processor, and each version also targets the selected accelerator.

- If you specify multiple `-tp` values and multiple `-ta` values:

  With 'N' `-tp` values and two `-ta` values, the compiler generates up to N+1 versions of the subprogram. It first generates up to N versions, for each `-tp` value, ignoring the accelerator regions, which is equivalent to using `-ta=host`. It then generates one additional version with the accelerator target.

Profiling Accelerator Kernels

This release supports the command line option:

```
-ta=nvidia,time
```

The `time` suboption links in a timer library, which collects and prints out simple timing information about the accelerator regions and generated kernels.

Example 10.1. Accelerator Kernel Timing Data

```
bb04.f90
  s1
   15: region entered 1 times
  time(us): total=1490738
                 init=1489138 region=1600
                 kernels=155 data=1445
  w/o init: total=1600 max=1600
                 min=1600 avg=1600
   18: kernel launched 1 times
  time(us): total=155 max=155 min=155 avg=155
```

In this example, a number of things are occurring:

- For each accelerator region, the file name `/proj/qa/tests/accel/bb04.f90` and subroutine or function name `s1` is printed, with the line number of the accelerator region, which in the example is `15`.
- The library counts how many times the region is entered (`1` in the example) and the microseconds spent in the region (in this example `1490738`), which is split into initialization time (in this example `1489138`) and execution time (in this example `1600`).
- The execution time is then divided into kernel execution time and data transfer time between the host and GPU.
- For each kernel, the line number is given, (`18` in the example), along with a count of kernel launches, and the total, maximum, minimum, and average time spent in the kernel, all of which are 155 in this example.

Related Accelerator Programming Tools

PGPROF pgcollect

The PGI profiler, PGPROF, has an **Accelerator tab** - that displays profiling information provided by the accelerator. This information is available in the file `pgprof.out` and is collected by using **pgcollect** on an executable binary compiled for an accelerator target. For more information on **pgcollect**, refer to Chapter 22, "pgcollect Reference," of the PGI Tools Guide.

NVIDIA CUDA Profile

You can use the NVIDIA CUDA Profiler with PGI-generated code for the NVIDIA GPUs. You may download the CUDA Profiler from the same website as the CUDA software:

```
www.nvidia.com/cuda
```

Documentation and support is provided by NVIDIA.

TAU - Tuning and Analysis Utility

You can use the TAU (Tuning and Analysis Utility), version 2.18.1+, with PGI-generated accelerator code. TAU instruments code at the function or loop level, and version 2.18.1 is enhanced with support to track performance in accelerator regions. TAU software and documentation is available at this website:

```
http://tau.uoregon.edu
```

Supported 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.

Intrinsics make the use of processor-specific enhancements easier because they provide a 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 section contains an overview of the Fortran intrinsics that the accelerator supports.

Supported Fortran Intrinsics Summary Table

Table 10.5 is an alphabetical summary of the supported Fortran intrinsics that the accelerator supports. These functions are specific to Fortran 90/95 unless otherwise specified.

Note

For complete descriptions of these intrinsics, refer to the Chapter 6, "Fortran Intrinsics" of the PGI Fortran Reference.

In most cases PGI provides support for all the data types for which the intrinsic is valid. When support is available for only certain data types, the middle column of the table specifies which ones, using the following codes:

I for integer
S for single precision real
D for double precision real
C for single precision complex
Z for double precision complex

Table 10.5. Supported Fortran Intrinsics

This intrinsic		Returns this value ...
ABS	I,S,D	absolute value of the supplied argument.
ACOS		arccosine of the specified value.
AINT		truncation of the supplied value to a whole number.
ANINT		nearest whole number to the supplied argument.
ASIN		arcsine of the specified value.
ATAN		arctangent of the specified value.
ATAN2		arctangent of the specified value.
COS	S,D	cosine of the specified value.
COSH		hyperbolic cosine of the specified value.
DBLE	S,D	conversion of the value to double precision real.
DPROD		double precision real product.
EXP	S,D	exponential value of the argument.
IAND		result of a bit-by-bit logical AND on the arguments.
IEOR		result of a bit-by-bit logical exclusive OR on the arguments.
INT	I,S,D	conversion of the value to integer type.
IOR		result of a bit-by-bit logical OR on the arguments.
LOG	S,D	natural logarithm of the specified value.
LOG10		base-10 logarithm of the specified value.
MAX		maximum value of the supplied arguments.
MIN		minimum value of the supplied arguments.
MOD	I	remainder of the division.
NINT		nearest integer to the real argument.
NOT		result of a bit-by-bit logical complement on the argument.
REAL	I,S,D	conversion of the argument to real.
SIGN		absolute value of A times the sign of B.
SIN	S,D	value of the sine of the argument.
SINH		hyperbolic sine of the argument.
SQRT	S,D	square root of the argument.
TAN		tangent of the specified value.
TANH		hyperbolic tangent of the specified value.

References related to Accelerators

- ISO/IEC 1539-1:1997, Information Technology - Programming Languages - Fortran, Geneva, 1997 (Fortran 95).
- American National Standard Programming Language C, ANSI X3.159-1989 (ANSI C).
- ISO/IEC 9899:1999, Information Technology - Programming Languages - C, Geneva, 1999 (C99).
- PGI Tools Guide, The Portland Group, Release 10.0, November, 2009. Available online at http://www.pgroup.com/doc/pgitools.pdf.
- PGI Fortran Reference, The Portland Group, Release 10.0, November, 2009. Available online at http://www.pgroup.com/doc/pgifortref.pdf

Chapter 11. Using Directives

It is often useful to be able to alter the effects of certain command line options or default behavior of the compiler. Fortran directives 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 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 to tune selected routines or loops.

PGI Proprietary Fortran Directives

PGI Fortran compilers support proprietary directives that may have any of the following forms:

```
!pgi$g directive
!pgi$r directive
!pgi$l directive
!pgi$ directive
```

Note

If the input is in fixed format, the comment character must begin in column 1 and either * or C is allowed in place of !.

The scope indicator controls the scope of the directive. This indicator occurs after the $. Some directives ignore the scope indicator.

The valid scopes, shown in the previous forms of the directive, are these:

g
: (global) indicates the directive applies to the end of the source file.

r
: (routine) indicates the directive applies to the next subprogram.

l
: (loop) indicates the directive applies to the next loop, but not to any loop contained within the loop body. Loop-scoped directives are only applied to `DO` loops.

blank
: indicates that the default scope for the directive is applied.

The body of the directive may immediately follow the scope indicator. Alternatively, any number of blanks may precede the name of the directive. Any names in the body of the directive, including the directive name, may not contain embedded blanks. Blanks may surround any special characters, such as a comma or an equal sign.

The directive name, including the directive prefix, may contain upper or lower case letters, and the case is not significant. Case is significant for any variable names that appear in the body of the directive if the command line option `-Mupcase` is selected. For compatibility with other vendors' directives, the prefix `cpgi$` may be substituted with `cdir$` or `cvd$`.

PGI Proprietary Optimization Directive Summary

The following table summarizes the supported Fortran directives. The following terms are useful in understanding the table.

- Functionality is a brief summary of the way to use the directive. For a complete description, refer to Chapter 21, "*Directives Reference*," on page 285.
- Many of the directives can be preceded by `NO`. The default entry indicates the default for the directive. N/A appears if a default does not apply.
- The scope entry indicates the allowed scope indicators for each directive, with `l` for loop, `r` for routine, and `g` for global. The default scope is surrounded by parentheses.

Note

The "*" in the scope indicates this:

For routine-scoped directive
: The scope includes the code following the directive until the end of the routine.

For globally-scoped directive
: The scope includes the code following the directive until the end of the file rather than for the entire file.

Note

The name of a directive may also be prefixed with `-M`.

For example, you can use the directive `-Mbounds`, which is equivalent to the directive `bounds` and you can use `-Mopt`, which is equivalent to `opt`.

Table 11.1. Proprietary Optimization-Related Fortran Directive Summary

Directive	Functionality	Default	Scope
altcode (noaltcode)	Do/don't generate alternate code for vectorized and parallelized loops.	altcode	(l)rg
assoc (noassoc)	Do/don't perform associative transformations.	assoc	(l)rg

Directive	Functionality	Default	Scope
bounds (nobounds)	Do/don't perform array bounds checking.	nobounds	(r)g*
cncall (nocncall)	Loops are considered for parallelization, even if they contain calls to user-defined subroutines or functions, or if their loop counts do not exceed usual thresholds.	nocncall	(l)rg
concur (noconcur)	Do/don't enable auto-concurrentization of loops.	concur	(l)rg
depchk (nodepchk)	Do/don't ignore potential data dependencies.	depchk	(l)rg
eqvchk (noeqvchk)	Do/don't check EQUIVALENCE s for data dependencies.	eqvchk	(l)rg
invarif (noinvarif)	Do/don't remove invariant if constructs from loops.	invarif	(l)rg
ivdep	Ignore potential data dependencies.	ivdep	(l)rg
lstval (nolstval)	Do/don't compute last values.	lstval	(l)rg
prefetch	Control how prefetch instructions are emitted		
opt	Select optimization level.	N/A	(r)g
safe_lastval	Parallelize when loop contains a scalar used outside of loop.	not enabled	(l)
tp	Generate PGI Unified Binary code optimized for specified targets.	N/A	(r)g
unroll (nounroll)	Do/don't unroll loops.	nounroll	(l)rg
vector (novector)	Do/don't perform vectorizations.	vector	(l)rg*
vintr (novintr)	Do/don't recognize vector intrinsics.	vintr	(l)rg

Scope of Fortran Directives and Command-Line options

During compilation the effect of a directive may be to either turn an option on, or turn an option off. Directives apply to the section of code following the directive, corresponding to the specified scope, which may include the following loop, the following routine, or the rest of the program. This section presents several examples that show the effect of directives as well as their scope.

Consider the following Fortran code:

```
integer maxtime, time
parameter (n = 1000, maxtime = 10)
double precision a(n,n), b(n,n), c(n,n)
  do time = 1, maxtime
    do i = 1, n
      do j = 1, n
        c(i,j) = a(i,j) + b(i,j)
      enddo
    enddo
  enddo
end
```

When compiled with `-Mvect`, both interior loops are interchanged with the outer loop.

```
$ pgfortran -Mvect dirvect1.f
```

Directives alter this behavior either globally or on a routine or loop by loop basis. To assure that vectorization is not applied, use the `novector` directive with global scope.

```
cpgi$g novector
 integer maxtime, time
 parameter (n = 1000, maxtime = 10)
 double precision a(n,n), b(n,n), c(n,n)
   do time = 1, maxtime
     do i = 1, n
       do j = 1, n
         c(i,j) = a(i,j) + b(i,j)
       enddo
     enddo
   enddo
 end
```

In this version, the compiler disables vectorization for the entire source file. Another use of the directive scoping mechanism turns an option on or off locally, either for a specific procedure or for a specific loop:

```
 integer maxtime, time
 parameter (n = 1000, maxtime = 10)
 double precision a(n,n), b(n,n), c(n,n)
cpgi$l novector
   do time = 1, maxtime
     do i = 1, n
       do j = 1, n
         c(i,j) = a(i,j) + b(i,j)
       enddo
     enddo
   enddo
 end
```

Loop level scoping does not apply to nested loops. That is, the directive only applies to the following loop. In this example, the directive turns off vector transformations for the top-level loop. If the outer loop were a timing loop, this would be a practical use for a loop-scoped directive.

Prefetch Directives

Today's processors are so fast that it is difficult to bring data into them quickly enough to keep them busy. 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.

When vectorization is enabled using the `–Mvect` or `–Mprefetch` compiler options, or an aggregate option such as `–fast` that incorporates `–Mvect`, the PGI compilers selectively emit instructions to explicitly prefetch data into the data cache prior to first use. You can control how these prefetch instructions are emitted by using prefetch directives.

For a list of processors that support prefetch instructions refer to the PGI Release Notes.

Prefetch Directive Syntax

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.

Prefetch Directive Format Requirements

> **Note**
>
> The sentinel for prefetch directives is `c$mem`, which is distinct from the `cpgi$` sentinel used for optimization directives. Any prefetch directives that use the `cpgi$` sentinel are ignored by the PGI compilers.

- The "c" must be in column 1.
- Either * or ! is allowed in place of c.
- The scope indicators g, r and l used with the `cpgi$` sentinel are not supported.
- The directive name, including the directive prefix, may contain upper or lower case letters and is case insensitive (case is not significant).
- If the command line option `-Mupcase` is used, any variable names that appear in the body of the directive are case sensitive.

Sample Usage of Prefetch Directive

Example 11.1. Prefetch Directive Use

This example uses prefetch directives to prefetch data in a matrix multiplication inner loop where a row of one source matrix has been gathered into a contiguous vector.

```
 real*8 a(m,n), b(n,p), c(m,p), arow(n)
...
 do j = 1, p
c$mem prefetch arow(1),b(1,j)
c$mem prefetch arow(5),b(5,j)
c$mem prefetch arow(9),b(9,j)
 do k = 1, n, 4
c$mem prefetch arow(k+12),b(k+12,j)
 c(i,j) = c(i,j) + arow(k) * b(k,j)
 c(i,j) = c(i,j) + arow(k+1) * b(k+1,j)
 c(i,j) = c(i,j) + arow(k+2) * b(k+2,j)
 c(i,j) = c(i,j) + arow(k+3) * b(k+3,j)
 enddo
 enddo
```

This pattern of prefetch directives the compiler emits prefetch instructions whereby elements of `arow` and `b` are fetched into the data cache starting four iterations prior to first use. By varying the prefetch distance in this way, it is sometimes possible to reduce the effects of main memory latency and improve performance.

!DEC$ Directives

PGI Fortran compilers for Microsoft Windows support several de-facto standard Fortran directives that help with inter-language calling and importing and exporting routines to and from DLLs. These directives all take the form:

```
!DEC$ directive
```

Format Requirements

You must follow the following format requirements for the directive to be recognized in your program:

- The directive must begin in line 1 when the file is fixed format or compiled with `-Mfixed`.
- The directive prefix `!DEC$` requires a space between the prefix and the directive keyword, such as `ATTRIBUTES`.
- The ! must begin the prefix when compiling Fortran 90/95 free-form format.
- The characters C or * can be used in place of ! in either form of the prefix when compiling F77-style fixed-form format.
- The directives are completely case insensitive.

Summary Table

The following table summarizes the supported !DEC$ directives. For a complete description of each directive, refer to the section "!DEC$ Directives," on page 290 in Chapter 21, "*Directives Reference*".

Table 11.2. !DEC$ Directives Summary Table

Directive	Functionality
ALIAS	Specifies an alternative name with which to resolve a routine.
ATTRIBUTES	Lets you specify properties for data objects and procedures.
DECORATE	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. This directive has no effect if ALIAS is not specified.
DISTRIBUTE	Tells the compiler at what point within a loop to split into two loops.
IGNORE_TKR	Directs the compiler to ignore the type, kind, and/or rank (/TKR/) of specified dummy arguments in a procedure interface.

Chapter 12. Creating and Using Libraries

A library is a collection of functions or subprograms that are grouped for reference and ease of linking. This chapter discusses issues related to PGI-supplied compiler libraries. Specifically, it addresses the creation of dynamically linked libraries, known as shared objects or shared libraries, and math libraries.

> **Note**
>
> This chapter does not duplicate material related to using libraries for inlining, described in "Creating an Inline Library," on page 83 or information related to run-time library routines available to OpenMP programmers, described in "Run-time Library Routines," on page 95.

PGI provides libraries that export C interfaces by using Fortran modules. It also provides additions to the supported library functionality, specifically, NARGS, a run-time function included in DFLIB. NARGS returns the total number of command-line arguments, including the command. The result is of type INTEGER(4). For example, NARGS returns 4 for the command-line invocation of PROG1 -g -c -a.

This chapter has examples that include the following options related to creating and using libraries.

`-Bdynamic`	`-def<file>`	`-implib <file>`	`-Mmakeimplib`
`-Bstatic`	`-dynamiclib`	`-l`	`-o`
`-c`	`-fpic`	`-Mmakedll`	`-shared`

PGI Runtime Libraries on Windows

The PGI runtime libraries on Windows are available in both static and dynamically-linked (DLL) versions. The static libraries are used by default.

- You can use the dynamically-linked version of the run-time by specifying `-Bdynamic` at both compile and link time.

> **Note**
>
> C++ on Windows does not support `-Bdynamic`.

- You can explicitly specify static linking, the default, by using `-Bstatic` at compile and link time.

For details on why you might choose one type of linking over another type, refer to "Creating and Using Dynamic-Link Libraries on Windows," on page 131.

Creating and Using Static Libraries on Windows

The Microsoft Library Manager (`LIB.EXE`) is the tool that is typically used to create and manage a static library of object files on Windows. `LIB` is provided with the PGI compilers as part of the Microsoft Open Tools. Refer to *www.msdn2.com* for a complete `LIB` reference - search for `LIB.EXE`. For a list of available options, invoke `LIB` with the `/?` switch.

For compatibility with legacy makefiles, PGI provides a wrapper for `LIB` and `LINK` called `ar`. This version of `ar` is compatible with Windows and object-file formats.

PGI also provides `ranlib` as a placeholder for legacy makefile support.

ar command

The `ar` command is a legacy archive wrapper that interprets legacy `ar` command line options and translates these to `LINK/LIB` options. You can use it to create libraries of object files.

Syntax:

The syntax for the `ar` command is this:

```
ar [options] [archive] [object file].
```

Where:

- The first argument must be a command line switch, and the leading dash on the first option is optional.
- The single character options, such as `-d` and `-v`, may be combined into a single option, as `-dv`.

  Thus, `ar dv`, `ar -dv,` and `ar -d -v` all mean the same thing.
- The first non-switch argument must be the library name.
- One (and only one) of `-d`, `-r`, `-t`, or `-x` must appear on the command line.

Options

The options available for the `ar` command are these:

–c
: This switch is for compatibility; it is ignored.

–d
: Deletes the named object files from the library.

–r
: Replaces in or adds the named object files to the library.

–t
Writes a table of contents of the library to standard out.

–v
Writes a verbose file-by-file description of the making of the new library to standard out.

–x
Extracts the named files by copying them into the current directory.

ranlib command

The `ranlib` command is a wrapper that allows use of legacy scripts and makefiles that use the `ranlib` command. The command actually does nothing; it merely exists for compatibility.

Syntax:

The syntax for the `ranlib` command is this:

```
ranlib [options] [archive]
```

Options

The options available for the `ranlib` command are these:

–help
Short help information is printed out.

–V
Version information is printed out.

Creating and Using Dynamic-Link Libraries on Windows

There are several differences between static and dynamic-link libraries on Windows. Libraries of either type are used when resolving external references for linking an executable, but the process differs for each type of library. When linking with a static library, the code needed from the library is incorporated into the executable. When linking with a DLL, external references are resolved using the DLL's import library, not the DLL itself. The code in the DLL associated with the external references does not become a part of the executable. The DLL is loaded when the executable that needs it is run. For the DLL to be loaded in this manner, the DLL must be in your path.

Static libraries and DLLs also handle global data differently. Global data in static libraries is automatically accessible to other objects linked into an executable. Global data in a DLL can only be accessed from outside the DLL if the DLL exports the data and the image that uses the data imports it.

The PGI Fortran compilers support the DEC$ ATTRIBUTES extensions `DLLIMPORT` and `DLLEXPORT`:

```
cDEC$ ATTRIBUTES DLLEXPORT :: object [,object] ...
cDEC$ ATTRIBUTES DLLIMPORT :: object [,object] ...
```

Here *c* is one of C, c, !, or *. `object` is the name of the subprogram or common block that is exported or imported. Further, common block names are enclosed within slashes (/), as shown here:

```
cDEC$ ATTRIBUTES DLLIMPORT :: intfunc
!DEC$ ATTRIBUTES DLLEXPORT :: /fdata/
```

For more information on these extensions, refer to "!DEC$ Directives," on page 127.

The examples in this section further illustrate the use of these extensions.

To create a DLL in PVF, select *File* | *New* | *Project...*, then select *PGI Visual Fortran*, and create a new Dynamic Library project.

To create a DLL from the command line, use the `-Mmakedll` option.

The following switches apply to making and using DLLs with the PGI compilers:

–Bdynamic
: 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.

  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. You must use the `-Bdynamic` flag when linking an executable against a DLL built by the PGI compilers.

  Note

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

–Bstatic
: Compile for and link to the static version of the PGI runtime libraries. This flag corresponds to the `/MT` flag used by Microsoft's `cl` compilers.

  On Windows, you must use`-Bstatic` for both compiling and linking.

–Mmakedll
: Generate a dynamic-link library or DLL. Implies `-Bdynamic`.

–Mmakeimplib
: Generate an import library without generating a DLL. Use this flag when you want to generate an import library for a DLL but are not yet ready to build the DLL itself. This situation might arise, for example, when building DLLs with mutual imports, as shown in Example 12.2, "Build DLLs Containing Mutual Imports: Fortran," on page 134.

–o <file>
: Passed to the linker. Name the DLL or import library <file>.

–def <file>
: When used with `-Mmakedll`, this flag is passed to the linker and a `.def` file named <file> is generated for the DLL. The `.def` file contains the symbols exported by the DLL. Generating a `.def` file is not required when building a DLL but can be a useful debugging tool if the DLL does not contain the symbols that you expect it to contain.

  When used with `-Mmakeimplib`, this flag is passed to `lib` which requires a `.def` file to create an import library. The `.def` file can be empty if the list of symbols to export are passed to `lib` on the command line or explicitly marked as `DLLEXPORT` in the source code.

–implib <file>
Passed to the colinker. Generate an import library named <file> for the DLL. A DLL's import library is the interface used when linking an executable that depends on routines in a DLL.

To use the PGI compilers to create an executable that links to the DLL form of the runtime, use the compiler flag `–Bdynamic`. The executable built will be 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.

The following examples outline how to use `–Bdynamic`, `–Mmakedll` and `–Mmakeimplib` to build and use DLLs with the PGI compilers.

Note

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

Example 12.1. Build a DLL: Fortran

This example builds a DLL from a single source file, `object1.f`, which exports data and a subroutine using `DLLEXPORT`. The source file, `prog1.f`, uses `DLLIMPORT` to import the data and subroutine from the DLL.

`object1.f`

```
 subroutine sub1(i)
!DEC$ ATTRIBUTES DLLEXPORT :: sub1
 integer i
 common /acommon/ adata
 integer adata
!DEC$ ATTRIBUTES DLLEXPORT :: /acommon/
 print *, "sub1 adata", adata
 print *, "sub1 i ", i
 adata = i
 end
```

`prog1.f`

```
 program prog1
 common /acommon/ adata
 integer adata
 external sub1
!DEC$ ATTRIBUTES DLLIMPORT:: sub1, /acommon/
 adata = 11
 call sub1(12)
 print *, "main adata", adata
 end
```

Step 1: 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
```

Step 2: Compile the main program:

```
% pgfortran -Bdynamic -o prog1 prog1.f -defaultlib:obj1
```

The `–Bdynamic` and `–Mmakedll` switches cause the compiler to link against the PGI runtime DLLs instead of the PGI runtime static libraries. The `–Bdynamic` switch is required when linking against any PGI-compiled

DLL, such as `obj1.dll`. The -defaultlib: switch specifies that `obj1.lib`, the DLL's import library, should be used to resolve imports.

Step 3: Ensure that `obj1.dll` is in your path, then run the executable `prog1` to determine if the DLL was successfully created and linked:

```
% prog1
sub1 adata 11
sub1 i 12
main adata 12
```

Should you wish to change `obj1.dll` without changing the subroutine or function interfaces, no rebuilding of `prog1` is necessary. Just recreate `obj1.dll` and the new `obj1.dll` is loaded at runtime.

Tip

Example 12.2. Build DLLs Containing Mutual Imports: Fortran

In this example we build two DLLs when each DLL is dependent on the other, and use them to build the main program.

In the following source files, `object2.f95` makes calls to routines defined in `object3.f95`, and vice versa. This situation of mutual imports requires two steps to build each DLL.

To link the first DLL, the import library for the second DLL must be available. Usually an import library is created when a DLL is linked. In this case, however, the second DLL cannot be linked without the import library for the first DLL. When such circular imports exist, an import library for one of the DLLs must be created in a separate step without creating the DLL. The PGI drivers call the Microsoft `lib` tool to create import libraries in this situation.

Once the DLLs are built, we can use them to build the main program.

`object2.f95`

```
 subroutine func_2a
 external func_3b
!DEC$ ATTRIBUTES DLLEXPORT :: func_2a
!DEC$ ATTRIBUTES DLLIMPORT :: func_3b
 print*,"func_2a, calling a routine in obj3.dll"
 call func_3b()
 end subroutine
 subroutine func_2b
!DEC$ ATTRIBUTES DLLEXPORT :: func_2b
 print*,"func_2b"
 end subroutine
```

`object3.f95`

```
 subroutine func_3a
 external func_2b
!DEC$ ATTRIBUTES DLLEXPORT :: func_3a
!DEC$ ATTRIBUTES DLLIMPORT :: func_2b
 print*,"func_3a, calling a routine in obj2.dll"
 call func_2b()
 end subroutine
```

```
 subroutine func_3b
!DEC$ ATTRIBUTES DLLEXPORT :: func_3b
 print*,"func_3b"
 end subroutine
```

prog2.f95

```
 program prog2
 external func_2a
 external func_3a
!DEC$ ATTRIBUTES DLLIMPORT :: func_2a
!DEC$ ATTRIBUTES DLLIMPORT :: func_3a
 call func_2a()
 call func_3a()
 end program
```

Step 1: Use `-Mmakeimplib` with the PGI compilers to build an import library for the first DLL without building the DLL itself.

```
% pgfortran -Bdynamic -c object2.f95
% pgfortran -Mmakeimplib -o obj2.lib object2.obj
```

> **Tip**
>
> The `-def=<deffile>` option can also be used with `-Mmakeimplib`. Use a `.def` file when you need to export additional symbols from the DLL. A `.def` file is not needed in this example because all symbols are exported using `DLLEXPORT`.

Step 2: Use the import library, `obj2.lib`, created in Step 1, to link the second DLL.

```
% pgfortran -Bdynamic -c object3.f95
% pgfortran -Mmakedll -o obj3.dll object3.obj -defaultlib:obj2
```

Step 3: Use the import library, `obj3.lib`, created in Step 2, to link the first DLL.

```
% pgfortran -Mmakedll -o obj2.dll object2.obj -defaultlib:obj3
```

Step 4: Compile the main program and link against the import libraries for the two DLLs.

```
% pgfortran -Bdynamic prog2.f95 -o prog2 -defaultlib:obj2 -defaultlib:obj3
```

Step 5: Execute `prog2` to ensure that the DLLs were created properly:

```
% prog2
func_2a, calling a routine in obj3.dll
func_3b
func_3a, calling a routine in obj2.dll
func_2b
```

Example 12.3. Import a Fortran module from a DLL

In this example we import a Fortran module from a DLL. We use the source file `defmod.f90` to create a DLL containing a Fortran module. We then use the source file `use_mod.f90` to build a program that imports and uses the Fortran module from `defmod.f90`.

defmod.f90

```
module testm
 type a_type
 integer :: an_int
 end type a_type
```

```
 type(a_type) :: a, b
!DEC$ ATTRIBUTES DLLEXPORT :: a,b
 contains
 subroutine print_a
!DEC$ ATTRIBUTES DLLEXPORT :: print_a
 write(*,*) a%an_int
 end subroutine
 subroutine print_b
!DEC$ ATTRIBUTES DLLEXPORT :: print_b
 write(*,*) b%an_int
 end subroutine
end module
```

usemod.f90

```
use testm
 a%an_int = 1
 b%an_int = 2
 call print_a
 call print_b
end
```

Step 1: Create the DLL.

```
% pgf90 -Mmakedll -o defmod.dll defmod.f90
Creating library defmod.lib and object defmod.exp
```

Step 2: Create the `exe` and link against the import library for the imported DLL.

```
% pgf90 -Bdynamic -o usemod usemod.f90 -defaultlib:defmod.lib
```

Step 3: Run the `exe` to ensure that the module was imported from the DLL properly.

```
% usemod
1
2
```

Using LIB3F

The PGI Fortran compilers include complete support for the de facto standard LIB3F library routines on both Linux and Windows operating systems. See the PGI Fortran Reference manual for a complete list of available routines in the PGI implementation of LIB3F.

LAPACK, BLAS and FFTs

Pre-compiled versions of the public domain LAPACK and BLAS libraries are included with the PGI compilers. The LAPACK library is called `liblapack.a` or on Windows, `liblapack.lib`. The BLAS library is called `libblas.a` or on Windows, `libblas.lib`. These libraries are installed to `$PGI\<target>\lib`, where <target> is replaced with the appropriate target name (win32, win64).

To use these libraries, simply link them in using the `-l` option when linking your main program:

```
% pgfortran myprog.f -llapack -lblas
```

Highly optimized assembly-coded versions of BLAS and certain FFT routines may be available for your platform. In some cases, these are shipped with the PGI compilers. See the current release notes for the PGI compilers you are using to determine if these optimized libraries exist, where they can be downloaded (if necessary), and how to incorporate them into your installation as the default.

Chapter 13. Using Environment Variables

Environment variables allow you to set and pass information that can alter the default behavior of the PGI compilers and the executables which they generate. This chapter includes explanations of the environment variables specific to PGI compilers. Other environment variables are referenced and documented in other sections of this User's Guide or the PGI Tools Guide.

- You use OpenMP environment variables to control the behavior of OpenMP programs. For consistency related to the OpenMP environment, the details of the OpenMP-related environment variables are included in Chapter 9, "*Using OpenMP*".
- You can use environment variables to control the behavior of the PGDBG debugger or PGPROF profiler. For a description of environment variables that affect these tools, refer to the PGI Tools Guide.

Setting Environment Variables

Before we look at the environment variables that you might use with the PGI compilers and tools, let's take a look at how to set environment variables. To illustrate how to set these variables in various environments, lets look at how a user might initialize the shell environment prior to using the PGI compilers and tools.

Setting Environment Variables on Windows

When you open the PVF Command Prompt, as described in "Commands Submenu", the environment is pre-initialized to use the PGI compilers and tools.

You may want to use other environment variables, such as the OpenMP ones. This section explains how to do that.

Suppose that your home directory is `C:\tmp`. The following examples show how you might set the temporary directory to your home directory, and then verify that it is set.

```
DOS> set TMPDIR=C:\tmp
DOS> echo %TMPDIR%
C:\tmp
DOS>
```

Note

To set the environment for programs run from within PVF, whether or not they are run in the debugger, use the environment properties described in the "Debugging Property Page," on page 328.

PGI-Related Environment Variables

For easy reference, the following table provides a quick listing of some OpenMP and all PGI compiler-related environment variables. This section provides more detailed descriptions of the environment variables specific to PGI compilers and the executables they generate. For information specific to OpenMP environment variables, refer to Table 9.4, "OpenMP-related Environment Variable Summary Table," on page 100 and to the complete descriptions in "OpenMP Environment Variables"

Table 13.1. PGI-Related Environment Variable Summary

Environment Variable	Description
FLEXLM_BATCH	(Windows only) When set to 1, prevents interactive pop-ups from appearing by sending all licensing errors and warnings to standard out rather than to a pop-up window.
FORTRANOPT	Allows the user to specify that the PGI Fortran compilers user VAX I/O conventions.
LM_LICENSE_FILE	Specifies the full path of the license file that is required for running the PGI software. On Windows, `LM_LICENSE_FILE` does not need to be set.
MPSTKZ	Increases the size of the stacks used by threads executing in parallel regions. The value should be an integer <n> concatenated with `M` or `m` to specify stack sizes of n megabytes.
MP_BIND	Specifies whether to bind processes or threads executing in a parallel region to a physical processor.
MP_BLIST	When `MP_BIND` is `yes`, this variable specifically defines the thread-CPU relationship, overriding the default values.
MP_SPIN	Specifies the number of times to check a semaphore before calling _sleep().
MP_WARN	Allows you to eliminate certain default warning messages.
NCPUS	Sets the number of processes or threads used in parallel regions.
NCPUS_MAX	Limits the maximum number of processors or threads that can be used in a parallel region.
NO_STOP_MESSAGE	If used, the execution of a plain STOP statement does not produce the message FORTRAN STOP.
OMP_DYNAMIC	Currently has no effect. Enables (`TRUE`) or disables (`FALSE`) the dynamic adjustment of the number of threads. The default is FALSE.

Environment Variable	Description
OMP_MAX_ACTIVE_LEVELS	Currently has no effect. Enables (TRUE) or disables (FALSE) nested parallelism. The default is FALSE.
OMP_NUM_THREADS	Specifies the number of threads to use during execution of parallel regions. Default is 1.
OMP_SCHEDULE	Specifies the type of iteration scheduling and, optionally, the chunk size to use for *omp for* and *omp parallel for* loops that include the run-time schedule clause. The default is STATIC with chunk size = 1.
OMP_STACKSIZE	Overrides the default stack size for a newly created thread.
OMP_WAIT_POLICY	Sets the behavior of idle threads, defining whether they spin or sleep when idle. The values are ACTIVE and PASSIVE. The default is ACTIVE.
PATH	Determines which locations are searched for commands the user may type.
PGI	Specifies, at compile-time, the root directory where the PGI compilers and tools are installed.
PGI_CONTINUE	If set, when a program compiled with -Mchkfpstk is executed, the stack is automatically cleaned up and execution then continues.
PGI_OBJSUFFIX	Allows you to control the suffix on generated object files.
PGI_STACK_USAGE	(Windows only) Allows you to explicitly set stack properties for your program.
PGI_TERM	Controls the stack traceback and just-in-time debugging functionality.
PGI_TERM_DEBUG	Overrides the default behavior when PGI_TERM is set to debug.
STATIC_RANDOM_SEED	Forces the seed returned by RANDOM_SEED to be constant.
TMP	Sets the directory to use for temporary files created during execution of the PGI compilers and tools; interchangeable with TMPDIR.
TMPDIR	Sets the directory to use for temporary files created during execution of the PGI compilers and tools.

PGI Environment Variables

You use the environment variables listed in Table 13.1 to alter the default behavior of the PGI compilers and the executables which they generate. This section provides more detailed descriptions about the variables in this table that are not OpenMP environment variables.

FLEXLM_BATCH

By default, on Windows the license server creates interactive pop-up messages to issue warning and errors. You can use the environment variable FLEXLM_BATCH to prevent interactive pop-up windows. To do this, set the environment variable FLEXLM_BATCH to 1.

The following csh example prevents interactive pop-up messages for licensing warnings and errors:

```
% set FLEXLM_BATCH = 1;
```

FORTRANOPT

FORTRANOPT allows the user to adjust the behavior of the PGI Fortran compilers.

- If FORTRANOPT exists and contains the value vaxio, the record length in the open statement is in units of 4-byte words, and the $ edit descriptor only has an effect for lines beginning with a space or a plus sign (+).
- If FORTRANOPT exists and contains the value format_relaxed, an I/O item corresponding to a numerical edit descriptor (such as F, E, I, and so on) is not required to be a type implied by the descriptor.
- In a non-Windows environment, if FORTRANOPT exists and contains the value crif, a sequential formatted or list-directed record is allowed to be terminated with the character sequence \r\n (carriage return, newline). This approach is useful when reading records from a file produced on a Window's system.

The following example causes the PGI Fortran compilers to use VAX I/O conventions:

```
% setenv FORTRANOPT vaxio
```

LM_LICENSE_FILE

The LM_LICENSE_FILE variable specifies the full path of the license file that is required for running the PGI software.

Note

LM_LICENSE_FILE is not required for PVF, but you can use it.

To set the environment variable LM_LICENSE_FILE to the full path of the license key file, do this:

1. Open the System Properties dialog: *Start* | *Control Panel* | *System*.
2. Select the *Advanced* tab.
3. Click the *Environment Variables* button.
   - If LM_LICENSE_FILE is not already an environment variable, create a new system variable for it. Set its value to the full path, including the name of the license key file, license.dat.
   - If LM_LICENSE_FILE already exists as an environment variable, append the path to the license file to the variable's current value using a semi-colon to separate entries.

MPSTKZ

MPSTKZ increases the size of the stacks used by threads executing in parallel regions. You typically use this variable with programs that utilize large amounts of thread-local storage in the form of private variables or local variables in functions or subroutines called within parallel regions. The value should be an integer <n> concatenated with M or m to specify stack sizes of n megabytes.

For example, the following setting specifies a stack size of 8 megabytes.

```
% setenv MPSTKZ 8M
```

MP_BIND

You can set MP_BIND to yes or y to bind processes or threads executing in a parallel region to physical processor. Set it to no or n to disable such binding. The default is to not bind processes to processors. This variable is an execution-time environment variable interpreted by the PGI run-time support libraries. It does not affect the behavior of the PGI compilers in any way.

Note

The MP_BIND environment variable is not supported on all platforms.

```
% setenv MP_BIND y
```

MP_BLIST

MP_BLIST allows you to specifically define the thread-CPU relationship.

Note

This variable is only in effect when MP_BIND is yes.

While the MP_BIND variable binds processors or threads to a physical processor, MP_BLIST allows you to specifically define which thread is associated with which processor. The list defines the processor-thread relationship order, beginning with thread 0. This list overrides the default binding.

For example, the following setting for MP_BLIST maps CPUs 3, 2, 1 and 0 to threads 0, 1, 2 and 3 respectively.

```
% setenv MP_BLIST=3,2,1,0
```

MP_SPIN

When a thread executing in a parallel region enters a barrier, it spins on a semaphore. You can use MP_SPIN to specify the number of times it checks the semaphore before calling _sleep() . These calls cause the thread to be re-scheduled, allowing other processes to run. The default value is 10000.

```
% setenv MP_SPIN 200
```

MP_WARN

MP_WARN allows you to eliminate certain default warning messages.

By default, a warning is printed to stderr if you execute an OpenMP or auto-parallelized program with NCPUS or OMP_NUM_THREADS set to a value larger than the number of physical processors in the system.

For example, if you produce a parallelized executable `a.exe` and execute as follows on a system with only one processor, you get a warning message.

```
> set OMP_NUM_THREADS=2
> a.exe
Warning: OMP_NUM_THREADS or NCPUS (2) greater than available cpus (1)
FORTRAN STOP
```

Setting `MP_WARN` to `NO` eliminates these warning messages.

NCPUS

You can use the `NCPUS` environment variable to set the number of processes or threads used in parallel regions. The default is to use only one process or thread, which is known as serial mode.

Note

`OMP_NUM_THREADS` has the same functionality as `NCPUS`. For historical reasons, PGI supports the environment variable `NCPUS`. If both `OMP_NUM_THREADS` and `NCPUS` are set, the value of `OMP_NUM_THREADS` takes precedence.

Warning

Setting `NCPUS` to a value larger than the number of physical processors or cores in your system can cause parallel programs to run very slowly.

NCPUS_MAX

You can use the `NCPUS_MAX` environment variable to limit the maximum number of processes or threads 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(), will cause the number of processes or threads to be set at the value of `NCPUS_MAX` rather than the value specified in the function call.

NO_STOP_MESSAGE

If the `NO_STOP_MESSAGE` variable exists, the execution of a plain `STOP` statement does not produce the message FORTRAN STOP. The default behavior of the PGI Fortran compilers is to issue this message.

PATH

The `PATH` variable sets the directories that are searched for commands that the user types. When using PGI products, it is important that you set your `PATH` to include the location of the PGI products.

Within the PVF IDE, the `PATH` variable can be set using the Environment and MPI Debugging properties on the "Debugging Property Page". The PVF Command Prompt, accessible from the PVF submenus of *Start* | *All Programs* | *PGI Visual Fortran*, opens with the `PATH` variable pre-configured for use of the PGI products.

Important

If you invoke a generic Command Prompt using *Start* | *All Programs* | *Accessories* | *Command Prompt*, then the environment is not pre-configured for PGI products.

PGI

The PGI environment variable specifies the root directory where the PGI compilers and tools are installed. This variable is recognized at compile-time. If it is not set, the default value depends on your system as well as which compilers are installed:

- On Windows, the default value is C:\Program Files\PGI, where C represents the system drive. If both 32- and 64-bit compilers are installed, the 32-bit compilers are in C:\Program Files (x86)\PGI .

In most cases, if the PGI environment variable is not set, the PGI compilers and tools dynamically determine the location of this root directory based on the instance of the compiler or tool that was invoked.

PGI_CONTINUE

You set the PGI_CONTINUE variable to specify the actions to take before continuing with execution. For example, if the PGI_CONTINUE environment variable is set and then a program that is compiled with -Mchkfpstk is executed, the stack is automatically cleaned up and execution then continues. If PGI_CONTINUE is set to verbose, the stack is automatically cleaned up, a warning message is printed, and then execution continues.

Note

There is a performance penalty associated with the stack cleanup.

PGI_OBJSUFFIX

You can set the PGI_OBJSUFFIX environment variable to generate object files that have a specific suffix. For example, if you set PGI_OBJSUFFIX to .o, the object files have a suffix of .o rather than .obj.

PGI_STACK_USAGE

(Windows only) The PGI_STACK_USAGE variable allows you to explicitly set stack properties for your program. When the user compiles a program with the -Mchkstk option and sets the PGI_STACK_USAGE environment variable to any value, the program displays the stack space allocated and used after the program exits. 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 -Mchkstk option, refer to –Mchkstk.

PGI_TERM

The PGI_TERM environment variable controls the stack traceback and just-in-time debugging functionality. The runtime libraries use the value of PGI_TERM to determine what action to take when a program abnormally terminates.

The value of PGI_TERM is a comma-separated list of options. The commands for setting the environment variable follow.

- In csh:

```
% setenv PGI_TERM option[,option...]
```

- In bash, sh, zsh, or ksh:

```
$ PGI_TERM=option[,option...]
$ export PGI_TERM
```

- In the Windows Command Prompt:

```
C:\> set PGI_TERM=option[,option...]
```

Table 13.2 lists the supported values for `option`. Following the table is a complete description of each option that indicates specifically how you might apply the option.

By default, all of these options are disabled.

Table 13.2. Supported PGI_TERM Values

[no]debug	Enables/disables just-in-time debugging (debugging invoked on error)
[no]trace	Enables/disables stack traceback on error
[no]signal	Enables/disables establishment of signal handlers for common signals that cause program termination
[no]abort	Enables/disables calling the system termination routine abort()

[no]debug

This enables/disables just-in-time debugging. The default is `nodebug`.

When `PGI_TERM` is set to `debug`, the following command is invoked on error, unless you use `PGI_TERM_DEBUG` to override this default.

```
pgdbg -text -attach <pid>
```

`<pid>` is the process ID of the process being debugged.

The PGI_TERM_DEBUG environment variable may be set to override the default setting. For more information, refer to "PGI_TERM_DEBUG," on page 145.

[no]trace

This enables/disables the stack traceback. The default is `notrace`.

[no]signal

This enables/disables establishing signal handlers for the most common signals that cause program termination. The default is `nosignal`. Setting `trace` and `debug` automatically enables `signal`. Specifically setting `nosignal` allows you to override this behavior.

[no]abort

This enables/disables calling the system termination routine abort(). The default is `noabort`. When `noabort` is in effect the process terminates by calling `_exit(127)`.

A few runtime errors just print an error message and call `exit(127)`, regardless of the status of `PGI_TERM`. These are mainly errors such as specifying an invalid environment variable value where a traceback would not be useful.

The abort routine exits with the status of the exception received; for example, if the program receives an access violation abort exits with status 0xC0000005.

For more information on why to use this variable, refer to "Stack Traceback and JIT Debugging," on page 145.

PGI_TERM_DEBUG

The `PGI_TERM_DEBUG` variable may be set to override the default behavior when `PGI_TERM` is set to `debug`.

The value of `PGI_TERM_DEBUG` should be set to the command line used to invoke the program. For example:

```
gdb --quiet --pid %d
```

The first occurrence of `%d` in the `PGI_TERM_DEBUG` string is replaced by the process id. The program named in the `PGI_TERM_DEBUG` string must be found on the current `PATH` or specified with a full path name.

STATIC_RANDOM_SEED

You can use `STATIC_RANDOM_SEED` to force the seed returned by the Fortran 90/95 `RANDOM_SEED` intrinsic to be constant. The first call to `RANDOM_SEED` without arguments resets the random seed to a default value, then advances the seed by a variable amount based on time. Subsequent calls to `RANDOM_SEED` without arguments reset the random seed to the same initial value as the first call. Unless the time is exactly the same, each time a program is run a different random number sequence is generated. Setting the environment variable `STATIC_RANDOM_SEED` to `YES` forces the seed returned by `RANDOM_SEED` to be constant, thereby generating the same sequence of random numbers at each execution of the program.

TMP

You can use `TMP` to specify the directory to use for placement of any temporary files created during execution of the PGI compilers and tools. This variable is interchangeable with `TMPDIR`.

TMPDIR

You can use `TMPDIR` to specify the directory to use for placement of any temporary files created during execution of the PGI compilers and tools.

Stack Traceback and JIT Debugging

When a programming error results in a run-time error message or an application exception, a program will usually exit, perhaps with an error message. The PGI run-time library includes a mechanism to override this default action and instead print a stack tracebackor start a debugger.

The stack traceback and just-in-time debugging functionality is controlled by an environment variable, `PGI_TERM`, described in "PGI_TERM," on page 143. The run-time libraries use the value of `PGI_TERM` to determine what action to take when a program abnormally terminates.

When the PGI runtime library detects an error or catches a signal, it calls the routine `pgi_stop_here()` prior to generating a stack traceback or starting the debugger. The `pgi_stop_here()` routine is a convenient spot to set a breakpoint when debugging a program.

Chapter 14. Distributing Files - Deployment

Once you have successfully built, debugged and tuned your application, you may want to distribute it to users who need to run it on a variety of systems. This chapter addresses how to effectively distribute applications built using PGI compilers and tools. The application must be installed in such a way that it executes accurately on a system other than the one on which it was built, and which may be configured differently.

Deploying Applications on Windows

Windows programs may be linked statically or dynamically.

- A statically linked program is completely self-contained, created by linking to static versions of the PGI and Microsoft runtime libraries.
- A dynamically linked program depends on separate dynamically-linked libraries (DLLs) that must be installed on a system for the application to run on that system.

Although it may be simpler to install a statically linked executable, there are advantages to using the DLL versions of the runtime, including these:

- Executable binary file size is smaller.
- Multiple processes can use DLLs at once, saving system resources.
- New versions of the runtime can be installed and used by the application without rebuilding the application.

Dynamically-linked Windows programs built with PGI compilers depend on dynamic run-time library files (DLLs). These DLLs must be distributed with such programs to enable them to execute on systems where the PGI compilers are not installed. These redistributable libraries include both PGI runtime libraries and Microsoft runtime libraries.

PGI Redistributables

PGI redistributable directories contain all of the PGI Linux runtime library shared object files or Windows dynamically-linked libraries that can be re-distributed by PGI 10.0 licensees under the terms of the PGI End-user License Agreement (EULA).

Microsoft Redistributables

The PGI products on Windows include Microsoft Open Tools. The Microsoft Open Tools directory contains a subdirectory named `redist`. PGI licensees may redistribute the files contained in this directory in accordance with the terms of the PGI End-User License Agreement.

Microsoft supplies installation packages, `vcredist_x86.exe` and `vcredist_x64.exe`, containing these runtime files. These files are available in the `redist` directory.

Code Generation and Processor Architecture

The PGI compilers can generate much more efficient code if they know the specific x86 processor architecture on which the program will run. When preparing to deploy your application, you should determine whether you want the application to run on the widest possible set of x86 processors, or if you want to restrict the application to run on a specific processor or set of processors. The restricted approach allows you to optimize performance for that set of processors.

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, all of which can have a profound effect on the performance of your application.

Processor-specific code generation is controlled by the `-tp` option, described in "–tp <target> [,target...]," on page 206. When an application is compiled without any `-tp` options, the compiler generates code for the type of processor on which the compiler is run.

Generating Generic x86 Code

To generate generic x86 code, use one of the following forms of the `-tp` option on your command line:

```
-tp px ! generate code for any x86 cpu type
```

```
-tp p6 ! generate code for Pentium 2 or greater
```

While both of these examples are good choices for portable execution, most users have Pentium 2 or greater CPUs.

Generating Code for a Specific Processor

You can use the `-tp` option to request that the compiler generate code optimized for a specific processor. The PGI Release Notes contains a list of supported processors.

Generating One Executable for Multiple Types of Processors

PGI unified binaries provide a low-overhead method for a single program to run well on a number of hardware platforms.

All 64-bit PGI compilers for Windows can produce PGI Unified Binary programs that contain code streams fully optimized and supported for both AMD64 and Intel EM64T processors using the `-tp` target option. You specify this option using PVF's Fortran | Target Processors property page. For more information on this property page, refer to "Fortran | Target Processors," on page 344.

The compilers generate and combine multiple binary code streams into one executable, where each stream is optimized for a specific platform. At runtime, this one executable senses the environment and dynamically selects the appropriate code stream.

Executable size is automatically controlled via unified binary culling. Only those functions and subroutines where the target affects the generated code have unique binary images, resulting in a code-size savings of 10-90% compared to generating full copies of code for each target.

Programs can use PGI Unified Binary technology even if all of the object files and libraries are not compiled as unified binaries. Like any other object file, you can use PGI Unified Binary object files to create programs or libraries. No special start up code is needed; support is linked in from the PGI libraries.

The `-Mpfi` option disables generation of PGI Unified Binary object files. Instead, the default target auto-detect rules for the host are used to select the target processor.

PGI Unified Binary Command-line Switches

The PGI Unified Binary command-line switch is an extension of the target processor switch, `-tp`, which may be applied to individual files during compilation using the PVF property pages described in Chapter 23, "*PVF Properties*," on page 313.

The target processor switch, `-tp`, accepts a comma-separated list of 64-bit targets and generates code optimized for each listed target.

The following example generates optimized code for three targets:

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

To use multiple `-tp` options within a PVF project, specify the comma-separated `-tp` list on both the Fortran | Command Line and the Linker | Command Line property pages, described in Chapter 23, "*PVF Properties*," on page 313.

A special target switch, `-tp x64`, is the same as `-tp k8-64, p7-64s`.

PGI Unified Binary Directives

PGI Unified binary directives may be applied to functions, subroutines, or whole files. The directives and pragmas cause the compiler to generate PGI Unified Binary code optimized for one or more targets. No special command line options are needed for these pragmas and directives to take effect.

The syntax of the Fortran directive is this:

```
pgi$[g|r| ] pgi tp [target]...
```

where the scope is g (global), r (routine) or blank. The default is r, routine.

For example, the following syntax indicates that the whole file, represented by g, should be optimized for both k8_64 and p7_64.

```
pgi$g pgi tp k8_64 p7_64
```

Chapter 15. Inter-language Calling

This chapter describes inter-language calling conventions for C, C++, and Fortran programs using the PGI compilers. The following sections describe how to call a Fortran function or subroutine from a C or C++ program and how to call a C or C++ function from a Fortran program. For information on calling assembly language programs, refer to Chapter 22, "*Run-time Environment*".

This chapter provides examples that use the following options related to inter-language calling. For more information on these options, refer to Chapter 18, "*Command-Line Options Reference*," on page 173.

`-c` `-Mnomain` `-Miface` `-Mupcase`

Overview of Calling Conventions

This chapter includes information on the following topics:

- Functions and subroutines in Fortran, C, and C++
- Naming and case conversion conventions
- Compatible data types
- Argument passing and special return values
- Arrays and indexes
- Win32 calling conventions

The sections "Inter-language Calling Considerations," on page 152 through describe how to perform inter-language calling using the Win64 convention. Default Fortran calling conventions for Win32 differ, although Win32 programs compiled using the `-Miface=unix` Fortran command-line option use the Win64 convention rather than the default Win32 conventions. All information in those sections pertaining to compatibility of arguments applies to Win32 as well. For details on the symbol name and argument passing conventions used on Win32 platforms, refer to "Win32 Calling Conventions," on page 160.

The concepts in this chapter apply equally to using inter-language calling in PVF. While all of the examples given are shown as being compiled at the command line, they can also be used within PVF. The primary difference for you to note is this: Visual Studio projects are limited to a single language. To mix languages, create a multi-project solution.

Tip

For inter-language examples that are specific to PVF, look in the directory:

```
$(VSInstallDir)\PGI Visual Fortran\Samples\interlanguage\
```

Inter-language Calling Considerations

In general, when argument data types and function return values agree, you can call a C or C++ function from Fortran as well as call a Fortran function from C or C++. When data types for arguments do not agree, you may need to develop custom mechanisms to handle them. For example, the Fortran `COMPLEX` type has a matching type in C99 but does not have a matching type in C90; however, it is still possible to provide inter-language calls but there are no general calling conventions for such cases.

Note

- If a C++ function contains objects with constructors and destructors, calling such a function from Fortran is not possible unless the initialization in the main program is performed from a C++ program in which constructors and destructors are properly initialized.
- C++ member functions cannot be declared `extern`, since their names will always be mangled. Therefore, C++ member functions cannot be called from C or Fortran.

Functions and Subroutines

Fortran, C, and C++ define functions and subroutines differently.

For a Fortran program calling a C or C++ function, observe the following return value convention:

- When a C or C++ function returns a value, call it from Fortran as a function.
- When a C or C++ function does not return a value, call it as a subroutine.

For a C/C++ program calling a Fortran function, the call should return a similar type. Table 15.1, "Fortran and C/C++ Data Type Compatibility," on page 153 lists compatible types. If the call is to a Fortran subroutine, a Fortran `CHARACTER` function, or a Fortran `COMPLEX` function, call it from C/C++ as a function that returns void. The exception to this convention is when a Fortran subroutine has alternate returns; call such a subroutine from C/C++ as a function returning `int` whose value is the value of the integer expression specified in the alternate `RETURN` statement.

Upper and Lower Case Conventions, Underscores

By default on Linux, Win64, and OSX systems, all Fortran symbol names are converted to lower case. C and C++ are case sensitive, so upper-case function names stay upper-case. When you use inter-language calling, you can either name your C/C++ functions with lower-case names, or invoke the Fortran compiler command with the option `-Mupcase`, in which case it will not convert symbol names to lower-case.

When programs are compiled using one of the PGI Fortran compilers on Linux, Win64, and OSX systems, an underscore is appended to Fortran global names (names of functions, subroutines and common blocks). This mechanism distinguishes Fortran name space from C/C++ name space. Use these naming conventions:

- If you call a C/C++ function from Fortran, you should rename the C/C++ function by appending an underscore.
- If you call a Fortran function from C/C++, you should append an underscore to the Fortran function name in the calling program.

Compatible Data Types

Table 15.1 shows compatible data types between Fortran and C/C++. Table 15.2, "Fortran and C/C++ Representation of the COMPLEX Type," on page 153 shows how the Fortran COMPLEX type may be represented in C/C++.

Tip

If you can make your function/subroutine parameters as well as your return values match types, you should be able to use inter-language calling.

Table 15.1. Fortran and C/C++ Data Type Compatibility

Fortran Type (lower case)	C/C++ Type	Size (bytes)
character x	char x	1
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	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	int x	4
logical*8	long x	8

Table 15.2. 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

Fortran Type (lower case)	C/C++ Type	Size (bytes)
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.

Fortran Named 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, here is a Fortran common block:

```
INTEGER I
COMPLEX C
DOUBLE COMPLEX CD
DOUBLE PRECISION D
COMMON /COM/ i, c, cd, d
```

This Fortran Common Block is represented in C with the following equivalent:

```
extern struct {
 int i;
 struct {float real, imag;} c;
 struct {double real, imag;} cd;
 double d;
} com_;
```

This same Fortran Common Block is represented in C++ with the following equivalent:

```
extern "C" struct {
 int i;
 struct {float real, imag;} c;
 struct {double real, imag;} cd;
 double d;
} com_;
```

Tip

For global or external data sharing, `extern "C"` is not required.

Argument Passing and Return Values

In Fortran, arguments are passed by reference, that is, the address of the argument is passed, rather than the argument itself. In C/C++, arguments are passed by value, except for strings and arrays, which are passed by reference. Due to the flexibility provided in C/C++, you can work around these differences. Solving the parameter passing differences generally involves intelligent use of the `&` and `*` operators in argument passing when C/C++ calls Fortran and in argument declarations when Fortran calls C/C++.

For strings declared in Fortran as type CHARACTER, an argument representing the length of the string is also passed to a calling function.

Passing by Value (%VAL)

When passing parameters from a Fortran subprogram to a C/C++ function, it is possible to pass by value using the %VAL function. If you enclose a Fortran parameter with %VAL(), the parameter is passed by value. For example, the following call passes the integer i and the logical bvar by value.

```
integer*1 i
logical*1 bvar
call cvalue (%VAL(i), %VAL(bvar))
```

Character Return Values

"Functions and Subroutines," on page 152 describes the general rules for return values for C/C++ and Fortran inter-language calling. There is a special return value to consider. When a Fortran function returns a character, two arguments need to be added at the beginning of the C/C++ calling function's argument list:

- The address of the return character or characters
- The length of the return character

Example 15.1, "Character Return Parameters" illustrates the extra parameters, tmp and 10, supplied by the caller:

Example 15.1. Character Return Parameters

```
! Fortran function returns a character
CHARACTER*(*) FUNCTION CHF(C1,I)
   CHARACTER*(*) C1
   INTEGER I
END
```

```
/* C declaration of Fortran function */
extern void chf_();
char tmp[10];
char c1[9];
int i;
chf_(tmp, 10, c1, &i, 9);
```

If the 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 must still be supplied, but is not used.

Note

The value of the character function is not automatically NULL-terminated.

Complex Return Values

When a Fortran function returns a complex value, an argument needs to be added at the beginning of the C/C++ calling function's argument list; this argument is the address of the complex return value. Example 15.2, "COMPLEX Return Values" illustrates the extra parameter, cplx, supplied by the caller.

Example 15.2. COMPLEX Return Values

```
COMPLEX FUNCTION CF(C, I)
   INTEGER I
 . . .
END
```

```
extern void cf_();
typedef struct {float real, imag;} cplx;
cplx c1;
int i;
cf_(&c1, &i);
```

Array Indices

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. If you adjust your array comparisons so that a Fortran second element is compared to a C/C++ first element, and adjust similarly for other elements, you should not have problems working with this difference. If this is not satisfactory, you can declare your Fortran arrays 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++ uses 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. For arrays other than single dimensional arrays, and square two-dimensional arrays, inter-language function mixing is not recommended.

Examples

This section contains examples that illustrate inter-language calling.

Example - Fortran Calling C

> **Note**
>
> There are other solutions to calling C from Fortran than the one presented in this section. For example, you can use the `iso_c_binding` intrinsic module which PGI does support. For more information on this module and for examples of how to use it, search the web using the keyword iso_c_binding.

Example 15.4, "C function f2c_func_" shows a C function that is called by the Fortran main program shown in Example 15.3, "Fortran Main Program f2c_main.f". Notice that each argument is defined as a pointer, since Fortran passes by reference. Also notice that the C function name uses all lower-case and a trailing "_".

Example 15.3. Fortran Main Program f2c_main.f

```
logical*1 bool1
 character letter1
 integer*4 numint1, numint2
 real numfloat1
 double precision numdoub1
 integer*2 numshor1
```

```
external f2c_func

call f2c_func_(bool1, letter1, numint1, numint2, numfloat1, numdoub1, numshor1)

write( *, "(L2, A2, I5, I5, F6.1, F6.1, I5)")
+ bool1, letter1, numint1, numint2, numfloat1,numdoub1, numshor1

end
```

Example 15.4. C function f2c_func_

```
#define TRUE 0xff
#define FALSE 0
void f2c_func_( bool1, letter1, numint1, numint2, numfloat1,\
 numdoub1, numshor1, len_letter1)
 char *bool1, *letter1;
 int *numint1, *numint2;
 float *numfloat1;
 double *numdoub1;
 short *numshor1;
 int len_letter1;
{
 *bool1 = TRUE;  *letter1 = 'v';
 *numint1 = 11;  *numint2 = -44;
 *numfloat1 = 39.6 ;
 *numdoub1 = 39.2;
 *numshor1 = 981;
}
```

Compile and execute the program `f2c_main.f` with the call to `f2c_func_` using the following command lines:

```
$ pgcc -c f2c_func.c
$ pgfortran f2c_func.o f2c_main.f
```

Executing the `f2c_main.exe` file should produce the following output:

```
T v 11 -44 39.6 39.2 981
```

Example - C Calling Fortran

Example 15.5, "C Main Program c2f_main.c" shows a C main program that calls the Fortran subroutine shown in Example 15.6, "Fortran Subroutine c2f_sub.f". Notice that each call uses the & operator to pass by reference. Also notice that the call to the Fortran subroutine uses all lower-case and a trailing "_".

Example 15.5. C Main Program c2f_main.c

```
void main () {
 char bool1, letter1;
 int numint1, numint2;
 float numfloat1;
 double numdoub1;
 short numshor1;
 extern void c2f_func_();
 c2f_sub_(&bool1,&letter1,&numint1,&numint2,&numfloat1,&numdoub1,&numshor1, 1);
 printf(" %s %c %d %d %3.1f %.0f %d\n",
 bool1?"TRUE":"FALSE", letter1, numint1, numint2,
 numfloat1, numdoub1, numshor1);
}
```

Example 15.6. Fortran Subroutine c2f_sub.f

```
subroutine c2f_func ( bool1, letter1, numint1, numint2,
+ numfloat1, numdoub1, numshor1)
   logical*1 bool1
   character letter1
   integer numint1, numint2
   double precision numdoub1
   real numfloat1
   integer*2 numshor1

   bool1 = .true.
   letter1 = "v"
    numint1 = 11
    numint2 = -44
    numdoub1 = 902
    numfloat1 = 39.6
    numshor1 = 299
    return
end
```

To compile this Fortran subroutine and C program, use the following commands:

```
$ pgcc -c c2f_main.c
$ pgfortran -Mnomain c2f_main.o c2_sub.f
```

Executing the resulting `c2fmain.exe` file should produce the following output:

```
TRUE v 11 -44 39.6 902 299
```

Example - Fortran Calling C++

The Fortran main program shown in Example 15.7, "Fortran Main Program f2cp_main.f calling a C++ function" calls the C++ function shown in Example 15.8, "C++ function f2cp_func.C".

Notice:

- Each argument is defined as a pointer in the C++ function, since Fortran passes by reference.
- The C++ function name uses all lower-case and a trailing "_":

Example 15.7. Fortran Main Program f2cp_main.f calling a C++ function

```
  logical*1 bool1
   character letter1
   integer*4 numint1, numint2
   real numfloat1
   double precision numdoub1
   integer*2 numshor1
 external f2cpfunc
 call f2cp_func (bool1, letter1, numint1,
 + numint2, numfloat1, numdoub1, numshor1)
   write( *, "(L2, A2, I5, I5, F6.1, F6.1, I5)")
   + bool1, letter1, numint1, numint2, numfloat1,
   + numdoub1, numshor1
 end
```

Example 15.8. C++ function f2cp_func.C

```
#define TRUE 0xff
#define FALSE 0
extern "C"
{
extern void f2cp_func_ (
 char *bool1, *letter1,
 int *numint1, *numint2,
 float *numfloat1,
 double *numdoub1,
 short *numshort1,
 int len_letter1)
{
 *bool1 = TRUE;     *letter1 = 'v';
 *numint1 = 11;     *numint2 = -44;
 *numfloat1 = 39.6; *numdoub1 = 39.2;   *numshort1 = 981;
}
}
```

Assuming the Fortran program is in a file `fmain.f`, and the C++ function is in a file `cpfunc.C`, create an executable, using the following command lines:

```
$ pgcpp -c f2cp_func.C
$ pgfortran f2cp_func.o f2cp_main.f -pgcpplibs
```

Executing the `fmain.exe` file should produce the following output:

```
T v 11 -44 39.6 39.2 981
```

Example - C++ Calling Fortran

Example 15.10, "Fortran Subroutine cp2f_func.f" shows a Fortran subroutine called by the C++ main program shown in Example 15.9, "C++ main program cp2f_main.C". Notice that each call uses the & operator to pass by reference. Also notice that the call to the Fortran subroutine uses all lower-case and a trailing "_":

Example 15.9. C++ main program cp2f_main.C

```
#include <iostream>
extern "C" { extern void cp2f_func_(char *,char *,int *,int *,
 float *,double *,short *); }
main ()
{
 char bool1, letter1;
 int numint1, numint2;
 float numfloat1;
 double numdoub1;
 short numshor1;

 cp2f_func(&bool1,&letter1,&numint1,&numint2,&numfloat1, &numdoub1,&numshor1);
 cout << " bool1 = ";
 bool1?cout << "TRUE ":cout << "FALSE "; cout <<endl;
 cout << " letter1 = " << letter1 <<endl;
 cout << " numint1 = " << numint1 <<endl;
 cout << " numint2 = " << numint2 <<endl;
```

```
 cout << " numfloat1 = " << numfloat1 <<endl;
 cout << " numdoub1 = " << numdoub1 <<endl;
 cout << " numshor1 = " << numshor1 <<endl;
}
```

Example 15.10. Fortran Subroutine cp2f_func.f

```
 subroutine cp2f_func ( bool1, letter1, numint1,
 + numint2, numfloat1, numdoub1, numshor1)
 logical*1 bool1
 character letter1
 integer numint1, numint2
 double precision numdoub1
 real numfloat1
 integer*2 numshor1
 bool1 = .true. ; letter1 = "v"
 numint1 = 11 ; numint2 = -44
 numdoub1 = 902 ; numfloat1 = 39.6 ; numshor1 = 299
 return
 end
```

To compile this Fortran subroutine and C++ program, use the following command lines:

```
$ pgfortran -c cp2f_func.f
$ pgcpp cp2f_func.o cp2f_main.C -pgf90libs
```

Executing this C++ main should produce the following output:

```
bool1 = TRUE
letter1 = v
numint1 = 11
numint2 = -44
numfloat1 = 39.6
numdoub1 = 902
numshor1 = 299
```

Note that you must explicitly link in the PGFORTRAN runtime support libraries when linking pgfortran-compiled program units into C or C++ main programs. When linking pgf77-compiled program units into C or C++ main programs, you need only link in `-lpgftnrtl`.

Win32 Calling Conventions

A calling convention is a set of conventions that describe the manner in which a particular routine is executed. A routine's calling conventions specify where parameters and function results are passed. For a stack-based routine, the calling conventions determine the structure of the routine's stack frame.

The calling convention for C/C++ is identical between most compilers on Win32 and Win64. However, Fortran calling conventions vary widely between legacy Win32 Fortran compilers and Win64 Fortran compilers.

Win32 Fortran Calling Conventions

Four styles of calling conventions are supported using the PGI Fortran compilers for Win32: Default, C, STDCALL, and UNIX.

- **Default** - Used in the absence of compilation flags or directives to alter the default.

- **C or STDCALL** - Used if an appropriate compiler directive is placed in a program unit containing the call. The C and STDCALL conventions are typically used to call routines coded in C or assembly language that depend on these conventions.
- **UNIX** - Used in any Fortran program unit compiled using the `-Miface=unix` (or `-Munix`) compilation flag.

The following table outlines each of these calling conventions.

Table 15.3. Calling Conventions Supported by the PGI Fortran Compilers

Convention	Default	STDCALL	C	UNIX
Case of symbol name	Upper	Lower	Lower	Lower
Leading underscore	Yes	Yes	Yes	Yes
Trailing underscore	No	No	No	Yes
Argument byte count added	Yes	Yes	No	No
Arguments passed by reference	Yes	No*	No*	Yes
Character argument length passed	After each char argument	No	No	End of argument list
First character of character string and passed by value	No	Yes	Yes	No
varargs support	No	No	Yes	Yes
Caller cleans stack	No	No	Yes	Yes

* Except arrays, which are always passed by reference even in the STDCALL and C conventions

Note

While it is compatible with the Fortran implementations of Microsoft and several other vendors, the C calling convention supported by the PGI Fortran compilers for Windows is not strictly compatible with the C calling convention used by most C/C++ compilers. In particular, symbol names produced by PGI Fortran compilers using the C convention are all lower case. The standard C convention is to preserve mixed-case symbol names. You can cause any of the PGI Fortran compilers to preserve mixed-case symbol names using the `-Mupcase` option, but be aware that this could have other ramifications on your program.

Symbol Name Construction and Calling Example

This section presents an example of the rules outlined in Table 15.3, "Calling Conventions Supported by the PGI Fortran Compilers," on page 161. In the pseudocode shown in the following examples, `%addr` refers to the address of a data item while `%val` refers to the value of that data item. Subroutine and function names are converted into symbol names according to the rules outlined in Table 15.3.

Consider the following subroutine call, where a is a double precision scalar, b is a real vector of size n, and n is an integer:

```
call work ( 'ERR', a, b, n)
```

- **Default** - The symbol name for the subroutine is constructed by pre-pending an underscore, converting to all upper case, and appending an @ sign followed by an integer indicating the total number of bytes occupied by the argument list. Byte counts for character arguments appear immediately following the corresponding argument in the argument list.

  The following example is pseudocode for the preceding subroutine call using Default conventions:

  ```
  call _WORK@20 (%addr('ERR'), 3, %addr(a), %addr(b), %addr(n))
  ```

- **STDCALL** - The symbol name for the subroutine is constructed by pre-pending an underscore, converting to all lower case, and appending an @ sign followed by an integer indicating the total number of bytes occupied by the argument list. Character strings are truncated to the first character in the string, which is passed by value as the first byte in a 4-byte word. The following is an example of the pseudocode for the work subroutine call using STDCALL conventions:

  ```
  call _work@20 (%val('E'), %val(a), %addr(b), %val(n))
  ```

  Notice in this case that there are still 20 bytes in the argument list. However, rather than five 4-byte quantities as in the Default convention, there are three 4-byte quantities and one 8-byte quantity (the double precision value of a).

- **C** - The symbol name for the subroutine is constructed by pre-pending an underscore and converting to all lower case. Character strings are truncated to the first character in the string, which is passed by value as the first byte in a 4-byte word. The following is an example of the pseudocode for the work subroutine call using C conventions:

  ```
  call _work (%val('E'), %val(a), %addr(b), %val(n))
  ```

- **UNIX** - The symbol name for the subroutine is constructed by pre-pending an underscore, converting to all lower case, and appending an underscore. Byte counts for character strings appear in sequence following the last argument in the argument list. The following is an example of the pseudocode for the work subroutine call using UNIX conventions:

  ```
  call _work_ (%addr('ERR'), %addr(a), %addr(b), %addr(n),3)
  ```

Using the Default Calling Convention

The Default calling convention is used if no directives are inserted to modify calling conventions and if neither the `-Miface=unix` (or `-Munix`) compilation flag is used. Refer to "Symbol Name Construction and Calling Example," on page 161 for a complete description of the Default calling convention.

Using the STDCALL Calling Convention

Using the STDCALL calling convention requires the insertion of a compiler directive into the declarations section of any Fortran program unit which calls the STDCALL program unit. This directive has no effect when either the `-Miface=unix` (or `-Munix`) compilation flag is used, meaning you cannot mix UNIX-style argument passing and STDCALL calling conventions within the same file.

In the following example syntax for the directive, `work` is the name of the subroutine to be called using STDCALL conventions:

```
!DEC$ ATTRIBUTES STDCALL :: work
```

You can list more than one subroutine, separating them by commas. Refer to "Symbol Name Construction and Calling Example," on page 161 for a complete description of the implementation of STDCALL.

Note

- The directive prefix !DEC$ requires a space between the prefix and the directive keyword ATTRIBUTES.
- The ! must begin the prefix when compiling using Fortran 90 freeform format.
- The characters C or * can be used in place of ! in either form of the prefix when compiling used fixed-form format.
- The directives are completely case insensitive.

Using the C Calling Convention

Using the C calling convention requires the insertion of a compiler directive into the declarations section of any Fortran program unit which calls the C program unit. This directive has no effect when the `-Miface=unix` (or `-Munix`) compilation flag is used, meaning you cannot mix UNIX-style argument passing and C calling conventions within the same file.

Syntax for the directive is as follows:

```
!DEC$ ATTRIBUTES C :: work
```

Where `work` is the name of the subroutine to be called using C conventions. More than one subroutine may be listed, separated by commas. Refer to "Symbol Name Construction and Calling Example," on page 161 for a complete description of the implementation of the C calling convention.

Using the UNIX Calling Convention

Using the UNIX calling convention is straightforward. Any program unit compiled using `-Miface=unix` or the `-Munix` compilation flag uses the UNIX convention.

Using the CREF Calling Convention

Using the CREF calling convention is straightforward. Any program unit compiled using `-Miface=cref` compilation flag uses the CREF convention.

Chapter 16. Programming Considerations for 64-Bit Environments

You can use the PGI Fortran compilers on 64-bit Windows operating systems to create programs that use 64-bit memory addresses. However, there are limitations to how this capability can be applied. The object file format used on Windows limits the total cumulative size of code plus static data to 2GB. This limit includes the code and statically declared data in the program and in system and user object libraries. Dynamically allocated data objects can be larger than 2GB. This chapter describes the specifics of how to use the PGI compilers to make use of 64-bit memory addressing.

The 64-bit Windowsenvironment maintains 32-bit compatibility, which means that 32-bit applications can be developed and executed on the corresponding 64-bit operating system.

Note

The 64-bit PGI compilers are 64-bit applications which cannot run on anything but 64-bit CPUs running 64-bit Operating Systems.

This chapter describes how to use the following options related to 64-bit programming.

`-i8`	`-tp`

Data Types in the 64-Bit Environment

The size of some data types can be different in a 64-bit environment. This section describes the major differences. Refer to Chapter 17, "*Fortran Data Types*" for detailed information.

Fortran Data Types

In Fortran, the default size of the INTEGER type is 4 bytes. The `-i8` compiler option may be used to make the default size of all INTEGER data in the program 8 bytes.

Large Dynamically Allocated Data

Dynamically allocated data objects in programs compiled by the 64-bit PGI compilers can be larger than 2GB. No special compiler options are required to enable this functionality. The size of the allocation is only limited by the system.

Compiler Options for 64-bit Programming

The usual switches that apply to 64-bit programmers seeking to increase the data range of their applications are in the following table.

Table 16.1. 64-bit Compiler Options

Option	Purpose	Considerations
–Mlargeaddressaware	[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. This is not applicable if the object file is placed in a DLL. Further, if an object file is compiled with this option, it must also be used when linking.
–Mlarge_arrays	Perform all array-location-to-address calculations using 64-bit integer arithmetic.	Slightly slower execution. Win64 does not support –Mlarge_arrays for static objects larger than 2GB.
–i8	All INTEGER functions, data, and constants not explicitly declared INTEGER*4 are assumed to be INTEGER*8.	Users should take care to explicitly declare INTEGER functions as INTEGER*4.

The following table summarizes the limits of these programming models:

Table 16.2. Effects of Options on Memory and Array Sizes

Combined Compiler Options	Addr. Math		Max Size Gbytes			Comments
	A	I	AS	DS	TS	
`-tp k8-32 or -tp p7`	32	32	2	2	2	32-bit linux86 programs
`-tp k8-64 or -tp p7-64`	64	32	2	2	2	64-bit addr

Column Legend	
A	Address Type - size in bits of data used for address calculations, 32-bit or 64-bit.
I	Index Arithmetic - bit-size of data used to index into arrays and other aggregate data structures. If 32-bit, total range of any single data object is limited to 2GB.

AS	Maximum Array Size - the maximum size in gigabytes of any single data object.
DS	*Maximum Data Size* - max size in gigabytes combined of all data objects in .bss
TS	Maximum Total Size - max size in gigabytes, in aggregate, of all executable code and data objects in a running program.

Practical Limitations of Large Array Programming

The 64-bit addressing capability of the Linux86-64 and Win64 environments can cause unexpected issues when data sizes are enlarged significantly. The following table describes the most common occurrences of practical limitations of large array programming.

Table 16.3. 64-Bit Limitations

array initialization	Initializing a large array with a data statement may result in very large assembly and object files, where a line of assembler source is required for each element in the initialized array. Compilation and linking can be very time consuming as well. To avoid this issue, consider initializing large arrays in a loop at runtime rather than in a data statement.
stack space	Stack space can be a problem for data that is stack-based. In Win64, stack space can be increased by using this link-time switch, where N is the desired stack size:`-Wl,-stack:N`
page swapping	If your executable is much larger than the physical size of memory, page swapping can cause it to run dramatically slower; it may even fail. This is not a compiler problem. Try smaller data sets to determine whether or not a problem is due to page thrashing.

Large Array and Small Memory Model in Fortran

The following example uses large, dynamically-allocated arrays. The code is divided into a main and subroutine so you could put the subroutine into a shared library. Dynamic allocation of large arrays saves space in the size of executable and saves time initializing data. Further, the routines can be compiled with 32-bit compilers, by just decreasing the parameter size.

Example 16.1. Large Array and Small Memory Model in Fortran

```
% cat mat_allo.f90
program mat_allo
 integer i, j
 integer size, m, n
 parameter (size=16000)
 parameter (m=size,n=size)
 double precision, allocatable::a(:,:),b(:,:),c(:,:)
 allocate(a(m,n), b(m,n), c(m,n))
 do i = 100, m, 1
 do j = 100, n, 1
 a(i,j) = 10000.0D0 * dble(i) + dble(j)
 b(i,j) = 20000.0D0 * dble(i) + dble(j)
 enddo
 enddo
 call mat_add(a,b,c,m,n)
```

```
 print *, "M =",m,",N =",n
 print *, "c(M,N) = ", c(m,n)
end
subroutine mat_add(a,b,c,m,n)
 integer m, n, i, j
 double precision a(m,n),b(m,n),c(m,n)
!$omp do
 do i = 1, m
 do j = 1, n
 c(i,j) = a(i,j) + b(i,j)
 enddo
 enddo
 return
end
% pgfortran –o mat_allo mat_allo.f90 –i8 –Mlarge_arrays -mp -fast
```

Part II. Reference Information

In Part I you learned how to use the PGI compilers as well as why certain options or tasks are useful in enhancing the effectiveness and efficiency of the PGI compilers and tools. You may now be ready to learn more about specific areas or specific topics. The chapters in this part of the guide provide more data and facts about the topics that you have already learned about, including information about:

- Data types, as described in Chapter 17, "Fortran Data Types" on page 169.
- Detailed information about each of the command-line options, as described in Chapter 18, "Command-Line Options Reference" on page 173.
- Details about the OpenMP directives, as described in Chapter 19, "OpenMP Reference Information" on page 241.
- PGI Accelerator directives, runtime routines, and environment variables, as described in Chapter 20, "PGI Accelerator Compilers Reference" on page 265.
- Details about PGI directives, as described in Chapter 21 "Directives Reference" on page 285.
- Information about run-time environments, as described in Chapter 22, "Run-time Environment" on page 295.
- PVF Property Pages, as described in Chapter 23, "PVF Properties" on page 313.
- PVF Build Macros, as described in Chapter 24, "PVF Build Macros" on page 359.
- Fortran module and library interfaces that PVF uses to support the Win32 API and Unix/Linux portability libraries, as described in Chapter 25, "Fortran Module/Library Interfaces for Windows" on page 363.
- Error messages, as described in Chapter 26, "Messages" on page 393.

Chapter 17. Fortran Data Types

This chapter describes the scalar and aggregate data types recognized by the PGI Fortran 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 xxvii. 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 17.2, "Real Data Type Ranges," on page 170 shows the range and approximate precision for Fortran real data types. Table 17.3, "Scalar Type Alignment," on page 170 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 17.1. 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
LOGICAL*4	32-bit value	true or false
LOGICAL*8	64-bit value	true or false

Fortran Data Type	Format	Range
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 17.2. 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 17.3. 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
INTEGER*8	8-byte
REAL*4	4-byte
REAL*8	8-byte

This Type...	...Is aligned on this size boundary
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’
```

Chapter 18. 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*," on page 51, 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 18.1. 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.
–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.
–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.
--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.
–module <moduledir>	Save/search for module files in directory <moduledir>.

Option	Description
–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.
–pgf77libs	Append PGF77 runtime libraries to the link line.
–pgf90libs	Append PGF90/PGF95/PGFORTRAN runtime libraries to the link line.
–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	Stops after the compiling phase and saves the assembly–language code in filename.s.
–show	Display driver's configuration parameters after startup.
–silent	Do not print warning messages.
–time	Print execution times for the various compilation steps.
–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 18.2. 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.

Option	Description
–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.
–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 18.3. 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.

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 18.4. 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.
–byteswapio	(Fortran only) Swap bytes from big-endian to little-endian or vice versa on input/output of unformatted data
–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.

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 as well as to Chapter 6, "*Using Command Line Options*," on page 51.

–#

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 Example 12.1, "Build a DLL: Fortran," on page 133.

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," on page 130.

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 Linux, refer to "Creating and Using Static Libraries on Windows," on page 130.

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.obj` 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 .

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

–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.

To set this option in PVF, use the Fortran | Preprocessor | Preprocessor Definitions property, described in "Preprocessor Definitions," on page 336.

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.

–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`.

To set this option in PVF, use the Fortran | General | Optimization property, described in "Optimization," on page 333.

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

–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.obj` 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.

To set this option in PVF, use the Fortran | General | Debug Information Format property, described in "Debug Information Format," on page 333.

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.obj` 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.

To set this option in PVF, use the Fortran | General | Debug Information Format property described in "Debug Information Format".

Related options: –g, –M<pgflag>

–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 18.5. 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.

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.

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'
```

To set this option in PVF, use the Fortran | General | Additional Include Directories property, described in "Additional Include Directories," on page 332, or the Fortran | Preprocessor | Additional Include Directories property, described in "Additional Include Directories," on page 335.

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. To set this option in PVF, use the Fortran	Floating Point Options	IEEE Arithmetic property, described in "IEEE Arithmetic," on page 342.
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.

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`.

To set this option in PVF, use the Fortran | Floating Point Options | Floating Point Exception Handling property, described in "Floating Point Exception Handling," on page 342

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 213.

Table 18.6. –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]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
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
cray	Force Cray Fortran (CF77) compatibility (pgf77, pgf95, pgfortran, and pghpf only).	Optimization
[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

pgflag	Description	Category
[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
dollar,char	Specifies the character to which the compiler maps the dollar sign code.	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
fixed	Instructs the compiler to assume F77-style fixed format source code (pgf95, pgfortran, and pghpf only).	Fortran Language
[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

pgflag	Description	Category
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.	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
[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
makedll	Generate a dynamic link library (DLL).	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.	Optimization
nomain	When the link step is called, don't include the object file that calls the Fortran main program. .	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

pgflag	Description	Category
nopgdllmain	Do not link the module containing the default DllMain() into the DLL.	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
nostdinc	Instructs the compiler to not search the standard location for include files. To set this option in PVF, use the Fortran | Preprocessor | Ignore Standard Include Path property.	Environment
nostdlib	Instructs the linker to not link in the standard libraries.	Environment
[no]onetrip	Determines whether each DO loop executes at least once.	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
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.	Optimization
[no]r8intrinsics	Determines how the compiler treats the intrinsics CMPLX and REAL.	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.	Code Generation
[no]reentrant	Specifies whether the compiler avoids optimizations that can prevent code from being reentrant.	Code Generation

pgflag	Description	Category
[no]ref_externals	Do/don't force references to names appearing in EXTERNAL statements.	Code Generation
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.	Fortran Language
[no]scalarsse	Do/don't use SSE/SSE2 instructions to perform scalar floating-point arithmetic.	Optimization
[no]second_underscore	Do/don't add the second underscore to the name of a Fortran global if its name already contains an underscore.	Code Generation
[no]signextend	Do/don't extend the sign bit, if it is set.	Code Generation
[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.	Fortran Language
[no]stride0	Do/do not generate alternate code for a loop that contains an induction variable whose increment may be zero.	Code Generation
unix	Uses UNIX calling and naming conventions for Fortran subprograms.	Code Generation
[no]unixlogical	Determines how the compiler treats logical values. .	Fortran Language
[no]unroll	Controls loop unrolling.	Optimization
[no]upcase	Determines whether the compiler preserves uppercase letters in identifiers..	Fortran Language
varargs	Forces Fortran program units to assume calls are to C functions with a varargs type interface.	Code Generation
[no]vect	Do/don't invoke the code vectorizer.	Optimization

–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.

To set this option in PVF, use the Fortran | Output | Module Path property, described in "Module Path," on page 332.

Related options: None.

–mp[=all, align,bind,[no]numa]

Instructs the compiler to interpret user-inserted OpenMP shared-memory parallel programming directives, 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.

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 sub-options 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 sub-option at link time.

bind
: Instructs the compiler to bind threads to cores. You specify this sub-option 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, refer to Chapter 9, "*Using OpenMP*".

To set this option in PVF, use the Fortran | Language | Process OpenMP Directives property, described in "Process OpenMP Directives," on page 338.

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.

To set this option (–O2 or –O3) in PVF, use the Fortran | Optimization | Global Optimizations property, described in "Global Optimizations," on page 334.

Table 18.7 shows the interaction between the `-O` option, `-g` option, `-Mvect`, and `-Mconcur` options.

Table 18.7. 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, see Chapter 7, "*Optimizing & Parallelizing*".

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 with a name comprised of the base file name, such as `myprog`, plus the extension `.exe`, for example: `myprog.exe`.

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

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

To set this option in PVF, use the Fortran | Output | Object File Name property, described in "Object File Name," on page 333.

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

[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.

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
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.

When executed using default (extended) precision, the find_precision program has a value of 1.8446744E +19. 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> .

Syntax:

-- pedantic

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:

–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

–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

–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

–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

–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".

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.

    Note

    Compiling with the CUDA 3.0 toolkit, either by using the `-ta=nvidia:cuda3.0` option or by adding `set CUDAVERSION=3.0` 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`

  fastmath
  : Use routines from the fast math library.

  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," on page 118.

[no]wait
: 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 Binary for Accelerators," on page 116.

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 sub-options 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 the section called "Using –tp to Generate a Unified Binary".

To set this option in PVF, use the Fortran | Target Processors | Unified Binary Information property, described in "Unified Binary Information," on page 351.

The following list contains the possible sub-options for `-tp` and the processors that each sub-option is intended to target:

athlon
: generate 32-bit code for AMD Athlon XP/MP and compatible processors.

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

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

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

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

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

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

istanbul
: generate 32-bit 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 32-bit 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 32-bit code for Pentium Pro/II/III and AthlonXP compatible processors.

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

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

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

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

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

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

piii
: generate 32-bit code for Pentium III and compatible processors, including support for single-precision vector code using SSE instructions.

px
: generate 32-bit 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. Same as px suboption.

shanghai
: generate 32-bit 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 to “Processor-Specific Optimization & the Unified Binary,” on page 73.

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.

To set this option in PVF, use the Fortran | Preprocessor | Undefine Preprocessor Definitions property, described in "Undefine Preprocessor Definitions," on page 336.

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.

To set this option in PVF, use the Fortran | General | Display Startup Banner property, described in "Display Startup Banner," on page 332.

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

–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 191.

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 191. These options are grouped according to categories and are listed with exact syntax, defaults, and notes concerning similar or related options.

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.

  To set this option in PVF, use the Fortran | Floating Point Options | Treat Denormalized Values as Zero property, described in "Treat Denormalized Values as Zero," on page 342.

–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.

To set this option in PVF, use the Fortran | Floating Point Options | Flush Denormalized Results to Zero property, described in "Flush Denormalized Results to Zero," on page 342.

–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 to Chapter 16, "*Programming Considerations for 64-Bit Environments*".

–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.

–Mmpi=msmpi - Select Microsoft MPI libraries.

For more information on these options, refer to Chapter 4, "*Using MPI in PVF*," on page 35.

Note

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.

`lines`
Perform PGI-style line-level profiling.

`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 Chapter 4, "*Using MPI in PVF*".

To set this option in PVF, use the Fortran | General | Profiling property, described in "Line-Level Profiling," on page 352. To enable profiling you must also set the Linker | General | Profiling property, described in "Line-Level Profiling," on page 352.

–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 .

–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.

–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.

–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.

–Mvarargs
force Fortran program units to assume procedure calls are to C functions with a varargs-type interface.

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.

–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.

–Mnostdinc
instructs the compiler to not search the standard location for include files. To set this option in PVF, use the Fortran | Preprocessor | Ignore Standard Include Path property, described in "Ignore Standard Include Path," on page 336.

–Mnostdlib
instructs the linker not to link in the standard libraries 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 pgf77, pgf95, and pgfortran compiler drivers.

Default: For arguments that you do not specify, the defaults are as follows:

backslash	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.

Note

Compiling with the CUDA 3.0 toolkit, either by using the `-ta=nvidia:cuda3.0` option or by adding `set CUDAVERSION=2.0` 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`

emu
: Enable Cuda Fortran emulation mode.

fastmath
: Use routines from the fast math library.

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.

–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, see Chapter 8, "*Using Function Inlining*".

–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 Chapter 8, "*Using Function Inlining*".

To set this option in PVF, use the Fortran | Optimization | Inlining property, described in "Inlining," on page 334.

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.

To set this option in PVF, use the Fortran | Optimization | Auto-Parallelization property, described in "Auto-Parallelization," on page 335.

Note

This option applies only on shared-memory multi-processor (SMP) or multi-core processor-based systems.

`–Mcray[=option[,option,...]]`
: 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 –Mnofpamisalign on all processors.

  Note

  Applicable only with one of these options: –tp barcelona or –tp barcelona-64

–Mf[=option]
: Instructs the compiler to use relaxed precision in the calculation of some intrinsic functions. Can result in improved performance at the expense of numerical accuracy.

  To set this option in PVF, use the Fortran | Floating Point Options | Floating Point Consistency property. For more information on this property, refer to "Floating Point Consistency," on page 342.

  The possible values for option are:

div
Perform divide using relaxed precision.

noorder
Perform reciprocal square root (1/sqrt) using relaxed precision.

order
Perform reciprocal square root (1/sqrt) using relaxed precision.

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.

–`Mnofprelaxed`
(default) instructs the compiler to not use relaxed precision in the calculation of intrinsic functions.

–`Mi4`
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 sub-options can be prefaced with no, which reverses or disables the effect of the sub-option if it's included in an aggregate sub-option 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`.

–M`lre`[=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 sub-option 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.

  To set this option in PVF, use the Fortran | Optimization | Use Frame Pointer property, described in "Use Frame Pointer," on page 335.

–Mnoi4
: 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
: the compiler promotes REAL variables and constants to DOUBLE PRECISION variables and constants, respectively. DOUBLE PRECISION elements are 8 bytes in length.

–Mnor8
: the compiler does not promote REAL variables and constants to DOUBLE PRECISION. REAL variables will be single precision (4 bytes in length).

–Mr8intrinsics
: the compiler treats the intrinsics CMPLX and REAL as DCMPLX and DBLE, respectively.

–Mnor8intrinsics
: the compiler does not promote the intrinsics CMPLX and REAL to DCMPLX and DBLE, respectively.

–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.

To set this option in PVF, use the Fortran | Optimization | Loop Unroll Count property, described in "Loop Unroll Count," on page 335.

–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.

To set this option in PVF, use the Fortran | Optimization Vectorization property, described in "Vectorization," on page 334.

–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
Instructs the vectorizer to enable partial loop vectorization through innemost 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.

–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` .

To set this option in PVF, use the Fortran | Output | Annotated ASM Listing property, described in "Annotate Assembly," on page 350

–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.

–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," on page 143.

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.

  To set this option in PVF, use the Fortran | Diagnostics | Warning Level property, described in "Warning Level," on page 349.

`–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.

  To set this option in PVF, use the Fortran | Diagnostics | Warning Level property, described in "Warning Level," on page 349.

–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.

To set this option in PVF, use the Fortran | Output | Assembler Output property, described in "Generate Assembly," on page 350.

–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
generate a dynamic link library (DLL).

–Mmakeimplib
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
: do not link the module containing the default DllMain() into the DLL. 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 is included in the Release Notes for each release; the PGFORTRAN-specific code in this routine must be incorporated into the custom version of DllMain() to ensure the appropriate function of your DLL.

–Mpreprocess
: perform cpp-like preprocessing on assembly and Fortran input source files.

  To set this option in PVF, use the Fortran | Preprocessor | Preprocess Source File property, described in "Preprocessor Definitions," on page 336.

–Mwritable_strings
: stores string constants in the writable data segment.

  Note

  Options –Xs and –Xst include –Mwritable_strings.

Chapter 19. OpenMP Reference Information

The PGF77, PGF95, and PGFORTRAN Fortran compilers support the OpenMP Fortran Application Program Interface.

This chapter contains detailed descriptions of each of the OpenMP Fortran directives that PGI supports. In addition, the section "Directive Clauses," on page 258 contains information about the clauses associated with these directives .

Tasks

Every part of an OpenMP program is part of a task. "Task Overview," on page 90 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 19.1. 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

Parallelization directives, as described in Chapter 9, "*Using OpenMP*", 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]
```

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," on page 95.

Note

Directives which are presented in pairs must be used in pairs.

This section describes the details of these directives that were summarized in Chapter 9, "*Using OpenMP*". For each directive , this section describes the overall purpose, the syntax, the clauses associated with it, the usage, and examples of how to use it.

ATOMIC

The OpenMP ATOMIC directive is semantically equivalent to a single statement in a CRITICAL...END CRITICAL directive.

Syntax:

```
!$OMP ATOMIC
```

Usage:

The ATOMIC directive is semantically equivalent to enclosing the following single statement in a CRITICAL / END CRITICAL directive pair.

The statements must be one of the following forms:

```
x = x operator expr
x = expr operator x
x = intrinsic (x, expr)
x = intrinsic (expr, 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, and operator is one of +, *, -, /, .AND., .OR., .EQV., or .NEQV..

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
```

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 synchronizes all threads at such a point in a program. Multiple barrier points are allowed within a parallel region. The BARRIER directive must either be executed by all threads executing the parallel region or by none of them.

CRITICAL ... END CRITICAL

The CRITICAL...END CRITICAL directive requires 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)]
```

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 defines 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
```

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 "DO...END DO " 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 >
```

Clauses:

{PRIVATE | LOCAL} (list)
{SHARED | SHARE} (list)
MP_SCHEDTYPE={SIMPLE | INTERLEAVE}
CHUNK=<integer_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 Clauses," on page 258.

DO...END DO

The OpenMP DO...END DO directive supports 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]
```

Clauses:

PRIVATE(list)
FIRSTPRIVATE(list)
LASTPRIVATE(list)
REDUCTION({operator | intrinsic} : list)
SCHEDULE (type [, chunk])
COLLAPSE (n)
ORDERED

Usage:

The real purpose of supporting parallel execution is the distribution of work across the available threads. The DO... END DO directive pair provides 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.

Examples:

```
IF (omp_get_thread_num() .EQ. 0)
THEN
  ...
ELSE IF (omp_get_thread_num() .EQ. 1)
THEN
   ...
ENDIF
```

Tips

Remember these items about clauses in the DO...END DO directives:

- 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 261 .
- The SCHEDULE clause specifies how iterations of the DO loop are divided up between threads. For more information on this clause, refer to “SCHEDULE,” on page 261.

- 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 ”.
- 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.
- 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 :

- The DO loop index variable is always private.
- !$OMP DO 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.
- Values of the loop control expressions and the chunk expressions must be the same for all threads executing the loop.

Example:

```
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
```

FLUSH

The OpenMP FLUSH directive ensures that processor-visible data item are written back to memory at the point at which the directive appears.

Syntax:

```
!$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 appears.

MASTER ... END MASTER

The MASTER...END MASTER directive allows 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
```

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 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
```

ORDERED

The OpenMP ORDERED directive allows 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
```

Usage:

The ORDERED directive can appear only in the dynamic extent of a DO or PARALLEL DO directive that includes the ORDERED clause. The structured code block between the ORDERED / END ORDERED directives 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:

- 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 cannot execute the same ORDERED directive more than once, and cannot execute more than one ORDERED directive.

PARALLEL ... END PARALLEL

The OpenMP PARALLEL...END PARALLEL directive supports 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
```

Clauses:

PRIVATE(list)
SHARED(list)
DEFAULT(PRIVATE | SHARED | NONE)
FIRSTPRIVATE(list)
REDUCTION([{operator | intrinsic}:] list)
COPYIN(list)
IF(scalar_logical_expression)
NUM_THREADS(scalar_integer_expression)

Usage:

This directive pair 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 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 must occur within the scope of a parallel region. Nested PARALLEL... END PARALLEL directive pairs 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 that specifies work distribution. For work distribution, see the DO, PARALLEL DO, or DOACROSS directives.

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
```

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]
```

Clauses:

PRIVATE(list)
SHARED(list)
DEFAULT(PRIVATE | SHARED | NONE)
FIRSTPRIVATE(list)
LASTPRIVATE(list)
REDUCTION([{operator | intrinsic}:] list)
COPYIN(list)
IF(scalar_logical_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 ," on page 250 and "DO...END DO ".

Note

The END PARALLEL DO directive is optional.

PARALLEL SECTIONS

The OpenMP PARALLEL SECTIONS / END SECTIONS directive pair 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]
```

Clauses:

PRIVATE(list)
SHARED(list)
DEFAULT(PRIVATE | SHARED | NONE)
FIRSTPRIVATE(list)
LASTPRIVATE(list)
REDUCTION({operator | intrinsic} : list)
COPYIN (list)
IF(scalar_logical_expression)
NUM_THREADS(scalar_integer_expression)

Usage:

The PARALLEL SECTIONS / END SECTIONS directive pair 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 ," on page 250 and "DO...END DO ".

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)
SHARED(list)
DEFAULT(PRIVATE | SHARED | NONE)
FIRSTPRIVATE(list)
LASTPRIVATE(list)
REDUCTION({operator | intrinsic} : list)
COPYIN (list)
IF(scalar_logical_expression)
NUM_THREADS(scalar_integer_expression)
SCHEDULE (type [, chunk])
COLLAPSE (n)
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 ," on page 250.

SECTIONS ... END SECTIONS

The OpenMP SECTIONS / END SECTIONS directive pair 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]
```

Clauses:

PRIVATE(list)
FIRSTPRIVATE(list)
LASTPRIVATE(list)
REDUCTION({operator | intrinsic} : list)

Usage:

The SECTIONS / END SECTIONS directive pair define sa 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 may only appear within the lexical extent of the enclosing SECTIONS / END SECTIONS directives. In addition, the code within the SECTIONS / END SECTIONS directives must be a structured block.

The available clauses are as defined in "PARALLEL ... END PARALLEL ," on page 250 and "DO...END DO ".

SINGLE ... END SINGLE

The SINGLE...END SINGLE directive designates 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]
```

Clauses:

PRIVATE(list) FIRSTPRIVATE(list) COPYPRIVATE(list)

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:

```
 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 SHOULD ONLY SEE THIS ONCE"
!$OMP END SINGLE
!$OMP END PARALLEL
   PRINT *, "A(0)=",A(0), " A(1)=", A(1)
 END
```

TASK

The OpenMP TASK directive defines an explicit task.

Syntax:

```
!$OMP TASK [Clauses]
< Fortran code executed as task>
!$OMP END TASK
```

Clauses:

IF(scalar_logical_expression)
UNTIED
DEFAULT(private | firstprivate | shared | none)
PRIVATE(list)
FIRSTPRIVATE(list)
SHARED(list)

Usage:

The TASK / END TASK directive pair defines 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:

- 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.

TASKWAIT

The OpenMP TASKWAIT directive specifies a wait on the completion of child tasks generated since the beginning of the current task.

Syntax:

```
!$OMP TASKWAIT
```

Clauses:

IF(scalar_logical_expression)	PRIVATE(list)
UNTIED	FIRSTPRIVATE(list)
DEFAULT(private | firstprivate | shared | none)	SHARED(list)

Usage:

The OpenMP TASKWAIT directive specifies 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:

- The TASKWAIT directive 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

The OpenMP THREADPRIVATE directive identifies a Fortran common block as being private to each thread.

Syntax:

```
!$OMP THREADPRIVATE (list)
```

Usage:

The `list` is a comma-separated list of named variables to be made private to each thread or named common blocks to be made private to each thread but global within the thread. Common block names must appear between slashes, such as `/common_block_name/`.

This directive must appear in the declarations section of a program unit after the declaration of any common blocks or variables listed. On entry to a parallel region, data in a THREADPRIVATE common block or variable is undefined unless COPYIN is specified on the PARALLEL directive. When a common block or variable that is initialized using DATA statements appears in a THREADPRIVATE directive, each thread's copy is initialized once prior to its first use.

Restrictions:

The following restrictions apply to the THREADPRIVATE directive:

- The THREADPRIVATE directive must appear after every declaration of a thread private common block.
- 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.

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 Clauses

Some directives 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, so the clauses that are valid are included with the description of the directive. Typically, if no data scope clause is specified for variables, the default scope is *shared*.

Table 9.2, “Directive Clauses Summary Table ,” on page 93 provides a brief summary of the clauses associated with OpenMP directives 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).

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 19.1. Initialization of REDUCTION Variables

Operator / Intrinsic	Initialization
+	0
*	1
-	0
.AND.	.TRUE.
.OR.	.FALSE.
.EQV.	.TRUE.

Operator / Intrinsic	Initialization
.NEQV.	.FALSE.
MAX	Smallest representable number
MIN	Largest representable number
IAND	All bits on
IOR	0
IEOR	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 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.

- 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.

Table 9.4, "OpenMP-related Environment Variable Summary Table," on page 100 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 that include the SCHEDULE(RUNTIME) clause, described in "SCHEDULE," on page 261. 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 ," on page 247.

Examples of the use of OMP_SCHEDULE are as follows:

```
% 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, described in "MPSTKZ," on page 141. 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`, described in "MP_SPIN," on page 141.
- 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 20. PGI Accelerator Compilers Reference

Chapter 10, "*Using an Accelerator*" describes the programming model that uses a collection of compiler directives to specify regions of code in Fortran 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 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 that PGI supports. In addition, the section "PGI Accelerator Directive Clauses," on page 271 contains information about the clauses associated with these directives.

PGI Accelerator Directives

This section provides detailed descriptions of the Fortran directives used to delineate accelerator regions and to augment information available to the compiler for scheduling of loops and classification of data.

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 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 )
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.
- 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 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 )
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 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 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.
- 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 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 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.

These directives create a visible device copy of the variable or array.

Syntax

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 )
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 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 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 `.false.` in Fortran, the host copy is executed.
  - When the condition in the `if` clause evaluates to `.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 )
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 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.
- 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.

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.

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.

acc_get_device

The `acc_get_device` routine returns the type of accelerator device being used.

Syntax

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 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 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_num_devices

The `acc_get_num_devices` routine returns the number of accelerator devices of the given type attached to the host.

Syntax

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 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 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 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_set_device

The `acc_set_device` routine specifies which type of device the runtime uses when executing an accelerator compute region.

Syntax

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 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 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_on_device

The `acc_on_device` routine tells the program whether it is executing on a particular device.

Syntax

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 `.true.` for Fortran; otherwise, it evaluates to `.false.` for Fortran.

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 21. Directives Reference

As we mentioned in Chapter 11, "*Using Directives*," on page 123, PGI Fortran compilers support proprietary directives.

Directives override corresponding command-line options. For usage information such as the scope and related command-line options, refer to "*Using Directives*".

This chapter contains detailed descriptions of PGI's proprietary directives.

PGI Proprietary Fortran Directive Summary

Directives are Fortran 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 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 to tune selected routines or loops.

As outlined in Chapter 11, "*Using Directives*," on page 123, 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 override corresponding command-line options. For usage information such as the scope and related command-line options, refer to Chapter 11, "*Using Directives*," on page 123.

altcode (noaltcode)

The `altcode` directive instructs the compiler to generate alternate code for vectorized or parallelized loops.

The `noaltcode` directive disables generation of alternate code.

Scope: This directive 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 toggles the effects of the `-Mvect=noassoc` command-line option, an optimization `-M` control.

Scope: This directive 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 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 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 alters the effects of the `-Mconcur` command-line option. The directive instructs the auto-parallelizer to enable auto-concurrentization of loops.

Scope: This directive 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 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 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.

invarif (noinvarif)

This directive 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 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 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 is:

```
c$mem prefetch <var1>[,<var2>[,...]]
```

where <varn> is any valid variable, member, or array element reference.

opt

The opt directive overrides the value specified by the command line option -On.

The syntax of this directive 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_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 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, a problem results since 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.

tp

You use the directive tp to specify one or more processor targets for which to generate code.

```
 cpgi$ tp [target]...
```

Note

The tp directive can only be applied at the routine or global level.

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 enables loop unrolling while nounroll disables loop unrolling.

Note

The unroll directive has no effect on vectorized loops.

The directive 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 novector disables vectorization. The directive 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 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

As mentioned in Chapter 11, "*Using Directives*," on page 123, 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.

For examples on how to use the prefetch directive, refer to "Prefetch Directives ," on page 126.

!DEC$ Directives

As mentioned in Chapter 11, "*Using Directives*," on page 123, 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 "!DEC$ Directives," on page 127

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 21.1 indicates which rules are ignored for which dummy arguments in the sample subroutine fragment:

Table 21.1. 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 22. 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 running a Windows operating system.

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).

Win32 Programming Model

This section defines compiler and assembly language conventions for the use of certain aspects of an x86 processor running a 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 Fortran 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 22.1. 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 22.2. 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	

Several key points concerning the stack frame:

- 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 22.3. 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 22.3, "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 22.4. 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 22.5. 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 22.6. 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 18, "*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 22.1. 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
```

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 Fortran 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 22.7. 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 22.8 shows the stack frame organization.

Table 22.8. 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	

Position	Contents	Frame
-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 22.9, "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 22.2. 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 22.9. 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 18, "*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 22.3. 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 22.10. 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

Fortran Type	Size (bytes)	Alignment (bytes)
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 22.11 provides the C/C++ data type corresponding to each Fortran data type.

Table 22.11. 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 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

Table 22.12 provides the Fortran and C/C++ representation of the COMPLEX type.

Table 22.12. 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 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

Fortran Type (lower case)	C/C++ Type	Size (bytes)
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`. Refer to "Argument Passing and Return Values," on page 154, for additional information.

Chapter 23. PVF Properties

As we mentioned in Chapter 2, "*Build with PVF*", there are a number of property pages that are available in a PVF project.

Property pages are grouped into categories that you can access from the Property Page dialog. Further, each of PVF's property pages contains one or more properties, or configuration options. The set of categories and property pages available vary, depending on the type of project.

The properties in a PVF project are divided into the following categories:

- General
- Debugging
- Fortran
- Linker
- Librarian
- Resources
- Build Events
- Custom Build Step

This chapter contains descriptions of each of PVF's property pages. The summary table, Table 23.1, lists the properties in alphabetical order for quick reference. Whenever applicable, the description shows the command line option associated with the property.

The remainder of this chapter provides detailed descriptions of the properties, organized as you would see them in the Property Page dialog: by category and property page.

Tip

The Fortran, Linker, and Librarian categories contain a Command Line property page where you can see the command line derived from the properties in that category. Options that are not supported by the PVF property pages can be added to the command line from this property page by entering them in the *Additional Options* field.

Table 23.1. PVF Property Page Summary

Use this Property...	From this Property Page...	To do this...
Accelerator Information	Fortran | Diagnostics	Generate information about accelerator regions. (`-Minfo=accel`)
Additional Arguments: job submit	Debugging [Cluster MPI]	Specify additional arguments to be passed to the `job submit` command.
Additional Arguments: mpiexec	Debugging [Cluster MPI]	Specify additional arguments to be passed to `mpiexec`.
Additional Arguments: mpiexec	Debugging [Local MPI]	Specify additional arguments to be passed to `mpiexec`.
Additional Dependencies	Custom Build Step | General	Specify additional input files to use as dependencies.
Additional Dependencies	Librarian | General	Specify one or more directories to the library search path.
Additional Dependencies	Linker | Input	Specify additional dependencies, such as libraries, to the link line.
Additional Include Directories	Fortran | General	Specify one or more directories to add to the compiler's include path.
Additional Include Directories	Fortran | Preprocessing	Specify one or more directories to add to the compiler's include path.
Additional Library Directories	Librarian | General	Specify one or more directories to add to the library search path.
Additional Library Directories	Linker | General	Specify one or more directories to add to the library search path.
AMD Athlon	Fortran | Target Processors	Optimize for AMD Athlon, AMD Opteron and compatible processors. (`-tp=k8-32`) (`-tp=k8-64`)
AMD Barcelona	Fortran | Target Processors	Optimize for AMD Opteron/Quadcore and compatible processors. (`-tp=barcelona-32`) (`-tp=barcelona-64`)
AMD Istanbul	Fortran | Target Processors	Optimize for AMD Istanbul processor-based systems. (`-tp=istanbul-32`) (`-tp=istanbul-64`)

Use this Property...	From this Property Page...	To do this...
AMD Shanghai	Fortran | Target Processors	Optimize for AMD Shanghai processor-based systems. (`-tp=shanghai-32`) (`-tp=shanghai-64`)
Annotate Assembly	Fortran | Diagnostics	Generate an assembly file (`.s`) that is annotated with source code. (`-Manno`)
Application Arguments	Debugging [All]	Pass command line arguments to the application when it is run or debugged.
Application Command	Debugging [All]	Specify the application to execute when you select *Start Debugging* or *Start Without Debugging* from the *Debug* menu.
Auto-Parallelization	Fortran | Optimization	Enable auto-parallelization of parallelizable code. (`-Mconcur`)
Build Log File	General	Specify the build log file that is produced when the project is built.
Build Log Level	General	Specify the level of detail to be included in the build log file.
Calling Convention	Fortran | External Procedures	Specify an alternate calling convention. `Default`: Accept the default calling convention. `C By Reference`: Use the CREF calling conventions. (`-Miface=cref`) `Unix`: [Win32 platform only] Use the UNIX calling convention. (`-Miface=unix`)
Case of External Names	Fortran | External Procedures	Specify the case used for Fortran external names. (`-Mnames`)
CCFF Information	Fortran | Diagnostics	Append common compiler feedback format information to object files. (`-Minfo=ccff`)
Command Line	Build Events | Pre-Build	Specify the command line that the build tool will run.
Command Line	Build Events | Pre-Link	Specify the command line that the build tool will run.

Use this Property...	From this Property Page...	To do this...
Command Line	Build Events | Post-Build	Specify the command line that the build tool will run.
Command Line	Custom Build Step | General	Specify the command line that the build tool will run.
Command Line	Fortran | Command Line	Specify options that you want used during compilation but that are not available through any of the Fortran property pages.
Command Line	Librarian | Command Line	Specify options to use at link time that are not available from the Librarian property pages.
Command Line	Linker | Command Line	Specify options to use at link time that are not available from the Linker property pages.
Command Line	Resources | Command Line	Add options to the Resource compiler's command line.
Configuration Type	General	Change the output type that the project produces.
CUDA Fortran Compute Capability	Fortran | Language	Either automatically generate code compatible with all applicable compute capabilities or use a manually selected set. (`-Mcuda=cc10|cc11|cc12|cc13|cc20`)
CUDA Fortran CC 1.0	Fortran | Language	When Enable CUDA Fortran is set to `Yes` and CUDA Fortran Compute Capability is set to `Manual`: `Yes`: Enables generating code for compute capability 1.0 (`-Mcuda=cc10`) `No`: Disables generating code for compute capability 1.0
CUDA Fortran CC 1.1	Fortran | Language	When Enable CUDA Fortran is set to `Yes` and CUDA Fortran Compute Capability is set to `Manual`: `Yes`: Enables generating code for compute capability 1.1 (`-Mcuda=cc11`) `No`: Disables generating code for compute capability 1.1

Use this Property...	From this Property Page...	To do this...
CUDA Fortran CC 1.2	Fortran | Language	When Enable CUDA Fortran is set to `Yes` and CUDA Fortran Compute Capability is set to `Manual`: `Yes`: Enables generating code for compute capability 1.2 (`-Mcuda=cc12`) `No`: Disables generating code for compute capability 1.2
CUDA Fortran CC 1.3	Fortran | Language	When Enable CUDA Fortran is set to `Yes` and CUDA Fortran Compute Capability is set to `Manual`: `Yes`: Enables generating code for compute capability 1.3 (`-Mcuda=cc13`) `No`: Disables generating code for compute capability 1.3
CUDA Fortran CC 2.0	Fortran | Language	When Enable CUDA Fortran is set to `Yes` and CUDA Fortran Compute Capability is set to `Manual`: `Yes`: Enables generating code for compute capability 2.0 (`-Mcuda=cc20`) `No`: Disables generating code for compute capability 2.0
CUDA Fortran Emulation	Fortran | Language	Enable CUDA Fortran emulation mode. (`-Mcuda=emu`)
CUDA Fortran Keep Binary	Fortran | Language	Keep the Cuda binary (.bin) file. (`-Mcuda=keepbin`).
CUDA Fortran Keep Kernel Source	Fortran | Language	Keep the kernel source files. (`-Mcuda=keepgpu`).
CUDA Fortran Keep PTX	Fortran | Language	Keep the portable assembly (.ptx) file for the GPU code. (`-Mcuda=keepptx.`)
CUDA Fortran Register Limit	Fortran | Language	Specify the maximum number of registers to use on the GPU. Leave blank for no limit. (`-Mcuda=maxregcount:n`)

Use this Property...	From this Property Page...	To do this...
CUDA Fortran Toolkit	Fortran | Language	When Enable CUDA Fortran is set to `Yes`, specifies the version of the Cuda toolkit targeted by the compilers. `Default`: The compiler selects the default CUDA toolkit version. `2.3`: Specifies use of toolkit version 2.3. `3.0`: Specifies use of toolkit version 3.0. (`-Mcuda=cuda2.3|cuda3.0`)
CUDA Fortran Use Fast Math Library	Fortran | Language	Use routines from the fast math library. (`-Mcuda=fastmath`)
CUDA Fortran Use Fused Multiply-Adds	Fortran | Language	Control generation of fused multiply-add (FMA) instructions. `Yes`: Enables generation of FMA. (default) `No`: Disables generation of FMA. (`-Mcuda=nofma`)
Debug Information Format	Fortran | General	Specify whether the compiler should generate debug information; and if so, in what format. (`-g`) (`-gopt`)
Description	Build Events | Pre-Build	Echo the contents of the Description property to the Output window when this event is fired.
Description	Build Events | Pre-Link	Echo the contents of the Description property to the Output window when this event is fired.
Description	Build Events | Post-Build	Echo the contents of the Description property to the Output window when this event is fired.
Description	Custom Build Step | General	Echo the contents of the Description property to the Output window when this event is fired.
Display Startup Banner	Fortran | General	Add the `-V` switch to the compilation line so the compiler's startup banner is displayed during compilation.
Enable CUDA Fortran	Fortran | Language	Enable CUDA Fortran. (`-Mcuda`)
Enable Limited DWARF	Fortran | Profiling	Generate limited DWARF information to use with performance profilers. (`-Mprof=dwarf`).

Use this Property...	From this Property Page...	To do this...
Environment	Debugging [All]	Specify environment variables to set for the application when it runs.
Excluded From Build	Build Events | Pre-Build	Specify whether this build event should be excluded from the build for the current configuration.
Excluded From Build	Build Events | Pre-Link	Specify whether this build event should be excluded from the build for the current configuration.
Excluded From Build	Build Events | Post-Build	Specify whether this build event should be excluded from the build for the current configuration.
Export Symbols	Linker | General	Specify whether a DLL will export symbols.
Extend Line Length	Fortran | Language	Extend the line length for fixed-format Fortran files to 132 characters. (`-Mextend`)
Extensions to Delete on Clean	General	Specify which files in the intermediate directory should be deleted when the project is cleaned or before it is rebuilt.
Floating Point Consistency	Fortran | Floating Point Options	Enable relaxed floating point accuracy in favor of speed. (`-Mfprelaxed`)
Floating Point Exception Handling	Fortran | Floating Point Options	Enable floating point exceptions through the compiler option: (`-Ktrap=fp`)
Flush Denormalized Results to Zero	Fortran | Floating Point Options	Accept the default handling of denormalized floating point results. `Yes`: Enable SSE flush-to-zero mode. (`-Mflushz`) `No`: Disable SSE flush-to-zero mode. (`-Mnoflushz`)
Fortran Dialect	Fortran | Language	Specify the Fortran language dialect to use during compilation.
Fortran Language Information	Fortran | Diagnostics	Generate information about Fortran language features. (`-Minfo=ftn`)
Function-Level Profiling	Fortran | Profiling	Generate code instrumented for function-level profiling. (`-Mprof=func`)

Use this Property...	From this Property Page...	To do this...
Generate Assembly	Fortran | Diagnostics	Generate an assembly file (.s) during compilation. (`-Mkeepasm`)
Generic x86 [Win32 only]	Fortran | Target Processors	Optimize for any x86 processor-based system. (`-tp=px-32`)
Generic x86-64 [x64 only]	Fortran | Target Processors	Optimize for any x86-64 processor-based system. (`-tp=px-64`)
Global Optimizations	Fortran | Optimization	Set the compiler's global optimization level by selecting a –O<level> option.
IEEE Arithmetic	Fortran | Floating Point Options	`Default`: Accept the default floating point arithmetic. `Yes`: Enable IEEE floating point arithmetic: (`-Kieee`). `No`: Disable IEEE floating point arithmetic: (`-Knoieee`).
Ignore Standard Include Path	Fortran | Preprocessing	Specify whether the preprocessor should ignore the standard include path. (`-Mnostdinc`).
Inlining	Fortran | Optimization	Enable inlining for certain subprograms. (`-Minline`).
Inlining Information	Fortran | Diagnostics	Generate information about inlining. (`-Minfo=inline`)
Intel Core 2	Fortran | Target Processors	Optimize for Intel Core 2 Duo and compatible processors. (`-tp=core2-32`) (`-tp=core2-64`)
Intel Core i7	Fortran | Target Processors	Optimize for Intel Core i7 processor-based systems. (`-tp=nehalem-32`) (`-tp=nehalem-64`)
Intel Penryn	Fortran | Target Processors	Optimize for Intel Penryn architecture and compatible processors. (`-tp=penryn-32`) (`-tp=penryn-64`)

Use this Property...	From this Property Page...	To do this...
Intel Pentium 4	Fortran | Target Processors	Optimize for Intel Pentium 4 and compatible processors. (`-tp=p7-32`) (`-tp=p7-64`)
Intermediate Directory	General	Specify a relative path to the intermediate file directory where the intermediate files (i.e., object files) are created when the project is built.
IPA Information	Fortran | Diagnostics	Generate information about inter-procedural analysis optimizations. (`-Minfo=ipa`)
Line-Level Profiling	Fortran | Profiling	Generate code instrumented for line-level profiling. (`-Mprof=lines`).
Location of job.exe	Debugging [Cluster MPI]	Override the default path to `job.exe` as specified in the system `PATH` variable.
Location of mpiexec	Debugging [Local MPI]	Override the default path to `mpiexec` as specified in the system `PATH` variable.
Loop Intensity Information	Fortran | Diagnostics	Generate compute intensity information about loops. (`-Minfo=intensity`)
Loop Optimization Information	Fortran | Diagnostics	Generate information about loop optimizations. (`-Minfo=loop`)
Loop Unroll Count	Fortran | Optimization	Specify unrolling by two or four. (`-Munroll`)
LRE Information	Fortran | Diagnostics	Generate information about loop-carried redundancy elimination. (`-Minfo=lre`)
Merge Environment	Debugging	Specify whether or not to merge the environment variables in the Environment property with the existing environment when the application is run or debugged.
Module Path	Fortran | General	Specify the location of module (`.mod`) files.
MPI	Fortran | Language	Specify whether or not to enable compiling and linking with the Microsoft MPI libraries.

Use this Property...	From this Property Page...	To do this...
MPI	Fortran | Profiling	Specify whether to access profiled MPI communication libraries. Must be used in conjunction with function-level or line-level profiling.
MPI Debugging	Debugging [All]	Specify the run and debug mode: `Disabled`: In serial mode. `Local`: Using MPI locally. `Cluster`: Using MPI on a cluster.
NVIDIA: Analysis Only	Fortran | Target Accelerators	Perform loop analysis only. Do not generate GPU code. (`-ta=nvidia:analysis`)
NVIDIA: Compute Capability	Fortran | Target Accelerators	Generate code for the specified compute capability. (`-ta=nvidia:cc10|cc11|cc12|cc13|cc20`)
NVIDIA: CC 1.0	Fortran | Target Accelerators	When Target NVIDIA Accelerator is set to `Yes` and NVIDIA Compute Capability is set to `Manual`: `Yes`: Enables generating code for compute capability 1.0 (`-ta=nvidia:cc10`) `No`: Disables generating code for compute capability 1.0
NVIDIA: CC 1.1	Fortran | Target Accelerators	When Target NVIDIA Accelerator is set to `Yes` and NVIDIA Compute Capability is set to `Manual`: `Yes`: Enables generating code for compute capability 1.1 (`-ta=nvidia:cc11`) `No`: Disables generating code for compute capability 1.1
NVIDIA: CC 1.2	Fortran | Target Accelerators	When Target NVIDIA Accelerator is set to `Yes` and NVIDIA Compute Capability is set to `Manual`: `Yes`: Enables generating code for compute capability 1.2 (`-ta=nvidia:cc12`) `No`: Disables generating code for compute capability 1.2

Use this Property...	From this Property Page...	To do this...
NVIDIA: CC 1.3	Fortran | Target Accelerators	When Target NVIDIA Accelerator is set to `Yes` and NVIDIA Compute Capability is set to `Manual`: `Yes`: Enables generating code for compute capability 1.3 (`-ta=nvidia:cc13`) `No`: Disables generating code for compute capability 1.3
NVIDIA: CC 2.0	Fortran | Target Accelerators	When Target NVIDIA Accelerator is set to `Yes` and NVIDIA Compute Capability is set to `Manual`: `Yes`: Enables generating code for compute capability 2.0 (`-ta=nvidia:cc20`) `No`: Disables generating code for compute capability 2.0
NVIDIA: CUDA Toolkit	Fortran | Target Accelerators	When Target NVIDIA Accelerator is set to `Yes`, specifies the Cuda toolkit version that is targeted by the compilers. `Default`: The compiler selects the default CUDA toolkit version. `2.3`: Specifies use of toolkit version 2.3. `3.0`: Specifies use of toolkit version 3.0. (`-ta=nvidia:cuda2.3|cuda3.0`)
NVIDIA: Enable Profiling	Fortran | Target Accelerators	Collect simple timing information for accelerator kernel profiling. (`-ta=nvidia:time`)
NVIDIA: Keep Kernel Binary	Fortran | Target Accelerators	Keep kernel binary (.bin) files. (`-ta=nvidia:keepbin`)
NVIDIA: Keep Kernel PTX	Fortran | Target Accelerators	Keep the portable assemply (`.ptx`) file for the GPU code. (`-ta=nvidia:keepptx`)
NVIDIA: Keep Kernel Source	Fortran | Target Accelerators	Keep kernel source files. (`-ta=nvidia:keepgpu`)
NVIDIA: Register Limit	Fortran | Target Accelerators	Specify the maximum number of registers to use on the GPU. Leave blank for no limit. (`-ta=nvidia:maxregcount:n`)

Use this Property...	From this Property Page...	To do this...
NVIDIA: Synchronous Kernel Launch	Fortran | Target Accelerators	When Target NVIDIA Accelerator is set to `Yes` use this property to wait for each kernel to finish before continuing in the host program. `Yes`: wait for kernel to finish. This is the default. `No`: do not wait for kernel to finish. (`-ta=nvidia:[no]wait`)
NVIDIA: Use 24-bit Subscript Multiplication	Fortran | Target Accelerators	Use 24-bit multiplication for subscripting. (`-ta=nvidia:mul24`)
NVIDIA: Use Fast Math Library	Fortran | Target Accelerators	Use routines from the fast math library. (`-ta=nvidia:fastmath`)
NVIDIA: Use Fused Multiply-Adds	Fortran | Target Accelerators	Control generation of fused multiply-add (FMA) instructions. `Yes`: Enables generation of FMA. (default) `No`: Disables generation of FMA. (`-ta=nvidia:nofma`)
Number of Cores	Debugging [Cluster MPI]	Select the number of cores on which to run.
Number of Processes	Debugging [Local MPI]	Select the number of processes to use to run.
Object File Name	Fortran | General	File level: Set the name of the object file using the `-o` switch. Project level: Set the location of the object files created by a build.
OpenMP Information	Fortran | Diagnostics	Generate information about OpenMP. (`-Minfo=mp`)
Optimization	Fortran | General	Select the overall code optimization.
Optimization	Fortran | Optimization	Select the overall code optimization.
Optimization Information	Fortran | Diagnostics	Generate information about general optimizations. (`-Minfo=opt`)
Output Directory	General	Specify a relative path to the output file directory. This directory is where the project's output files will be built.
Output File	Librarian | General	Override the default output file name.
Output File	Linker | General	Override the default output file name.

Use this Property...	From this Property Page...	To do this...
Outputs	Custom Build Step | General	Specify the files generated by the custom build step.
Parallelization Information	Fortran | Diagnostics	Generate information about parallel optimizations (`-Minfo=par`)
Preprocessor Definitions	Fortran | Preprocessing	Add one or more preprocessor definitions to compilation.
Preprocess Source File	Fortran | Preprocessing	Specify whether the compiler should preprocess source files or not.
Process OpenMP Directives	Fortran | Language	Enable OpenMP 3.0 language extensions. (`-mp`)
Runtime Library	Fortran | Code Generation	Specify what type of runtime libraries should be used during linking. (`-Bstatic`) (`-Bdynamic`) **Note**. You must keep the runtime libraries consistent within a solution.
Standard Error	Debugging [Cluster MPI]	Specify a file to be used for standard error for MPI cluster running or debugging.
Standard Input	Debugging [Cluster MPI]	Specify a file to be used as standard input for MPI cluster running or debugging.
Standard Output	Debugging [Cluster MPI]	Specify a file to be used as standard output for MPI cluster running or debugging.
String Length Arguments	Fortran | External Procedures	Specify where string length arguments are placed in the argument list. (`-Miface=[no]mixed_str_len_arg`)
Suppress CCFF Information	Fortran | Profiling	Suppress profiling's default generation of CCFF information. (`-Mprof=noccff`)
Target Host	Fortran | Target Accelerators	Generate code just for the host if no accelerator is selected. Otherwise, generate PGI Unified Binary Code for the host and accelerator. (`-ta=host`)
Target NVIDIA Accelerator	Fortran | Target Accelerators	Select NVIDIA accelerator target. (`-ta=nvidia`)

Use this Property...	From this Property Page...	To do this...
Treat Backslash as Character	Fortran | Language	`Yes`: Treat the backslash ('\') as a regular character. (`-Mbackslash`) `No`: Treat the backslash ('\') as an escape character. (`-Mnobackslash`)
Treat Denormalized Values as Zero	Fortran | Floating Point Options	`Default`: Accept the default treatment of denormalized floating point values. `Yes`: Enable the treatment of denormalized floating point values as zero. (`-Mdaz`) `No`: Disable the treatment of denormalized floating point values as zero. (`-Mnodaz`)
Undefine Preprocessor Definitions	Fortran | Preprocessing	Undefine one or more preprocessor definitions.
Unified Binary Information	Fortran | Diagnostics	Generate information specific to the PGI Unified Binary (`-Minfo=unified`)
Use Frame Pointer	Fortran | Optimization	Specify if generated code uses a frame pointer. `Yes`: Generate code that uses a frame pointer. (`-Mframe`) `No`: Generate code that does not use a frame pointer. (`-Mnoframe`)
Vectorization	Fortran | Optimization	Specify the type of vectorization to perform. (`-Mvect`)
Vectorization Information	Fortran | Diagnostics	Generate vectorization information. (`-Minfo=vect`)
Warning Level	Fortran | Diagnostics	Select the level of diagnostic reporting you want the compiler to use. (`-Minform=inform|warn|severe|fatal`)
Working Directory	Debugging [Cluster MPI]	Specify the application's working directory for MPI cluster running or debugging.

Use this Property...	From this Property Page...	To do this...
Working Directory	Debugging [Local MPI]	Specify the application's working directory for local MPI running or debugging.
Working Directory	Debugging [Serial]	Specify the application's working directory for serial running or debugging.

General Property Page

This section contains the properties that are included on the General property page.

General

Output Directory

Use this property to specify a relative path to the output file directory. This directory is where the project's output files are built.

Intermediate Directory

Use this property to specify a relative path to the intermediate file directory. This directory is where the intermediate files (i.e., object files) are created when the project is built.

Extensions to Delete on Clean

Use this property to specify which files in the intermediate directory should be deleted when the project is cleaned or before it is rebuilt. This property uses a semi-colon-delimited wildcard specification for the files.

Configuration Type

Use this property to change the output type that the project produces.

When you create a project, you specify the type of output that the project produces: executable, static library, or dynamic library. If you want to change the output type, use this property to do so.

Build Log File

Use this property to specify the build log file that is produced when the project is built.

Build Log Level

Use this property to specify the level of detail to be included in the build log file.

Note

Any setting above Default can produce large amounts of output and may potentially slow down the building of your project.

Debugging Property Page

This section contains the properties that are included on the Debugging property page.

Debugging

Application Command

Use this property to specify the application to execute when you select *Start Debugging* or *Start Without Debugging* from the *Debug* menu.

- If the Startup Project in your solution is a PVF project that builds an executable, there is probably no need to change this property.
- If the Startup Project in your solution is a PVF project that builds a DLL or static library, you must use the Command property to specify an application to execute when you run (with or without debugging).

Note

To use the PVF debug engine, the Startup Project must be a PVF project. If, for example, your main executable is built by a Visual C++ project that links against a PVF project, you would designate the PVF project as the Startup Project; and in its Debugging | Application Command property, you would specify the path to the executable built by the Visual C++ project.

Tip

The Startup Project is the project listed in boldface in the solution explorer. You can change the Startup Project by right-clicking on any project in the solution explorer and selecting *Set as Startup Project* from the context menu.

Application Arguments

Use this property to pass command line arguments to the application when it is run or debugged.

Environment

Use this property to specify any environment variables to set for the application when it runs. One common use of this property is to augment the `PATH` environment variable. For example, if the application requires DLLs to run but the general environment is not set to find these, the path to these DLLs could be added to the `PATH` environment variable.

For more information on `PATH`, refer to "PATH," on page 142.

If the Merge Environment property is set to `Yes`, then the contents of the Environment property are merged with the existing environment when the application is run or debugged.

Merge Environment

Use this property to merge the environment variables in the Environment property with the existing environment when the application is run or debugged. To do this, set the Merge Environment property to `Yes`.

MPI Debugging

Use this property to enable MPI debugging and select between local MPI debugging and cluster MPI debugging.

The value selected for this property determines which properties are displayed following it on the Debugging property page.

Important

If you change the value of this property and the displayed properties do not change, be sure to click Apply in the property page dialog box.

- When MPI Debugging is set to *Disabled*, the application is run or debugged in serial mode.
- When MPI Debugging is set to *Local*, the application is run or debugged using `mpiexec`. All processes launched are local to the system on which the application is run.
- When MPI Debugging is set to *Cluster*, the application is launched for running or debugging using the Microsoft HPC Job Manager via the `job submit` command. Some or all of the processes launched as part of execution may be on remote nodes.

  A PGI license that supports running or debugging MPI processes on remote clusters is required for this option to be available. For information about upgrading your current license to a cluster license, contact sales@pgroup.com.

Working Directory

[Serial]

Use this property to specify the application's working directory when it is run or debugged serially. By default, the working directory is set to the solution directory.

This property is displayed when the MPI Debugging property is set to *Disabled*.

Number of Processes

[Local MPI]

Use this property to specify the number of MPI processes to use when the application is run or debugged. The number of processes is passed to `mpiexec` using the `-n` option.

This property is displayed when the MPI Debugging property is set to *Local*.

Working Directory

[Local MPI]

Use this property to specify the application's working directory when it is run or debugged using local MPI. By default, the working directory is set to the solution directory.

This property is displayed when the MPI Debugging property is set to *Local*.

Additional Arguments: mpiexec

[Local MPI]

Use this property to specify additional arguments to be passed to `mpiexec` when the application is run or debugged.

This property is displayed when the MPI Debugging property is set to *Local*.

Location of mpiexec

[Local MPI]

Use this property to override the default path to `mpiexec` as specified in the system PATH variable.

This property is displayed when the MPI Debugging property is set to *Local*.

Number of Cores

[Cluster MPI]

Use this property to specify the number of cores with which to run or debug the MPI application. The value of this property is passed to the `job submit` command using the `/numcores` option.

This property is displayed when the MPI Debugging property is set to *Cluster*.

Working Directory

[Cluster MPI]

Use this property to specify the application's working directory when it is run or debugged using MPI on a cluster. The value of this property will be passed to the `job submit` command using the `/workdir` option. By default, the working directory is set to the solution directory.

This property is displayed when the MPI Debugging property is set to *Cluster*.

Note

The working directory specified for cluster debugging must be a directory specified as *shared* among all the nodes on the cluster. Execution of the application fails if the working directory is not marked *shared*. For assistance designating a directory as *shared*, contact the system administrator.

Standard Input

[Cluster MPI]

Use this property to specify a file to be used as standard input by the MPI application running on a cluster. This file is passed to the `job submit` command using the `/stdin` option.

This property is displayed when the MPI Debugging property is set to *Cluster*.

By default, the `job submit` command searches for the input file in the application's working directory. If the input file is in a location other than the working directory, specify a full path to the file.

Standard Output

[Cluster MPI]

Use this property to specify a file to be used for standard output by the MPI application running on a cluster. This file will be passed to the `job submit` command using the `/stdout` option.

This property is displayed when the MPI Debugging property is set to *Cluster*.

By default, the `job submit` command creates the output file in the application's working directory. To create the output file in a location other than the working directory, specify a full path to the file.

Standard Error

[Cluster MPI]

Use this property to specify a file to be used for standard error by the MPI application running on a cluster. This file is passed to the `job submit` command using the `/stderr` option.

This property is displayed when the MPI Debugging property is set to *Cluster*.

By default, the `job submit` command creates the error file in the application's working directory. If the error file is in a location other than the working directory, specify a full path to the file.

Note

To capture the standard error output by an MPI cluster application, a file must be specified in the *Standard Error* property.

Additional Arguments: job submit

[Cluster MPI]

Use this property to specify additional arguments to be passed to the `job submit` command when the application is run or debugged.

This property is displayed when the MPI Debugging property is set to *Cluster*.

Additional Arguments: mpiexec

[Cluster MPI]

Use this property to specify additional arguments to be passed to `mpiexec` when the application is run or debugged.

This property is displayed when the MPI Debugging property is set to *Cluster*.

Location of job.exe

[Cluster MPI]

Use this property to override the default path to `job.exe` as specified in the system PATH variable.

This property is displayed when the MPI Debugging property is set to *Cluster*.

Fortran Property Pages

This section contains the property pages that are included in the Fortran category. This category is further divided into the following property pages, displayed in the following order:

- General
- Optimization
- Preprocessing
- Code Generation
- Language
- Floating Point Options
- External Procedures
- Target Processors
- Target Accelerators
- Diagnostics
- Profiling
- Command Line

The following sections describe the properties available on each property page.

Fortran | General

The following properties are available from the Fortran | General property page.

Display Startup Banner

Use this property to determine whether to display the compiler's startup banner during compilation.

Changing the property to `Yes` adds the `-V` switch to the compilation line, which causes the compiler to display the startup banner during compilation.

For more information on `-V`, refer to Chapter 18, "*Command-Line Options Reference*".

Additional Include Directories

Use this property to add one or more directories to the compiler's include path.

For every path that is added to this property, PVF adds `-I<path>` to the compilation line.

There are two ways to add directories to this property:

- Type the information directly into the property page box.

  Use a semi-colon (';') to separate each directory.

- Click the ellipsis ('...') button in the property page box to open the *Additional Include Directories* dialog box.

  Enter each directory on its own line in this box. Do not use semi-colons to separate directories; the semi-colons are added automatically when the box is closed.

  ### Note

  This property is also available from the Fortran | Preprocessing property page.

Module Path

Use this property to specify the location of module (`.mod`) files.

For every directory that is added to this property, PVF adds `-module <dir>` to the compilation line, causing the compiler to search each listed directory for modules during compilation.

Note

The first directory in the list is also the module output directory, which is where PVF puts all module files created when the project is built.

There are two ways to add directories to this property:

- Type the information directly into the property page box.

  Use a semi-colon (';') to separate each path.

- Click the ellipsis ('...') button in the property page box to open the *Module Path* dialog box.

  Enter each directory on its own line in this box. Do not use semi-colons to separate entries; the semi-colons are added automatically when the box is closed.

Object File Name

Use of this property depends on whether it is being applied to a file or a project:

- File level: Use this property to set the name of the object file. Setting the name adds the `-o` switch to the compilation line.

  For more information on `-o`, refer to "–o ," on page 198.

- Project level: Use this property to set the location of the object files created by a build.

  To change the default location for the object files, specify a different directory name for this property.

Note

You must append a backslash (\) to the directory path or the value of this property will be interpreted as a file.

Debug Information Format

Use this property to specify whether the compiler should generate debug information and if so, in what format.

- The richest debugging experience is obtained when this option is set to "Full Debug Information (–g)."
- If you are debugging a project built with optimizations, you may want to select "Full Debug Information with Full Optimization (–gopt)." This selection prevents the generation of debug information from affecting optimizations.

For more information on `-g`, refer to "–g," on page 183. For more information on `-gopt`, refer to "–gopt," on page 184.

Optimization

Use this property to select the overall code optimization.

This property can be set to one of the following values:

- **No Optimization** - the default value for Debug configurations.
- **Maximize Speed** - the default value for Release Configurations.
- **Maximize Speed Across the Whole Program**

Note

This property is also available from the Fortran | Optimization property page.

Fortran | Optimization

The following properties are available from the Fortran | Optimization property page.

Optimization

Use this property to select the overall code optimization.

This property can be set to one of the following values:

- **No Optimization** - the default value for Debug configurations.
- **Maximize Speed** - the default value for Release Configurations.
- **Maximize Speed Across the Whole Program**

Note

This property is also available from the Fortran | General property page.

Global Optimizations

Use this property to set the compiler's global optimization level. Setting this property adds one of the `-O` options to the compilation line.

For more information on `-O`, refer to "–O<level> ," on page 197.

Vectorization

Use this property to specify the type of vectorization to perform.

The PVF compilers use the `-Mvect` options to vectorize code that is vectorizable. When you use this property the appropriate option is added to the compilation line.

For more information on `-Mvect`, refer to "Optimization Controls," on page 223.

Inlining

Use this property to enable inlining of certain subprograms.

Setting this property to `Yes` adds the `-Minline` switch to the compilation command line.

For more information on `-Minline`, refer to "Optimization Controls," on page 223.

Use Frame Pointer

Use this property to specify whether to generate code that uses a frame pointer.

Setting this property to `Yes` adds the `-Mframe` switch to the compilation command line and PVF compilers generate code that uses a frame pointer.

Setting this property to `No`, the default, adds the `-Mnoframe` switch to the compilation command line and PVF compilers generate code that does not use frame pointers.

For more information on `-Mframe`, refer to "Optimization Controls," on page 223.

Loop Unroll Count

Use this property to select the appropriate value for unrolling.

Loop unrolling is a common optimization. This property allows you to specify unrolling by two or four. Using this option adds the `-Munroll` option to the compilation line.

For more information on `-Munroll`, refer to "Optimization Controls," on page 223.

Auto-Parallelization

Use this property to auto-parallelize code that is parallelizable. Using this option adds the `-Mconcur` option to the compilation line.

For more information on `-Mconcur`, refer to "Optimization Controls," on page 223.

Fortran | Preprocessing

The following properties are available from the Fortran | Preprocessing Property page.

Preprocess Source File

Use this property to specify whether the compiler should preprocess source files.

Setting this property to `Yes` adds the `-Mpreprocess` switch to the compilation command line.

For more information on `-Mpreprocess`, refer to "Miscellaneous Controls," on page 233.

Additional Include Directories

Use this property to add one or more directories to the compiler's include path.

For every path that is added to this property, PVF adds `-I<path>` to the compilation line.

There are two ways to add directories to this property:

- Type the information directly into the property page box.

Use a semi-colon (';') to separate each directory.

- Click the ellipsis ('...') button in the property page box to open the *Additional Include Directories* dialog box.

  Enter each directory on its own line in this box. Do not use semi-colons to separate directories; the semi-colons are added automatically when the box is closed.

For more information on `-I<path>`, refer to "–I ," on page 186.

Note

This property is also available from the Fortran | General property page.

Ignore Standard Include Path

Use this property to specify whether the preprocessor should ignore the standard include path.

Setting this property to `Yes` adds the `-Mnostdinc` switch to the compilation command line.

For more information on `-Mnostdinc`, refer to "Environment Controls," on page 217.

Preprocessor Definitions

Use this property to add one or more preprocessor definitions to compilation.

For every definition that is added to this property, PVF adds `-D<definition>` to the compilation line.

There are two ways to add definitions to this property:

- Type the information directly into the property page box.

  Use a semi-colon (';') to separate each definition.

  For example, `DEF1;DEF2=2` defines DEF1, and defines DEF2 and initializes it to 2.

- Click the ellipsis ('...') button in the property page box to open the *Preprocessor Definitions* dialog box.

  Enter each definition on its own line in this box. Do not use semi-colons to separate definitions; the semi-colons are added automatically when the box is closed.

For more information on `-D<definition>`, refer to "–D," on page 180.

Undefine Preprocessor Definitions

Use this property to undefine one or more preprocessor definitions.

For every definition that is added to this property, PVF adds `-U<definition>` to the compilation line.

There are two ways to add definitions to this property:

- Type the information directly into the property page box.

Use a semi-colon (';') to separate each definition.

For example, `DEF1;DEF2` undefines DEF1 and DEF2.

- Click the ellipsis ('...') button in the property page box to open the *Undefine Preprocessor Definitions* dialog box.

  Enter each definition on its own line in this box. Do not use semi-colons to separate definitions; the semi-colons are added automatically when the box is closed.

For more information on `-U<definition>`, refer to "–U ," on page 210.

Fortran | Code Generation

The following properties are available from the Fortran | Code Generation property page.

Runtime Library

Use this property to specify the type of runtime libraries to use during linking.

Default: Depends on the project:

- *For executable and static library projects:* multi-threaded static libraries.

  Using this option adds the `-Bstatic` option to the compilation line. This choice corresponds to Microsoft's `/MT` compilation option.

  For more information on `-Bstatic`, refer to "–Bstatic," on page 178.

- *For dynamic-link library projects:* multi-threaded DLL libraries.

  Using this option adds the `-Bdynamic` option to the compilation line. This choice corresponds to Microsoft's `/MD` compilation option.

  For more information on `-Bstatic`, refer to "–Bdynamic," on page 178.

  #### Note

  It is important to keep the type of runtime libraries consistent within a solution. PVF projects that build DLLs should link to the multi-threaded DLL runtime, and projects that link to these PVF DLLs should also use the multi-threaded DLL runtime.

Fortran | Language

The following properties are available from the Fortran | Language property page.

Fortran Dialect

Use this property to select the Fortran dialect to use during compilation.

PVF supports two Fortran language dialects: Fortran 95 and FORTRAN 77. The dialect determines which PGI compiler driver is used during compilation.

- **Default:** The dialect is set to Fortran 95, even for fixed-format `.f` files, and the pgfortran driver is used.
- **Fortran 77:** Use the pgf77 driver. You can select the FORTRAN 77 dialect at the project or file level.

Treat Backslash as Character

Use this property to specify how the compilers should treat the backslash (\) character.

Default: PVF treats the backslash (\) as a regular character.

This default action is equivalent to adding the `-Mbackslash` switch to compilation.

If you want the backslash character to be treated as an escape character, which is how C and C++ compilers handle backslashes, set this property to `No`.

For more information on `-Mbackslash`, refer to "Fortran Language Controls," on page 218.

Extend Line Length

Use this property to extend the line length for fixed-format Fortran files to 132 characters.

Fixed-format Fortran files limit the accepted line length to 72 characters. To extend the line length for these types of files to 132 characters, set this property to `Yes`, which adds the `-Mextend` switch to the PVF compilation line.

For more information on `-Mextend`, refer to "Fortran Language Controls," on page 218.

Process OpenMP Directives

Use this property to enable OpenMP 3.0 language extensions.

Setting this property to `Yes` adds the `-mp` switch to the PVF compilation and link lines.

For more information on `-mp`, refer to "–mp[=all, align,bind,[no]numa] ," on page 196.

MPI

Use this property to enable compilation and linking using the Microsoft MPI headers and libraries.

Setting this property to `Microsoft MPI` adds the `-Mmpi=msmpi` switch to the PVF compilation and link lines.

Enable CUDA Fortran

Use this property to enable CUDA Fortran.

Setting this property to `Yes` adds the `-Mcuda` switch to the PVF compilation and link lines and activiates access to these additional properties:

CUDA Fortran Register Limit
CUDA Fortran Use Fused Multiply-Adds
CUDA Fortran Use Fast Math Library

CUDA Fortran Toolkit
CUDA Fortran Compute Capability
CUDA Fortran Keep Binary
CUDA Fortran Keep Kernel Source
CUDA Fortran Keep PTX
CUDA Fortran Emulation

Important

If you select `Yes` and the additional properties do not appear, click Apply in the Property page dialog.

For more information on `-Mcuda`, refer to "Optimization Controls," on page 223.

CUDA Fortran Register Limit

When Enable CUDA Fortran is set to `Yes`, use this property to specify the number of registers to use on the GPU.

Setting this property to an integer value, n, adds the `-Mcuda=maxregcount:n` switch to the PVF compilation and link lines.

Leaving this property blank indicates no limit to the number of registers to use on the GPU.

For more information on `-Mcuda`, refer to "Optimization Controls," on page 223.

CUDA Fortran Use Fused Multiply-Adds

When Enable CUDA Fortran is set to `Yes`, use this property to control the generation of fused multiply-add (FMA) instructions.

Setting this property to `Yes` enables generation of FMA instructions; this is the default.

Setting this property to `No` adds the `-Mcuda=nofma` switch to the PVF compilation and link lines.

CUDA Fortran Use Fast Math Library

When Enable CUDA Fortran is set to `Yes`, use this property to use routines from the fast math library.

Setting this property to `Yes` adds the `-Mcuda=fastmath` switch to compilation and linking.

CUDA Fortran Toolkit

When Enable CUDA Fortran is set to `Yes`, use this property to specify the version of the CUDA toolkit that is targeted by the compilers.

- **Default:** The compiler selects the default CUDA toolkit version.
- **2.3:** Use version 2.3 of the CUDA toolkit. This selection adds the `-Mcuda=cuda2.3` switch to the PVF compilation and link lines.
- **3.0:** Use version 3.0 of the CUDA toolkit. This selection adds the `-Mcuda=cuda3.0` switch to the PVF compilation and link lines.

Note

Compiling with the CUDA 3.0 toolkit, either by setting the appropriate property as just described or by adding `set CUDAVERSION=3.0` 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 more information on `-Mcuda`, refer to "Fortran Language Controls," on page 218.

CUDA Fortran Compute Capability

When Enable CUDA Fortran is set to `Yes`, use this property to either automatically generate code compatible with all applicable compute capabilities, or to direct the compiler to use a manually-selected set.

Select either `Automatic` or `Manual`.

- **Automatic:** Let the compiler generate code for all applicable compute capabilities. This is the default.
- **Manual:** Choose one or more compute capabilities to target. The compiler generates code for each capability specified.

  If you select `Manual`, then you can select any or all of the following compute capabilities that are described in the next sections.

  CUDA Fortran CC 1.0
  CUDA Fortran CC 1.1
  CUDA Fortran CC 1.2
  CUDA Fortran CC 1.3
  CUDA Fortran CC 2.0

Important

If you select `Manual` and the additional properties do not appear, click Apply in the Property page dialog.

For more information on `-Mcuda`, refer to "Optimization Controls," on page 223.

CUDA Fortran CC 1.0

When Enable CUDA Fortran is set to `Yes` and CUDA Fortran Compute Capability is set to `Manual`, use this property to generate code for CUDA compute capability 1.0.

Setting this property to `Yes` adds the `-Mcuda=cc10` switch to the PVF compilation and link lines.

CUDA Fortran CC 1.1

When Enable CUDA Fortran is set to `Yes` and CUDA Fortran Compute Capability is set to `Manual`, use this property to generate code for CUDA compute capability 1.1.

Setting this property to Yes adds the -Mcuda=cc11 switch to the PVF compilation and link lines.

CUDA Fortran CC 1.2

When Enable CUDA Fortran is set to Yes and CUDA Fortran Compute Capability is set to Manual, use this property to generate code for CUDA compute capability 1.2.

Setting this property to Yes adds the -Mcuda=cc12 switch to the PVF compilation and link lines.

CUDA Fortran CC 1.3

When Enable CUDA Fortran is set to Yes and CUDA Fortran Compute Capability is set to Manual, use this property to generate code for CUDA compute capability 1.3.

Setting this property to Yes adds the -Mcuda=cc13 switch to the PVF compilation and link lines.

CUDA Fortran CC 2.0

When Enable CUDA Fortran is set to Yes and CUDA Fortran Compute Capability is set to Manual, use this property to generate code for CUDA compute capability 2.0.

Setting this property to Yes adds the -Mcuda=cc20 switch to the PVF compilation and link lines.

CUDA Fortran Keep Binary

Use this property to keep the CUDA binary (.bin) file.

Setting this property to Yes adds the -Mcuda=keepbin switch to the PVF compilation and link lines.

For more information on -Mcuda, refer to "Optimization Controls," on page 223.

CUDA Fortran Keep Kernel Source

When Enable CUDA Fortran is set to Yes, use this property to keep the kernel source files.

Setting this property to Yes adds the -Mcuda=keepgpu switch to the PVF compilation and link lines.

For more information on -Mcuda, refer to "Optimization Controls," on page 223.

CUDA Fortran Keep PTX

When Enable CUDA Fortran is set to Yes, use this property to keep the portable assembly (.ptx) file for the GPU code.

Setting this property to Yes adds the -Mcuda=keepptx switch to the PVF compilation and link lines.

For more information on -Mcuda, refer to "Optimization Controls," on page 223.

CUDA Fortran Emulation

When Enable CUDA Fortran is set to Yes, use this property to enable CUDA Fortran emulation mode.

Setting this property to Yes adds the -Mcuda=emu switch to the PVF compilation and link lines.

For more information on `-Mcuda`, refer to "Optimization Controls," on page 223.

Fortran | Floating Point Options

The following properties are available from the Fortran | Floating Point Options property page.

Floating Point Exception Handling

Use this property to enable floating point exceptions.

Setting this property to `Yes` adds the `-Ktrap=fp` option to compilation.

For more information on `-Ktrap`, refer to "–K<flag>," on page 187.

Floating Point Consistency

Use this property to enable relaxed floating point accuracy in favor of speed.

Setting this property to `Yes` adds the `-Mfprelaxed` option to compilation.

For more information on `-Mfprelaxed`, refer to "Optimization Controls," on page 223.

Flush Denormalized Results to Zero

Use this property to specify how to handle denormalized floating point results.

- **Default**: Accepts the default handling of denormalized floating point results.
- **Yes**: Enables SSE flush-to-zero mode using the `-Mflushz` compilation option.
- **No**: Disables SSE flush-to-zero mode using the `-Mnoflushz` compilation option.

For more information on `-M[no]flushz`, refer to "Code Generation Controls," on page 213.

Treat Denormalized Values as Zero

Use this property to specify how to treat denormalized floating point values.

- **Default**: Accept the default treatment of denormalized floating point values.
- **Yes**: Enable the treatment of denormalized floating point values as zero using the `-Mdaz` compilation option.
- **No**: Disable the treatment of denormalized floating point values as zero using the `-Mnodaz` compilation option.

For more information on `-M[no]daz`, refer to "Code Generation Controls," on page 213.

IEEE Arithmetic

Use this option to specify IEEE floating point arithmetic.

- **Default**: Accept the default floating point arithmetic.

- **Yes**: Enable IEEE floating point arithmetic using the `-Kieee` compilation option.
- **No**: Disable IEEE floating point arithmetic using the `-Knoieee` compilation option.

For more information on `-K[no]ieee`, refer to "–K<flag>," on page 187.

Fortran | External Procedures

The following properties are available from the Fortran | External Procedures property page.

Calling Convention

Use this property to specify an alternate Fortran calling convention.

- **Default:** Accept the default calling convention.
- **C By Reference:** Use the CREF calling convention. Adds `-Miface=cref` to compilation. On both Win32 and x64 platforms, no trailing underscores are used with this option. On the x64 platform, this option also causes Fortran externals to be uppercase and lengths of character arguments to be put at the end of the argument list.
- **Unix:** [Win32 platform only] Use the Unix calling convention. Adds `-Miface=unix` to compilation. No trailing underscores are used with this option.

For more information on `-Miface`, refer to "Miscellaneous Controls," on page 233.

String Length Arguments

Use this property to change where string length arguments are placed in the argument list.

- **Default:** Use the calling convention's default placement for passing string length arguments.
- **After Every String Argument:** Lengths of character arguments are placed immediately after their corresponding argument. This option adds `-Miface=mixed_str_len_arg` to compilation.
- **After All Arguments:** Places lengths of character arguments at the end of the argument list. This option adds `-Miface=nomixed_str_len_arg` to the compilation.

Note

The *After Every String Argument* and *After All Arguments* options only have an effect when using the C By Reference calling convention.

For more information on `-Miface`, refer to "Miscellaneous Controls," on page 233.

Case of External Names

Use this property to specify the case used for Fortran external names.

- **Default:** Use the calling convention's default case for external names.
- **Lower Case:** Make Fortran external names lower case. This option adds `-Mnames=lowercase` to the compilation.

- **Upper Case:** Make Fortran external names upper case. This option adds `-Mnames=uppercase` to the compilation.

  Note

  The Lower Case and Upper Case options only have an effect when using the C By Reference calling convention.

Fortran | Target Processors

The properties that are available from the Fortran | Target Processors property page depend on the platform you are using. The platform selection box in the center of the Property Pages dialog indicates the platform: `x64` or `Win32`.

Note

x64 Platform
: You can target multiple processors for optimization on the x64 platform.

Win32 Platform
: You can target only one processor at a time for optimization on the Win32 platform. If you select `Yes` for more than one processor, a compiler error occurs.

The Target Processors properties add the `-tp=<target>` option to compilation. For more information on the `-tp` switch referenced throughout the following descriptions, refer to –tp.

AMD Athlon

Use this property to optimize for AMD Athlon64, AMD Opteron and compatible processors.

x64: Setting this property to `Yes` adds the `-tp=k8-64` switch to the compilation.

Win32: Setting this property to `Yes` adds the `-tp=k8-32` switch to the compilation.

AMD Barcelona

Use this property to optimize for AMD Opteron/Quadcore and compatible processors.

x64: Setting this property to `Yes` adds the `-tp=barcelona-64` switch to the compilation.

Win32: Setting this property to `Yes` adds the `-tp=barcelona-32` switch to the compilation.

AMD Istanbul

Use this property to optimize for AMD Istanbul processor-based systems.

x64: Setting this property to `Yes` adds the `-tp=istanbul-64` switch to the compilation.

Win32: Setting this property to `Yes` adds the `-tp=istanbul-32` switch to the compilation.

AMD Shanghai

Use this property to optimize for AMD Shanghai processor-based systems.

x64: Setting this property to `Yes` adds the `-tp=shanghai-64` switch to the compilation.

Win32: Setting this property to `Yes` adds the `-tp=shanghai-32` switch to the compilation.

Intel Core 2

Use this property to optimize for Intel Core 2 Duo and compatible processors.

x64: Setting this property to `Yes` adds the `-tp=core2-64` switch to the compilation.

Win32: Setting this property to `Yes` adds the `-tp=core2-32` switch to the compilation.

Intel Core i7

Use this property to optimize for Intel Core i7 (Nehalem) processor-based systems.

x64: Setting this property to `Yes` adds the `-tp=nehalem-64` switch to the compilation.

Win32: Setting this property to `Yes` adds the `-tp=nehalem-32` switch to the compilation.

Intel Penryn

Use this property to optimize for Intel Penryn architecture and compatible processors.

x64: Setting this property to `Yes` adds the `-tp=penryn-64` switch to the compilation.

Win32: Setting this property to `Yes` adds the `-tp=penryn-32` switch to the compilation.

Intel Pentium 4

Use this property to optimize for Intel Pentium 4 and compatible processors.

Win32: Setting this property to `Yes` adds the `-tp=p7-32` switch to the compilation.

Generic x86 [Win32 only]

Use this property to optimize for any x86 processor-based system.

x64: N/A

Win32: Setting this property to `Yes` adds the `-tp=px-32` switch to the compilation.

Generic x86-64 [x64 only]

Use this property to optimize for any x86-64 processor-based system.

x64: Setting this property to `Yes` adds the `-tp=px-64` switch to the compilation.

Win32: N/A

Fortran | Target Accelerators

The following properties are available from the Fortran | Target Accelerators property page.

For more information about the PGI Accelerator, refer to Chapter 10, "*Using an Accelerator*". For information on the options in this section, refer to –ta and "Miscellaneous Controls," on page 233.

Target NVIDIA Accelerator

Use this property to select NVIDIA accelerator target.

Setting this property to `Yes` adds the `-ta=nvidia` switch to the compilation line and activates access to these additional properties:

NVIDIA: Register Limit
NVIDIA: Use Fused Multiply-Adds
NVIDIA: Use Fast Math Library
NVIDIA: Use 24-bit Subscript Multiplication
NVIDIA: Synchronous Kernel Launch
NVIDIA: CUDA Toolkit
NVIDIA: Compute Capability
NVIDIA: Keep Kernel Binary
NVIDIA: Keep Kernel Source
NVIDIA: Keep Kernel PTX
NVIDIA: Enable Profiling
NVIDIA: Analysis Only

Important

If you change the value of this property and the displayed properties do not change, be sure to click Apply in the property page dialog box.

NVIDIA: Register Limit

Use this property to specify the number of registers to use on the GPU.

Setting this property to an integer value, n, adds the `-ta=nvidia:maxregcount:n` switch to the PVF compilation and link lines.

Leaving this property blank indicates no limit to the number of registers to use on the GPU.

NVIDIA: Use Fused Multiply-Adds

When Target NVIDIA Accelerator is set to `Yes`, use this property to control the generation of fused multiply-add (FMA) instructions.

Setting this property to `Yes` enables generation of FMA instructions; this is the default.

Setting this property to `No` adds the `-ta=nvidia:nofma` switch to compilation and linking.

NVIDIA: Use Fast Math Library

When Target NVIDIA Accelerator is set to `Yes`, use this property to use routines from the fast math library.

Setting this property to `Yes` adds the `-ta=nvidia:fastmath` switch to compilation and linking.

NVIDIA: Use 24-bit Subscript Multiplication

When Target NVIDIA Accelerator is set to `Yes`, use this property to use 24-bit multiplication for subscripting.

Setting this property to `Yes` adds the `-ta=nvidia:mul24` switch to compilation and linking.

NVIDIA: Synchronous Kernel Launch

When Target NVIDIA Accelerator is set to `Yes`, use this property to wait for each kernel to finish before continuing in the host program.

Setting this property to `Yes` adds the `-ta=nvidia:wait` switch to compilation and linking and overrides the nowait clause. This is the default.

Setting this property to `No` adds the `-ta=nvidia:nowait` switch to compilation and linking.

NVIDIA: CUDA Toolkit

When Target NVIDIA Accelerator is set to `Yes`, use this property to specify the version of the NVIDIA CUDA toolkit that is targeted by the compilers:

- **Default:** The compiler selects the default CUDA toolkit version.
- **2.3:** Use version 2.3 of the CUDA toolkit. This selection adds the `-ta=nvidia:cuda2.3` switch to the PVF compilation and link lines.
- **3.0:** Use version 3.0 of the CUDA toolkit. This selection adds the `-ta=nvidia:cuda3.0` switch to the PVF compilation and link lines.

Note

Compiling with the CUDA 3.0 toolkit, either by setting the appropriate property as just described or by adding `set CUDAVERSION=2.0` 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 more information on `-ta=nvidia`, refer to –ta.

NVIDIA: Compute Capability

When Target NVIDIA Accelerator is set to `Yes`, use this property to either automatically generate code compatible with all applicable compute capabilities, or to direct the compiler to use a manually-selected set.

Select either `Automatic` or `Manual`.

- **Automatic:** Let the compiler generate code for all applicable compute capabilities. This is the default.
- **Manual:** Choose one or more compute capabilities to target. The compiler generates code for each capability specified.

If you select `Manual`, then you can select any or all of the following compute capabilities that are described in the next sections.

NVIDIA: CC 1.0
NVIDIA: CC 1.1
NVIDIA: CC 1.2
NVIDIA: CC 1.3
NVIDIA: CC 2.0

Important

If you select `Manual` and the additional properties do not appear, click Apply in the Property page dialog.

For more information on `-ta=nvidia`, refer to "–ta=nvidia(,nvidia_suboptions),host," on page 204.

NVIDIA: CC 1.0

When Target NVIDIA Accelerator is set to `Yes` and NVIDIA Compute Capability is set to `Manual`, use this property to generate code for NVIDIA compute capability 1.0.

Setting this property to `Yes` adds the `-ta=nvidia:cc10` switch to the PVF compilation and link lines.

NVIDIA: CC 1.1

When Target NVIDIA Accelerator is set to `Yes` and NVIDIA Compute Capability is set to `Manual`, use this property to generate code for NVIDIA compute capability 1.1.

Setting this property to `Yes` adds the `-ta=nvidia:cc11` switch to the PVF compilation and link lines.

NVIDIA: CC 1.2

When Target NVIDIA Accelerator is set to `Yes` and NVIDIA Compute Capability is set to `Manual`, use this property to generate code for NVIDIA compute capability 1.2.

Setting this property to `Yes` adds the `-ta=nvidia:cc12` switch to the PVF compilation and link lines.

NVIDIA: CC 1.3

When Target NVIDIA Accelerator is set to `Yes` and NVIDIA Compute Capability is set to `Manual`, use this property to generate code for NVIDIA compute capability 1.3.

Setting this property to `Yes` adds the `-ta=nvidia:cc13` switch to the PVF compilation and link lines.

NVIDIA: CC 2.0

When Target NVIDIA Accelerator is set to `Yes` and NVIDIA Compute Capability is set to `Manual`, use this property to generate code for NVIDIA compute capability 2.0.

Setting this property to `Yes` adds the `-ta=nvidia:cc20` switch to the PVF compilation and link lines.

NVIDIA: Keep Kernel Binary

When Target NVIDIA Accelerator is set to `Yes`, use this property to keep kernel binary (.bin) files.

Setting this property to `Yes` adds the `-ta=nvidia:keepbin` switch to compilation and linking.

NVIDIA: Keep Kernel Source

When Target NVIDIA Accelerator is set to `Yes`, use this property to keep kernel source files.

Setting this property to `Yes` adds the `-ta=nvidia:keepgpu` switch to compilation and linking.

NVIDIA: Keep Kernel PTX

When Target NVIDIA Accelerator is set to `Yes`, use this property to keep the portable assembly (.ptx) file for the GPU code.

Setting this property to `Yes` adds the `-ta=nvidia:keepptx` switch to compilation and linking.

NVIDIA: Enable Profiling

When Target NVIDIA Accelerator is set to `Yes`, use this property to collect simple timing information for accelerator kernel profiling.

Setting this property to `Yes` adds the `-ta=nvidia:time` switch to compilation and linking.

NVIDIA: Analysis Only

When Target NVIDIA Accelerator is set to `Yes`, use this property to perform loop analysis only; the conpiler does not generate GPU code.

Setting this property to `Yes` adds the `-ta=nvidia:analysis` switch to compilation and linking.

Target Host

Use this property to generate code just for the host if no accelerator is selected. Otherwise, generate PGI Unified Binary Code for host and accelerator.

Setting this property to `Yes` adds the `-ta=host` switch to compilation and linking.

Fortran | Diagnostics

The following properties are available from the Fortran | Diagnostics property page. These properties allow you to add switches to the compilation line that control the amount and type of information that the compiler provides.

For more information on the options referenced in these pages, refer to "Miscellaneous Controls," on page 233 and –Minfo.

Warning Level

Use this property to select the level of diagnostic reporting you want the compiler to use.

There are several levels of the `-Minform` option available through this property. For more information on this option, refer to "Miscellaneous Controls," on page 233.

Generate Assembly

Use this property to generate an assembly file for each compiled source file.

Setting this property to `Yes` adds the `-Mkeepasm` switch to the compilation line.

For more information on `-Mkeepasm`, refer to "Miscellaneous Controls," on page 233.

Annotate Assembly

Use this property to generate assembly files and to annotate the assembly with source code.

Setting this property to `Yes` adds the `-Manno` switch to the compilation line.

For more information on `-Manno`, refer to "Miscellaneous Controls," on page 233.

Accelerator Information

Use this property to generate information about accelerator regions.

Setting this property to `Yes` adds the `-Minfo=accel` switch to the compilation line.

CCFF Information

Use this property to append common compiler feedback format (CCFF) information to object files.

Setting this property to `Yes` adds the `-Minfo=ccff` switch to the compilation line.

Fortran Language Information

Use this property to generate information about Fortran language features.

Setting this property to `Yes` adds the `-Minfo=ftn` switch to the compilation line.

Inlining Information

Use this property to generate information about inlining.

Setting this property to `Yes` adds the `-Minfo=inline` switch to the compilation line.

IPA Information

Use this property to generate information about interprocedural analysis (IPA) optimizations.

Setting this property to `Yes` adds the `-Minfo=ipa` switch to the compilation line.

Loop Intensity Information

Use this property to generate compute intensity information about loops.

Setting this property to `Yes` adds the `-Minfo=intensity` switch to the compilation line.

Loop Optimization Information

Use this property to generate information about loop optimizations.

Setting this property to `Yes` adds the `-Minfo=loop` switch to the compilation line.

LRE Information

Use this property to generate information about loop-carried redundancy (LRE) elimination.

Setting this property to `Yes` adds the `-Minfo=lre` switch to the compilation line.

OpenMP Information

Use this property to generate information about OpenMP.

Setting this property to `Yes` adds the `-Minfo=mp` switch to the compilation line.

Optimization Information

Use this property to generate information about general optimizations.

Setting this property to `Yes` adds the `-Minfo=opt` switch to the compilation line.

Parallelization Information

Use this property to generate information about parallel optimizations.

Setting this property to `Yes` adds the `-Minfo=par` switch to the compilation line.

Unified Binary Information

Use this property to generate information specific to the PGI Unified Binary.

Setting this property to `Yes` adds the `-Minfo=unified` switch to the compilation line.

Vectorization Information

Use this property to generate vectorization information.

Setting this property to `Yes` adds the `-Minfo=vect` switch to the compilation line.

Fortran | Profiling

The following properties are available from the Fortran | Profiling property page so a run can be profiled with the PGI profiler, PGPROF.

Once your application is built, running it generates one or more `pgprof.out` files.

PGPROF is included with PVF. You launch it from the PVF start menu via *Start* | *All Programs* | *PGI Visual Fortran* | *Profiler* | *PGPROF Performance Profiler*.

For quick start information on PGPROF, refer to the "Profile an MPI Application," on page 38. For specific PGPROF documentation, launch PGPROF and open the documentation available from the PGPROF Help menu.

For more information on the `-Mprof` option, refer to "Code Generation Controls," on page 213.

Function-Level Profiling

Use this property to generate code instrumented for function-level profiling.

Setting this property to `Yes` adds the `-Mprof=func` switch to the compiling and linking lines.

Line-Level Profiling

Use this property to generate code instrumented for line-level profiling.

Setting this property to `Yes` adds the `-Mprof=lines` switch to the compiling and linking lines.

MPI

Use this property to access profiled MPI communication libraries.

Note

You must use this property in conjunction with function-level or line-level profiling. Be certain to set one of these properties to `Yes`.

Setting this property to `Microsoft MPI` adds the `-Mprof=msmpi` switch to the PVF compiling and linking lines.

Suppress CCFF Information

Use this property to suppress profiling's default generation of CCFF information.

Setting this property to `Yes` adds the `-Mprof=noccff` switch to the compiling and linking lines.

Enable Limited DWARF

Use this property to generate limited DWARF information which can be used with performance profilers.

Setting this property to `Yes` adds the `-Mprof=dwarf` switch to the compiling and linking lines.

Fortran | Command Line

The following properties are available from the Fortran | Command Line property page.

Command Line

This property page contains two boxes.

- The first box, titled *All options*, is a read-only description of what the compilation line will be. This description is based on the values of the properties set in the Fortran property pages.
- The second box, titled *Additional options*, allows you to specify any other options that you want the compiler to use. Use this box when the option you need is not available through any of the Fortran property pages.

For more information on all the compiler options that are available, refer to Chapter 18, "*Command-Line Options Reference*," on page 173.

Linker Property Pages

This section contains the property pages that are included in the Linker category. The Linker property page category is available for projects that build an executable or a dynamically linked library (DLL).

Linker | General

The following properties are available from the Linker | General property page.

Output File

Use this property to override the default output file name.

Providing the file name and the file's extension is equivalent to using the `-o` switch.

Note

You must provide the file's extension.

For more information on `-o`, refer to "–o ," on page 198.

Additional Library Directories

Use this property to add one or more directories to the library search path.

For every directory path that is added to this property, PVF adds `/LIBPATH:[dir]` to the link line.

There are two ways to add directories to this property:

- Type the information directly into the property page box.

  Use a semi-colon (';') to separate each directory.

- Click the ellipsis ('...') button in the property page box to open the *Additional Library Directories* dialog box.

  Enter each directory on its own line in this box. Do not use semi-colons to separate directories; the semi-colons are added automatically when the box is closed.

Tip

To add directories, use this property. To add libraries, use the *Additional Dependencies* property on the Linker | Input property page.

Export Symbols

Use this property to specify whether the DLL will export symbols. This property is only visible for DLL project types.

Linker | Input

The following properties are available from the Linker | Input property page.

Additional Dependencies

Use this property to specify additional dependencies, such as libraries, to the link line.

There are two ways to add libraries to this property:

- Type the information directly into the property page box.

  ### Note

  Use spaces, not semi-colons, to separate multiple libraries. If the name of a library contains a space, use double quotes around that library name.

- Click the ellipsis ('...') button in the property page box to open the *Additional Dependencies* dialog box.

  Enter each library on its own line in this box.

  ### Note

  If you enter two libraries on the same line in this box, PVF interprets these as a single library whose name contains spaces.

  ### Tip

  When you close this dialog box, review the contents of the property to make sure that any spaces or double quotes automatically added by PVF are appropriate for your project.

Linker | Command Line

The following properties are available from the Linker | Command Line property page.

Command Line

This property page contains two boxes.

- The first box, titled *All options*, is a read-only description of what the link line will be. This value is based on the values of the properties set in the Linker property pages.
- The second box, titled *Additional options* allows you to specify options that you want the linker to use. Use this box when the option you need is not available through any of the Linker property pages.

  For more information on all the compiler options that are available, refer to Chapter 18, "*Command-Line Options Reference*," on page 173.

Librarian Property Pages

This section contains the property pages that are included in the Librarian category. The Librarian property pages are available for projects that build static libraries.

Librarian | General

The following properties are available from the Librarian | General property page.

Output File

Use this property to override the default output file name.

Providing the file name and the file's extension is equivalent to using the `-o` switch.

Note

You must provide the file's extension.

For more information on `-o`, refer to "–o ," on page 198.

Additional Library Directories

Use this property to add one or more directories to the library search path.

For every directory path that is added to this property, PVF adds `/LIBPATH:<dir>` to the link line.

There are two ways to add directories to this property:

- Type the information directly into the property page box.

  Use a semi-colon (';') to separate each directory.

- Click the ellipsis ('...') button in the property page box to open the *Additional Library Directories* dialog box.

  Enter each directory on its own line in this box. Do not use semi-colons to separate directories; the semi-colons are added automatically when the box is closed.

Tip

To add directories, use this property. To add libraries, use the Additional Dependencies property.

Additional Dependencies

Use this property to specify additional dependencies, such as libraries, to the link line.

There are two ways to add libraries to this property:

- Type the information directly into the property page box.

  ### Note

  Use spaces, not semi-colons, to separate multiple libraries. If the name of a library contains a space, use double quotes around that library name.

- Click the ellipsis ('...') button in the property page box to open the *Additional Dependencies* dialog box.

Enter each library on its own line in this box.

Note

If you enter two libraries on the same line in this box, PVF interprets these as a single library whose name contains spaces.

Tip

When you close this dialog box, review the contents of the property to make sure that any spaces or double quotes automatically added by PVF are appropriate for your project.

Librarian | Command Line

The following properties are available from the Librarian | Command Line property page.

Command Line

This property page contains two boxes.

- The first box, titled *All options*, is a read-only description of what the link line will be. This value is based on the values of the properties set in the Librarian property pages.
- The second box, titled *Additional options*, allows you to specify options that you want the librarian to use, even though these options are not available through any of the Librarian property pages.

  For more information on all the compiler options that are available, refer to Chapter 18, "*Command-Line Options Reference*," on page 173.

Resources Property Page

This section contains the property pages that are included in the Resources category.

Resources | Command Line

The following properties are available from the Resources | Command Line property page.

Command Line

Use this property to add options to the Resource compiler's command line.

PVF's support of resources is somewhat limited at this time and the property pages in this category reflect that. To add options to the Resource compiler's command line, use the *Additional options* box on this property page.

Build Events Property Page

This section contains the property pages that are included in the Build Events category. Build events include three types of events: Pre-Build, Pre-Link, and Post-Build.

The Build Events property pages provide an opportunity to specify actions, in addition to compiling and linking, that you want to have happen during the process of a build.

Build Event

The name of the build event describes when the event will be fired.

- The Pre-Build Event is run before a build starts.
- The Pre-Link Event is run after compilation but before linking.
- The Post-Build Event is run after the build completes.

> **Note**
>
> Build events will not be run if a project is up-to-date.

The properties for a build event are the same for all three types of build events.

Command Line

Use this property to specify the command line that the build tool will run.

This property is at the core of the build event. For example, to add a time stamp to a build, you could use `time /t` as the build event's command line.

Description

Use this property to provide feedback to the Output window. The contents of the Description property is echoed to the Output window when this event is fired.

Excluded From Build

Use this property to specify whether this build event should be excluded from the build for the current configuration.

Custom Build Step Property Page

This section contains the property pages that are included in the Custom Build Step category.

You can define a custom build step either for a project or for an individual file. Custom build steps can only be defined for files that are not Fortran or resource files.

Custom Build Step | General

The following properties are available from the Custom Build Step | General property page.

Command Line

Use this property to specify the command line that the build tool will run. This property is at the core of the custom build step.

Description

Use this property to provide feedback to the Output window. The contents of the Description property is echoed to the Output window when the custom build step runs.

Outputs

Use this property to specify the files generated by the custom build step.

Use semi-colons (';') to separate multiple output files.

Note

When a custom build step is specified at the file-level, this property must be non-empty or the custom build step will be skipped.

Additional Dependencies

Use this property to specify any additional input files to use for the custom build.

Note

The custom build step is run when an additional dependency is out of date.

There are two ways to add files to this property:

- Type the information directly into the property page box.

  Use a semi-colon (';') to separate each directory.

- Click the ellipsis ('...') button in the property page box to open the *Additional Dependencies* dialog box.

  Enter each file on its own line in this box. Do not use semi-colons to separate directories; the semi-colons are added automatically when the box is closed.

Chapter 24. PVF Build Macros

PVF implements a subset of the build macros supported by Visual C++ along with a few PVF-specific macros. The macro names are not case-sensitive, and they should be usable in any string field in a property page. Unless otherwise noted, macros that evaluate to directory names end with a trailing backslash ('\').

In general these items can only be changed if there is an associated PVF project or file property. For example, `$(VCInstallDir)` cannot be changed, while `$(IntDir)` can be changed by modifying the General | Intermediate Directory property.

Table 24.1 lists the build macros that PVF supports:

Table 24.1. PVF Build Macros

Macro Name	Description
$(Configuration)	The name of the current project configuration (for example, "Debug").
$(ConfigurationName)	The name of the current project configuration (for example, "Debug").
$(ConfigurationType)	The type of the current project configuration - one of the following: “Application” “StaticLibrary” “DynamicLibrary”
$(DevEnvDir)	The installation directory of Visual Studio.
$(InputDir)	The directory of the input file. If the project is the input, then this macro is equivalent to $(ProjectDir).
$(InputExt)	The file extension of the input file, including the ‘.’ before the file extension. If the project is the input, then this macro is equivalent to $(ProjectExt).
$(InputFileName)	The file name of the input file. If the project is the input, then this macro is equivalent to $(ProjectFileName).
$(InputName)	The base name of the input file. If the project is the input, then this macro is equivalent to $(ProjectName).
$(InputPath)	The full path name of the input file. If the project is the input, then this macro is equivalent to $(ProjectPath).

Macro Name	Description
$(IntDir)	The path to the directory for intermediate files, relative to the project directory, as set by the Intermediate Directory property.
$(OpenToolsDir)	[PVF only]. The location of the Open Tools installation directory, including files needed for building Microsoft Windows applications for both 32-bit and 64-bit environments.
$(OutDir)	The path to the directory for output files, relative to the project directory, as set by the Output Directory property.
$(OutputPath)	The path to the directory for output files, relative to the project directory, as set by the Output Directory property.
$(OutputType)	The type of the current project output - one of the following: “exe” “staticlibrary” “library”
$(PGITools32Dir)	[PVF only]. The location of the active PGI toolset for 32-bit targets. This directory is the parent of `bin`, `lib`, and `include` directories containing executables, libraries, and include files for the PGI development environment.
$(PGIToolsDir)	[PVF only]. The location of the active PGI toolset for 64-bit targets. This directory is the parent of `bin`, `lib`, and `include` directories containing executables, libraries, and include files for the PGI development environment.
$(Platform)	The name of the current project platform (for example, "x64").
$(PlatformArchitecture)	The name of the current project platform architecture. For Win32: **32** For x64: **64**
$(PlatformName)	The name of the current project platform (for example, "x64").
$(PlatformShortName)	The description of the architecture ABI for the current project platform. For Win32: **x86** For x64: **amd64**
$(ProjectDir)	The directory of the project.
$(ProjectExt)	The file extension of the project file, including the ‘.’ before the file extension.
$(ProjectFileName)	The file name of the project file.
$(ProjectName)	The base name of the project.
$(ProjectPath)	The full path name of the project.
$(SolutionDir)	The directory of the solution.
$(SolutionExt)	The file extension of the solution file, including the ‘.’ before the file extension.

Macro Name	Description
$(SolutionFileName)	The file name of the solution file.
$(SolutionName)	The base name of the solution.
$(SolutionPath)	The full path name of the solution.
$(TargetDir)	The directory of the primary output file of the build.
$(TargetExt)	The file extension of the primary output file of the build, including the '.' before the file extension.
$(TargetFileName)	The file name of the primary output file of the build.
$(TargetPath)	The full path name of the primary output file of the build.
$(VCInstallDir)	The Visual C++ installation directory. If Visual C++ is not installed, this macro may evaluate to a directory that does not exist.
$(VSInstallDir)	The Visual Studio installation directory.

Chapter 25. Fortran Module/Library Interfaces for Windows

PGI Visual Fortran provides access to a number of libraries that export C interfaces by using Fortran modules. PVF uses this mechanism to support the Win32 API and Unix/Linux portability libraries. This chapter describes the Fortran module library interfaces that PVF 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 10.0-0 release, look for your files in this location:

```
C:/Program Files/PGI/win32/10.0-0/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 25.1. Fortran Data Type Mappings

Windows Data Type	Fortran Data Type
BOOL	LOGICAL(4)
BYTE	BYTE
CHAR	CHARACTER
SHORT, WORD	INTEGER(2)
DWORD, INT, LONG	INTEGER(4)

Windows Data Type	Fortran Data Type
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 169.

Using DFLIB and DFPORT

PVF includes Fortran module interfaces to libraries supporting some standard C library and Unix/Linux system call functionality. These functions are provided by the `DFLIB` and `DFPORT` modules. To utilize these modules, add the appropriate `USE` statement:

```
use dflib
```

```
use dfport
```

DFLIB

The following table 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 363.

Table 25.2. 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.
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.

Routine	Result	Description
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.

DFPORT

The following table 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 363.

Table 25.3. 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
```

For information on the arguments and functionality of a given routine, refer to Table 1.1, "PVF Win32 API Module Mappings". The 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.

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 25.4. 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
GetPath
GetPolyFillMode
GetRegionData
GetStockObject
GetSystemPaletteUse
GetTextCharset
GetTextExtentExPoint
GetTextExtentPoint32
GetTextMetrics
GetWindowExtEx
GetWorldTransform
InvertRgn
LPtoDP
MoveToEx
OffsetViewportOrgEx
PatBlt
PlayEnhMetaFile
PlayMetaFileRecord
PolyBezierTo
Polyline
PolyPolyline
PtVisible
RectInRegion
RemoveFontResource
ResizePalette
SaveDC
SelectClipPath
SelectPalette
SetBitmapBits
SetBkMode
SetColorAdjustment
SetDCPenColor
SetDIBits
SetGraphicsMode
GetOutlineTextMetrics
GetPixel
GetRandomRgn
GetRgnBox
GetStretchBltMode
GetTextAlign
GetTextCharsetInfo
GetTextExtentExPointI
GetTextExtentPointI
GetViewportExtEx
GetWindowOrgEx
GradientFill
LineDD
MaskBlt
OffsetClipRgn
OffsetWindowOrgEx
PathToRegion
PlayEnhMetaFileRecord
PlgBlt
PolyDraw
PolylineTo
PolyTextOut
RealizePalette
RectVisible
RemoveFontResourceEx
RestoreDC
ScaleViewportExtEx
SelectClipRgn
SetAbortProc
SetBitmapDimensionEx
SetBoundsRect
SetColorSpace
SetDeviceGammaRamp
SetDIBitsToDevice
SetICMMode
GetPaletteEntries
GetPixelFormat
GetRasterizerCaps
GetROP2
GetSystemPaletteEntries
GetTextCharacterExtra
GetTextColor
GetTextExtentPoint
GetTextFace
GetViewportOrgEx
GetWinMetaFileBits
IntersectClipRect
LineTo
ModifyWorldTransform
OffsetRgn
PaintRgn
Pie
PlayMetaFile
PolyBezier
Polygon
PolyPolygon
PtInRegion
Rectangle
RemoveFontMemResourceEx
ResetDC
RoundRect
ScaleWindowExtEx
SelectObject
SetArcDirection
SetBkColor
SetBrushOrgEx
SetDCBrushColor
SetDIBColorTable
SetEnhMetaFileBits
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
ExitThread
FatalAppExit
FileTimeToDosDateTime
FileTimeToSystemTime
FillConsoleOutputCharacter
FindActCtxSectionString
FindClose
FindFirstChangeNotification
FindFirstFileEx
FindFirstVolumeMountPoint
FindNextFile
FindNextVolumeMountPoint
FindResourceEx
FindVolumeMountPointClose
FlsFree
FlsSetValue
FlushFileBuffers
FlushViewOfFile
FreeConsole
FreeLibrary
FreeResource
GenerateConsoleCtrlEvent
GetBinaryType
GetCommConfig
GetCommModemStatus
GetCommState
GetCompressedFileSize
GetConsoleAlias
GetConsoleAliasesLength
GetConsoleAliasExesLength
GetConsoleCursorInfo
GetConsoleFontSize
GetConsoleOutputCP
GetConsoleScreenBufferInfo
ExitProcess
ExpandEnvironmentStrings
FatalExit
FileTimeToLocalFileTime
FillConsoleOutputAttribute
FindActCtxSectionGuid
FindAtom
FindCloseChangeNotification
FindFirstFile
FindFirstVolume
FindNextChangeNotification
FindNextVolume
FindResource
FindVolumeClose
FlsAlloc
FlsGetValue
FlushConsoleInputBuffer
FlushInstructionCache
FormatMessage
FreeEnvironmentStrings
FreeLibraryAndExitThread
FreeUserPhysicalPages
GetAtomName
GetCommandLine
GetCommMask
GetCommProperties
GetCommTimeouts
GetComputerName
GetConsoleAliases
GetConsoleAliasExes
GetConsoleCP
GetConsoleDisplayMode
GetConsoleMode
GetConsoleProcessList
GetConsoleSelectionInfo

GetConsoleTitle
GetConsoleWindow
GetCurrentActCtx
GetCurrentConsoleFont
GetCurrentDirectory
GetCurrentProcess
GetCurrentProcessId
GetCurrentProcessorNumber
GetCurrentThread
GetCurrentThreadId
GetDefaultCommConfig
GetDevicePowerState
GetDiskFreeSpace
GetDiskFreeSpaceEx
GetDllDirectory
GetDriveType
GetEnvironmentStrings
GetEnvironmentVariable
GetExitCodeProcess
GetExitCodeThread
GetFileAttributes
GetFileAttributesEx
GetFileInformationByHandle
GetFileSize
GetFileSizeEx
GetFileTime
GetFileType
GetFirmwareEnvironmentVariable
GetFullPathName
GetHandleInformation
GetLargePageMinimum
GetLargestConsoleWindowSize
GetLastError
GetLocalTime
GetLogicalDrives
GetLogicalDriveStrings
GetLogicalProcessorInformation
GetLongPathName
GetMailslotInfo
GetModuleFileName
GetModuleHandle
GetModuleHandleEx
GetNamedPipeHandleState
GetNamedPipeInfo
GetNativeSystemInfo
GetNumaAvailableMemoryNode
GetNumaHighestNodeNumber
GetNumaNodeProcessorMask
GetNumaProcessorNode
GetNumberOfConsoleInputEvents
GetNumberOfConsoleMouseButtons
GetOverlappedResult
GetPriorityClass
GetPrivateProfileInt
GetPrivateProfileSection
GetPrivateProfileSectionNames
GetPrivateProfileString
GetPrivateProfileStruct
GetProcAddress
GetProcessAffinityMask
GetProcessHandleCount
GetProcessHeap
GetProcessHeaps
GetProcessId
GetProcessIdOfThread
GetProcessIoCounters
GetProcessPriorityBoost
GetProcessShutdownParameters
GetProcessTimes
GetProcessVersion

GetProcessWorkingSetSize
GetProfileInt
GetProfileString
GetShortPathName
GetStdHandle
GetSystemFirmwareTable
GetSystemRegistryQuota
GetSystemTimeAdjustment
GetSystemWindowsDirectory
GetTapeParameters
GetTapeStatus
GetTempPath
GetThreadId
GetThreadPriority
GetThreadSelectorEntry
GetTickCount
GetVersion
GetVolumeInformation
GetVolumePathName
GetWindowsDirectory
GlobalAddAtom
GlobalCompact
GlobalFindAtom
GlobalFlags
GlobalGetAtomName
GlobalLock
GlobalMemoryStatusEx
GlobalSize
GlobalUnlock
GlobalWire
HeapCompact
HeapDestroy
HeapLock
HeapReAlloc
HeapSize
GetProcessWorkingSetSizeEx
GetProfileSection
GetQueuedCompletionStatus
GetStartupInfo
GetSystemDirectory
GetSystemInfo
GetSystemTime
GetSystemTimeAsFileTime
GetSystemWow64Directory
GetTapePosition
GetTempFileName
GetThreadContext
GetThreadIOPendingFlag
GetThreadPriorityBoost
GetThreadTimes
GetTimeZoneInformation
GetVersionEx
GetVolumeNameForVolumeMountPoint
GetVolumePathNamesForVolumeName
GetWriteWatch
GlobalAlloc
GlobalDeleteAtom
GlobalFix
GlobalFree
GlobalHandle
GlobalMemoryStatus
GlobalReAlloc
GlobalUnfix
GlobalUnWire
HeapAlloc
HeapCreate
HeapFree
HeapQueryInformation
HeapSetInformation
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
SetFileApisToANSI
SetFileApisToOEM
SetFileAttributes
SetFilePointer
SetFilePointerEx
SetFileShortName
SetFileTime
SetFileValidData
SetFirmwareEnvironmentVariable
SetHandleCount
SetHandleInformation
SetInformationJobObject
SetLastError
SetLocalTime
SetMailslotInfo
SetMessageWaitingIndicator
SetNamedPipeHandleState
SetPriorityClass
SetProcessAffinityMask
SetProcessPriorityBoost
SetProcessShutdownParameters
SetProcessWorkingSetSize
SetProcessWorkingSetSizeEx
SetStdHandle
SetSystemTime
SetSystemTimeAdjustment
SetTapeParameters
SetTapePosition
SetThreadAffinityMask
SetThreadContext
SetThreadExecutionState
SetThreadIdealProcessor
SetThreadPriority
SetThreadPriorityBoost
SetThreadStackGuarantee
SetTimerQueueTimer
SetTimeZoneInformation
SetUnhandledExceptionFilter
SetupComm
SetVolumeLabel
SetVolumeMountPoint
SetWaitableTimer
SignalObjectAndWait
SizeofResource
Sleep
SleepEx
SuspendThread
SwitchToFiber
SwitchToThread
SystemTimeToFileTime
SystemTimeToTzSpecificLocalTime
TerminateJobObject
TerminateProcess
TerminateThread
TlsAlloc
TlsFree
TlsGetValue
TlsSetValue
TransactNamedPipe
TransmitCommChar
TryEnterCriticalSection
TzSpecificLocalTimeToSystemTime
UnhandledExceptionFilter
UnlockFile
UnlockFileEx
UnmapViewOfFile
UnregisterWait
UnregisterWaitEx
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 26. 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 Chapter 6, "*Using Command Line Options*".

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, see Chapter 6, "*Using Command Line Options*".

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 specified by the environment variables `$TMP` or `$TMPDIR` 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

N

O

P

R

S

T

U

V

W