PGI® COMPILERS & TOOLS

DEBUGGER USER'S GUIDE

Version 2017

NVIDIA

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

PREFACE

This guide describes how to use the PGI debugger to debug serial and parallel applications built with PGI Fortran, C, and C++ compilers for X86, AMD64 and Intel 64 processor-based systems. It contains information about how to use the debugger, as well as detailed reference information on commands and its graphical interface.

Intended Audience

This guide is intended for application programmers, scientists and engineers proficient in programming with the Fortran, C, and/or C++ languages. The PGI tools are available on a variety of operating systems for the X86, AMD64, and Intel 64 hardware platforms. This guide assumes familiarity with basic operating system usage.

Documentation

All documentation for PGI compilers and tools is available online at http://www.pgroup.com/resources/docs.htm

Compatibility and Conformance to Standards

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

For further information, refer to the following:

- *American National Standard Programming Language FORTRAN*, ANSI X3. -1978 (1978).
- *ISO/IEC 1539-1 : 1991, Information technology – Programming Languages – Fortran,* Geneva, 1991 (Fortran 90).
- *ISO/IEC 1539-1 : 1997, Information technology – Programming Languages – Fortran,* Geneva, 1997 (Fortran 95).
- *ISO/IEC 1539-1 : 2004, Information technology – Programming Languages – Fortran,* Geneva, 2004 (Fortran 2003).
- *ISO/IEC 1539-1 : 2010, Information technology – Programming Languages – Fortran,* Geneva, 2010 (Fortran 2008).

- *Fortran 95 Handbook Complete ISO/ANSI Reference*, Adams et al, The MIT Press, Cambridge, Mass, 1997.
- *The Fortran 2003 Handbook*, Adams et al, Springer, 2009.
- *OpenMP Application Program Interface*, Version 3.1, July 2011, http://www.openmp.org.
- *Programming in VAX Fortran*, Version 4.0, Digital Equipment Corporation (September, 1984).
- *IBM VS Fortran*, IBM Corporation, Rev. GC26-4119.
- Military Standard, Fortran, DOD Supplement to American National Standard Programming Language Fortran, ANSI x.3-1978, MIL-STD-1753 (November 9, 1978).
- *American National Standard Programming Language C*, ANSI X3.159-1989.
- ISO/IEC 9899:1999, Information technology – Programming Languages – C, Geneva, 1999 (C99).
- ISO/IEC 9899:2011, Information Technology – Programming Languages – C, Geneva, 2011 (C11).
- ISO/IEC 14882:2011, Information Technology – Programming Languages – C++, Geneva, 2011 (C++11).

Organization

The PGI Debugger User's Guide contains these thirteen sections describing the PGI symbolic debugger for Fortran, C, C++ and assembly language programs.

Getting Started
contains information on how to start using the debugger, including a description of how to build a program for debug and how to invoke the debugger.

The Graphical User Interface
describes how to use the debugger's graphical user interface (GUI).

Command Line Options
describes the debugger's command-line options.

Command Language
provides detailed information about the debugger's command language, which can be used from the command-line user interface or from the Command tab of the graphical user interface.

Command Summary
provides a brief summary table of the PGI debugger commands with a brief description of the command as well as information about the category of command use.

Assembly-Level Debugging
contains information on assembly-level debugging; basic debugger operations, commands, and features that are useful for debugging assembly code; and how to access registers.

Sourec-Level Debugging
contains information on language-specific issues related to source debugging.

Platform-Specific Features
contains platform-specific information as it relates to debugging.

Parallel Debugging Overview
contains an overview of the debugger's parallel debugging capabilities.

Parallel Debugging with OpenMP
describes the debugger's parallel debugging capabilities and how to use them with OpenMP.

Parallel Debugging with MPI
describes the debugger's parallel debugging capabilities and how to use them with MPI.

Parallel Debugging of Hybred Applications
describes the debugger's parallel debugging capabilities and how to use them with hybrid applications.

Command Reference
provides reference information about each of the debugger's commands, organized by area of use.

Conventions

This guide uses the following conventions:

italic
is used for emphasis.

`Constant Width`
is used for filenames, directories, arguments, options, examples, and for language statements in the text, including assembly language statements.

Bold
is used for commands.

[item1]
in general, square brackets indicate optional items. In this case item1 is optional. In the context of p/t-sets, square brackets are required to specify a p/t-set.

{ 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 a reduced fixed point size.

`C/C++`
C/C++ language statements are shown in the test of this guide using a reduced fixed point size.

The PGI compilers and tools are supported on a wide variety of Linux, macOS and Windows operating systems running on 64-bit x86-compatible processors, and on Linux running on OpenPOWER processors. (Currently, the PGI debugger is supported on

x86-64/x64 only.) See the Compatibility and Installation section on the PGI website for a comprehensive listing of supported platforms.

Support for 32-bit development was deprecated in PGI 2016 and is no longer available as of the PGI 2017 release. PGI 2017 is only available for 64-bit operating systems and does not include the ability to compile 32-bit applications for execution on either 32- or 64-bit operating systems.

Terminology

If there are terms in this guide with which you are unfamiliar, PGI provides a glossary of terms which you can access at www.pgroup.com/support/definitions.htm

Related Publications

The following documents contain additional information related to the X86 architecture and the compilers and tools available from The Portland Group.

- PGI Fortran Reference Manual describes the FORTRAN 77, Fortran 90/95, Fortran 2003, and HPF statements, data types, input/output format specifiers, and additional reference material related to the use of PGI Fortran compilers.
- System V Application Binary Interface Processor Supplement by AT&T UNIX System Laboratories, Inc. (Prentice Hall, Inc.).
- FORTRAN 95 HANDBOOK, Complete ANSI/ISO Reference (The MIT Press, 1997).
- Programming in VAX Fortran, Version 4.0, Digital Equipment Corporation (September, 1984).
- IBM VS Fortran, IBM Corporation, Rev. GC26-4119.
- The C Programming Language by Kernighan and Ritchie (Prentice Hall).
- C: A Reference Manual by Samuel P. Harbison and Guy L. Steele Jr. (Prentice Hall, 1987).
- The Annotated C^{++} Reference Manual by Margaret Ellis and Bjarne Stroustrup, AT&T Bell Laboratories, Inc. (Addison-Wesley Publishing Co., 1990)
- PGI Compiler User's Guide, PGI Reference Manual, PGI Release Notes, FAQ, Tutorials, http://www.pgroup.com/
- MPI-CH http://www.unix.mcs.anl.gov/MPI/mpich/
- OpenMP http://www.openmp.org

Chapter 1. GETTING STARTED

The PGI debugger is a symbolic debugger for Fortran, C, C++ and assembly language programs. It provides debugger features, such as execution control using breakpoints, single-stepping, and examination and modification of application variables, memory locations, and registers.

The debugger supports debugging of certain types of parallel applications:

- Multi-threaded and OpenMP applications.
- MPI applications.
- Hybrid applications, which use multiple threads or OpenMP as well as multiple MPI processes.

Multi-threaded and OpenMP applications may be run using more threads than the available number of CPUs, and MPI applications may allocate more than one process to a cluster node. The debugger supports debugging the listed types of applications regardless of how well the number of threads match the number of CPUs or how well the number of processes match the number of cluster nodes.

1.1. Definition of Terms

Throughout this manual we use several debugging-specific terms. The *program* is the executable being debugged. The *platform* is the combination of the operating system and processors(s) on which the program runs. The *program architecture* is the platform for which the program was built.

Remote debugging introduces a few more terms. *Remote debugging* is the process of running the debugger on one system (the *client*) and using it to debug a program running on a different system (the *server*). *Local debugging*, by contrast, occurs when the debugger and program are running on the same system. A *connection* is the set of information the debugger needs to begin debugging a program. This information always includes the program name and whether debugging will be local or remote.

Additional terms are defined as needed. Terminology specific to parallel debugging is introduced in Parallel Debugging Overview.

1.2. Building Applications for Debug

To build a program for debug, compile with the `-g` option. With this option, the compiler generates information about the symbols and source files in the program and includes it in the executable file. The option `-g` also sets the compiler optimization to level zero (no optimization) unless you specify optimization options such as `-O`, `-fast`, or `-fastsse` on the command line. Optimization options take effect whether they are listed before or after `-g` on the command line.

1.2.1. Debugging Optimized Code

Programs built with `-g` and optimization levels higher than `-O0` can be debugged, but due to transformations made to the program during optimization, source-level debugging may not be reliable. Assembly-level debugging (e.g., accessing registers, viewing assembly code, etc.) is reliable, even with optimized code. Programs built without `-g` can be debugged; however, information about types, local variables, arguments and source file line numbers are not available. For more information on assembly-level debugging, refer to Assembly-Level Debugging.

In programs built with both `-g` and optimization levels higher than `-O0`, some optimizations may be disabled or otherwise affected by the `-g` option, possibly changing the program behavior. An alternative option, `-gopt`, can be used to build programs with full debugging information, but without modifying program optimizations. Unlike `-g`, the `-gopt` option does not set the optimization to level zero.

1.2.2. Building for Debug on Windows

To build an application for debug on Windows platforms, applications must be linked with the –g option as well as compiled with –g. This process results in the generation of debug information stored in a '`.dwf`' file and a '`.pdb`' file. The PGI compiler driver should always be used to link applications; except for special circumstances, the linker should not be invoked directly.

1.3. User Interfaces

The debugger includes both a command-line interface (CLI) and a graphical user interface (GUI).

1.3.1. Command Line Interface (CLI)

Text commands are entered one line at a time through the command-line interface. A number of command-line options can be used when launching the debugger.

For information on these options and how they are interpreted, refer to Command Line Options, Command Language, and Command Reference.

1.3.2. Graphical User Interface

The GUI, the default user interface, supports command entry through a point-and-click interface, a view of source and assembly code, a full command-line interface panel, and several other graphical elements and features. There may be minor variations in the appearance of the GUI from system to system, depending on the type of display hardware available, the settings for various defaults and the window manager used. Except for differences caused by those factors, the basic interface remains the same across all systems.

For more information on the GUI, refer to The Graphical User Interface.

1.4. Co-installation Requirements

There are co-installation requirements for the PGI debugger.

1.4.1. Java Virtual Machine

The debugger GUI depends on the Java Virtual Machine (JVM) which is part of the Java Runtime Environment (JRE). The debugger requires that the JRE be a specific minimum version or above.

Command-line mode debugging does not require the JRE.

Linux and macOS

When PGI software is installed on Linux or macOS, the version of Java required by the debugger is also installed. The debugger uses this version of Java by default. You can override this behavior in two ways: set your PATH to include a different version of Java; or, set the PGI_JAVA environment variable to the full path of the Java executable. The following example uses a bash command to set PGI_JAVA:

```
$ export PGI_JAVA=/home/myuser/myjava/bin/java
```

Windows

If an appropriately-versioned JRE is not already on your Windows system, the PGI software installation process installs it for you. The PGI command shell and Start menu links are automatically configured to use the JRE. If you choose to skip the JRE-installation step or want to use a different version of Java to run the debugger, then set your PATH to include the Java bin directory or use the PGI_JAVA environment variable to specify the full path to the java executable.

The command-line mode debugger does not require the JRE.

1.5. Start Debugging

You can start debugging a program right away by launching the PGI debugger and giving it the program name. For example, to load `your_program` into the debugger, do:

```
$ pgdbg your_program
```

Now you are ready to set breakpoints and start debugging.

You can also launch the debugger without a program. Once the debugger is up, use the Connections tab to specify the program to debug. To load the specified program into the debugger, use the Connections tab's Open button.

1.6. Program Load

When the debugger loads a program, it reads symbol information from the executable file and then loads the application into memory. For large applications this process can take a few moments.

1.7. Initialization Files

An initialization file can be useful for defining common aliases, setting breakpoints, and for other startup commands. If an initialization file named `.pgdbgrc` exists in the current directory or in your home directory, as defined by the environment variable HOME, the debugger opens this file when it starts up and executes the commands in it.

If an initialization file is found in the current directory, then the initialization file in the home directory, if there is one, is ignored. However, a `script` command placed in the initialization file can be used to execute the initialization file in the home directory or any other file.

1.8. Program Architecture

The debugger supports debugging 64-bit programs.

Chapter 2.
THE GRAPHICAL USER INTERFACE

The debugger's default user interface is a graphical user interface or GUI. There may be minor variations in the appearance of the GUI from host to host, depending on the type of display hardware available, the settings for various defaults and the window manager used. Except for differences caused by those factors, the basic interface remains the same across all systems.

2.1. Main Components

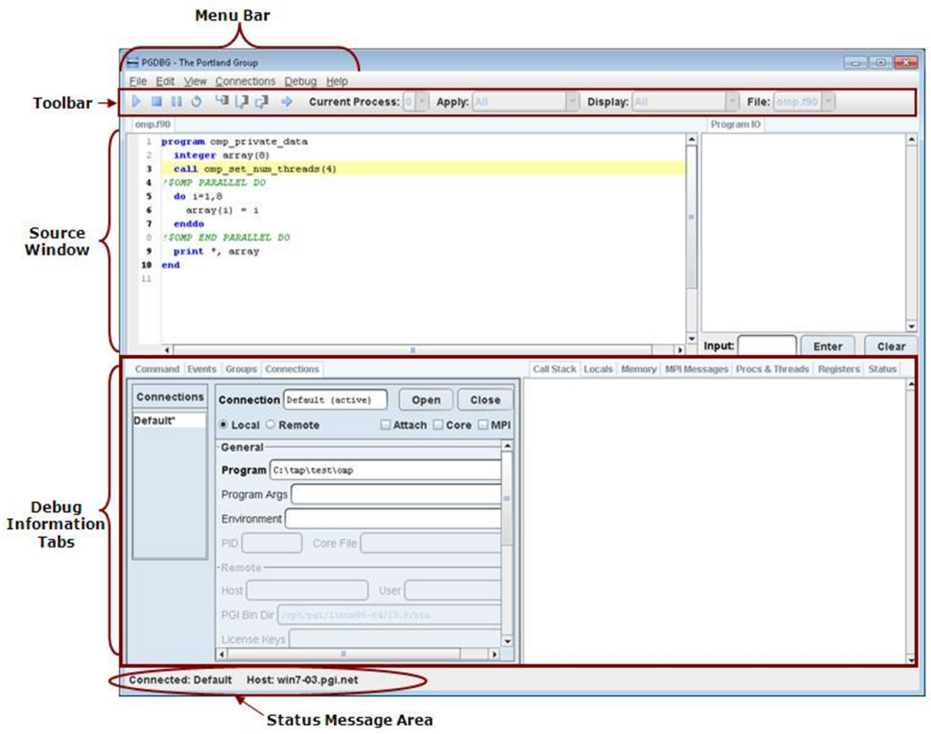


Figure 1 Default Appearance of the Debugger GUI

Figure 1 shows the debugger GUI as it appears when it is invoked for the first time.

The GUI can be resized according to the conventions of the underlying window manager. Changes in window size and other settings are saved and used in subsequent invocations of the debugger. To prevent changes to the default settings from being saved, uncheck the Save Settings on Exit item on the Edit menu.

As the illustration shows, the GUI is divided into several areas: the menu bar, main toolbar, source window, program I/O window, and debug information tabs.

The source window and all of the debug information tabs are dockable tabs. A dockable tab can be separated from the main window by either double-clicking the tab or dragging the tab off the main window. To return the tab to the main window, double-click it again or drag it back onto the main window. You can change the placement of any dockable tab by dragging it from one location to another. Right-click on a dockable tab to bring up a context menu with additional options, including and option to close the tab. To reopen a closed tab, use the View menu. To return the GUI to its original state, use the Edit menu's *Restore Default Settings...* option.

The following sections explain the parts of the GUI and how they are used in a debug session.

2.2. Source Window

The source window, illustrated in Figure 2, displays the source code for the current location. Use the source window to control the debug session, step through source files, set breakpoints, and browse source code.

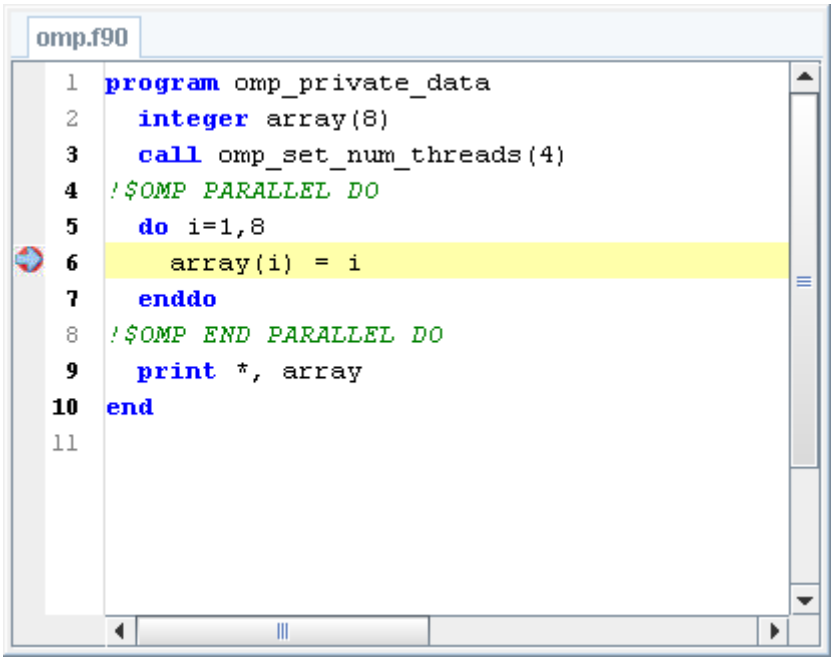


Figure 2 Source Window

The source window contains a number of visual aids that allow you to know more about the execution of your code. The following sections describe these features.

2.2.1. Source and Assembly Displays

Source code debugging is the default unless source information is unavailable in which case debugging will be shown using disassembly. When debugging in source code, use the View | Show Assembly menu option to switch to assembly-level debugging. When disassembly is shown, use options on the View menu to toggle on or off the display of source code, assembly addresses, and bytecode. Source or assembly is always shown for the current process or thread.

Source code line numbers are shown, when known, in their own column to the left of the source or assembly display. A grayed-out line number indicates a non-executable source line. Some examples of non-executable source lines are comments, non-applicable preprocessed code, some routine prologues, and some variable declarations. Non-executable source lines can also exist for otherwise executable code when a program is compiled at certain optimization levels. Breakpoints and other events cannot be set on non-executable lines.

To the left of the line numbers is another column called the gutter. The gutter is where the program counter and debug events like breakpoints are shown. The program counter is represented by a blue arrow and shows where program execution is during a debug session. Breakpoints can be set on executable source lines and any assembly line just by left clicking in the gutter. A red sphere will appear at the line where the breakpoint was set. To delete a breakpoint, either left click on any red breakpoint sphere or right click and select the option from the context menu. By right clicking in the gutter at a place without an existing breakpoint, the context menu will pop up and provide the option of setting a hit count breakpoint. After selecting this option, a dialog will appear for configuring the hit count number and break condition, i.e. break when the hit count is equal to, greater than or a multiple of the specified number.

2.2.2. Source Window Context Menu

The source window supports a context menu that provides convenient access to commonly used features. Right-click in the source window to bring up this context menu. If text is selected when the context menu opens, the selection is used by the context menu options.

In the example in Figure 3, the variable `array(i)` is highlighted and the context menu is set to print its value as a decimal integer:

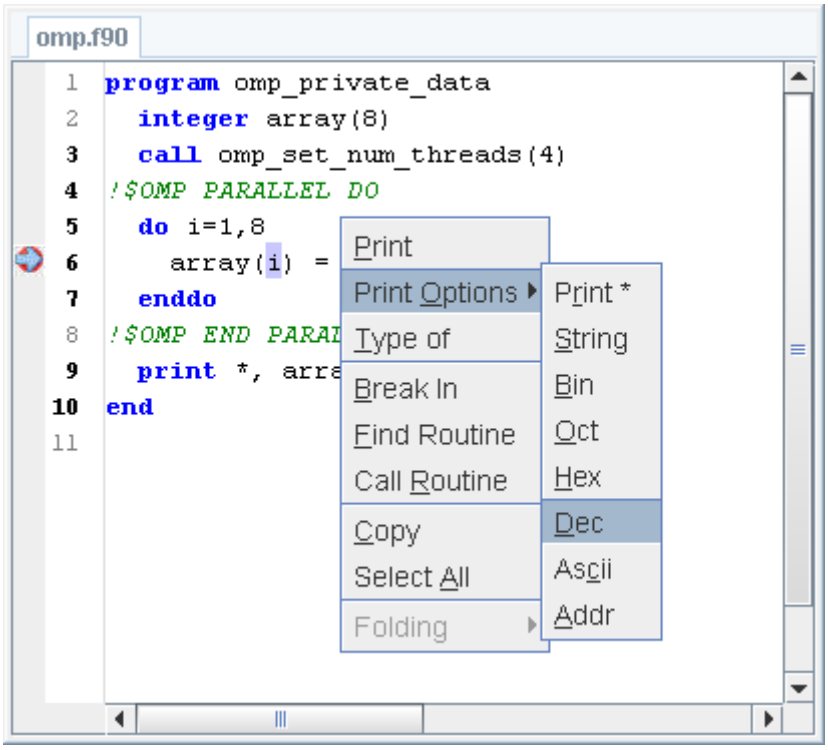


Figure 3 Context Menu

The context menu in Figure 3 provides shortcuts to the Type Of, the Break In, Find Routine..., and Call Routine menu options.

2.3. Main Toolbar

The debugger's main toolbar contains several buttons and four drop-down lists.

2.3.1. Buttons

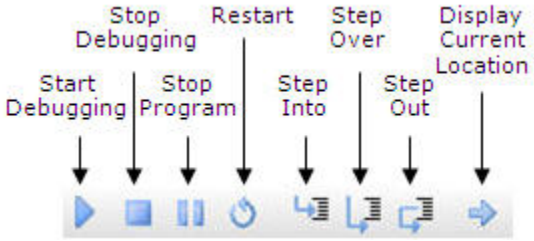


Figure 4 Buttons on Toolbar

Most of the buttons on the main toolbar have corresponding entries on the Debug menu. The functionality invoked from the toolbar is the same as that achieved by selecting the menu item. Refer to the "Debug Menu" descriptions for details on how Start Debugging (Continue), Stop Debugging, Stop Program, Restart, Step Into, Step Over, Step Out, and Display Current Location work.

2.3.2. Drop-Down Lists

As illustrated in Figure 5, the main toolbar contains four drop-down lists. A drop-down list displays information while also offering an opportunity to change the displayed information if other choices are available. When no or one choice is available, a drop-down list is grayed-out. When more than one choice is available, the drop-down arrow in the component can be clicked to display the available choices.

Figure 5 Drop-Down Lists on Toolbar

Current Process or Current Thread

The first drop-down list displays the current process or current thread. The list's label changes depending on whether processes or threads are described. When more than one process or thread is available, use this drop-down list to specify which process or thread should be the current one. The current process or thread controls the contents of the source and disassembly display tabs. The function of this drop-down list is the same as that of the Procs & Threads tab in the debug information tabs.

Apply

The second drop-down list is labeled Apply. The selection in the Apply drop-down determines the set of processes and threads to which action commands are applied. Action commands are those that control program execution and include, for example, **`cont`**, **`step`**, **`next`**, and **`break`**. By default, action commands are applied to all processes and threads. When more than one process or thread exists, you have additional options in this drop-down list from which to choose. The Current Group option designates the process and thread group selected in the Groups tab, and the Current Process and Current Thread options designate the process or thread selected in the Current Process or Current Thread drop-down.

Display

The third drop-down list is labeled Display. The selection in the Display drop-down determines the set of processes and threads to which data display commands are applied. Data display commands are those that print the values of expressions and program state and include, for example, **`print`**, **`names`**, **`regs`** and **`stack`**. The options in the Display drop-down are the same as those in the Apply drop-down but can be changed independently.

File

The fourth drop-down list is labeled File. It displays the source file that contains the current target location. It can be used to select another file for viewing in the source window.

2.4. Program I/O Window

Program output is displayed in the Program IO tab's central window. Program input is entered into this tab's Input field.

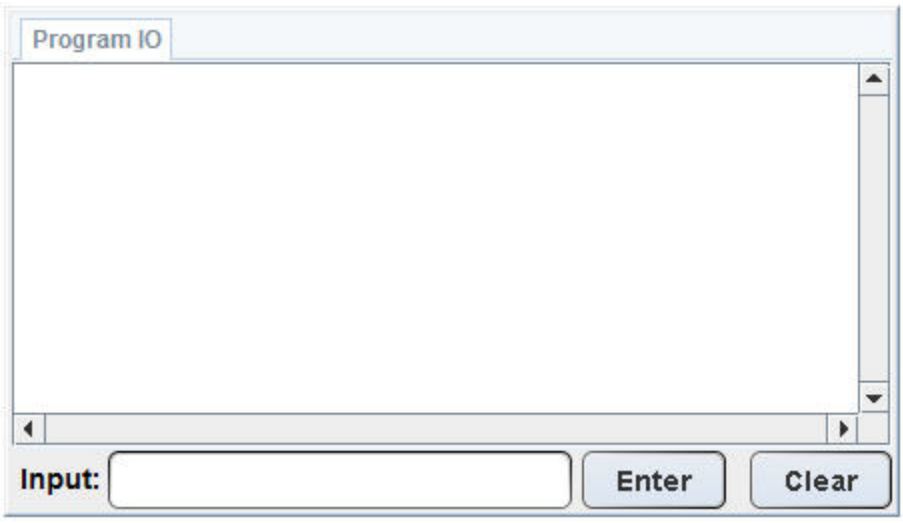


Figure 6 Program I/O Window

2.5. Debug Information Tabs

Debug information tabs take up the lower half of the debugger GUI. Each of these tabs provides a particular function or view of debug information. The following sections discuss the tabs as they appear from left-to-right in the GUI's default configuration.

2.5.1. Command Tab

The Command tab provides an interface in which to use the PGI debugger's command language. Commands entered in this panel are executed and the results are displayed there.

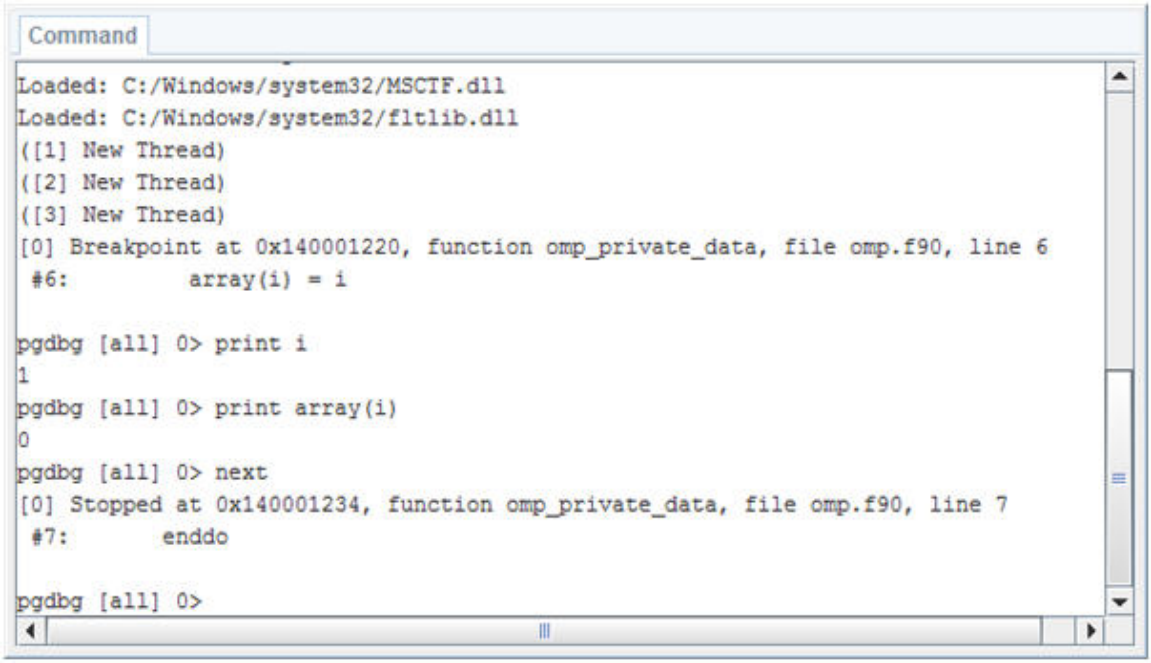


Figure 7 Command Tab

Using this tab is much like interacting with the debugger in text mode; the same list of commands is supported. For a complete list of commands, refer to Command Summary.

2.5.2. Events Tab

The Events tab displays the current set of events held by the debugger. Events include breakpoints and watchpoints, as shown in the following illustration.

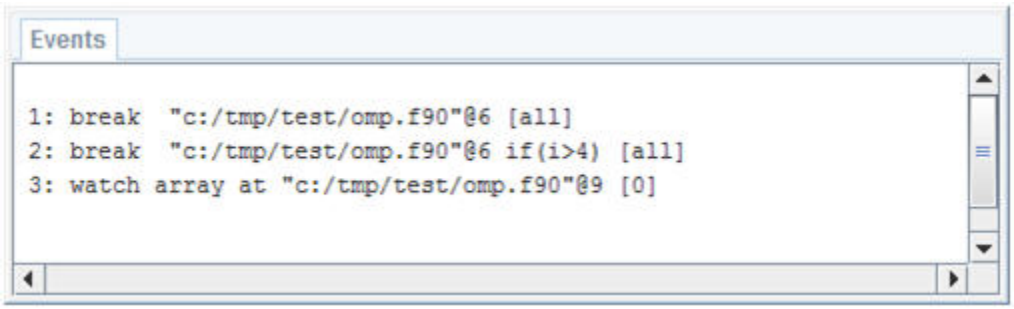


Figure 8 Events Tab

2.5.3. Groups Tab

The Groups tab displays the current set of user-defined groups of processes and threads. The group selected (highlighted) in the Groups tab defines the Current Group as used by the Apply and Display drop-down lists. In the following illustration, the 'evens' group is the Current Group.

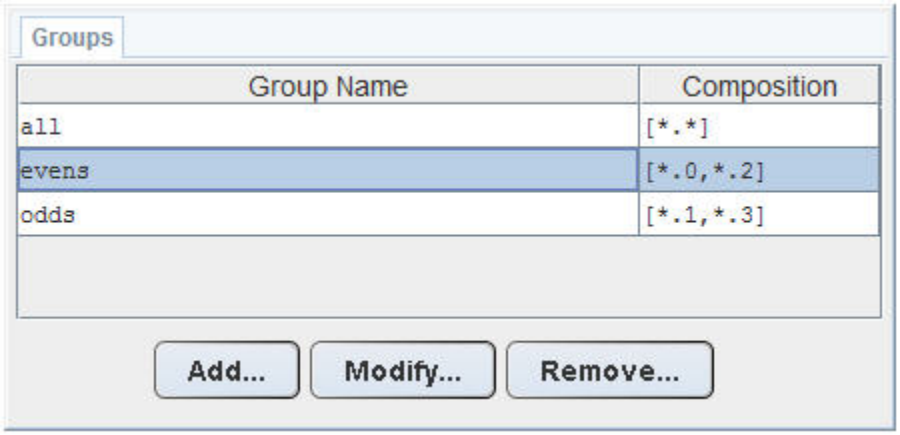


Figure 9 Groups Tab

To change the set of defined groups use the Add..., Modify..., and Remove... buttons on the Groups tab.

A defined group of processes and threads is also known as a process/thread-set or p/t-set. For more information on p/t-sets, refer to Process/Thread Sets in Parallel Debugging Overview.

2.5.4. Connections Tab

A *connection* is the set of information the debugger needs to begin debugging a program. The Connections tab provides the interface to specifying information for a particular connection, and allows you to create and save multiple connections. Saved connections persist from one invocation of the debugger to the next. When you launch the debugger, the Default connection is created for you. If you launched the debugger with an executable, the Program field is filled in for you.

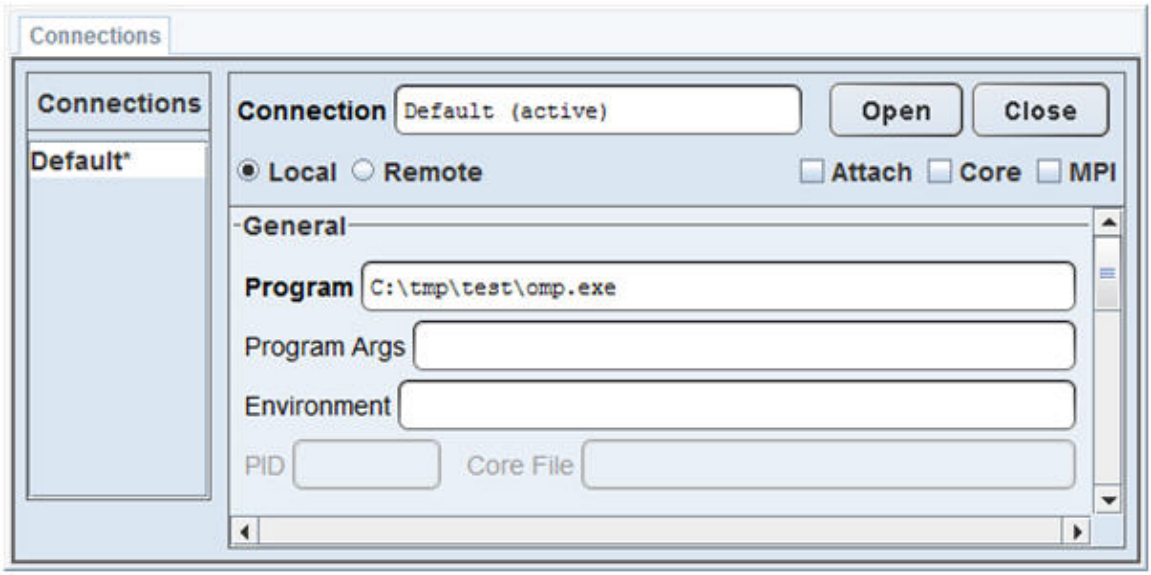


Figure 10 Connections Tab

Fields required by the debugger for program launch are **bold**. Fields not applicable to the current configuration options are grayed-out. To display a tooltip describing the use of a field, hover over its name.

Use the Connections menu to manage your connections.

2.5.5. Call Stack Tab

The Call Stack tab displays the current call stack. A blue arrow indicates the current stack frame.

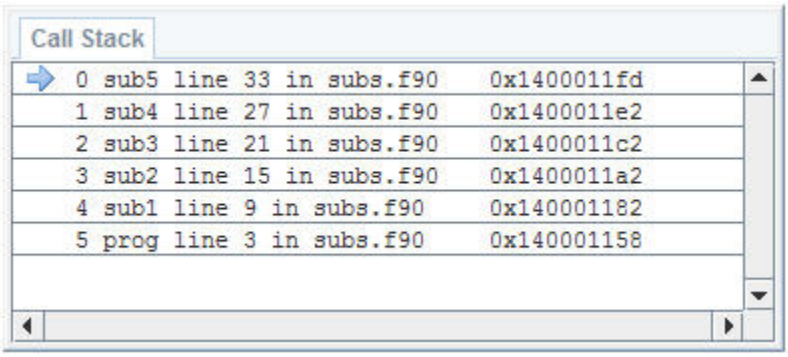


Figure 11 Call Stack Tab

Double-click in any call frame to move the debugging scope to that frame. A hollow arrow is used to indicate when the debug scope is in a frame other than the current frame.

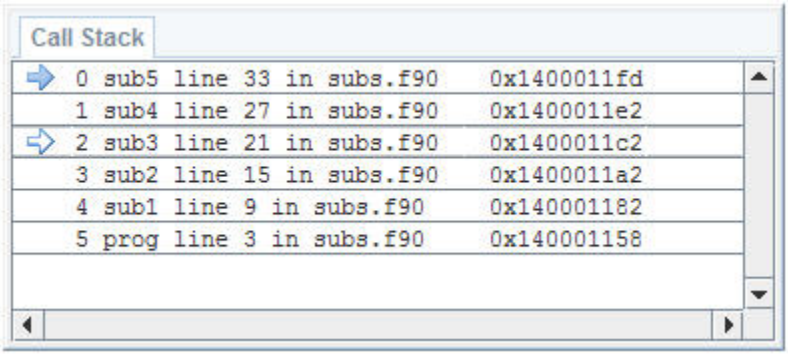


Figure 12 Call Stack Outside Current Frame

You can also navigate the call stack using the Up and Down options on the Debug menu.

2.5.6. Locals Tab

The Locals tab displays the current set of local variables and each of their values.

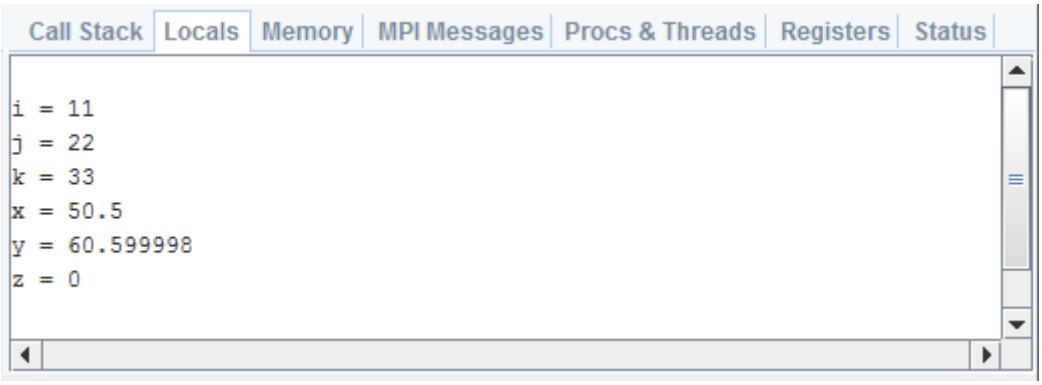


Figure 13 Locals Tab

2.5.7. Memory Tab

The Memory tab displays a region of memory starting with a provided Address which can be a memory address or a symbol name. One element of memory is displayed by default, but this amount can be changed via the Count field. Figure 14 illustrates this process.

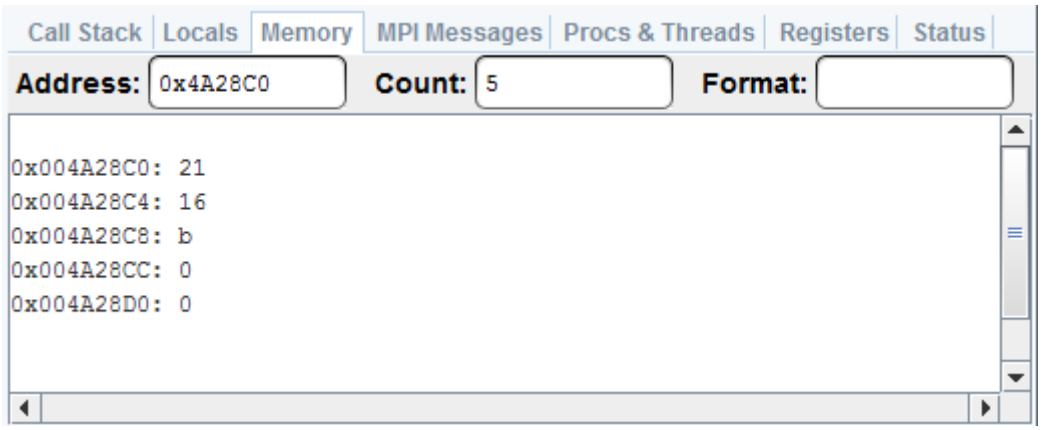


Figure 14 Memory Tab

The default display format for memory is hexadecimal. The display format can be changed by providing a printf-like format descriptor in the Format field. A detailed description of the supported format strings is available in Memory Access in Command Reference.

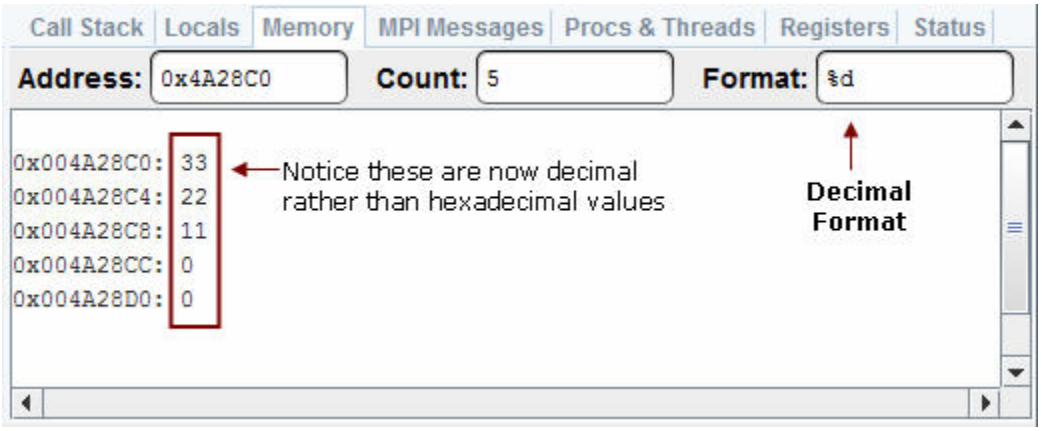


Figure 15 Memory Tab in Decimal Format

2.5.8. MPI Messages Tab

The MPI Messages tab provides a listing of the MPI message queues as illustration in Figure 16.

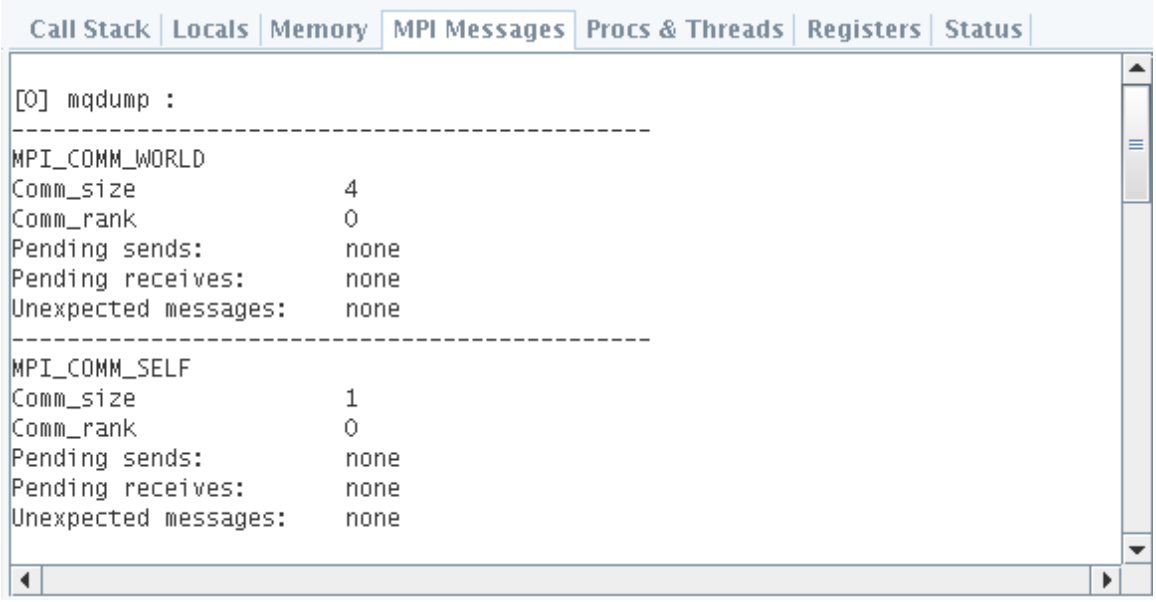


Figure 16 MPI Messages Tab

Message queue information applies only to MPI applications. When debugging a non-MPI application, this tab is empty. Additionally, message queue information is not supported by Microsoft MPI so this tab contains no data on Windows.

2.5.9. Procs & Threads Tab

The Procs & Threads tab provides a graphical display of the processes and threads in a debug session.

The Process Grid in Figure 17 has four processes. The thicker border around process 0 indicates that it is the current process; its threads are represented pictorially. Thread 0.0, as the current thread of the current process, has the thickest border. Clicking on any process or thread in this grid changes that process or thread to be the current process or thread.

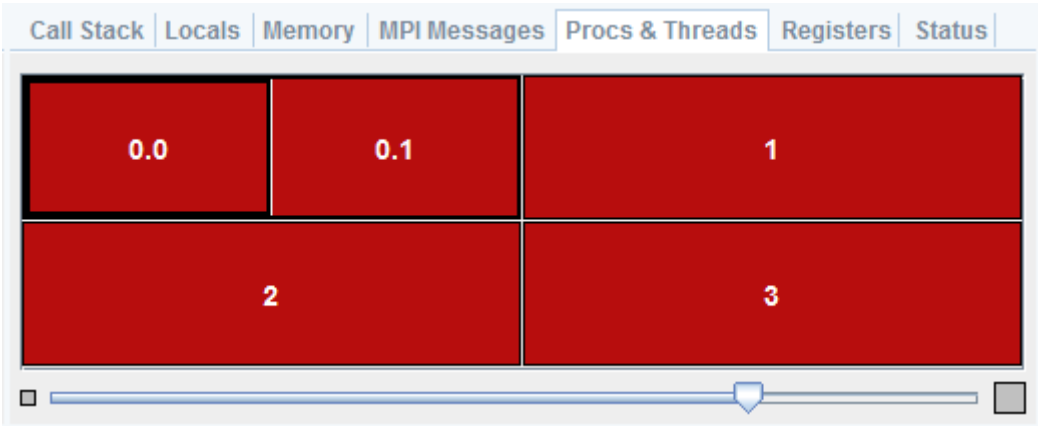


Figure 17 Process (Thread) Grid Tab

Use the slider at the bottom of the grid to zoom in and out.

The color of each element indicates the state of that process or thread. For a list of colors and states, refer to Table 1.

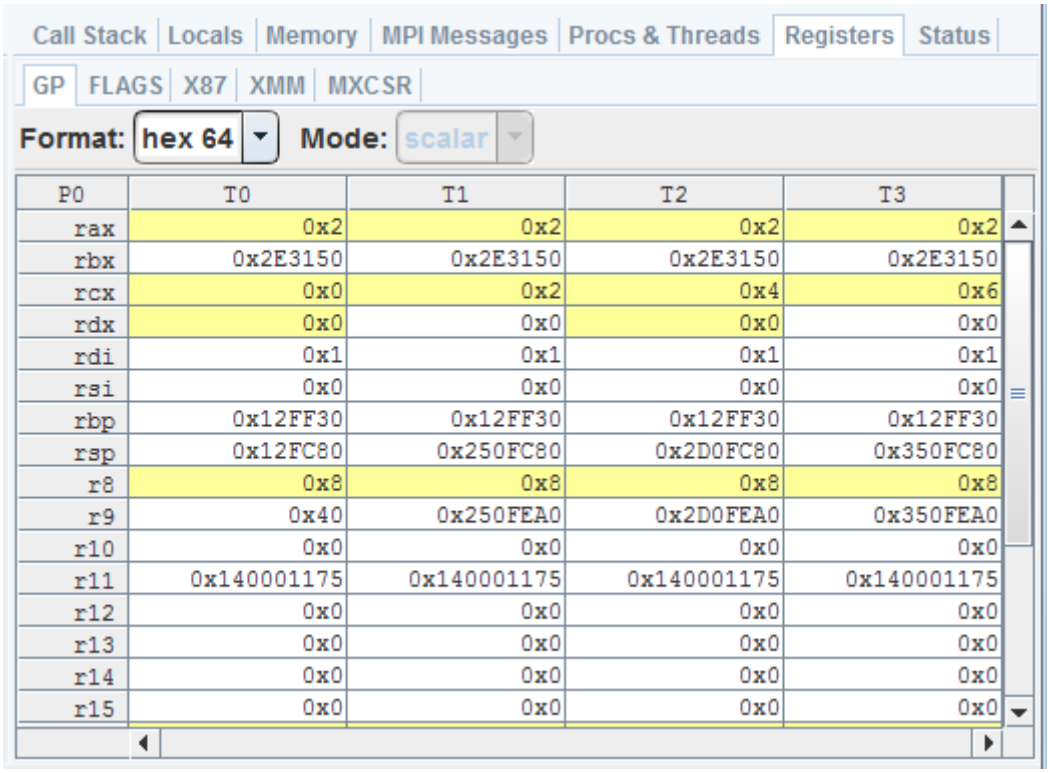
Table 1 Colors Describing Thread State

Option	Description
Stopped	Red
Signaled	Blue
Running	Green
Terminated	Black

2.5.10. Registers Tab

The target machine's architecture determines the number and type of system registers. Registers are organized into groups based on their type and function. Each register group is displayed in its own tab contained in the Registers tab. Registers and their values are displayed in a table. Values are shown for all the threads of the currently selected process.

In Figure 18, the General Purpose registers are shown for threads 0-3 of process 0.



P0	T0	T1	T2	T3
rax	0x2	0x2	0x2	0x2
rbx	0x2E3150	0x2E3150	0x2E3150	0x2E3150
rcx	0x0	0x2	0x4	0x6
rdx	0x0	0x0	0x0	0x0
rdi	0x1	0x1	0x1	0x1
rsi	0x0	0x0	0x0	0x0
rbp	0x12FF30	0x12FF30	0x12FF30	0x12FF30
rsp	0x12FC80	0x250FC80	0x2D0FC80	0x350FC80
r8	0x8	0x8	0x8	0x8
r9	0x40	0x250FEA0	0x2D0FEA0	0x350FEA0
r10	0x0	0x0	0x0	0x0
r11	0x140001175	0x140001175	0x140001175	0x140001175
r12	0x0	0x0	0x0	0x0
r13	0x0	0x0	0x0	0x0
r14	0x0	0x0	0x0	0x0
r15	0x0	0x0	0x0	0x0

Figure 18 General Purpose Registers

The values in the registers table are updated each time the program stops. Values that change from one stopping point to the next are highlighted in yellow.

Register values can be displayed in a variety of formats. The formatting choices provided for each register group depends on the type of registers in the group. Use the Format drop-down list to change the displayed format.

Vector registers, such as XMM and YMM registers, can be displayed in both scalar and vector modes. Change the Mode drop-down list to switch between these two modes.

2.5.11. Status Tab

The Status tab provides a text summary of the status of the program being debugged. The state and location of each thread of each process is shown. In Figure 19, each of four processes has two threads.

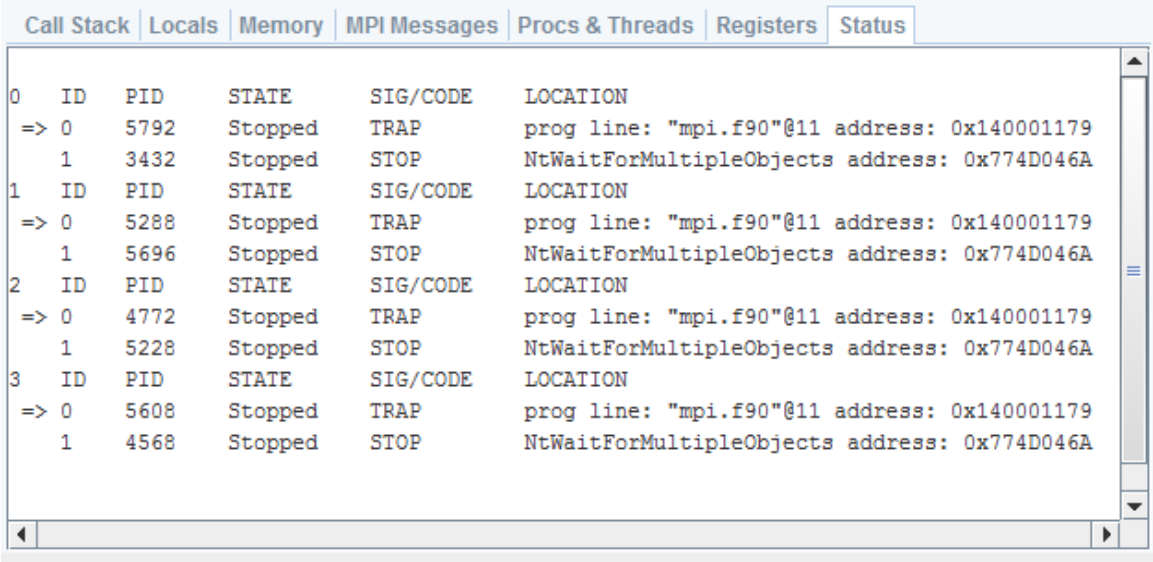


Figure 19 Status Tab

2.6. Menu Bar

The main menu bar contains these menus: File, Edit, View, Connections, Debug and Help. This section describes these menus and their contents.

You can navigate the menus using the mouse or the system's mouseless modifier (typically the Alt key). Use the mouseless modifier together with a menu's mnemonic, usually a single character, to select a menu and then a menu item. Menu mnemonics are indicated with an underscore. For example, the File menu appears as File which indicates that 'F' is the mnemonic.

Keyboard shortcuts, such as Ctrl+V for Edit | Paste, are available for some actions. Where a keyboard shortcut is available, it is shown in the GUI on the menu next to the menu item.

Menu items that contain an ellipsis (...) launch a dialog box to assist in performing the menu's action.

2.6.1. File Menu

Exit
End the current debug session and close all windows.

2.6.2. Edit Menu

Copy
Copy selected text to the system's clipboard.

Paste
Paste selected text to the system's clipboard.

Find
Perform a string search in the current source window.

Find Routine...
Find a routine. If symbol and source information is available for the specified routine, the routine is displayed in the source panel.

Restore Default Settings
Restore the GUI's various settings to their initial default state illustrated in Default Appearance of the Debugger GUI.

Revert to Saved Settings
Restore the GUI to the state that it was in at the start of the debug session.

Save Settings on Exit
By default, the debugger saves the state (size and settings) of the GUI on exit on a per-system basis. To prevent settings from being saved from one invocation of the debugger to another, uncheck this option. This option must be unchecked prior to every exit since the debugger always defaults to saving the GUI state.

2.6.3. View Menu

Use the View menu to customize the debugger's display of tabs, source code, and assembly. Some of the items on this menu contain a check box next to the name of a tab.

- When the check box is checked, the tab is visible.
- When the check box is not checked, the tab is hidden.

View menu items that correspond to tabs include Call Stack, Command, Connections, Events, Groups, Locals, Memory, MPI Messages, Procs & Threads, Program I/O, Source, and Status.

Show Assembly/ Show Source
Toggle (turn on or off) the display of assembly code during a debug session. Source code can be shown only when source information is available.

Disable/Enable Syntax Coloring
Toggle syntax coloring of source code.

Hide Source in Assembly/Add Source to Assembly
When assembly code is shown, toggle the display of source code (when available).

Hide Addresses/Show Addresses
When assembly code is shown, toggle the display of assembly addresses.

Show Bytecode/Hide Bytecode
When assembly code is shown, toggle the display of assembly bytecode.

Registers
The Registers menu item opens a submenu containing items representing every subtab on the Registers tab. Recall that each subtab represents a register group and the set of register groups is system and architecture dependent. Use the Registers submenu to hide or show tabs for register groups. Use the Show Selected item to hide or show the Registers tab itself.

Font...
Use the font chooser dialog box to select the font and size used in the source window and debug information tabs. The default font is named *monospace* and the default size is *12*.

Show Tool Tips
Tool tips are small temporary messages that pop up when the mouse pointer hovers over a component in the GUI. They provide additional information on the

functionality of a component. Tool tips are enabled by default. Uncheck the Show Tools Tips option to prevent them from popping up.

Reload Source
Update the source window.

2.6.4. Connections Menu

Use the items under this menu to manage the connections displayed in the Connections list on the Connections tab.

Connect Default
Open the currently displayed connection. When the debugger starts, this connection is named 'Default.' When a different connection is selected, the name of this menu option changes to reflect the name of the selected connection. This menu option works the same way that the Open button on the Connections tab works.

New
Create a new connection.

Save
Save changes to all the connections.

Save As
Save the selected connection as a new connection.

Rename
Change the name of the selected connection.

Delete
Delete the selected connection.

2.6.5. Debug Menu

The items under this menu control the execution of the program.

Go
Run or continue running the program.

Stop Program
Stop the running program. This action halts the running processes or threads. For more information, refer to the halt command.

Stop Debugging
Stop debugging the program.

Restart Program
Start the program from the beginning.

Step
Continue and stop after executing one source line or one assembly-level instruction depending on whether source or assembly is displayed. Step steps *into* called routines. For more information, refer to the step and stepi commands.

Next
Continue and stop after executing one source line or one assembly-level instruction depending on whether source or assembly is displayed. Next steps *over* called routines. For more information, refer to the next and nexti commands.

Step Out
Continue and stop after returning to the caller of the current routine. For more information, refer to the stepout command.

Set Breakpoint...
Set a breakpoint at the first executable source line in the specified routine.
Call Routine
Specify a routine to call. For more information, refer to the call command.
Display Current Location
Display the current program location in the Source panel. For more information, refer to the arrive command.
Up
Enter the scope of the routine up one level in the call stack. For more information, refer to the up command.
Down
Enter the scope of the routine down one level in the call stack. For more information, refer to the down command.
Custom
Opens a separate window where you can enter a variety of debugger commands.

2.6.6. Help Menu

Debugger Guide
Launch your defulat browser to view the PGI Debugger Guide (this document) online.
About PGI Debugger
This option displays a dialog box with version and copyright information on the PGI debugger. It also contains sales and support points of contact.

Chapter 3. COMMAND LINE OPTIONS

The debugger accepts a variety of options when it is invoked from the command line. This section describes these options and how they can be used.

3.1. Command-Line Options Syntax

```
pgdbg arguments program arg1 arg2 ... argn
```

The optional *arguments* may be any of the command-line arguments described in this chapter. The *program* parameter is the name of the executable file being debugged. The optional arguments *arg1 arg2 ... argn* are the command-line arguments to the program.

3.2. Command-Line Options

-attach <*pid*>
Attach to a running process with the process ID <pid>.

-c <*pgdbg_cmd*>
Execute the debugger command pgdbg_cmd before executing the commands in the startup file.

-cd <*workdir*>
Sets the working directory to the specified directory.

-core <*corefile*>
Analyze the core dump named corefile. [Linux only]

–help
Display a list of command-line arguments (this list).

–dryrun
Display commands that would be executed without executing them.

–I <*directory*>
Add <*directory*> to the list of directories that the debugger uses to search for source files. You can use this option multiple times to add multiple directories to the search path.

–jarg, <*javaarg*>
Pass specified argument(s) (separated by commas) to java, e.g. -jarg,-Xmx256m.

–java <*jrepath*>
Add a jrepath directory to the JVM search path. Multiple '-java' options are allowed.
–nomin
Do not minimize the debugger's console shell on startup. [Windows only]
–s <*pgdbg_script*>
Runs the provided debugger command script instead of the configuration file: `pgdbgrc` [Linux,macOS] or `pgdbg_rc` [Windows].
–show
Print debugger configuration information.
–text
Run the debugger using a command-line interface (CLI). The default is for the debugger to launch in graphical user interface (GUI) mode.
–V
Display the version of the debugger being run.
–v
Enable verbose output; display commands as they are run.

3.3. Command-Line Options for MPI Debugging

–mpi[=<*launcher_path*>
Debug an MPI program. Here the term *launcher* means the MPI launch program. The debugger uses **`mpiexec`** as the default launcher. If the location of the launcher in your MPI distribution is not in your PATH environment variable, you must provide the debugger with the full path to the launcher, including the name of the launch tool itself. If the location of the launcher is in your PATH, then you just need to provide the name of the launcher, and then only if the launcher is not **`mpiexec`**.
–sgimpi[=<*launcher_path*>]
Debug an SGI MPI (MPT) program. The debugger uses **`mpirun`** as the default launcher for SGI MPI debugging. If the location of **`mpirun`** in your installation of SGI MPI is not in your PATH environment variable, you must provide the debugger with the full path to **`mpirun`**, including the name **`mpirun`** itself. If the location of **`mpirun`** is in your PATH, then you can use –sgimpi without a sub-option.
–program_args
Pass subsequent arguments to the program under debug; required when passing program arguments to an MPI program.
–pgserv[=<*pgserv_path*>]
[Optional] Specify path for pgserv, the per-node debug agent.

3.4. I/O Redirection

The command shell interprets any I/O redirection specified on the debugger command line. For a description of how to redirect I/O using the run command, refer to Process Control.

Chapter 4. COMMAND LANGUAGE

The debugger supports a command language that is capable of evaluating complex expressions. The command language is composed of commands, constants, symbols, locations, expressions, and statements.

You can use the command language by invoking the debugger's command-line interface with the –text option, or in the Command tab of the debugger's graphical user interface, as described in The Graphical User Interface.

4.1. Command Overview

Commands are named operations, which take zero or more arguments and perform some action. Commands may also return values that may be used in expressions or as arguments to other commands.

4.1.1. Command Syntax

Commands are entered one line at a time.

- Lines are delimited by a carriage return.
- Each line must consist of a command and its arguments, if any.
- You can place multiple commands on a single line by using the semi-colon (;) as a delimiter.

4.1.2. Command Modes

There are two command modes: **pgi** and **dbx**.

- The pgi command mode maintains the original PGI debugger command interface.
- In dbx mode, the debugger uses commands compatible with the Unix-based dbx debugger.

PGI and dbx commands are available in both command modes, but some command behavior may be slightly different depending on the mode. The mode can be set while the debugger is running by using the **pgienv** command.

4.2. Constants

The debugger supports C language style integer (hex, octal and decimal), floating point, character, and string constants.

4.3. Symbols

The debugger uses the symbolic information contained in the executable object file to create a symbol table for the target program. The symbol table contains symbols to represent source files, subroutines, types (including structure, union, pointer, array, and enumeration types), variables, and arguments. The debugger's command-line interface is case-sensitive with respect to symbol names; a symbol name on the command line must match the name as it appears in the object file.

4.4. Scope Rules

Since several symbols in a single application may have the same name, scope rules are used to bind program identifiers to symbols in the symbol table. The debugger uses the concept of a search scope for looking up identifiers. The search scope represents a subroutine, a source file, or global scope. When the user enters a name, the debugger first tries to find the symbol in the search scope. If the symbol is not found, the containing scope (source file or global) is searched, and so forth, until either the symbol is located or the global scope is searched and the symbol is not found.

Normally, the search scope is the same as the current scope, which is the subroutine where execution is currently stopped. The current scope and the search scope are both set to the current subroutine each time execution of the program stops. However, you can use the **`enter`** command to change the search scope.

A scope qualifier operator @ allows selection of out-of-scope identifiers. For example, if f is a routine with a local variable i, then:

```
f@i
```

represents the variable i local to f. Identifiers at file scope can be specified using the quoted file name with this operator. The following example represents the variable i defined in file `xyz.c`.

```
"xyz.c"@i
```

4.5. Register Symbols

To provide access to the system registers, the debugger maintains symbols for them. Register names generally begin with $ to avoid conflicts with program identifiers. Each register symbol has a default type associated with it, and registers are treated like global variables of that type, except that their address may not be taken. For more information on register symbols, refer to SSE Register Symbols.

4.6. Source Code Locations

Some commands must refer to source code locations. Source file names must be enclosed in double quotes. Source lines are indicated by number, and may be qualified by a quoted filename using the scope qualifier operator. Further, a range of lines is indicated using the range operator ":".

Here are some examples:

`break 37`	sets a breakpoint at line 37 of the current source file.
`break "xyz.c"@37`	sets a breakpoint at line 37 of the source file xyz.c.
`list 3:13`	lists lines 3 through 13 of the current file.
`list "xyz.c"@3:13`	lists lines 3 through 13 of the source file xyz.c.

Some commands accept both line numbers and addresses as arguments. In these commands, it is not always obvious whether a numeric constant should be interpreted as a line number or an address. The description for these commands says which interpretation is used. However, the debugger provides commands to convert from source line to address and vice versa. The **`line`** command converts an address to a line, and the **`addr`** command converts a line number to an address.

Here are some examples:

`line 37`	means “line 37”
`addr 0x1000`	means "address 0x1000"
`addr {line 37}`	means "the address associated with line 37"
`line {addr 0x1000}`	means "the line associated with address 0x1000"

4.7. Lexical Blocks

Line numbers are used to name lexical blocks. The line number of the first instruction contained by a lexical block is used to indicate the start scope of the lexical block.

In the following example, there are two variables named var. One is declared in function main, and the other is declared in the lexical block starting at line 5. The lexical block has the unique name "lex.c"@main@5. The variable var declared in "lex.c"@main@5 has the unique name "lex.c"@main@5@var. The output of the **`whereis`** command that follows shows how these identifiers can be distinguished.

```
lex.c:
1 main()
2 {
3   int var = 0;
4   {
5     int var = 1;
6     printf("var %d/n",var);
7   }
8   printf("var %d/n",var)
9 }
```

```
pgdbg> n
Stopped at 0x8048b10, function main, file
/home/demo/pgdbg/ctest/lex.c,
line 6
#6: printf("var %d/n",var);
```

```
pgdbg> print var
1
```

```
pgdbg> which var
"lex.c"@main@5@var
```

```
pgdbg> whereis var
variable: "lex.c"@main@var
variable: "lex.c"@main@5@var
```

```
pgdbg> names "lex.c"@main@5
 var = 1
```

4.8. Statements

Although the debugger's command-line input is processed one line at a time, statement constructs allow multiple commands per line, as well as conditional and iterative execution. The statement constructs roughly correspond to the analogous C language constructs. Statements may be of the following forms.

- *Simple Statement:* A command and its arguments. For example:

  ```
  print i
  ```

- *Block Statement:* One or more statements separated by semicolons and enclosed in curly braces. Note: these may only be used as arguments to commands or as part of `if` or `while` statements. For example:

  ```
  if(i>1) {print i; step }
  ```

- *If Statement*: The keyword if, followed by a parenthesized expression, followed by a block statement, followed by zero or more `else if` clauses, and at most one else clause. For example:

  ```
  if(i>j) {print i} else if(i<j) {print j} else {print "i==j"}
  ```

- *While Statement:* The keyword while, followed by a parenthesized expression, followed by a block statement. For example:

  ```
  while(i==0) {next}
  ```

Multiple statements may appear on a line separated by a semicolon. The following example sets breakpoints in routines main and xyz, continues, and prints the new current location.

```
break main; break xyz; cont; where
```

However, since the **where** command does not wait until the program has halted, this statement displays the call stack at some arbitrary execution point in the program. To control when the call stack is printed, insert a **wait** command, as shown in this example:

```
break main; break xyz; cont; wait; where
```

Any value returned by the last statement on a line is printed.

Statements can be parallelized across multiple threads of execution. For more information, refer to Parallel Statements.

4.9. Events

Breakpoints, watchpoints, and other mechanisms used to define the response to certain conditions are collectively called *events*.

- An event is defined by the conditions under which the event occurs and by the action taken when the event occurs.
- A breakpoint occurs when execution reaches a particular address.

  The default action for a breakpoint is simply to halt execution and prompt the user for commands.
- A watchpoint occurs when the value of an expression changes.
- A hardware watchpoint occurs when the specified memory location is accessed or modified.

4.9.1. Event Commands

The debugger supports six basic commands for defining events. Each command takes a required argument and may also take one or more optional arguments. The basic commands are **break**, **watch**, **hwatch**, **trace**, **track**, and **do**.

Event Command Descriptions

- The **break** command takes an argument specifying a breakpoint location. Execution stops when that location is reached.
- The **watch** command takes an expression argument. Execution stops and the new value is printed when the value of the expression changes.
- The **hwatch** command takes a data address argument, which can be either an identifier or a variable name. Execution stops when memory at that address is written.
- The **trace** command activates source line tracing, as specified by the arguments you supply.
- The **track** command is like **watch** except that execution continues after the new value is printed.
- The **do** command takes a list of commands as an argument. The commands are executed whenever the event occurs.

Event Command Arguments

The six event commands share a common set of optional arguments. The optional arguments provide the ability to make the event definition more specific. They are:

at *line*
Event occurs at indicated line.

at *addr*
Event occurs at indicated address.

in *routine*
Event occurs throughout indicated routine.

if (*condition*)
Event occurs only when condition is true.

do {*commands*}
When event occurs, execute commands.

The optional arguments may appear in any order after the required argument and should not be delimited by commas.

Event Command Examples

Here are some event definition examples:

```
watch i at 37 if(y>1)
```
This event definition says to stop and print the value of i whenever line 37 is executed and the value of y is greater than 1.

```
do {print xyz} in f
```
This event definition says that at each line in the routine f print the value of xyz.

```
break func1 if (i==37)
  do {print a[37]; stack}
```
This event definition says to print the value of a[37] and do a stack trace when i is equal to 37 in routine func1.

4.9.2. Event Command Action

It is useful to know when events take place.

- Event commands that do not explicitly define a location occur at each source line in the program. Here are some examples:

  ```
  do {where}
  ```
  prints the current location at the start of each source line.

  ```
  trace a.b
  ```
  prints the value of a.b each time the value has changed.

  ```
  track a.b
  ```
  prints the value of a.b at the start of each source line if the value has changed.

Events that occur at every line can be useful, but they can make program execution very slow. Restricting an event to a particular address minimizes the impact on program execution speed, and restricting an event that occurs at every

line to a single routine causes execution to be slowed only when that routine is executed.

- The debugger supports instruction-level versions of several commands, such as **breaki**, **watchi**, **tracei**, **tracki**, and **doi**. The basic difference in the instruction-level version is that these commands interpret integers as addresses rather than line numbers, and events occur at each instruction rather than at each line.
- When multiple events occur at the same location, all event actions are taken before the prompt for input. Defining event actions that resume execution is allowed but discouraged, since continuing execution may prevent or defer other event actions.

  For example, the following syntax creates an ambiguous situation:

  ```
  break 37 do {continue}
  ```

  ```
  break 37 do {print i}
  ```

  With this sequence, it is not clear whether `i` will ever be printed.
- Events only occur after the **continue** and **run** commands. They are ignored by **step**, **next**, **call**, and other commands.
- Identifiers and line numbers in events are bound to the current scope when the event is defined.

  For example, the following command sets a breakpoint at line 37 in the current file.

  ```
  break 37
  ```

  The following command tracks the value of whatever variable `i` is currently in scope.

  ```
  track i
  ```

  If `i` is a local variable, then it is wise to add a location modifier (at or in) to restrict the event to a scope where i is defined. Scope qualifiers can also specify lines or variables that are not currently in scope. Events can be parallelized across multiple threads of execution. See Parallel Events for details.

4.10. Expressions

The debugger supports evaluation of expressions composed of constants, identifiers, commands that return values, and operators.

The following rules apply:

- To use a value returned by a command in an expression, the command and arguments must be enclosed in curly braces.

  For example, the following command invokes the **pc** command to compute the current address, adds 8 to it, and sets a breakpoint at that address.

  ```
  breaki {pc}+8
  ```

  Similarly, the following command compares the start address of the current routine with the start address of routine xyz. It prints the value 1 if they are equal and 0 if they are not.

  ```
  print {addr {func}}=={addr xyz}
  ```

- The @ operator, introduced previously, may be used as a scope qualifier. Its precedence is the same as the C language field selection operators "." and "->" .
- The debugger recognizes a range operator ":" which indicates array sub-ranges or source line ranges. The precedence of ':' is between '||' and '='.

  Here are a few examples that use the range operator:

`print a[1:10]`	prints elements 1 through 10 of the array a.
`list 5:10`	lists source lines 5 through 10.
`list "xyz.c"@5:10`	lists lines 5 through 10 in file xyz.c.

  The general format for the range operator is [lo : hi : step] where:

lo	is the array or range lower bound for this expression.
hi	is the array or range upper bound for this expression.
step	is the step size between elements.

- An expression can be evaluated across many threads of execution by using a prefix p/t-set. For more details, refer to Current vs. Prefix p/t sets.

Table 2 shows the C language operators that the debugger supports. The operator precedence is the same as in the C language.

Table 2 PGI Debugger Operators

Operator	Description		Operator	Description
*	indirection		<=	less than or equal
.	direct field selection		>=	greater than or equal
->	indirect field selection		!=	not equal
[ ]	C/ C++ array index		&&	logical and
()	routine call		||	logical or
&	address of		!	logical not
+	add		|	bitwise or
(type)	cast		&	bitwise and
-	subtract		~	bitwise not
/	divide		^	bitwise exclusive or
*	multiply		()	FORTRAN array index
=	assignment		%	FORTRAN field selector
==	comparison		<<	left shift
>>	right shift			

4.11. Ctrl+C

The effect of Ctrl+C is different when debugging using the command-line interface or the GUI, and when debugging serial or parallel code.

4.11.1. Command-Line Debugging

If the program is not running, Ctrl+C can be used to interrupt long-running debugger commands. For example, a command requesting disassembly of thousands of instructions might run for a long time, and it can be interrupted by Ctrl+C. In such cases the program is not affected.

If the program is running, entering Ctrl+C at the debugger command prompt halts execution of the program. This is useful in cases where the program 'hangs' due to an infinite loop or deadlock.

Sending Ctrl+C, also known as SIGINT, to a program while it is in the middle of initializing its threads, by calling omp_set_num_threads() or entering a parallel region, may kill some of the threads if the signal is sent before each thread is fully initialized. Avoid sending SIGINT in these situations. Note that when the number of threads employed by a program is large, thread initialization may take a while.

4.11.2. GUI Debugging

If the program is running, entering Ctrl+C in the Input field of the Program IO tab sends SIGINT to the program.

4.11.3. MPI Debugging

Sending Ctrl+C to a running MPICH1 program, support for which is now deprecated, is not recommended. For details, refer to Use halt instead of Ctrl+C. Use the debugger's **`halt`** command as an alternative to sending Ctrl+C to a running program. The debugger's command prompt must be available in order to issue a **`halt`** command. The command prompt is available while threads are running if **`pgienv threadwait none`** is set.

As described in Using Continue, when debugging an MPI job via the following command, the debugger spawns the job in a manner that prevents console-generated interrupts from directly reaching the MPI launcher or any of the MPI processes.

```
$ pgdbg -mpi ...
```

In this case, typing Ctrl+C only interrupts the debugger, leaving the MPI processes running. When debugger's thread wait mode is not set to none, you can halt the MPI job after using Ctrl+C by entering the debugger's **`halt`** command, even if no command prompt is generated.

Chapter 5. COMMAND SUMMARY

This chapter contains a brief summary of the debugger's commands. For a detailed description of each command, grouped by category of use, refer to Command Reference.

If you are viewing an online version of this manual, you can select the hyperlink under the selection category to jump to that section in the manual.

5.1. Notation Used in Command Sections

The command sections that follow use these conventions for the command names and arguments, when the command accepts one.

- Command names may be abbreviated by omitting the portion of the command name enclosed in brackets ([]).
- Argument names are chosen to indicate what kind of argument is expected.
- Arguments enclosed in brackets([]) are optional.
- Two or more arguments separated by a vertical line (|) indicate that any one of the arguments is acceptable.
- An ellipsis (...) indicates an arbitrarily long list of arguments.
- Other punctuation (commas, quotes, etc.) should be entered as shown.

For example, the following syntax indicates that the command **`list`** may be abbreviated to **`lis`**, and that it can be invoked without any arguments or with *one* of the following arguments: an integer count, a line range, a routine name, or a line and a count.

```
lis[t] [count | lo:hi | routine | line,count]
```

5.2. Command Summary

Table 3 PGI Debugger Commands

Name	Arguments	Category
ad[dr]	[n | line n | routine | var | arg]	Conversions

Name	Arguments	Category
	Creates an address conversion under certain conditions.	
al[ias]	[name [string]] Create or print aliases.	Miscellaneous
args	 Print the current program arguments.	Process Control
arri[ve]	 Print location information for the current location.	Program Locations
asc[ii]	exp [,...exp] Evaluate and print as an ascii character.	Printing Variables and Expressions
as[sign]	var=exp Set variable `var` to the value of the expression `exp`.	Symbols and Expressions
att[ach]	pid [exe] Attach to a running process with process ID pid. Use `exe` to specify the absolute path of the executable file.	Process Control
bin	exp [,...exp] Evaluate and print the expressions. Integer values are printed in base 2.	Printing Variables and Expressions
b[reak]	[line | routine] [if (condition)] [do {commands}] [hit [>|*] <num>] When arguments are specified, sets a breakpoint at the indicated line or routine. When no arguments are specified, prints the current breakpoints.	Events
breaki	[addr | routine] [if (condition)] [do {commands}] [hit [>|*] <num>] When arguments are specified, sets a breakpoint at the indicated address or routine. When no arguments are specified, prints the current breakpoints.	Events
breaks	 Displays all the existing breakpoints	Events
call	routine [(exp,...)] Call the named routine.	Symbols and Expressions
catch	[number [,number...]] With arguments, catches the specified signals and runs the program as though the signal was not sent. With no arguments, prints the list of signals being caught.	Events

Name	Arguments	Category
cd	[dir] Change to the $HOME directory or to the specified directory dir.	Program Locations
clas[s]	[class] Return the current class or enter the scope of the specified `class`.	Scope
classe[s]	 Print the C++ class names.	Target
clear	[all | routine | line | addr {addr}] With arguments, clears the indicated breakpoints. When no arguments are specified, this command clears all breakpoints at the current location.	Events
con[nect]	[-t name [args] | -d path [args] | -f file [name [args]]] Prints the current connection and the list of possible connection targets.	Target
c[ont]	 Continue execution from the current location.	Process Control
de[bug]	[target [arg1 _ argn]] Load the specified program with optional command-line arguments.	Process Control
dec	exp [,...exp] Evaluate and print the expressions. Integer values are printed in decimal.	Printing Variables and Expressions
decl[aration]	name Print the declaration for the symbol based on its type according to the symbol table.	Symbols and Expressions
decls	[routine | "sourcefile" | {global}] Print the declarations of all identifiers defined in the indicated scope. If no scope is given, print the declarations for global scope.	Scope
defset	name [p/t-set] Assign a name to a process/thread set. Define a named set.	Process-Thread Sets
del[ete]	event-number | all | 0 | event-number [,.event-number.] Delete the event `event-number` or all events (delete 0 is the same as delete all). Multiple event numbers can be supplied if they are separated by commas. Use	Events

Name	Arguments	Category
	`delete` without arguments to list events by event-number.	
det[ach]	Detach from the current running process.	Process Control
dir[ectory]	[pathname] Add the directory pathname to the search path for source files. If no argument is specified, the currently defined directories are printed.	Miscellaneous
disab[le]	event-number | all With arguments, disables the event `event-number` or all events. When no arguments are specified, prints both enabled and disabled events by event-number.	Printing Variables and Expressions
dis[asm]	[count | lo:hi | routine | addr, count] Disassemble memory. If no argument is given, disassemble four instructions starting at the current address.	Program Locations
disc[onnect]	Close connection to target.	Events
display	[exp [,...exp]] With one or more arguments, print expression `exp` at every breakpoint. Without arguments, list the expressions for the debugger to automatically display at breakpoints.	Printing Variables and Expressions
do	{commands} [at line | in routine] [if (condition)] Define a `do` event. Without the optional arguments `at` or `in`, the commands are executed at each line in the program.	Events
doi	{commands} [at addr | in routine] [if (condition)] Define a `doi` event. If neither the `at` or `in` argument is specified, then the commands are executed at each instruction in the program.	Events
down	[number] Enter scope of routine down one level or `number` levels on the call stack.	Scope
du[mp]	[addr [,count [,format]]] Dump the contents of a region of memory. The output is formatted according to a printf-like format descriptor.	Memory Access
edit	[filename | routine]	Program Locations

Name	Arguments	Category
	Edit the specified file or file containing the subroutine. If no argument is supplied, edit the current file starting at the current location(Command-line interface only).	
enab[le]	[event-number | all] With arguments, this command enables the event `event-number` or all events. When no arguments are specified, prints both enabled and disabled events by event-number.	Events
en[ter]	[routine | "sourcefile" | global] Set the search scope to be the indicated symbol, which may be a subroutine, source file or global. Using no argument is the same as using global.	Scope
entr[y]	[routine] Return the address of the first executable statement in the program or specified subroutine.	Symbols and Expressions
fil[e]	[filename] Change the source file to the file filename and change the scope accordingly. With no argument, print the current file.	Program Locations
files	 Return the list of known source files used to create the executable file.	Scope
focus	[p/t-set] Set the target process/thread set for commands. Subsequent commands are applied to the members of this set by default.	Process-Thread Sets
fp	 Return the current value of the frame pointer.	Register Access
func[tion]	[addr | line] Return a subroutine symbol. If no argument is specified, return the current routine.	Conversions
glob[al]	 Return a symbol representing global scope.	Scope
halt	[command] Halt the running process or thread.	Process Control
he[lp]	[command] If no argument is specified, print a brief summary of all the commands. If a `command` name is specified,	Miscellaneous

Name	Arguments	Category
	print more detailed information about the use of that command.	
hex	exp [,...exp] Evaluate and print expressions as hexadecimal integers.	Printing Variables and Expressions
hi[story]	[num] List the most recently executed commands. With the `num` argument, resize the history list to hold num commands.	Miscellaneous
hwatch	addr | var [if (condition)] [do {commands}] Define a hardware watchpoint.	Events
hwatchb[oth]	addr | var [if (condition)] [do {commands}] Define a hardware read/write watchpoint.	Events
hwatchr[ead]	addr | var [if (condition)] [do {commands}] Define a hardware read watchpoint	Events
ignore	[number [,number...]] Ignore the specified signals and do not deliver them to the program. When no arguments are specified, prints the list of signals being ignored.	Events
language	 Print the name of the language of the current file.	Miscellaneous
lin[e]	[n | routine | addr] Create a source line conversion. If no argument is given, return the current source line.	Conversions
lines	[routine] Print the lines table for the specified routine. If no argument is specified, prints the lines table for the current routine.	Program Locations
lis[t]	[count | line,count | lo:hi | routine] With no argument, list 10 lines centered at the current source line. If an argument is specified, list lines based on information requested.	Program Locations
lo[ad]	[prog [args]] Without options, print the name and arguments of the program being debugged. With arguments, invoke the debugger using the specified program and program arguments, if any.	Process Control

Name	Arguments	Category
log	filename Keep a log of all commands entered by the user and store it in the named file.	Miscellaneous
lv[al]	expr Return the lvalue of the expression *expr*.	Symbols and Expressions
mq[dump]	 Dump MPI message queue information for the current process.	Memory Access
names	[routine | "sourcefile" | {global}] Print the names of all identifiers defined in the indicated scope. If no scope is specified, use the search scope.	Scope
nat[ive]	[command] Without arguments, print a list of the available target commands. With a command argument, send the native command directory to the target.	Target
n[ext]	[count] Stop after executing one or count source line(s) in the current subroutine.	Process Control
nexti	[count] Stop after executing one or count instruction(s) in the current subroutine.	Process Control
nop[rint]	exp Evaluate the expression but do not print the result.	Miscellaneous
oct	exp [,...exp] Evaluate and print expressions as octal integers.	Printing Variables and Expressions
pc	 Return the current program address.	Register Access
pgienv	[command] Define the debugger environment. With no arguments, display the debugger settings.	Miscellaneous
p[rint]	exp1 [,...expn] Evaluate and print one or more expressions.	Printing Variables and Expressions
printf	"format_string", expr,...expr	Printing Variables and Expressions

Name	Arguments	Category
	Print expressions in the format indicated by the format string.	
proc	[id] Set the current process to the process identified by id. When issued with no argument, lists the location of the current thread of the current process in the current program.	Process Control
procs	 Print the status of all active processes, listing each process by its logical process ID.	Process Control
pwd	 Print the current working directory.	Program Locations
q[uit]	 Terminate the debugging session.	Process Control
regs	regs [-info] [-grp=grp1[,grp2...]] [-fmt=fmt1[,fmt2...]] [-mode=vector|scalar] Print a formatted display of the names and values of registers. Specify the register group(s) with the `-grp` option and formatting with the `-fmt` option. Use `-info` to see a listing of available register groups and formats.	Register Access
rep[eat]	[first, last] | [first: last:n] | [num] | [-num] Repeat the execution of one or more previous history list commands.	Miscellaneous
rer[un]	[arg0 arg1 ... argn] [< inputfile] [[> | >& | >> | >>&] outputfile] Like the **`run`** command with one exception: if no args are specified with **`rerun`**, then no args are used when the program is launched.	Process Control
ret[addr]	 Return the current return address.	Register Access
ru[n]	[arg0 arg1 ... argn] [< inputfile] [> outputfile] Execute program from the beginning. If arguments arg0, arg1, and so on are specified, they are set up as the command-line arguments of the program. Otherwise, the arguments for the previous **`run`** command are used.	Process Control
rv[al]	expr Return the rvalue of the expression *expr*.	Symbols and Expressions
sco[pe]	 Return a symbol for the search scope.	Scope

Name	Arguments	Category
scr[ipt]	filename Open the indicated file and execute the contents as though they were entered as commands. If you use~ before the filename, it is expanded to the value of the environment variable HOME.	Miscellaneous
set	var = exp Set variable var to the value of expression.	Symbols and Expressions
setargs	[arg1 , arg2, ... argn] Set program arguments to be used by the current program.	Process Control
setenv	name [value] Print value of environment variable name. With a specified value, set name to value.	Miscellaneous
sh[ell]	[arg0 , arg1, ... argn] Fork a shell (defined by $SHELL) and give it the indicated arguments (the default shell is sh). Without arguments, invokes an interactive shell, and executes until a "^D" is entered.	Miscellaneous
siz[eof]	name Return the size, in bytes, of the variable type name; or, if the name refers to a routine, returns the size in bytes of the subroutine.	Symbols and Expressions
sle[ep]	[time] Pause for time seconds. If no time is specified, pause for one second.	Miscellaneous
sou[rce]	filename Open the indicated file and execute the contents as though they were entered as commands. If you use ~ before the filename, it is expanded to the value of $HOME.	Miscellaneous
sp	 Return the current stack pointer address.	Register Access
stackd[ump]	[count] Print a formatted dump of the call stack. This command displays a hex dump of the stack frame for each active subroutine.	Program Locations
stack[trace]	[count]	Program Locations

Name	Arguments	Category
	Print the call stack. For each active subroutine print the subroutine name, source file, line number, and current address, provided that this information is available.	
stat[us]	Display all the event definitions, including an event number by which the event can be identified.	Events
s[tep]	[count | up] Step into the current subroutine and stop after executing one or count source line(s). If the up argument is specified, stops execution after stepping out of the current subroutine.	Process Control
stepi	[count | up] Step into the current subroutine and stop after executing one or count source line(s). If the up argument is specified, stops execution after stepping out of the current subroutine.	Process Control
stepo[ut]	Stop after returning to the caller of the current subroutine.	Process Control
stop	[at line | in routine] [var] [if (condition)] [do {commands}] Set a breakpoint at the indicated subroutine or line. Break when the value of the indicated variable var changes.	Events
stopi	[at addr | in routine] [var] [if (condition)] [do {commands}] Set a breakpoint at the indicated address or subroutine. Break when the value of the indicated variable var changes.	Events
str[ing]	exp [,...exp] Evaluate and print expressions as null-terminated character strings, up to a maximum of 70 characters.	Printing Variables and Expressions
sync	[routine | line] Advance the current process/thread to a specific program location, ignoring any user-defined events.	Process Control
synci	[routine | addr] Advance the current process/thread to a specific program location, ignoring any user-defined events.	Process Control
thread	[number]	Process Control

Name	Arguments	Category
	Set the current thread to the thread identified by number; where number is a logical thread ID in the current process' active thread list. When issued with no argument, list the current program location of the currently active thread.	
threads	Prints the status of all active threads, grouped by process.	Process Control
trace	[at line | in routine] [var | routine] [if (condition)] do {commands} Activates source line tracing as specified by the arguments supplied.	Events
tracei	[at addr | in routine] [var] [if (condition)] do {commands} Activates instruction tracing as specified by the arguments supplied.	Events
track	expression [at line | in routine] [if (condition)] [do {commands}] Define a track event.	Events
tracki	expression [at addr | in routine] [if (condition)] [do {commands}] Define an assembly-level track event.	Events
type	expr Return the type of the expression.	Symbols and Expressions
unal[ias]	name Remove the alias definition for name, if one exists.	Miscellaneous
unb[reak]	line | routine | all Remove a breakpoint from the statement line or subroutine, or remove all breakpoints.	Events
unbreaki	addr | routine | all Remove a breakpoint from the address addr or the subroutine, or remove all breakpoints.	Events
undefset	[name | -all] Remove a previously defined process/thread set from the list of process/thread sets	Process-Thread Sets
undisplay	[all | 0 | exp] Remove all expressions specified by previous display commands. With an argument or several arguments,	Printing Variables and Expressions

Name	Arguments	Category
	remove the expression exp from the list of display expressions.	
u[p]	[number] Move up one level or number levels on the call stack.	Scope
use	[dir] Print the current list of directories or add dir to the list of directories to search. If the first character in pathname is ~, the value of $HOME is substituted for this character.	Miscellaneous
viewset	name List the members of a process/thread set that currently exist as active threads or list defined p/t-sets.	Process-Thread Sets
wait	[any | all | none] Inserts explicit wait points in a command stream.	Process Control
wa[tch]	expression [at line | in routine] [if (condition)] [do {commands}] Define a watch event. The given expression is evaluated, and subsequently, each time the value of the expression changes, the program stops and the new value is printed.	Events
watchi	expression [at addr | in routine] [if(condition)] [do {commands}] Define an assembly-level watch event	Events
whatis	[name] With no arguments, prints the declaration for the current subroutine. With argument name, prints the declaration for the symbol name.	Symbols and Expressions
when	[at line | in routine] [if (condition)] do {commands} Execute commands at every line in the program, at a specified line in the program or in the specified subroutine.	Events
wheni	[at addr | in routine] [if(condition)] do {commands} Execute commands at each address in the program. If an address is specified, the commands are executed each time the address is reached.	Events
w[here]	[count] Print the call stack. For each active subroutine print the subroutine name, subroutine arguments, source file,	Program Locations

Name	Arguments	Category
	line number, and current address, provided that this information is available.	
whereis	name Print all declarations for name.	Symbols and Expressions
which	name Print full scope qualification of symbol name.	Scope
whichsets	[p/t-set] List all defined p/t-sets to which the members of a process/thread set belong.	Process-Thread Sets
/	/ [string] / Search forward for the specified string of characters in the current source file.	Program Locations
?	?[string] ? Search backward for the specified string of characters in the current source file.	Program Locations
!	History modification Executes a command from the command history list. The command executed depends on the information that follows the !.	Miscellaneous
^	History modification Quick history command substitution ^old^new^<modifier> this is equivalent to !:s/old/new/	Miscellaneous

Chapter 6. ASSEMBLY-LEVEL DEBUGGING

This chapter provides information about assembly-level debugging, including an overview about what to expect if you are using assembly-level debugging or if you did not compile your program for debugging.

6.1. Assembly-Level Debugging Overview

The PGI debugger supports debugging regardless of how a program was compiled. In other words, the debugger does not require that the program under debug be compiled with debugging information, such as using `-g`. It can debug code that is lacking debug information, but because it is missing information about symbols and line numbers, it can only access the program at the assembly level. The debugger also supports debugging at the assembly level if debug symbols are available.

As described in Building Applications for Debug, the richest debugging experience is available when the program is compiled using `-g` or `-gopt` with no optimization. When a program is compiled at higher levels of optimization, less information about source-level symbols and line numbers is available, even if the program was compiled with `-g` or `-gopt`. In such cases, if you want to find the source of a problem without rebuilding the program, you may need to debug at the assembly level.

If a program has been "stripped" of all symbols, either by the linker or a separate utility, then debugging will be at the assembly level. The debugger is only able to examine or control the program in terms of memory addresses and registers.

6.1.1. Assembly-Level Debugging on Windows

When applications are built without `-g` on Windows systems, the resulting binary, the `.exe` file, does not contain any symbol information. The Microsoft linker stores symbol information in a program database, a `.pdb` file. To generate a `.pdb` file using the PGI compiler drivers, you must use `-g` during the link step. You can do this even if you did not use `-g` during the compile step. Having this `.pdb` file available provides the debugger with enough symbol information to map addresses to routine names.

6.1.2. Assembly-Level Debugging with Fortran

To refer to Fortran symbol names when debugging at the assembly level, you must translate names so these match the calling convention in use by the compiler. For code compiled by the PGI compilers, in most cases this means translating Fortran names to lower case and appending an underscore. For example, a routine that appears in the source code as "`VADD`" would be referred to in the debugger as "`vadd_`".

Name translation is only necessary for assembly-level debugging. When debugging at the source level, you may refer to symbol names as they appear in the source.

A special symbol, `MAIN_`, is created by PGFORTRAN to refer to the main program. PGFORTRAN generates this special symbol whether or not there is a PROGRAM statement. One way to run to the beginning of a Fortran program is to set a breakpoint on `MAIN_`, then run.

6.1.3. Assembly-Level Debugging with C++

C++ symbol names are "mangled" names. For the names of C++ methods, the names are modified to include not only the name as it appears in the source code, but information about the enclosing class hierarchy, argument and return types, and other information. The names are long and arcane. At the source level these names are translated by the debugger to the names as they appear in the source. At the assembly level, these names are in the mangled form. Translation is not easy and not recommended. If you have no other alternative, you can find information about name mangling in the *PGI Compiler User's Guide*.

6.1.4. Assembly-Level Debugging Using the PGI Debugger GUI

This section describes some basic operations for assembly-level debugging using the PGI debugger GUI. If you encounter the message "`Can't find main function compiled -g`" on startup, assembly-level debugging is required.

To get into a program in this situation, you can select the Debug | Set Breakpoint... menu option. For example, to stop at program entry, in Fortran you could enter `MAIN_` in response to the dialog query, while in C or C++ you could enter `main`.

Debug information tabs that are useful in assembly-level debugging include the Call Stack, Memory, and Register tabs. Disassembly is automatically shown in the source pane when source files are available. You can also switch from source to disassembly debugging by selecting the View | Show Assembly menu option.

6.1.5. Assembly-Level Debugging Using the PGI Debugger CLI

This section describes some basic operations for assembly-level debugging using the PGI debugger's command-line interface. When you invoke the debugger's CLI and are presented with a message telling you that "`NOTE: Can't find main function compiled -g`", assembly-level debugging is required.

To get into the program, you can set a breakpoint at a named routine. To stop at program entry, for example, in Fortran you could use

```
pgdbg>   break MAIN_
```

and in C/ C++ you could use

```
pgdbg>  break main
```

Some useful commands for assembly-level debugging using the debugger's command-line interface include:

run

run the program from the beginning

cont

continue program execution from the current point

nexti

single-step one instruction, stepping over calls

stepi

single-step one instruction, stepping into calls

breaki

set a breakpoint at a given address

regs

display the registers

print $<regname>

display the value of the specified register

For more information on register names, refer to SSE Register Symbols.

dump

dump memory locations

stacktrace

display the current call stack.

stackdump

display the current call stack.

6.2. SSE Register Symbols

X64 processors and x86 processors starting with Pentium III provide SSE (Streaming SIMD Enhancements) registers and a SIMD floating-point control/status register.

Each SSE register may contain four 32-bit single-precision or two 64-bit floating-point values. The **regs** command reports these values individually in both hexadecimal and floating-point format. The debugger provides command notation to refer to these values individually or all together.

The component values of each SSE register can be accessed using the same syntax that is used for array subscripting. Pictorially, the SSE registers can be thought of as follows:

[32-bit]

127 96	95 64	63 32	31 0
$xmm0[3]	$xmm0[2]	$xmm0[1]	$xmm0[0]
$xmm1[3]	$xmm1[2]	$xmm1[1]	$xmm1[0]
$xmm2[3]	$xmm2[2]	$xmm2[1]	$xmm2[0]

To access $xmm0[3], the 32-bit single-precision floating point value that occupies bits 96 – 127 of SSE register 0, use the following command:

```
pgdbg> print $xmm0[3]
```

To set $xmm2[0] to the value of $xmm3[2], use the following command:

```
pgdbg> set $xmm2[0] = $xmm3[2]
```

[64-bit]

127 64	63 0
$xmm0d[1]	$xmm0d[0]
$xmm1d[1]	$xmm1d[0]
$xmm2d[1]	$xmm2d[0]

To access the 64-bit floating point values in xmm0, append the character 'd' (for double precision) to the register name and subscript as usual, as illustrated in the following commands:

```
pgdbg> print $xmm0d[0]
```

```
pgdbg> print $xmm0d[1]
```

In most cases, the debugger detects when the target environment supports SSE registers. In the event the debugger does not allow access to SSE registers on a system that should have them, set the PGDBG_SSE environment variable to on to enable SSE support.

Chapter 7. SOURCE-LEVEL DEBUGGING

This section describes source-level debugging, including debugging Fortran and C++.

7.1. Debugging Fortran

7.1.1. Fortran Types

The debugger displays Fortran type declarations using Fortran type names. The only exception is Fortran character types, which are treated as arrays of the C type char.

7.1.2. Arrays

Fortran array subscripts and ranges are accessed using the Fortran language syntax convention, denoting subscripts with parentheses and ranges with colons.

PGI compilers for the linux86-64 platform (Intel 64 or AMD64) support large arrays (arrays with an aggregate size greater than 2GB). You can enable large array support by compiling using these options:`-mcmodel=medium -Mlarge_arrays`. The debugger provides full support for large arrays and large subscripts.

The debugger supports arrays with non-default lower bounds. Access to such arrays uses the same subscripts that are used in the program.

The debugger also supports adjustable arrays. Access to adjustable arrays may use the same subscripting that is used in the program.

7.1.3. Operators

In general, the debugger uses C language style operators in expressions and supports the Fortran array index selector "()" and the Fortran field selector "%" for derived types.

However, `.eq.`, `.ne.`, and so forth are not supported. You must use the analogous C operators ==, !=, and so on, instead.

The precedence of operators matches the C language, which may in some cases be different than that used in Fortran.

See PGI Debugger Commands for a complete list of operators and their definition.

7.1.4. Name of the Main Routine

If a PROGRAM statement is used, the name of the main routine is the name in the program statement. You can always use the following command to set a breakpoint at the start of the main routine.

```
break MAIN
```

7.1.5. Common Blocks

Each subprogram that defines a common block has a local static variable symbol to define the common. The address of the variable is the address of the common block. The type of the variable is a locally-defined structure type with fields defined for each element of the common block. The name of the variable is the common block name, if the common block has a name, or _BLNK_ otherwise.

For each member of the common block, a local static variable is declared which represents the common block variable. Thus given declarations:

```
common /xyz/ a, b
integer a
integer b
```

then the entire common block can be printed out using,

```
print xyz
```

Individual elements can be accessed by name. For example:,

```
print a, b
```

7.1.6. Internal Procedures

To unambiguously reference an internal procedure, qualify its name with the name of its host using the scoping operator @.

For example:

```
subroutine sub1 ()
    call internal_proc ()
    contains
    subroutine internal_proc ()
        print *, "internal_proc in sub1"
    end subroutine internal_proc
end subroutine

subroutine sub2 ()
    call internal_proc ()
    contains
    subroutine internal_proc ()
        print *, "internal_proc in sub2"
    end subroutine internal_proc
end subroutine

program main
    call sub1 ()
    call sub2 ()
end program
```

```
pgdbg> whereis internal_proc
function:       "/path/ip.f90"@sub1@internal_proc
function:       "/path/ip.f90"@sub2@internal_proc
```

```
pgdbg> break sub1@internal_proc
(1)breakpoint set at: internal_proc line: "ip.f90"@5 address: 0x401E3C 1
```

```
pgdbg> break sub2@internal_proc
(2)breakpoint set at: internal_proc line: "ip.f90"@13 address: 0x401EEC 2
```

7.1.7. Modules

A member of a Fortran 90 module can be accessed during debugging.

```
module mod
    integer iMod
end module
subroutine useMod()
    use mod
    iMod = 1000
end subroutine
program main
    call useMod()
end program
```

- If the module is in the current scope, no qualification is required to access the module's members.

  ```
  pgdbg> b useMod
  (1)breakpoint set at: usemod line: "modv.f90"@7 address: 0x401CC4
  1

  Breakpoint at 0x401CC4, function usemod, file modv.f90, line 7
   #7:         iMod = 1000
  ```

  ```
  pgdbg> p iMod
  0
  ```

- If the module is not in the current scope, use the scoping operator @ to qualify the member's name.

```
Breakpoint at 0x401CF0, function main, file modv.f90, line 11
 #11:         call useMod()
```

```
pgdbg> p iMod
"iMod" is not defined in the current scope
```

```
pgdbg> p mod@iMod
0
```

7.1.8. Module Procedures

A module procedure is a subroutine contained within a module. A module procedure itself can contain internal procedures. The scoping operator @ can be used when working with these types of subprograms to prevent ambiguity.

```
module mod
    contains
    subroutine mod_proc1()
        call internal_proc()
        contains
        subroutine internal_proc()
            print *, "internal_proc in mod_proc1"
        end subroutine
    end subroutine
    subroutine mod_proc2()
        call internal_proc()
        contains
        subroutine internal_proc()
            print *, "internal_proc in mod_proc2"
        end subroutine
    end subroutine
end module
```

```
program main
    use mod
    call mod_proc1
    call mod_proc2
end program
```

```
pgdbg> whereis internal_proc
function:       "/path/modp.f90"@mod@mod_proc1@internal_proc
function:       "/path/modp.f90"@mod@mod_proc2@internal_proc

pgdbg> break mod@mod_proc1@internal_proc
(1)breakpoint set at: internal_proc line: "modp.f90"@7 address: 0x401E3C
1
pgdbg> break mod@mod_proc2@internal_proc
(2)breakpoint set at: internal_proc line: "modp.f90"@14 address: 0x401EEC
2
```

7.2. Debugging C++

7.2.1. Calling C++ Instance Methods

To use the **`call`** command to call a C++ instance method, the object must be explicitly passed as the first parameter to the call. For example, suppose you were given the

following definition of class `Person` and the appropriate implementation of its methods:

```
class Person
{
    public:
    char name[10];
    Person(char * inName);
    void print();
};

int main ()
{
    Person * pierre;
    pierre = new Person("Pierre");
    pierre->print();
    return 0;
}
```

Call the instance method `print` on object `pierre` as follows:

```
pgdbg> call Person::print(pierre)
```

Notice that `pierre` must be explicitly passed into the method because it is the `this` pointer. You can also specify the class name to remove ambiguity.

Chapter 8. PLATFORM-SPECIFIC FEATURES

This section describes debugger features specific to particular platforms, such as pathname conventions, debugging with core files, and signals.

8.1. Pathname Conventions

The debugger uses the forward slash character (/) internally as the path component separator on all platforms. The backslash (\) is used as the escape character in the debugger's command language.

On Windows systems, use backslash as the path component separator in the fields of the Connections tab. Use the forward slash as the path component separator when using a debugger command in the Command tab or in the CLI. The forward slash separator convention is still in effect when using a drive letter to specify a full path. For example, to add the Windows pathname `C:/Temp/src` to the list of searched source directories, use the command:

```
pgdbg> dir C:/Temp/src
```

To set a breakpoint at line 10 of the source file specified by the relative path `sub1\main.c`, use this command:

```
pgdbg> break "sub1/main.c":10
```

8.2. Debugging with Core Files

The debugger supports debugging of core files on Linux platforms. In the GUI, select the Core option on the Connections tab to enable core file debugging. Fill in the Program and Core File fields and open the connection to load the core file.

You can also start core file debugging from the command line. To do this, use the following options:

```
$ pgdbg -core coreFileName programName
```

Core files (or core dumps) are generated when a program encounters an exception or fault. For example, one common exception is the segmentation violation, which can be

caused by referencing an invalid memory address. The memory and register states of the program are written into a core file so that they can be examined by a debugger.

The shell environment in which the application runs must be set up to allow core file creation. On many systems, the default user setting `ulimit` does not allow core file creation.

Check the `ulimit` as follows:

For sh/bash users:

```
$ ulimit -c
```

For csh/tcsh users:

```
% limit coredumpsize
```

If the core file size limit is zero or something too small for the application, it can be set to unlimited as follows:

For sh/bash users:

```
$ ulimit -c unlimited
```

For csh/tcsh users:

```
% limit coredumpsize unlimited
```

See the Linux shell documentation for more details. Some versions of Linux provide system-wide limits on core file creation.

The core file is normally written into the current directory of the faulting application. It is usually named `core` or `core.pid` where *pid* is the process ID of the faulting thread. If the shell environment is set correctly and a core file is not generated in the expected location, the system core dump policy may require configuration by a system administrator.

Different versions of Linux handle core dumping slightly differently. The state of all process threads are written to the core file in most modern implementations of Linux. In some new versions of Linux, if more than one thread faults, then each thread's state is written to separate core files using the `core.pid` file naming convention previously described. In older versions of Linux, only one faulting thread is written to the core file.

If a program uses dynamically shared objects (i.e., shared libraries named lib*.so), as most programs on Linux do, then accurate core file debugging requires that the program be debugged on the system where the core file was created. Otherwise, slight differences in the version of a shared library or the dynamic linker can cause erroneous information to be presented by the debugger. Sometimes a core file can be debugged successfully on a different system, particularly on more modern Linux systems, but you should take care when attempting this.

When debugging core files, the debugger:

- Supports all non-control commands.
- Performs any command that does not cause the program to run.
- Generates an error message in for any command that causes the program to run.
- May provide the status of multiple threads, depending on the type of core file created.

The debugger does not support multi-process core file debugging.

8.3. Signals

The debugger intercepts all signals sent to any of the threads in a multi-threaded program and passes them on according to that signal's disposition as maintained by (see the **catch** and **ignore** commands), except for signals that cannot be intercepted or signals used by the debugger internally.

8.3.1. Signals Used Internally by the Debugger

SIGTRAP and SIGSTOP are used by Linux for communication of application events to the debugger. Management of these signals is handled internally. Changing the disposition of these signals in the debugger (via **catch** and **ignore**) results in undefined behavior.

8.3.2. Signals Used by Linux Libraries

Some Linux thread libraries use SIGRT1 and SIGRT3 to communicate among threads internally. Other Linux thread libraries, on systems that do not have support for real-time signals in the kernel, use SIGUSR1 and SIGUSR2. Changing the disposition of these signals in the debugger (via **catch** and **ignore**) results in undefined behavior.

Target applications compiled with the options `-pg` or profiled with `pgprof` generate numerous SIGPROF signals. Although SIGPROF can be handled via the **ignore** command, debugging of applications built for sample-based profiling is not recommended.

Chapter 9. PARALLEL DEBUGGING OVERVIEW

This section provides an overview of how to debug parallel applications. It includes important definitions and background information on how the debugger represents processes and threads.

9.1. Overview of Parallel Debugging Capability

The debugger is a parallel application debugger capable of debugging multi-process MPI applications, multi-thread and OpenMP applications, and hybrid multi-thread/multi-process applications that use MPI to communicate between multi-threaded or OpenMP processes.

For specific information on multi-thread and OpenMP debugging, refer to Parallel Debugging with OpenMP.

For specific information on multi-process MPI debugging, refer to Parallel Debugging with MPI.

9.1.1. Graphical Presentation of Threads and Processes

The graphical user interface components that provide support for parallelism are described in detail in The Graphical User Interface.

9.2. Basic Process and Thread Naming

Because the debugger can debug multi-threaded, multi-process, and hybrid multi-threaded/multi-process applications, it provides a convention for uniquely identifying each thread in each process. This section gives a brief overview of this naming convention and how it is used to provide adequate background for the subsequent sections. A more detailed discussion of this convention, including advanced techniques for applying it, is provided in Thread and Process Grouping and Naming.

The debugger identifies threads in an OpenMP application using the OpenMP thread IDs. Otherwise, the debugger assigns arbitrary IDs to threads, starting at zero and incrementing in order of thread creation.

The debugger identifies processes in an MPI application using MPI rank (in communicator MPI_COMM_WORLD). Otherwise, the debugger assigns arbitrary IDs to processes; starting at zero and incrementing in order of process creation. Process IDs are unique across all active processes.

In a multi-threaded/multi-process application, each thread can be uniquely identified across all processes by prefixing its thread ID with the process ID of its parent process. For example, thread 1.4 identifies the thread with ID 4 in the process with ID 1.

An OpenMP application logically runs as a collection of threads with a single process, process 0, as the parent process. In this context, a thread is uniquely identified by its thread ID. The process ID prefix is implicit and optional. For more information on debugging threads, refer to Thread-only Debugging.

An MPI program logically runs as a collection of processes, each made up of a single thread of execution. Thread 0 is implicit to each MPI process. A process ID uniquely identifies a particular process, and thread ID is implicit and optional. For more information on process debugging, refer to Process-only Debugging.

A hybrid, or multilevel, MPI/OpenMP program requires the use of both process and thread IDs to uniquely identify a particular thread. For more information on multilevel debugging, refer to Multilevel Debugging.

A serial program runs as a single thread of execution, thread 0, belonging to a single process, process 0. The use of thread IDs and process IDs is allowed but unnecessary.

9.3. Thread and Process Grouping and Naming

This section describes how to name a single thread, how to group threads and processes into sets, and how to apply debugger commands to groups of processes and threads.

9.3.1. Debug Modes

The debugger can operate in four debug modes. The mode determines a short form for uniquely naming threads and processes. The debug mode is set automatically or by the **`pgienv mode`** command.

Table 4 Debug Modes

Debug Mode	Program Characterization
Serial	A single thread of execution
Threads-only	A single process, multiple threads of execution
Process-only	Multiple processes, each process made up of a single thread of execution
Multilevel	Multiple processes, at least one process employing multiple threads of execution

The debugger initially operates in serial mode reflecting a single thread of execution. Thread IDs can be ignored in serial debug mode since there is only a single thread of execution.

The command prompt displays the ID of the current thread according to the current debug mode. For a description of the command prompt, refer to The Command Prompt.

The debug mode can be changed at any time during a debug session.

To change debug mode manually, use the **pgienv** command.

```
pgienv mode [serial|thread|process|multilevel]
```

9.3.2. Threads-only Debugging

Enter threads-only mode to debug a program with a single multi-threaded process. As a convenience the process ID portion can be omitted. The debugger automatically enters threads-only debug mode from serial debug mode when it detects and attaches to new threads.

Table 5 Thread IDs in Threads-only Debug Mode

`1`	Thread 1 of process 0 (*. 1)
`*`	All threads of process 0 (*. *)
`0.7`	Thread 7 of process 0 (multilevel names are valid in threads-only mode)

In threads-only debug mode, status and error messages are prefixed with thread IDs depending on context.

9.3.3. Process-only Debugging

Enter process-only mode to debug an application consisting of single-threaded processes. As a convenience, the thread ID portion can be omitted. The debugger automatically enters process-only debug mode from serial debug mode when multiple processes are detected.

Table 6 Process IDs in Process-only Debug Mode

`0`	All threads of process 0 (0.*)
`*`	All threads of all processes (*.*)
`1.0`	Thread 0 of process 1 (multilevel names are valid in process-only mode)

In process-only debug mode, status and error messages are prefixed with process IDs depending on context.

9.3.4. Multilevel Debugging

The name of a thread in multilevel debug mode is the thread ID prefixed with its parent process ID. This forms a unique name for each thread across all processes. This naming scheme is valid in all debug modes. The debugger changes automatically to multilevel debug mode when at least one MPI process creates multiple threads.

Table 7 Thread IDs in Multilevel Debug Mode

`0.1`	Thread 1 of process 0
`0.*`	All threads of process 0
`*`	All threads of all processes

In multilevel debugging, mode status and error messages are prefixed with process/thread IDs depending on context.

9.4. Process/Thread Sets

You use a process/thread set (p/t-set) to restrict a debugger command to apply to a particular set of threads. A p/t-set is a set of threads drawn from all threads of all processes in the target program. Use p/t-set notation to define a p/t-set.

9.4.1. Named p/t-sets

In the following sections, you will notice frequent references to three named p/t-sets:

- The *target p/t-set* is the set of processes and threads to which a debugger command is applied. The target p/t-set is initially defined by the debugger to be the set [all] which describes all threads of all processes.
- A *prefix p/t-set* is defined when p/t-set notation is used to prefix a debugger command. For the prefixed command, the target p/t-set is the prefix p/t-set.
- The *current p/t-set* is the p/t set currently set in the debugger's environment. You can use the **`focus`** command to define the current p/t-set. Unless a prefix p/t-set overrides it, the current p/t set is used as the target p/t-set.

9.4.2. p/t-set Notation

The following rules describe how to use and construct p/t-sets:

Use a prefix p/t-set with a simple command:

```
[p/t-set prefix] command parm0, parm1, ...
```

Use a prefix p/t-set with a compound command:

```
[p/t-set prefix] simple-command [;simple-command ...]
```

p/t-id:

```
{integer|*}.{integer|*}
```

Use *p/t-id* optional notation when process-only or threads-only debugging is in effect. For more information, refer to the **`pgienv`** command.

p/t-range:

```
p/t-id:p/t-id
```

p/t-list:

```
{p/t-id|p/t-range} [, {p/t-id|p/t-range} ...]
```

p/t-set

```
[[!]{p/t-list|set-name}]
```

p/t-sets in Threads-only Debug Mode

`[0,4:6]`	Threads 0, 4, 5, and 6
`[*]`	All threads
`[*.1]`	Thread 1. Multilevel notation is valid in threads-only mode
`[*.*]`	All threads

p/t-sets in Process-only Debug Mode

`[0,2:3]`	Processes 0, 2, and 3 (equivalent to [0.*,2:3.*])
`[*]`	All processes (equivalent to [*.*])
`[0]`	Process 0 (equivalent to [0.*])
`[*.0]`	Process 0. Multilevel syntax is valid in process-only mode.
`[0:2.*]`	Processes 0, 1, and 2. Multilevel syntax is valid in process-only debug mode.

p/t-sets in Multilevel Debug Mode

`[0.1,0.3,0.5]`	Thread 1,3, and 5 of process 0
`[0.*]`	All threads of process 0
`[1.1:3]`	Thread 1, 2, and 3 of process 1
`[1:2.1]`	Thread 1 of processes 1 and 2
`[clients]`	All threads defined by named set clients
`[1]`	Incomplete; invalid in multilevel debug mode

9.4.3. Dynamic vs. Static p/t-sets

The **`defset`** command can be used to define both dynamic and static p/t-sets.

Defining a Dynamic p/t-set

The members of a dynamic p/t-set are those active threads described by the p/t-set at the time that the p/t-set is used. By default, a p/t-set is dynamic. Threads and processes are created and destroyed as the target program runs and, therefore, membership in a dynamic set varies as the target program executes.

`defset clients [*.1:3]`	Defines a dynamic named set '`clients`' whose members are threads 1, 2, and 3 of all processes that are currently active whenever

	'clients' is used. Membership in clients changes as processes are created and destroyed.

Defining a Static p/t-set

Membership in a static set is fixed when it is defined. The members of a static p/t-set are those threads described by that p/t-set when it is defined. Use a '!' to specify a static set.

`defset clients [!*.1:3]`	Defines a state named set 'clients' whose members are threads 1, 2, and 3 of those processes that are currently active at the time of the definition.

p/t-sets defined with defset are not mode-dependent and are valid in any debug mode.

9.4.4. Current vs. Prefix p/t-set

The current p/t-set is set by the **focus** command. The current p/t-set is described by the debugger prompt and depends on debug mode. For a description of the command prompt, refer to The Command Prompt. You can use a p/t-set to prefix a command that overrides the current p/t-set. The prefix p/t-set becomes the target p/t-set for the command. The target p/t-set defines the set of threads that will be affected by a command.

- In the following command line, the target p/t-set is the current p/t-set:

```
pgdbg [all] 0.0> cont
Continue all threads in all processes
```

- In contrast, a prefix p/t-set is used in the following command so that the target p/t-set is the prefix p/t-set, shown in this example in bold:

```
pgdbg [all] 0.0> [0.1:2] cont
 Continue threads 1 and 2 of process 0 only
```

In both of the above examples, the current p/t-set is the debugger-defined set [all]. In the first case, [all] is the target p/t-set. In the second case, the prefix p/t-set overrides [all] and becomes the target p/t-set. The **continue** command is applied to all active threads in the target p/t-set. Also, using a prefix p/t-set does not change the current p/t-set.

9.4.5. p/t-set Commands

You can use the following commands to collect threads and processes into logical groups.

- Use **defset** and **undefset** to manage a list of named p/t-sets.
- Use **focus** to set the current p/t-set.
- Use **viewset** to view the active members described by a particular p/t-set, or to list all the defined p/t-sets.
- Use **whichsets** to describe the p/t-sets to which a particular process/thread belongs.

Table 8 p/t-set Commands

Command	Description
defset	Define a named process/thread set. This set can later be referred to by name. The debugger stores a list of named sets.
focus	Set the target process/thread set for commands. Subsequent commands are applied to the members of this set by default..
undefset	Undefine a previously defined process/thread set. The set is removed from the list. The debugger-defined p/t-set [all] cannot be removed.
viewset	List the members of a process/thread set that currently exist as active threads, or list all the defined p/t-sets..
whichsets	List all defined p/t-sets to which the members of a process/thread set belong..

Examples of the p/t-set commands in the previous table follow.

Use **defset** to define the p/t-set `initial` to contain only thread 0:

```
pgdbg [all] 0> defset initial [0]
"initial" [0] : [0]
```

Use the **focus** command to change the current p/t-set to `initial`:

```
pgdbg [all] 0> focus [initial]
[initial] : [0]
[0]
```

Advance the thread using the current p/t-set, which is `initial`:

```
pgdbg [initial] 0> next
```

The **whichsets** command shows that thread 0 is a member of two defined p/t-sets:

```
pgdbg [initial] 0> whichsets [initial]
Thread 0 belongs to:
all
initial
```

The **viewset** command displays all threads that are active and are members of defined p/t-sets:

```
pgdbg [initial] 0> viewset
"all" [*.*] : [0.0,0.1,0.2,0.3]
"initial" [0] : [0]
```

You can use the **focus** command to set the current p/t-set back to [all]:

```
pgdbg [initial] 0> focus [all]
[all] : [0.0,0.1,0.2,0.3]
[*.*]
```

The **undefset** command undefines the initial p/t-set:

```
pgdbg [all] 0> undefset initial
p/t-set name "initial" deleted.
```

9.4.6. Using Process/Thread Sets in the GUI

The previous examples illustrate how to manage named p/t-sets using the command-line interface. A similar capability is available in the GUI. Figure 9 provides an overview of the Groups tab.

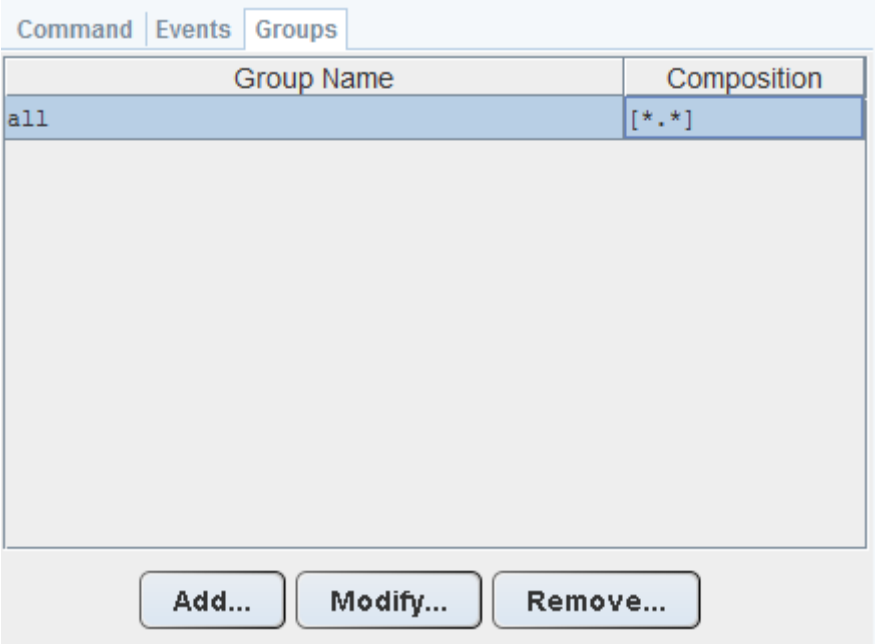


Figure 20 Groups Tab

The Groups tab contains a table with two columns: a Group Name column and a p/t-set Composition column. The entries in the Composition column are the same p/t-sets used in the command-line interface.

Using this tab you can create, select, modify and remove p/t-sets.

9.4.6.1. Create a p/t-set

To create a p/t-set in the Groups tab:

1. Click the Add button. This opens a dialog box similar to the one in Figure 21.
2. Enter the name of the p/t-set in the Group Name field and enter the p/t-set in the Composition field.
3. Click OK to add the p/t-set.

The new p/t-set appears in the Groups table. Clicking the Cancel button or closing the dialog box aborts the operation.

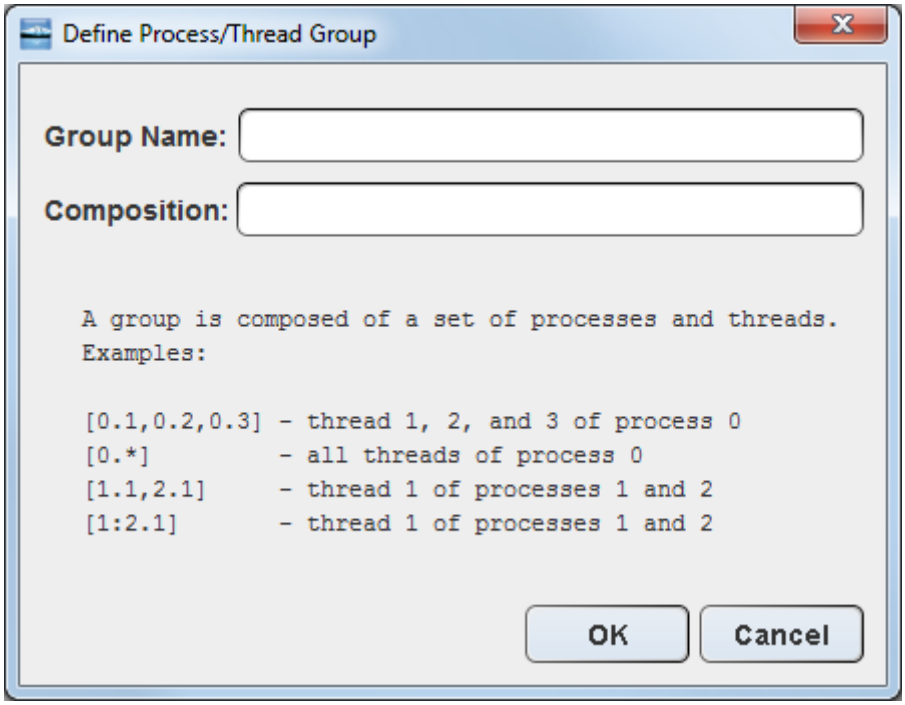


Figure 21 Process/Thread Group Dialog Box

9.4.6.2. Select a p/t-set

To select a p/t-set, click the desired p/t-set in the table. The selected p/t-set defines the Current Group used in the Apply and Display drop-down lists on the main toolbar.

9.4.6.3. Modify a p/t-set

To modify an existing p/t-set, select the desired group in the Group table and click the Modify... button. You see a dialog box similar to that in Figure 21, except that the Group Name and Composition fields contain the selected group's name and p/t-set respectively. You can edit the information in these fields and click OK to save the changes.

9.4.6.4. Remove a p/t-set

To remove an existing p/t-set, select the desired item in the Groups Table and click the Remove... button. The debugger displays a dialog box asking for confirmation of the removal request.

9.4.7. p/t-set Usage

When Current Group is selected in either the Apply or Display drop-down lists on the main toolbar, the currently selected p/t-set in the Groups tab defines the Current Group.

9.5. Command Set

For the purpose of parallel debugging, the debugger's command set is divided into three disjoint subsets according to how each command reacts to the current p/t-set. Process level and thread level commands can be parallelized. Global commands cannot be parallelized.

Table 9 Parallel Commands

Commands	Action
Process Level Commands	Parallel by current p/t-set or prefix p/t-set
Thread Level Commands	Parallel by prefix p/t-set only; current p/t-set is ignored.
Global Commands	Non-parallel commands

9.5.1. Process Level Commands

The process level commands are the debugger's control commands.

The control commands apply to the active members of the current p/t-set by default. A prefix set can be used to override the current p/t-set. The target p/t-set is the prefix p/t-set if present.

cont	next	step	stepout	synci
halt	nexti	stepi	sync	wait

Apply the **next** command to threads 1 and 2 of process 0:

```
pgdbg [all] 0.0> focus [0.1:2]
pgdbg [0.1:2] 0.0> next
```

Apply the **next** command to thread 3 of process 0 using a prefix p/t-set:

```
pgdbg [all] 0.0> [0.3] n
```

9.5.2. Thread Level Commands

The following commands are not concerned with the current p/t-set. When no p/t-set prefix is used, these commands execute in the context of the current thread of the current process by default. That is, thread level commands ignore the current p/t-set. Thread level commands can be applied to multiple threads by using a prefix p/t-set. When a prefix p/t-set is used, the commands in this section are executed in the context of each active thread described by the prefix p/t-set. The target p/t-set is the prefix p/t-set if present, or the current thread (not the current p/t-set) if no prefix p/t-set exists.

The thread level commands are:

addr	ascii	assign	bin	break*
dec	decl	disasm	do	doi
dump	entry	fp	func	hex

hwatch	line	lines	lval	noprint
oct	pc	pf	print	regs
retaddr	rval	scope	set	sizeof
sp	stack	stackdump	string	track
tracki	watch	watchi	whatis	where

* breakpoints and variants (**stop**, **stopi**, **break**, **breaki**): if no prefix p/t-set is specified, [all] is used (overriding current p/t-set).

The following actions occur when a prefix p/t-set is used:

- The threads described by the prefix are sorted per process by thread ID in increasing order.
- The processes are sorted by process ID in increasing order, and duplicates are removed.
- The command is then applied to the threads in the resulting list in order.

Without a prefix p/t-set, the **print** command executes in the context of the current thread of the current process, thread 0.0, printing rank 0:

```
pgdbg [all] 0.0> print myrank
0
```

With a prefix p/t-set, the thread members of the prefix are sorted and duplicates are removed. The **print** command iterates over the resulting list:

```
pgdbg [all] 0.0> [2:3.*,1:2.*] print myrank
[1.0] print myrank:
1
[2.0] print myrank:
2
[2.1] print myrank:
2
[2.2] print myrank:
2
[3.0] print myrank:
3
[3.2] print myrank:
3
[3.1] print myrank:
3
```

9.5.3. Global Commands

The rest of the debugger commands ignore threads and processes, or are defined globally for all threads across all processes. The current p/t-set and prefix p/t-set (if any) are ignored.

The following is a list of commands that are defined globally.

?	defset	funcs	quit	threads
/	delete	help	repeat	unalias
alias	directory	history	rerun	unbreak
arrive	disable	ignore	run	undefset
breaks	display	log	script	use

call	edit	pgienv	shell	viewset
catch	enable	proc	source	wait
cd	files	procs	status	whereis
debug	focus	pwd	thread	whichsets

9.6. Process and Thread Control

The debugger supports thread and process control everywhere in the program. Threads and processes can be advanced in groups anywhere in the program.

The control commands are:

cont	next	step	stepout	synci
halt	nexti	stepi	sync	wait

To describe those threads to be advanced, set the current p/t-set or use a prefix p/t-set.

A thread inherits the control operation of the current thread when it is created. If the current thread single-steps over an `_mp_init` call (found at the beginning of every OpenMP parallel region) using the **`next`** command, then all threads created by `_mp_init` step into the parallel region as if by the **`next`** command.

A process inherits the control operation of the current process when it is created. So if the current process returns from a call to `MPI_Init` under the control of a **`cont`** command, the new process does the same.

9.7. Configurable Stop Mode

The debugger supports configuration of how threads and processes stop in relation to one another. The debugger defines two **`pgienv`** environment variables, threadstop and procstop, for this purpose. The debugger defines two stop modes, synchronous (sync) and asynchronous (async).

Table 10 Stop Modes

Command	Result
sync	Synchronous stop mode; when one thread stops at a breakpoint (event), all other threads are stopped soon after.
async	Asynchronous stop mode; each thread runs independently of the other threads. One thread stopping does not affect the behavior of another.

Thread stop mode is set using the **`pgienv`** command as follows:

```
pgienv threadstop [sync|async]
```

Process stop mode is set using the **`pgienv`** command as follows:

```
pgienv procstop [sync|async]
```

The default is asynchronous for both thread and process stop modes. When debugging an OpenMP program, the debugger automatically enters synchronous thread stop mode in serial regions, and asynchronous thread stop mode in parallel regions.

The **pgienv** environment variables `threadstopconfig` and `procstopconfig` can be set to automatic (auto) or user defined (user) to enable or disable this behavior:

```
pgienv threadstopconfig [auto|user]
pgienv procstopconfig [auto|user]
```

Selecting the user-defined stop mode prevents the debugger from changing stop modes automatically. Automatic stop configuration is the default for both threads and processes.

9.8. Configurable Wait Mode

Wait mode describes when the debugger will accept the next command. The wait mode is defined in terms of the execution state of the program. Wait mode describes to the debugger which processes/threads must be stopped before it will accept the next command.

In certain situations, it is desirable to be able to enter commands while the program is running and not stopped at an event. The debugger prompt does not appear until all processes/threads are stopped. However, a prompt may be available before all processes/threads have stopped. Pressing <enter> at the command line brings up a prompt if it is available. The availability of the prompt is determined by the current wait mode and any pending **wait** commands.

The debugger accepts a compound statement at each prompt. Each compound statement is a sequence of semicolon-separated commands, which are processed immediately in order.

The wait mode describes when to accept the next compound statement. The debugger supports three wait modes, which can be applied to processes and/or threads.

Table 11 Wait Modes

Command	Result
all	The prompt is available only after all threads have stopped since the last control command.
any	The prompt is available only after at least one thread has stopped since the last control command.
none	The prompt is available immediately after a control command is issued.

- Thread wait mode describes which threads the debugger will wait for before accepting new commands.

  Thread wait mode is set using the **pgienv** command as follows:

  ```
  pgienv threadwait [any|all|none]
  ```
- Process wait mode describes which processes the debugger will wait for before accepting new commands.

Process wait mode is set using the **pgienv** command as follows:

```
pgienv procwait [any|all|none]
```

If process wait mode is set to none, then thread wait mode is ignored.

The debugger CLI defaults to:

```
threadwait all
procwait any
```

If the target program goes MPI parallel, then **procwait** is changed to `none` automatically.

If the target program goes thread parallel, then **threadwait** is changed to none automatically. The **pgienv** environment variable `threadwaitconfig` can be set to automatic (auto) or user defined (user) to enable or disable this behavior.

```
pgienv threadwaitconfig [auto|user]
```

Selecting the user defined wait mode prevents the debugger from changing wait modes automatically. Automatic wait mode is the default thread wait mode.

The GUI defaults to:

```
threadwait none
procwait none
```

Setting the wait mode may be necessary when invoking the GUI using the -s (script file) option. This step ensures that the necessary threads are stopped before the **next** command is processed.

The debugger also provides a **wait** command that can be used to insert explicit wait points in a command stream. **wait** uses the target p/t-set by default, which can be set to wait for any combination of processes/threads. You can use the **wait** command to insert wait points between the commands of a compound command.

The **pgienv** variables `threadwait` and `procwait` can be used to configure the behavior of **wait**. For more information, refer to **pgienv** usage in Configurable Wait Mode.

Table 12 describes the behavior of wait.

Suppose `S` is the target p/t-set. In the table,

- `P` is the set of all processes described by `S`.
- `p` is a single process.
- `T` is the set of all threads described by `S`.
- `t` is a single thread.

Table 12 Wait Behavior

Command	threadwait	procwait	Wait Set
wait	all any none	all	Wait for T

Command	threadwait	procwait	Wait Set
wait	all	any none	Wait for all threads in at least one p in P
wait	any none	any none	Wait for all t in T for at least one p in P
wait all	all any none	all	Wait for T
wait all	all	any none	Wait for all threads of at least one p in P
wait all	any none	any none	Wait for all t in T for at least one p in P
wait any	all	all	Wait for at least one thread for each process p in P
wait any	all any none	any none	Wait for at least one t in T
wait any	any none	all	Wait for at least one thread in T for each process p in P
wait none	all any none	all any none	Wait for no threads

9.9. Status Messages

The debugger can produce a variety of status messages during a debug session. This feature can be useful in the CLI if the graphical aids provided by the GUI are unavailable. Use the pgienv command to enable or disable the types of status messages produced by setting the verbose environment variable to an integer-valued bit mask:

```
pgienv verbose <bitmask>
```

The values for the bit mask, listed in the following table, control the type of status messages desired.

Table 13 Status Messages

Value	Type	Information
0x0	Standard	Disable all messages.
0x1	Standard	Report status information on current process/thread only. A message is printed when the current thread stops and when threads and processes

Value	Type	Information
		are created and destroyed. Standard messaging is the default and cannot be disabled.
0x2	Thread	Report status information on all threads of current processes. A message is reported each time a thread stops. If process messaging is also enabled, then a message is reported for each thread across all processes. Otherwise, messages are reported for threads of the current process only.
0x4	Process	Report status information on all processes. A message is reported each time a process stops. If thread messaging is also enabled, then a message is reported for each thread across all processes. Otherwise, messages are reported for the current thread only of each process.
0x8	SMP	Report SMP events. A message is printed when a process enters or exits a parallel region, or when the threads synchronize. The debugger's OpenMP handler must be enabled.
0x16	Parallel	Report process-parallel events (default).
0x32	Symbolic debug information	Report any errors encountered while processing symbolic debug information (e.g. ELF, DWARF2).

9.10. The Command Prompt

The debugger command prompt reflects the current debug mode, as described in Debug Modes.

In serial debug mode, the command prompt looks like this:

```
pgdbg>
```

In threads-only debug mode, the debugger displays the current p/t-set in square brackets followed by the ID of the current thread:

```
pgdbg [all] 0>
Current thread is 0
```

In process-only debug mode, the debugger displays the current p/t-set in square brackets followed by the ID of the current process:

```
pgdbg [all] 0>
Current process is 0
```

In multilevel debug mode, the debugger displays the current p/t-set in square brackets followed by the ID of the current thread prefixed by the id of its parent process:

```
pgdbg [all] 1.0>
Current thread 1.0
```

The **pgienv promptlen** variable can be set to control the number of characters devoted to printing the current p/t-set at the prompt.

9.11. Parallel Events

This section describes how to use a p/t-set to define an event across multiple threads and processes. Events, such as breakpoints and watchpoints, are user-defined events. User-defined events are thread-level commands, described in Thread Level Commands.

Breakpoints, by default, are set across all threads of all processes. A prefix p/t-set can be used to set breakpoints on specific processes and threads. For example:

```
i) pgdbg [all] 0> b 15
ii) pgdbg [all] 0> [all] b 15
iii) pgdbg [all] 0> [0.1:3] b 15
```

(i) and (ii) are equivalent. (iii) sets a breakpoint only in threads 1,2,3 of process 0.

By default, all other user events are set for the current thread only. A prefix p/t-set can be used to set user events on specific processes and threads. For example:

```
i) pgdbg [all] 0> watch glob
ii) pgdbg [all] 0> [*] watch glob
```

(i) sets a watchpoint for glob on thread 0 only. (ii) sets a watchpoint for glob on all threads that are currently active.

When a process or thread is created, it inherits all of the breakpoints defined for the parent process or thread. All other events must be defined explicitly after the process or thread is created. All processes must be stopped to add, enable, or disable a user event.

Events may contain if and do clauses. For example:

```
pgdbg [all] 0> [*] break func if (glob!=0) do {set f = 0}
```

The breakpoint fires only if glob is non-zero. The do clause is executed if the breakpoint fires. The if and do clauses execute in the context of a single thread. The conditional in the if clause and the body of the do execute in the context of a single thread, the thread that triggered the event. The conditional definition as above can be restated as follows:

```
[0] if (glob!=0) {[0] set f = 0}
[1] if (glob!=0) {[1] set f = 0}
...
```

When thread 1 hits func, glob is evaluated in the context of thread 1. If glob evaluates to non-zero, f is bound in the context of thread 1 and its value is set to 0.

Control commands can be used in do clauses, however they only apply to the current thread and are only well defined as the last command in the do clause. For example:

```
pgdbg [all] 0> [*] break func if (glob!=0) do {set f = 0; c}
```

If the **`wait`** command appears in a `do` clause, the current thread is added to the wait set of the current process. For example:

```
pgdbg [all] 0> [*] break func if (glob!=0) do {set f = 0; c; wait}
```

If conditionals and `do` bodies cannot be parallelized with prefix p/t-sets. For example, the following command is illegal:

```
pgdbg [all] 0> break func if (glob!=0) do {[*] set f = 0} ILLEGAL
```

The body of a do statement cannot be parallelized.

9.12. Parallel Statements

This section describes how to use a p/t-set to define a statement that executes for multiple threads and processes.

9.12.1. Parallel Compound/Block Statements

Each command in a compound statement is executed in order. The target p/t-set is applied to all statements in a compound statement. The following two examples (i) and (ii) are equivalent:

```
i) pgdbg [all] 0>[*] break main; cont; wait; print f@11@;i
ii) pgdbg [all] 0>[*] break main; [*]cont; [*]wait; [*]print f@11@;i
```

Use the **`wait`** command if subsequent commands require threads to be stopped, as the **`print`** command in the example does.

The `threadwait` and `procwait` environment variables do not affect how commands within a compound statement are processed. These **`pgienv`** environment variables describe under what conditions (state of program) the debugger should accept the next (compound) statement.

9.12.2. Parallel If, Else Statements

A prefix p/t-set can be used to parallelize an if statement. An if statement executes in the context of the current thread by default. The following example:

```
pgdbg [all] 0> [*] if (i==1) {break func; c; wait} else {sync func2}
```

is equivalent to the following pseudo-code:

```
for the subset of [*] where (i==1)
break func; c; wait; for the subset of [*] where (i!=1) sync func2
```

9.12.3. Parallel While Statements

A prefix p/t-set can be used to parallelize a while statement. A while statement executes in the context of the current thread by default. The following example:

```
pgdbg [all] 0> [*] while (i<10) {n; wait; print i}
```

is equivalent to the following pseudo-code:

```
loop:
if the subset of [*] is the empty set
goto done
endif
for the subset [s] of [*] where (i<10)
[s]n; [s]wait; [s]print i;
endfor
goto loop
```

The while statement terminates when either the subset of the target p/t-set matching the while condition is the empty set, or a return statement is executed in the body of the while.

9.12.4. Return Statements

The return statement is defined only in serial context since it cannot return multiple values. When return is used in a parallel statement, it returns the last value evaluated.

Chapter 10. PARALLEL DEBUGGING WITH OPENMP

This section provides information on how to debug OpenMP applications. Before reading this section, review the information in Parallel Debugging Overview.

10.1. OpenMP and Multi-thread Support

The debugger provides full control of threads in parallel regions. Commands can be applied to all threads, a single thread, or a group of threads. The debugger uses the native thread numbering scheme for OpenMP applications to identify threads; for other types of multi-threaded applications thread numbering is arbitrary. OpenMP private data can be accessed accurately for each thread. The debugger provides understandable status displays regarding per-thread state and location.

Advanced features provide for configurable thread stop modes and wait modes, allowing debugger operation that is concurrent with application execution.

10.2. Multi-thread and OpenMP Debugging

The debugger automatically attaches to new threads as they are created during program execution. The debugger reports when a new thread is created and the thread ID of the new thread is printed.

```
([1] New Thread)
```

The system ID of the freshly created thread is available through the **threads** command. You can use the **procs** command to display information about the parent process.

The debugger maintains a conceptual current thread. When using the command-line interface, the current thread is chosen by using the **thread** command.

```
pgdbg [all] 2> thread 3
pgdbg [all] 3>
```

When using the GUI, the current thread can be selected using the Current Thread drop-down list or by clicking in the Thread Grid. A subset of debugger commands known as thread-level commands apply only to the current thread. For more information, refer to Thread Level Commands.

The **threads** command lists all threads currently employed by an active program. It displays each thread's unique thread ID, system ID (OS process ID), execution state (running, stopped, signaled, exited, or killed), signal information and reason for stopping, and the current location if stopped or signaled. An arrow (=>) indicates the current thread. The process ID of the parent is printed in the top left corner. The **threads** command does not change the current thread.

```
pgdbg [all] 3> threads
0 ID PID STATE SIGNAL LOCATION
=> 3 18399 Stopped SIGTRAP main line: 31 in "omp.c" address: 0x80490ab
 2 18398 Stopped SIGTRAP main line: 32 in "omp.c" address: 0x80490cf
 1 18397 Stopped SIGTRAP main line: 31 in "omp.c" address: 0x80490ab
 0 18395 Stopped SIGTRAP f line: 5 in "omp.c" address: 0x8048fa0
```

In the GUI, thread state is represented by a color in the process/thread grid.

Table 14 Thread State Is Described Using Color

Thread State	Description	Color
Stopped	The thread is stopped at a breakpoint, or was directed to stop by the debugger.	Red
Signaled	The thread is stopped due to delivery of a signal.	Blue
Running	The thread is running.	Green
Exited or Killed	The thread has been killed or has exited.	Black

10.3. Debugging OpenMP Private Data

The debugger supports debugging of OpenMP private data for all supported languages. When an object is declared private in the context of an OpenMP parallel region, it essentially means that each thread team has its own copy of the object. This capability is shown in the following Fortran and C/C++ examples, where the loop index variable i is private by default.

FORTRAN example:

```
 program omp_private_data
   integer array(8)
   call omp_set_num_threads(2)
!$OMP PARALLEL DO
   do i=1,8
     array(i) = i
   enddo
!$OMP END PARALLEL DO
   print *, array
   end
```

C/ C++ example:

```
#include <omp.h>
int main ()
{
  int i;
  int array[8];
  omp_set_num_threads(2);
```

```
#pragma omp parallel
{
#pragma omp for
  for (i = 0; i < 8; ++i) {
    array[i] = i;
  }
}
  for (i = 0; i < 8; ++i) {
    printf("array[%d] = %d\n",i, array[i]);
  }
  }
```

Compile the examples with a PGI compiler. The display of OpenMP private data in the resulting executables as debugged is as follows:

```
pgdbg [all] 0> [*] print i
[0] print i:
1
[1] print i:
5
```

The example specifies [*] for the p/t-set to execute the **print** command on all threads. Figure 22 shows the values for i in the GUI using a Custom Window.

All Threads is selected in the Context drop-down list to display the value on both threads.

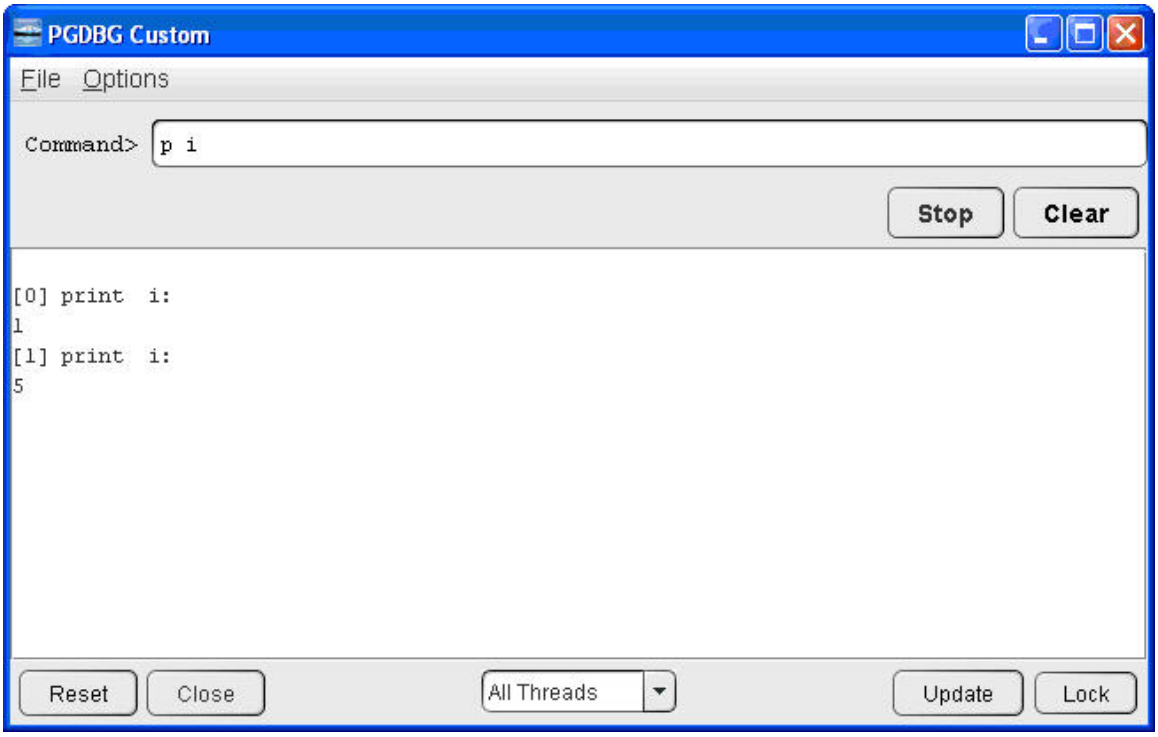


Figure 22 OpenMP Private Data in Debugger GUI

Chapter 11. PARALLEL DEBUGGING WITH MPI

The debugger is capable of debugging multi-process MPI and hybrid multi-thread/multi-process applications. Hybrid applications use MPI to communicate between multi-threaded or OpenMP processes. This section begins with a general overview of how to debug parallel MPI applications before detailing how to launch MPI applications under debug using the various supported platforms and versions of MPI.

For information on compiling a program using MPI, refer to 'Using MPI' in the *PGI Compiler User's Guide* available at www.pgroup.com/resources/docs.htm.

11.1. MPI and Multi-Process Support

The debugger can debug MPI applications running on the local system and, on Linux, applications distributed across a cluster. MPI applications must be started under debugger control. Process identification uses the MPI rank within MPI_COMM_WORLD.

MPI debugging is supported on Linux, macOS, and Windows. Application debugging is supported up to a maximum of 256 processes and 64 threads per process, but may be limited by your PGI license keys. A PGI floating license is required to enable the debugger's distributed debugging capabilities.

11.2. MPI on Linux

PGI products for Linux ship with a PGI-built version of Open MPI. In addition, users with permanent license can download PGI-built versions of MVAPICH2 and MPICH v3.

PGI products for Linux support SGI MPI. Any SGI MPI program configured to build with PGI compilers can be debugged with the PGI debugger without any additional configuration considerations.

11.3. MPI on macOS

PGI products for macOS ship with a PGI-built version of MPICH v3.

11.4. MPI on Windows

PGI products on Windows ship with Microsoft's HPC Pack 2012 SP1 MS-MPI redistributable package; MS-MPI is installed by default. The PGI debugger can debug MS-MPI programs running locally; cluster debugging is not supported.

11.5. Deprecated Support for MPICH1, MPICH2, MVAPICH1

PGI has deprecated support for MPICH1, MPICH2 and MVAPICH1. For instructions on debugging an application using one of these distributions of MPI, please refer to the documentation for release 13.10 (or prior).

11.6. Building an MPI Application for Debugging

In general, simply add `-g` to the compilation and linking of an MPI application to enable the generation of debug information. Instead of invoking a compiler directly, most versions of MPI include wrapper files (i.e., `mpicc`, `mpif90`). That said, the PGI compilers have direct support for some MPI distributions via the `-Mmpi` option. For example, sub-options to `-Mmpi` are available for MPICH, SGI, and MS-MPI.

11.7. The MPI Launch Program

In the following sections, the term *launcher* indicates the MPI launch program. The debugger uses `mpiexec` as the default launcher. If the location of the launcher in your MPI distribution is not in your PATH environment variable, you must provide the debugger with the full path to the launcher, including the name of the launch tool itself. If the location of the launcher is in your PATH, then you just need to provide the name of the launcher and then only if the launcher is not mpiexec.

11.7.1. Launch Debugging Using the Connection Tab

You can use the debugger's Connections tab to start debug sessions of programs that were built using MPICH, MS-MPI, MVAPICH2 and Open MPI. Programs built with SGI MPI can be debugged with the debugger GUI but the session must be started when the debugger is launched.

To configure an MPI debug session, first select the MPI check box near the top of the Connections tab to enable its MPI-specific fields. Use the Command field to specify the MPI launch program and the Arguments field to pass arguments (if any) to the MPI launch program. The same rules discussed earlier about specifying the MPI launch program apply here as well. The examples in the following table use eight processes and a file named 'hosts' to illustrate how you might go about filling in the Command and Argument fields for different MPI distributions:

MPI Distribution	Command	Arguments
MPICH	mpiexec	-np 8
MS-MPI	mpiexec	-n 8
MVAPICH2	mpirun_rsh	-rsh -hostfile hosts-np 8
Open MPI	mpiexec	-np 8

11.7.2. Launch Debugging From the Command Line

Debugging sessions for all MPI distributions supported by PGI can be started when the debugger is launched. The requirements of the MPI distribution itself determine the command required to start the debugger. The debugger's text and graphical modes are both supported from a command line launch.

11.7.3. MPICH

To launch debugging of an MPICH program from the command line, use this command format:

```
pgdbg [-text] -mpi[:<launcher_path>] <launcher_args> [ -program_args
 arg1,...argn ]
```

The launcher for MPICH v3 is `mpiexec`. If the path to `mpiexec` is not part of the PATH environment variable, then you must specify the full path to mpiexec in the **pgdbg** command line. If `mpiexec` is in your PATH, you don't have to supply an argument to the `-mpi` option at all because `mpiexec` is the default launcher name.

For example, to debug an MPICH program named `cpi`, which takes one program argument, using four processes, use a command like this one:

```
$ pgdbg -mpi -np 4 cpi -program_args 11000
```

11.7.4. MS-MPI

MS-MPI applications can be run and debugged locally. Debugging is not supported on Windows clusters.

To invoke the PGI debugger to debug an MS-MPI application locally, use this command format:

```
pgdbg -mpi[=<launcher_path>] <launcher_args> [-program_args arg1,...argn]
```

When MS-MSPI is installed, it adds the location of `mpiexec` to the system's PATH environment variable so you do not need to supply an argument to the `-mpi` option. If the path to `mpiexec` is not part of your PATH, then you must specify the full path to `mpiexec` in the **pgdbg** command line.

For example, to debug an MS-MPI application named `prog` using four processes running on the host system, use a command like this one:

```
$ pgdbg -mpi -n 4 prog.exe
```

11.7.5. MVAPICH2

To launch debugging of an MVAPICH2 program from the command line, use this command format:

```
pgdbg [-text] -mpi=<launcher_path> <launcher_args> [ -program_args
 arg1,...argn ]
```

For MVAPICH2, the MPI launcher is `mpirun_rsh`, so use `-mpi:mpirun_rsh`.

If the path to `mpirun_rsh` is not in your PATH environment variable, then you must specify the full path to `mpirun_rsh` using the `-mpi` option.

For example, to start debugging an MVAPICH2 program called `fpi` using four processes and a host file called 'hosts' use a command like:

```
$ pgdbg -mpi=mpirun_rsh -rsh -hostfile hosts -np 4 ./fpi
```

11.7.6. Open MPI

Open MPI debugging start-up is the same as MPICH debugging start-up. To launch debugging of an Open MPI program from the command line, use this command format:

```
pgdbg [-text] -mpi[=<launcher_path>] <launcher_args> [ -program_args
 arg1,...argn ]
```

The launcher for Open MPI is `mpiexec`. If the path to `mpiexec` is not part of the PATH environment variable, then you must specify the full path to `mpiexec` in the `pgdbg` command line. If `mpiexec` is in your PATH, you don't have to supply an argument to the `-mpi` option at all because `mpiexec` is the default launcher name.

For example, to debug an Open MPI program named `cpi`, which takes one program argument, using four processes, use a command like this one:

```
$ pgdbg -mpi -np 4 cpi -program_args 11000
```

11.7.7. SGI MPI

Use the debugger's `-sgimpi` option instead of `-mpi` when you want to debug an SGI MPI program. Otherwise the command format for launching SGI MPI debugging is similar to that used when debugging programs built with other distributions of MPI:

```
pgdbg [-text] -sgimpi[=<launcher_path>] <launcher_args> [ -program_args
 arg1,...argn ]
```

The SGI MPI launch program is `mpirun`. You can use `-sgimpi` without an argument if the location of `mpirun` is in your PATH. If `mpirun` is not in your PATH, then you must specify the full path to it, including the `mpirun` command, as part of the `-sgimpi` option.

When running or debugging an SGI MPI program, you need to include the SGI MPI lib directory in the LD_LIBRARY_PATH environment variable.

For example, provided `mpirun` is in your PATH, to debug an SGI MPI program named `fpi` using four processes, use a command like:

```
$ pgdbg -sgimpi -np 4 fpi
```

When an SGI MPI debugging session starts up, a number of messages are printed to the command prompt. These messages reflect how the debugger is setting up the session and can be safely ignored.

Program input from `stdin` is disabled when running an SGI MPI program using the debugger.

11.8. Process Control

Here are some general things to consider when debugging an MPI program:

- Use the Groups tab (p/t-sets in the CLI) to focus on a set of processes. Be mindful of process dependencies.
- For a running process to receive a message, the sending process must be allowed to run.
- Process synchronization points, such as MPI_Barrier, do not return until all processes have hit the sync point.
- MPI_Finalize acts as an implicit barrier except when using the now deprecated MPICH1, where Process 0 returns while Processes 1 through n-1 exit.

You can apply a control command, such as **`cont`** or **`step`**, to a stopped process while other processes are running. A control command applied to a running process is applied to the stopped threads of that process and is ignored by its running threads.

The debugger automatically switches to process wait mode `none` as soon as it attaches to its first MPI process. See the pgienv command and Configurable Wait Mode for details.

The debugger automatically attaches to new MPI processes as they are created by the running MPI application. The debugger displays an informational message as it attaches to the freshly created processes.

```
 ([1] New Process)
```

The MPI global rank is printed with the message.

You can use the `procs` command to list the host and the PID of each process by rank. The current process is indicated by an arrow (=>). You can use the `proc` command to change the current process by process ID.

```
pgdbg [all] 0.0> proc 1; procs
Process 1: Thread 0 Stopped at 0x804a0e2, function main, file MPI.c, line 30
 #30: aft=time(&aft);
   ID  IPID   STATE    THREADS HOST
    0  24765  Stopped   1      local
 => 1  17890  Stopped   1      red2.wil.st.com
```

The execution state of a process is described in terms of the execution state of its component threads. For a description of how thread state is represented in the GUI, refer to Thread State Is Described Using Color.

The debugger command prompt displays the current process and the current thread. In the above example, the current process was changed to process 1 by the `proc 1` command and the current thread of process 1 is 0; this is written as 1.0:

```
pgdbg [all] 1.0>
```

For a complete description of the prompt format, refer to Process and Thread Control.

The following rules apply during a debug session:

- The debugger maintains a conceptual current process and current thread.
- Each active process has a thread set of size >=1.
- The current thread is a member of the thread set of the current process.

Certain commands, when executed, apply only to the current process or the current thread. For more information, refer to Process Level Commands and Thread Level Commands.

The PGI license keys restrict the total number of MPI processes that can be debugged. In addition, there are internal limits on the number of threads per process that can be debugged.

11.9. Process Synchronization

Use the debugger's **sync** command to synchronize a set of processes to a particular point in the program. The following command runs all processes to MPI_Finalize:

```
pgdbg [all] 0.0> sync MPI_Finalize
```

The following command runs all threads of process 0 and process 1 to MPI_Finalize:

```
pgdbg [all] 0.0> [0:1.*] sync MPI_Finalize
```

A **sync** command only successfully syncs the target processes if the sync address is well defined for each member of the target process set, and all process dependencies are satisfied. If these conditions are not met, a member could wait forever for a message. The debugger cannot predict if a text address is in the path of an executing process.

11.10. MPI Message Queues

The debugger can dump MPI message queues. When using the CLI, use the mqdump command, described in Memory Access. When using the GUI, the message queues are displayed in the MPI Messages debug information tab.

The following error message may appear in the MPI Messages tab or when invoking **mqdump**:

```
ERROR: MPI Message Queue library not found.
Try setting 'PGDBG_MQS_LIB_OVERRIDE'environment variable
or set via the debugger command: pgienv mqslib <path>.
```

If this message is displayed, then the `PGDBG_MQS_LIB_OVERRIDE` environment variable should be set to the absolute path of `libtvmpich.so` or another shared object that is compatible with the version of MPI being used. The default path can also be overridden via the **pgienv** variable **mqslib**.

Microsoft MPI does not currently provide support for dumping message queues.

11.11. MPI Groups

The debugger identifies each process by its MPI_COMM_WORLD rank. In general, the debugger currently ignores MPI groups.

11.12. Use halt instead of Ctrl+C

Entering Ctrl+C at the debugger's command line can be used to halt all running processes. However, this is not the preferred method to use while debugging an MPICH1 program. (MPICH1 support has been deprecated.) The debugger automatically switches to process wait mode none (**`pgienv procwait none`**) as soon as it attaches to its first MPI process.

Setting **`pgienv procwait none`** allows commands to be entered while there are running processes, which allows the use of the **`halt`** command to stop running processes without the use of Ctrl+C.

`halt` cannot interrupt a **`wait`** command. Ctrl+C must be used for this.

In MPI debugging, **`wait`** should be used with care.

11.13. SSH and RSH

By default, the debugger uses rsh for communication between remote debugger components. The debugger can also use `ssh` for secure environments. The environment variable `PGRSH` should be set to `ssh` or `rsh` to indicate the desired communication method.

If you opt to use `ssh` as the mechanism for launching the debugger's remote components, you may want to do some additional configuration. The default configuration of `ssh` can result in a password prompt for each remote cluster node on which the debugger runs. Check with your network administrator to make sure that you comply with your local security policies when configuring `ssh`.

The following steps provide one way to configure SSH to eliminate this prompt. These instructions assume `$HOME` is the same on all nodes of the cluster.

```
$ ssh-keygen -t dsa
$ eval ´ssh-agent -s´
$ ssh-add
<make sure that $HOME is not group-writable>
$ cd $HOME/.ssh
$ cat id_dsa.pub >> authorized_keys
```

Then for each cluster node you use in debugging, use:

```
$ ssh <host>
```

A few things that are important related to this example are these:

- The **ssh-keygen** command prompts for a passphrase that is used to authenticate to the ssh-agent during future sessions. The passphrase can be anything you choose.
- Once you answer the prompts to make the initial connection, subsequent connections should not require further prompting.
- The **ssh-agent -s** command is correct for sh or bash shells. For csh shells, use **ssh-agent -c**.

After logging out and logging back in, the ssh-agent must be restarted and reauthorized. For example, in a bash shell, this is accomplished as follows:

```
$ eval ´ssh-agent -s´
$ ssh-add
```

You must enter the passphrase that was initially given to ssh-add to authenticate to the ssh-agent.

For further information, consult your ssh documentation.

11.14. Using the CLI

11.14.1. Setting DISPLAY

To use MPI debugging in text mode, be certain that the DISPLAY variable is undefined in the shell that is invoking mpirun. If this variable is set, you can undefine it by using one of the following commands:

For sh/bash users, use this command:

```
$ unset DISPLAY
```

For csh/tcsh users, use this command:

```
% unsetenv DISPLAY
```

11.14.2. Using Continue

When debugging an MPI job after invoking the debugger's CLI with the `-mpi` option, each process is stopped before the first assembly instruction in the program. Continuing execution using **step** or **next** is not appropriate; instead, use the **cont** command.

Chapter 12. PARALLEL DEBUGGING OF HYBRID APPLICATIONS

The debugger supports debugging hybrid multi-thread/multi-process applications that use MPI to communicate between multi-threaded or OpenMP processes. Multi-threaded and OpenMP applications may be run using more threads than the available number of CPUs, and MPI applications may allocate more than one process to a cluster node. The debugger supports debugging the supported types of applications regardless of how well the requested number of threads matches the number of CPUs or how well the requested number of processes matches the number of cluster nodes.

12.1. Multilevel Debug Mode

As described in Debug Modes, the debugger can operate in four debug modes. The mode determines a short form for uniquely naming threads and processes.

The debug mode is set automatically or can be set manually using the pgienv command.

When the debugger detects multilevel debugging, it sets the debug mode to multilevel. To manually set the debug mode to multilevel, use the **pgienv** command:

```
pgdbg> pgienv mode multilevel
```

12.2. Multilevel Debugging

The name of a thread in multilevel debug mode is the thread ID prefixed with its parent process ID. This forms a unique name for each thread across all processes. This naming scheme is valid in all debug modes. The debugger changes automatically to multilevel debug mode from process-only debug mode or threads-only debug mode when at least one MPI process creates multiple threads.

Thread IDs in multilevel debug mode

`0.1`	Thread 1 of process 0
`0.*`	All threads of process 0
`*`	All threads of all processes

In multilevel debug, mode status and error messages are prefixed with process/thread IDs depending on context. Further, in multilevel debug mode, the debugger displays the current p/t-set in square brackets followed by the ID of the current thread prefixed by the ID of its parent process:

```
pgdbg [all] 1.0>
Current thread 1.0
```

For more information on p/t sets, refer to Process/Thread Sets.

Chapter 13. COMMAND REFERENCE

This section describes the debugger's command set in detail, grouping the commands by these categories:

Conversions	Miscellaneous	Process-Thread Sets	Scope
Events	Printing Variables and Expressings	Program Locations	Symbols and Expressions
Memory Access	Process Control	Register Access	Target

For an alphabetical listing of all the commands, with a brief description of each, refer to the Command Summary.

13.1. Notation Used in Command Sections

The command sections that follow use these conventions for the command names and arguments, when the command accepts one.

- Command names may be abbreviated by omitting the portion of the command name enclosed in brackets ([]).
- Argument names are italicized.
- Argument names are chosen to indicate what kind of argument is expected.
- Arguments enclosed in brackets ([]) are optional.
- Two or more arguments separated by a vertical line (|) indicate that any one of the arguments is acceptable.
- An ellipsis (...) indicates an arbitrarily long list of arguments.
- Other punctuation, such as commas and quotes, must be entered as shown.

Syntax examples

Example 1:

```
lis[t] [count | lo:hi | routine | line,count]
```

This syntax indicates that the command **`list`** may be abbreviated to **`lis`**, and that it can be invoked without any arguments or with *one* of the following: an integer count, a line range, a routine name, or a line and a count.

Example 2:

```
att[ach] pid [exe]
```

This syntax indicates that the command **attach** may be abbreviated to **att**, and, when invoked, must have a process ID argument, *pid*. Optionally you can specify an executable file, *exe*.

13.2. Process Control

The following commands control the execution of the target program. The debugger lets you easily group and control multiple threads and processes. For more details, refer to Basic Process and Thread Naming.

13.2.1. attach

```
att[ach] pid [exe]
```

Attach to a running process with process ID `pid`. Use `exe` to specify the absolute path of the executable file. For example, `attach 1234` attempts to attach to a running process whose process ID is 1234. You may enter something like `attach 1234 /home/demo/a.out` to attach to a process ID 1234 called `/home/demo/a.out`.

The PGI debugger attempts to infer the arguments of the attached program. If the debugger fails to infer the argument list, then the program behavior is undefined if the run or rerun command is executed on the attached process.

The `stdio` channel of the attached process remains at the terminal from which the program was originally invoked.

The **attach** command is not supported for MPI programs.

Note that on Ubuntu systems `ptrace` is, by default, restricted from attaching to non-child processes. As a result, the PGI debugger, as well as other debuggers utilizing `ptrace` on Linux, is not able to attach to a process for debugging. There are two workarounds:

1) To temporarily allow attaching for debug, modify the file `/proc/sys/kernel/yama/ptrace-scope` and change the content

```
1
```

to

```
0
```

2) To permanently allow attaching for debug, modify the file `/etc/sysctl.d/10-ptrace.conf` and change the line

```
kernel.yama.ptrace_scope = 1
```

to

```
kernel.yama.ptrace_scope = 0
```

13.2.2. cont

```
c[ont]
```

Continue execution from the current location.

13.2.3. debug

```
de[bug] [target [ arg1...
argn]]
```

Load the specified target program with optional command-line arguments.

13.2.4. detach

```
det[ach]
```

Detach from the current running process.

13.2.5. halt

```
halt [command]
```

Halt the running process or thread.

13.2.6. load

```
lo[ad] [program [args]]
```

Without arguments, **load** prints the name and arguments of the program being debugged. With arguments, **load** loads the specified *program* for debugging. Provide program arguments as needed.

13.2.7. next

```
n[ext] [count]
```

Stop after executing one source line in the current routine. This command steps over called routines. The *count* argument stops execution only after executing *count* source lines.

13.2.8. nexti

```
nexti [count]
```

Stop after executing one instruction in the current routine. This command steps over called routines. The *count* argument stops execution only after executing *count* instructions.

13.2.9. proc

```
proc [id]
```

Set the current process to the process identified by *id*. When issued with no argument, **proc** lists the location of the current thread of the current process in the current program. For information on how processes are numbered, refer to Using the CLI.

13.2.10. procs

```
procs
```

Print the status of all active processes, listing each process by its logical process ID.

13.2.11. quit

```
q[uit]
```

Terminate the debugging session.

13.2.12. rerun

```
rer[un] [arg0
arg1 ... argn] [< inputfile ] [ [ > | >& | >> | >>& ] outputfile ]
```

The **rerun** command is the same as **run** with one exception: if no args are specified with **rerun**, then no args are used when the program is launched.

13.2.13. run

```
ru[n] [arg0 arg1
... argn] [< inputfile ] [ [ > | >& | >> | >>& ] outputfile ]
```

Execute the program from the beginning. If arguments *arg0*, *arg1*, and so on are specified, they are set up as the command-line arguments of the program. Otherwise, the arguments that were used with the previous **run** command are used. Standard input and standard output for the target program can be redirected using < or > and an input or output filename.

13.2.14. setargs

```
setargs [arg1, arg2, ... argn]
```

Set program arguments for use by the **run** command. The **rerun** command does not use the arguments specified by **setargs**.

13.2.15. step

```
s[tep] [count | count]
```

Stop after executing one source line. This command steps into called routines. The *count* argument stops execution after executing *count* source lines. The *up* argument stops execution after stepping out of the current routine (see stepout).

13.2.16. stepi

```
stepi [count | up]
```

Stop after executing one instruction. This command steps into called routines. The *count* argument stops execution after executing *count* instructions. The *up* argument stops the execution after stepping out of the current routine (see stepout).

13.2.17. stepout

```
stepo[ut]
```

Stop after returning to the caller of the current subroutine. This command sets a breakpoint at the current return address and continues execution to that point. For

this command to work correctly, it must be possible to compute the value of the return address. Some subroutines, particularly terminal (i.e. leaf) subroutines at higher optimization levels, may not set up a stack frame. Executing **`stepout`** from such a routine causes the breakpoint to be set in the caller of the most recent routine that set up a stack frame. This command halts execution immediately upon return to the calling subroutine.

13.2.18. sync

```
sy[nc] line | func
```

Advance to the specified source location, either the specified *line* or the first line in the specified function `func`, ignoring any user-defined events.

13.2.19. synci

```
synci  addr | func
```

Advance to the specified address `addr`, or to the first address in the specified function *`func`*, ignoring any user-defined events.

13.2.20. thread

```
thread [number]
```

Set the current thread to the thread identified by *number*; where *number* is a logical thread ID in the current process' active thread list. When issued with no argument, **`thread`** lists the current program location of the currently active thread.

13.2.21. threads

```
threads
```

Print the status of all active threads. Threads are grouped by process. Each process is listed by its logical process ID. Each thread is listed by its logical thread ID.

13.2.22. wait

```
wait [any | all | none]
```

Return the debugger's prompt only after specific processes or threads stop.

13.3. Process-Thread Sets

The following commands deal with defining and managing process thread sets. For a detailed discussion of process-thread sets, refer to Process/Thread Sets.

13.3.1. defset

```
defset name [p/t-set]
```

Assign a *name* to a process/thread set. In other words, define a named set of processes/threads. This set can then be referred to by its *name*. A list of named sets is stored by the debugger.

13.3.2. focus

```
focus [p/t-set]
```

Set the target process/thread set for debugger commands. Subsequent commands are applied to the members of this set by default.

13.3.3. undefset

```
undefset [name | -all]
```

Remove a previously defined process/thread set from the list of process/thread sets. The debugger-defined p/t-set [all] cannot be removed.

13.3.4. viewset

```
viewset [name]
```

List the active members of the named process/thread set. If no process/thread set is given, list the active members of all defined process/thread sets.

13.3.5. whichsets

```
whichsets [p/t-set]
```

List all defined p/t-sets to which the members of a process/thread set belong. If no process/thread set is specified, the target process/thread set is used.

13.4. Events

The following commands deal with defining and managing events.

13.4.1. break

```
b[reak]
b[reak] line [if (condition)] [do {commands}] [hit [>|*] <num>]
b[reak] routine [if (condition)] [do {commands}] [hit [>|*] <num>]
```

When no arguments are specified, the **break** command prints the current breakpoints. Otherwise, set a breakpoint at the indicated line or routine. If a routine is specified, and the routine was compiled for debugging, then the breakpoint is set at the start of the first statement in the routine (after the routine's prologue code). If the routine was not compiled for debugging, then the breakpoint is set at the first instruction of the routine, prior to any prologue code. This command interprets integer constants as line numbers. To set a breakpoint at an address, use the addr command to convert the constant to an address, or use the breaki command.

When a condition is specified with *if*, the breakpoint occurs only when the specified condition is true. If *do* is specified with a command or several commands as an argument, the command or commands are executed when the breakpoint occurs. If *hit* is specified with a number as an argument, the breakpoint occurs only when the hit count is equal to that number. Optional operators (greater than, multiple of) can be used to modify the hit count condition.

The following table provides examples of using **break** to set breakpoints at various locations.

This break command...	Sets breakpoints...
`break 37`	at line `37` in the current file
`break "xyz.c"@37`	at line `37` in the file `xyz.c`
`break main`	at the first executable line of routine main
`break {addr 0xf0400608}`	at address `0xf0400608`
`break {line}`	at the current line
`break {pc}`	at the current address

The following command stops when routine `xyz` is entered only if the argument `n` is greater than 10.

```
break xyz if(xyz@n > 10)
```

The next command prints the value of `n` and performs a stack trace every time line 100 in the current file is reached.

```
break 100 do {print n; stack}
```

The next command stops at line 111 when the hit count is a multiple of 5.

```
break 111 hit *5
```

13.4.2. breaki

```
breaki
breaki routine [if (condition)] [do {commands}] [hit [>|*] <num>]
breaki addr [if (condition)] [do {commands}] [hit [>|*] <num>]
```

When no arguments are specified, the **breaki** command prints the current breakpoints. Otherwise, this command sets a breakpoint at the indicated address *addr* or *routine*.

- If a routine is specified, the breakpoint is set at the first address of the routine. This means that when the program stops at this type of breakpoint the prologue code which sets up the stack frame will not yet have been executed. As a result, values of stack arguments may not yet be correct.
- Integer constants are interpreted as addresses.
- To specify a line, use the lines command to convert the constant to a line number, or use the break command.
- The *if*, *do* and *hit* arguments are interpreted in the same way as for the break command.

The following table provides examples of setting breakpoints using **breaki**.

This breaki command...	Sets breakpoints...
`breaki 0xf0400608`	at address `0xf0400608`
`breaki {line 37}`	at line `37` in the current file
`breaki "xyz.c"@37`	at line `37` in the file `xyz.c`
`breaki main`	at the first executable address of routine main
`breaki {line}`	at the current line
`breaki {pc}`	at the current address

In the following example, when `n` is greater than 3, the following command stops and prints the new value of `n` at address `0x6480`:

```
breaki 0x6480 if(n>3) do {print "n=", n}
```

13.4.3. breaks

```
breaks
```

Display all the existing breakpoints.

13.4.4. catch

```
catch  [sig:sig] [sig [, sig...]]
```

When no arguments are specified, the **catch** command prints the list of signals being caught. With the *sig:sig* argument, this command catches the specified range of signals. With a list of signals, catch the signals with the specified number(s). When signals are caught, the debugger intercepts the signal and does not deliver it to the program. The program runs as though the signal was never sent.

13.4.5. clear

```
clear [ all | routine|  line | {addr addr}]
```

Clear one or more breakpoints. Use the *all* argument to clear all breakpoints. Use the *routine* argument to clear all breakpoints from the first statement in the specified routine. Use the *line* number argument to clear all breakpoints from the specified line number in the current source file. Use the *addr* argument to clear breakpoints from the specified address *addr*.

When no arguments are specified, the **clear** command clears all breakpoints at the current location.

13.4.6. delete

```
del[ete] [event-number | 0 | all |  event-number [, event-number...] ]
```

Use the **delete** command without arguments to list all defined events by their event-number.

Use the **delete** command with arguments to delete events. Delete all events with *all* or delete just the event with the specified *event-number*. Using **delete 0**, is the same as using **delete all**.

13.4.7. disable

```
disab[le] [event-number | all ]
```

When no arguments are specified, the **disable** command prints both enabled and disabled events by event number.

With arguments, this command disables the event specified by *event-number* or *all* events. Disabling an event definition suppresses actions associated with the event, but leaves the event defined so that it can be used later. (See the enable command.)

13.4.8. do

```
do {commands} [if (condition)]
do {commands} at line [if (condition)]
do {commands} in routine [if (condition)]
```

Define a **do** event. This command is similar to watch except that instead of defining an expression, it defines a list of commands to be executed. Without the optional arguments *at* or *in*, the commands are executed at each line in the program.

Use *at* with a *line* number to specify the commands to be executed each time that line is reached. Use *in* with a *routine* to specify the commands to be executed at each line in the routine. The optional *if* argument has the same meaning that it has with the watch command. If a condition is specified, the **do** commands are executed only when the condition is true.

13.4.9. doi

```
doi {commands} [if (condition)]
doi {commands} at addr [if (condition)]
doi {commands} in routine [if (condition)]
```

Define a **doi** event. This command is similar to `watchi` except that instead of defining an expression, `doi` defines a list of commands to be executed. If an address `addr` is specified, then the commands are executed each time that the specified address is reached. If a *routine* is specified, then the commands are executed at each instruction in the routine. If neither an address nor a routine is specified, then the commands are executed at each instruction in the program. The optional *if* argument has the same meaning that it has in the do and watch commands. If a condition is specified, the **doi** commands are executed only when the condition is true.

13.4.10. enable

```
enab[le] [event-number | all ]
```

Without arguments, the **enable** command prints both enabled and disabled events by event number.

With arguments, this command enables the event *event-number* or *all* events.

13.4.11. hwatch

```
hwatch addr | var [if (condition)] [do {commands}]
```

Define a hardware watchpoint. This command uses hardware support to create a watchpoint for a particular address or variable. The event is triggered by hardware when the byte at the given address is written. This command is only supported on systems that provide the necessary hardware and software support.

Only one hardware watchpoint can be defined at a time.

When the optional *if* argument is specified, the event action is only triggered if the expression is true. When the optional *do* argument is specified, then the commands are executed when the event occurs.

13.4.12. hwatchboth

```
hwatchb[oth] addr | var [if (condition)] [do {commands}]
```

Define a hardware read/write watchpoint. This event is triggered by hardware when the byte at the given address or variable is either read or written. As with hwatch, system hardware and software support must exist for this command to be supported. The optional *if* and *do* arguments have the same meaning as for the hwatch command.

13.4.13. hwatchread

```
hwatchb[oth] addr | var [if (condition)] [do {commands}]
```

Define a hardware read watchpoint. This event is triggered by hardware when the byte at the given address or variable is read. As with hwatch, system hardware and software support must exist for this command to be supported. The optional *if* and *do* arguments have the same meaning as for the hwatch command.

13.4.14. ignore

```
ignore [sig:sig]  [sig [, sig...]]
```

Without arguments, the **ignore** command prints the list of signals being ignored. With the *sig:sig* argument, this command ignores the specified range of signals. With a list of signals, the command ignores signals with the specified numbers.

When a particular signal number is ignored, signals with that number sent to the program are not intercepted by the PGI debugger; rather, the signals are delivered to the program.

For information on intercepting signals, refer to catch.

13.4.15. status

```
stat[us]
```

Display all the event definitions, including an event number by which each event can be identified.

13.4.16. stop

```
stop var
stop at line [if (condition)][do {commands}]
stop in routine [if(condition)][do {commands}]
stop if (condition)
```

Break when the value of the indicated variable *var* changes. Use the *at* argument and a *line* to set a breakpoint at a line number. Use the *in* argument and a *routine* name to set a breakpoint at the first statement of the specified routine. When the *if* argument is used, the debugger stops when the condition is true.

13.4.17. stopi

```
stopi var
stopi at address [if (condition)][do {commands}]
stopi in routine [if (condition)][do {commands}]
stopi if (condition)
```

Break when the value of the indicated variable *var* changes. Set a breakpoint at the indicated address or routine. Use the *at* argument and an *address* to specify an address at which to stop. Use the *in* argument and a *routine* name to specify the first address of the specified routine at which to stop. When the *if* argument is used, the debugger stops when the condition is true.

13.4.18. trace

```
trace var [if (condition)][do {commands}]
trace routine [if (condition)][do {commands}]
trace at line [if (condition)][do {commands}]
trace in routine [if (condition)][do {commands}]
trace inclass class [if (condition)][do {commands}]
```

Use *var* to activate tracing when the value of *var* changes. Use *routine* to activate tracing when the subprogram *routine* is called. Use *at* to display the specified *line* each time it is executed. Use *in* to display the current line while in the specified *routine*. Use *inclass* to display the current line while in each member function of the specified *class*. If a condition is specified, tracing is only enabled if the condition evaluates to true. The *do* argument defines a list of commands to execute at each trace point.

Use the **pgienv speed** command to set the time in seconds between trace points. Use the clear command to remove tracing for a line or routine.

13.4.19. tracei

```
tracei var [if (condition)][do {commands}]
tracei at addr [if (condition)][do {commands}]
tracei in routine [if (condition)][do {commands}]
tracei inclass class [if (condition)][do {commands}]
```

Activate tracing at the instruction level. Use *var* to activate tracing when the value of *var* changes. Use *at* to display the instruction at *addr* each time it is executed. Use *in* to display memory instructions while in the subprogram *routine*. Use *inclass* to display memory instructions while in each member function of the specified *class*. If a condition is specified, tracing is only enabled if the condition evaluates to true. The *do* argument defines a list of commands to execute at each trace point.

Use the **pgienv speed** command to set the time in seconds between trace points. Use the clear command to remove tracing for a line or routine.

13.4.20. track

```
track expression [at line | in func] [if (condition)][do {commands}]
```

Define a track event. This command is equivalent to watch except that execution resumes after the new value of the expression is printed.

13.4.21. tracki

```
tracki expression [at addr | in func] [if (condition)][do {commands}]
```

Define an assembly-level track event. This command is equivalent to watchi except that execution resumes after the new value of the expression is printed.

13.4.22. unbreak

```
unb[reak] line | routine| all
```

Remove a breakpoint from the specified *line* or *routine*, or remove *all* breakpoints.

13.4.23. unbreaki

```
unbreaki addr | routine | all
```

Remove a breakpoint from the specified address *addr* or *routine*, or remove *all* breakpoints.

13.4.24. watch

```
wa[tch] expression
wa[tch] expression [if (condition)][do {commands}]
wa[tch] expression at line [if (condition)][do {commands}]
wa[tch] expression in routine [if (condition)][do {commands}]
```

Define a watch event. The given expression is evaluated, and subsequently, each time the value of the expression changes, the program stops and the new value of the expression is printed. If a *line* is specified, the expression is only evaluated at that line. If a *routine* is specified, the expression is evaluated at each line in the routine. If no location is specified, the expression is evaluated at each line in the program. If a *condition* is specified, the expression is evaluated only when the condition is true. If *commands* are specified using *do*, they are executed whenever the expression is evaluated and its value changes.

The watched expression may contain local variables, although this is not recommended unless a routine or address is specified to ensure that the variable is only evaluated when it is in the current scope.

Using watchpoints indiscriminately can dramatically slow program execution.

Using the *at* and *in* arguments speeds up execution by reducing the amount of single-stepping and expression evaluation that must be performed to watch the expression. For example:

```
watch i at 40
```

may not slow program execution noticeably, while

```
watch i
```

slows execution considerably.

13.4.25. watchi

```
watchi expression
watchi expression [if (condition)][do {commands}]
watchi expression at addr [if (condition)][do {commands}]
watchi expression in routine [if (condition)][do {commands}]
```

Define an assembly-level watch event. This command functions similarly to the watch command with two exceptions: 1) the argument interprets integers as addresses rather than line numbers and 2) the *expression* is evaluated at every instruction rather than at every line.

This command is useful when line number information is limited, which may occur when debug information is not available or assembly must be debugged. Using **`watchi`** causes programs to execute more slowly than watch.

13.4.26. when

```
when do {commands} [if (condition)]
when at line do {commands} [if (condition)]
when in routine do {commands} [if (condition)]
```

Execute *commands* at every line in the program, at a specified *line* in the program, or in the specified *routine*. If an optional *condition* is specified, commands are executed only when the *condition* evaluates to true.

13.4.27. wheni

```
wheni do {commands} [if (condition)]
wheni at addr do {commands} [if (condition)]
wheni in routine do {commands} [if (condition)]
```

Execute *commands* at each address in the program. If an address *addr* is specified, the commands are executed each time the address is reached. If a *routine* is specified, the commands are executed at each line in the routine. If an optional *condition* is specified, commands are executed whenever the *condition* evaluates to true.

13.5. Program Locations

This section describes the debugger's program location commands.

13.5.1. arrive

```
arri[ve]
```

Print location information for the current location.

13.5.2. cd

```
cd [dir]
```

Change the current directory to the `$HOME` directory or to the specified directory *dir*.

13.5.3. disasm

```
dis[asm] [ count | lo:hi | routine | addr, count ]
```

Disassemble memory.

If no argument is given, disassemble four instructions starting at the current address. If an integer *count* is given, disassemble *count* instructions starting at the current address. If an address range (*lo:hi*) is given, disassemble the memory in the range. If a *routine* is given, disassemble the entire routine. If the routine was compiled for debugging and source code is available, the source code is interleaved with the disassembly. If an address *addr* and a *count* are both given, disassemble *count* instructions starting at the provided address.

13.5.4. edit

```
edit [filename | routine]
```

Use the editor specified by the environment variable $EDITOR to edit a file.

If no argument is supplied, edit the current file starting at the current location. To edit a specific file, provide the *filename* argument. To edit the file containing the subprogram *routine*, specify the routine name.

This command is only supported in the CLI.

13.5.5. file

```
file [filename]
```

Change the source file to the file *filename* and change the scope accordingly. With no argument, print the current file.

13.5.6. lines

```
lines [routine]
```

Print the lines table for the specified *routine*. With no argument, prints the lines table for the current routine.

13.5.7. list

```
lis[t] [ count | line,num |  lo:hi |  routine[,num] ]
```

Provide a source listing.

By default, **`list`** displays ten lines of source centered at the current source line. If a *count* is given, list the specified number of lines. If a *line* and *count* are both given, start the listing of *count* lines at *line*. If a line range (*lo:hi*) is given, list the indicated source lines

in the current source file. If a *routine* name is given, list the source code for the indicated routine. If a *number* is specified with *routine*, list the first *number* lines of the source code for the indicated routine.

```
list [dbx mode]
```

The **list** command works somewhat differently when the debugger is in dbx mode.

```
lis[t] [ line | first,last | routine | file ]
```

By default, list displays ten lines of source centered at the current source line. If a *line* is provided, the source at that line is displayed. If a range of line numbers is provided (*first*,*last*), lines from the first specified line to the last specified line are displayed. If a *routine* is provided, the display listing begins in that routine. If a *file* name is provided, the display listing begins in that file. File names must be quoted.

13.5.8. pwd

```
pwd
```

Print the current working directory.

13.5.9. stackdump

```
stackd[ump] [count]
```

Print the call stack. This command displays a hex dump of the stack frame for each active routine. This command is an assembly-level version of the stacktrace command. If a *count* is specified, display a maximum of *count* stack frames.

13.5.10. stacktrace

```
stack[trace] [count]
```

Print the call stack. Print the available information for each active routine, including the routine name, source file, line number, and current address. This command also prints the names and values of any arguments, when available. If a *count* is specified, display a maximum of *count* stack frames. The **stacktrace** and **where** commands are equivalent.

13.5.11. where

```
w[here] [count]
```

Print the call stack. Print the available information for each active routine, including the routine name, source file, line number, and current address. This command also prints the names and values of any arguments, when available. If a *count* is specified, display a maximum of *count* stack frames. The **where** and **stacktrace** commands are equivalent.

13.5.12. /

```
/
/string/
```

Search forward for a *string* of characters in the current source file. With a specified string, search for the next occurrence of *string* in the current source file.

13.5.13. ?

```
?
?string?
```

Search backward for a *string* of characters in the current source file. Without arguments, search for the previous occurrence of *string* in the current source file.

13.6. Printing Variables and Expressions

This section describes the PGI debugger commands used for printing and setting variables. The primary print commands are **print** and **printf**, described at the beginning of this section. The rest of the commands for printing provide alternate methods for printing.

13.6.1. print

```
p[rint] exp1 [,...expn]
```

Evaluate and print one or more expressions. This command is invoked to print the result of each line of command input. Values are printed in a format appropriate to their type. For values of structure type, each field name and value is printed. Character pointers are printed as a hex address followed by the character string.

Character string constants print out literally using a comma-separated list. For example:

```
pgdbg> print "The value of i is ", i
```

Prints this:

```
"The value of i is", 37
```

The array sub-range operator (:) prints a range of an array. The following examples print elements 0 through 9 of the array a:

C/ C++[example 1:

```
pgdbg> print a[0:9]
a[0:4]: 0 1 2 3 4
a[5:9]: 5 6 7 8 9
```

FORTRAN example 1:

```
pgdbg> print a(0:9)
a(0:4): 0 1 2 3 4
a(5:9): 5 6 7 8 9
```

Notice that the output is formatted and annotated with index information. The debugger formats array output into columns. For each row, the first column prints an index expression which summarizes the elements printed in that row. Elements associated with each index expression are then printed in order. This is especially useful when printing slices of large multidimensional arrays.

The debugger also supports array expression strides. Below are examples for C/ C++ and FORTRAN.

C/ C++ example 2:

```
pgdbg> print a[0:9:2]
a[0:8]: 0 2 4 6 8
```

FORTRAN example 2:

```
pgdbg> print a(0:9:2)
a(0:8): 0 2 4 6 8
```

The print statement may be used to display members of derived types in FORTRAN or structures in C/ C++. Here are examples.

C/ C++ example 3:

```
typedef struct tt {
 int a[10];
 }TT;
 TT d = {0,1,2,3,4,5,6,7,8,9};
 TT * p = &d;
```

```
pgdbg> print d.a[0:9:2]
 d.a[0:8:2]: 0 2 4 6 8
pgdbg> print p->a[0:9:2]
 p->a[0:7:2]: 0 2 4 6
 p->a[8]: 8
```

FORTRAN example 3:

```
 type tt
 integer, dimension(0:9) :: a
 end type
 type (tt) :: d
 data d%a / 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 /
```

```
 pgdbg> print d%a(0:9:2)
 d%a(0:8:2): 0 2 4 6 8
```

13.6.2. printf

```
printf "format_string", expr,...expr
```

Print expressions in the format indicated by the format string. This command behaves like the C library function printf. For example:

```
pgdbg> printf "f[%d]=%G",i,f[i]
f[3]=3.14
```

The **pgienv stringlen** command sets the maximum number of characters that print with a **print** command. For example, the char declaration below:

```
char *c="a whole bunch of chars over 1000 chars long....";
```

By default, the **print c** command prints only the first 512 (default value of stringlen) bytes. Printing of C strings is usually terminated by the terminating null character. This limit is a safeguard against unterminated C strings.

13.6.3. ascii

```
asc[ii] exp [,...exp]
```

Evaluate and print *exp* as an ASCII character. Control characters are prefixed with the '^' character; for example, 3 prints as ^c. Otherwise, values that cannot be printed as characters are printed as integer values prefixed by ´\'. For example, 250 is printed as \250.

13.6.4. bin

```
bin exp [,...exp]
```

Evaluate and print the expressions. Integer values are printed in base2.

13.6.5. dec

```
dec exp [,...exp]
```

Evaluate and print the expressions. Integer values are printed in decimal.

13.6.6. display

```
display [ exp [,...exp] ]
```

Without arguments, list the expressions for the debugger to automatically display at breakpoints. With one or more arguments, print expression *exp* at every breakpoint. See also the undisplay command.

13.6.7. hex

```
hex exp [,...exp]
```

Evaluate and print expressions as hexadecimal integers.

13.6.8. oct

```
oct exp [,...exp]
```

Evaluate and print expressions as octal integers.

13.6.9. string

```
str[ing] exp [,...exp]
```

Evaluate and print expressions as null-terminated character strings. This command prints a maximum of 70 characters.

13.6.10. undisplay

```
undisplay 0 | all |  exp [,...exp]
```

Remove all expressions specified by previous display commands. With an argument or several arguments, remove the expression *exp* from the list of display expressions.

13.7. Symbols and Expressions

This section describes the commands that deal with symbols and expressions.

13.7.1. assign

```
as[sign] var = exp
```

Set variable *var* to the value of the expression *exp*. The variable can be any valid identifier accessed properly for the current scope. For example, given a C variable declared `'int * i'`, you can use the following command to assign the value 9999 to it.

```
assign *i = 9999
```

13.7.2. call

```
call routine [(exp,...)]
```

Call the named *routine*. C argument passing conventions are used. Breakpoints encountered during execution of the routine are ignored. Fortran functions and subroutines can be called, but the argument values are passed according to C conventions. The debugger may not always be able to access the return value of a Fortran function if the return value is an array. In the example below, the debugger calls the routine `foo` with four arguments:

```
pgdbg> call foo(1,2,3,4)
```

If a signal is caught during execution of the called routine, the debugger stops the execution and asks if you want to cancel the **`call`** command. For example, suppose a command is issued to call `foo` as shown above, and for some reason a signal is sent to the process while it is executing the call to `foo`. In this case, the debugger prints the following prompt:

```
PGDBG Message: Thread [0] was signalled while executing a function
reachable from the most recent PGDBG command line call to foo. Would you
like to cancel this command line call? Answering yes will revert the register
state of Thread [0] back to the state it had prior to the last call to foo
from the command line. Answering no will leave Thread [0] stopped in the call
to foo from the command line.
Please enter 'y' or 'n' > y
Command line call to foo cancelled
```

Answering yes to this question returns the register state of each thread back to the state they had before invoking the **`call`** command. Answering no to this question leaves each thread at the point they were at when the signal occurred.

Answering no to this question and continuing execution of the called routine may produce unpredictable results.

13.7.3. declaration

```
decl[aration] name
```

Print the declaration for the symbol name based on its type according to the symbol table. The symbol must be a variable, argument, enumeration constant, routine, structure, union, enum, or typedef tag.

For example, given the C declarations:

```
int i, iar[10];
struct abc {int a; char b[4]; struct
abc *c;}val;
```

the **decl** command provides the following output:

```
pgdbg> decl I
int i
```

```
pgdbg> decl iar
int iar[10]
```

```
pgdbg> decl val
struct abc val
```

```
pgdbg> decl abc
struct abc {
 int a;
 char b[4];
 struct abc *c;
};
```

13.7.4. entry

```
entr[y] [routine]
```

Return the address of the first executable statement in the program or specified *routine*. This is the first address after the routine's prologue code.

13.7.5. lval

```
lv[al] expr
```

Return the lvalue of the expression *expr*. The lvalue of an expression is the value it would have if it appeared on the left hand side of an assignment statement. Roughly speaking, an lvalue is a location to which a value can be assigned. This may be an address, a stack offset, or a register.

13.7.6. rval

```
rv[al] expr
```

Return the rvalue of the expression *expr*. The rvalue of an expression is the value it would have if it appeared on the right hand side of an assignment statement. The type of the expression may be any scalar, pointer, structure, or function type.

13.7.7. set

```
set var=expression
```

Set variable *var* to the value of *expression*. The variable can be any valid identifier accessed properly for the current scope. For example, given a C variable declared `int * i`, the following command could be used to assign the value 9999 to it.

```
pgdbg> set *i = 9999
```

13.7.8. sizeof

```
siz[eof] name
```

Return the size, in bytes, of the variable type *name*. If *name* refers to a routine, **sizeof** returns the size in bytes of the subprogram.

13.7.9. type

```
type expr
```

Return the type of the expression *expr*. The expression may contain structure reference operators (. , and ->), dereference (*), and array index ([ ]) expressions. For example, given the C declarations:

```
int i, iar[10];
struct abc {int a; char b[4];
struct abc *c;}val;
```

the **type** command provides the following output:

```
pgdbg> type i
int
pgdbg> type iar
int [10]
pgdbg> type val
struct abc
pgdbg> type val.a
int
```

```
pgdbg> type val.abc->b[2]
char
```

```
pgdbg> whatis
whatis name
```

With no arguments, print the declaration for the current routine.

With the argument *name*, print the declaration for the symbol *name*.

13.8. Scope

The following commands deal with program scope. For a discussion of scope meaning and conventions, refer to Scope Rules.

13.8.1. class

```
clas[s] [class]
```

Without arguments, **class** returns the current class. With a *class* argument, enter the scope of class *class*.

13.8.2. classes

```
classse[s]
```

Print the C++ class names.

13.8.3. decls

```
decls [routine | "sourcefile" |  {global} ]
```

Print the declarations of all identifiers defined in the indicated scope. If no scope is given, print the declarations for the current search scope.

13.8.4. down

```
down [number]
```

Enter the scope of the routine down one level or *number* levels on the call stack.

13.8.5. enter

```
en[ter] [routine | "sourcefile" | global ]
```

Set the search scope to be the indicated scope, which may be a *routine*, *file* or *global*. Using **enter** with no argument is the same as using **enter global**.

13.8.6. files

```
files
```

Return the list of known source files used to create the executable file.

13.8.7. global

```
glob[al]
```

Return a symbol representing global scope. This command is useful in combination with the scope operator @ to specify symbols with global scope.

13.8.8. names

```
names [routine | "sourcefile" | global ]
```

Print the names of all identifiers defined in the indicated scope. If no scope is specified, use the search scope.

13.8.9. scope

```
sco[pe]
```

Return a symbol for the search scope. The search scope is set to the current routine each time program execution stops. It may also be set using the enter command. The search scope is always searched first for symbols.

13.8.10. up

```
up [number]
```

Enter the scope of the routine up one or *number* levels from the current routine on the call stack.

13.8.11. whereis

```
whereis name
```

Print all declarations for *name*.

13.8.12. which

```
which name
```

Print the full scope qualification of symbol *name*.

13.9. Register Access

System registers can be accessed by name. For details on referring to registers in the debugger, refer to Register Symbols.

13.9.1. fp

```
fp
```

Return the current value of the frame pointer.

13.9.2. pc

```
pc
```

Return the current program address.

13.9.3. regs

```
regs
regs -info
regs -grp=grp1[,grp2...]
regs -fmt=fmt1[,fmt2...]
regs -mode=scalar|vector
```

Print the names and values of registers. By default, **`regs`** prints the General Purpose registers. Use the `-grp` option to specify one or more register groups, the `-fmt` option to specify one or more display formats, and `-mode` to specify scalar or vector mode. Use the `-info` option to display the register groups on the current system and the display formats available for each group. All optional arguments with the exception of `-info` can be used with the others.

13.9.4. retaddr

```
ret[addr]
```

Return the current return address.

13.9.5. sp

```
sp
```

Return the current value of the stack pointer.

13.10. Memory Access

The following commands display the contents of arbitrary memory locations. For each of these commands, the *addr* argument may be a variable or identifier.

13.10.1. dump

```
du[mp] address[, count[,format-string]]
```

This command dumps the contents of a region of memory. The output is formatted according to a descriptor. Starting at the indicated address, values are fetched from memory and displayed according to the format descriptor. This process is repeated *count* times.

Interpretation of the format descriptor is similar to that used by printf. Format specifiers are preceded by %.

The recognized format descriptors are for decimal, octal, hex, or unsigned:

```
%d, %D, %o, %O, %x, %X, %u, %U
```

Default size is machine dependent. The size of the item read can be modified by either inserting 'h' or 'l' before the format character to indicate half word or long word. For example, if your machine's default size is 32-bit, then %hd represents a 16-bit quantity. Alternatively, a 1, 2, or 4 after the format character can be used to specify the number of bytes to read.

Fetch and print a character.

```
%c
```

Fetch and print a float (lower case) or double (upper case) value using printf f, e, or g format.

```
%f, %F, %e, %E, %g, %G
```

Fetch and print a null terminated string.

```
%s
```

Interpret the next object as a pointer to an item specified by the following format characters. The pointed-to item is fetched and displayed.

```
%p
```

Pointer to int. Prints the address of the pointer, the value of the pointer, and the contents of the pointed-to address, which is printed using hexadecimal format.

```
%px
```

Fetch an instruction and disassemble it.

```
%i
```

Display address about to be dumped.

```
%w, %W
```

Display nothing while advancing or decrementing the current address by *n* bytes.

```
%z<n>, %Z<n>, %z<-n>, %Z<-n>
```

Display nothing while advancing the current address as needed to align modulo *n*.

```
 %a<n>, %A<n>
```

Display nothing while advancing the current address as needed to align modulo n.

13.10.2. mqdump

```
mq[dump]
```

Dump MPI message queue information for the current process. For more information on **mqdump**, refer to MPI Message Queues.

13.11. Conversions

The commands in this section are useful for converting between different kinds of values. These commands accept a variety of arguments, and return a value of a particular kind.

13.11.1. addr

```
ad[dr]  [n | line n | routine | var |  arg ]
```

Create an address conversion under these conditions:

- If an integer is given, return an address with the same value.
- If a line is given, return the address corresponding to the start of that line.
- If a routine is given, return the first address of the routine.
- If a variable or argument is given, return the address where that variable or argument is stored.

For example,

```
breaki {line {addr 0x22f0}}
```

13.11.2. function

```
func[tion] [[addr...] | [line...] ]
```

Return a routine symbol. If no argument is specified, return the current routine. If an address is given, return the routine containing *addr*. An integer argument is interpreted as an address. If a *line* is specified, return the routine containing that line.

13.11.3. line

```
lin[e] [ n | routine |  addr ]
```

Create a source line conversion. If no argument is given, return the current source line. If an integer *n* is given, return it as a line number. If a *routine* is given, return the first line of the routine. If an address is given, return the line containing that address.

For example, the following command returns the line number of the specified address:

```
line {addr 0x22f0}
```

13.12. Target

The following commands are applicable to system architectures for which multiple debugging environment targets are available. The commands in this section do not apply to the x86-64 environments.

13.12.1. connect

```
con[nect]
con[nect] -t target [args]
con[nect] -d path [args]
con[nect] -f file
con[nect] -f file name [args]
```

Without arguments, connect prints the current connection and the list of possible connection targets. Use *-t* to connect to a specific target. Use *-d* to connect to a target specified by *path*. Use *-f* to print a list of possible targets as contained in a *file*, or to connect to a target selected by *name* from the list defined in *file*. Pass configuration arguments to the target as appropriate.

13.12.2. disconnect

```
disc[onnect]
```

Close connection to the current target.

13.12.3. native

```
nati[ve] [command]
```

Without arguments **`native`** prints the list of available target commands. Given a *command* argument, **`native`** sends *command* directly to the target.

13.13. Miscellaneous

The following commands provide shortcuts, mechanisms for querying, customizing and managing the debugger environment, and access to operating system features.

13.13.1. alias

```
al[ias]  [ name [string] ]
```

Create or print aliases.

- If no arguments are given, print all the currently defined aliases.
- If just a *name* is given, print the alias for that name.
- If both a *name* and *string* are given, make *name* an alias for *string*. Subsequently, whenever *name* is encountered it is replaced by *string*.

Although *string* may be an arbitrary string, *name* must not contain any space characters.

For example, the following statement creates an alias for xyz.

```
alias xyz print "x= ",x,"y= ",y,"z= ",z;
cont
```

Now whenever xyz is typed, the PGI debugger responds as though the following command was typed:

```
print "x= ",x,"y= ",y,"z= ",z;
cont
```

13.13.2. directory

```
dir[ectory] [pathname]
```

Add the directory *pathname* to the search path for source files.

If no argument is specified, the currently defined directories are printed. This command assists in finding source code that may have been moved or is otherwise not found by the PGI debugger's default search mechanisms.

For example, the following statement adds the directory `morestuff` to the list of directories to be searched.

```
dir morestuff
```

Now, source files stored in `morestuff` are accessible to the debugger.

If the first character in pathname is ~, then $HOME replaces that character.

13.13.3. help

```
help [command]
```

If no argument is specified, print a brief summary of all the commands. If a *command* is specified, print more detailed information about the use of that command.

13.13.4. history

```
history [num]
```

List the most recently executed commands. With the *num* argument, resize the history list to hold *num* commands.

History allows several characters for command substitution:

!! [modifier]	Execute the previous command.
! num [modifier]	Execute command number *num*.
!-num [modifier]	Execute the command that is *num* commands from the most current command
!string [modifier]	Execute the most recent command starting with string.
!?string? [modifier]	Execute the most recent command containing string.
^	Command substitution. For example, `^old^new^`<modifier> is equivalent to `!:s/old/new/`.

There are two possible history modifiers. To substitute the value *new* for the value *old* use:

```
:s/old/new/
```

To print the command without executing it use:

```
:p
```

Use the **pgienv** history command to toggle whether or not the history record number is displayed. The default value is on.

13.13.5. language

```
lang[uage]
```

Print the name of the language of the current file.

13.13.6. log

```
log filename
```

Keep a log of all commands entered by the user and store it in the named file. This command may be used in conjunction with the script command to record and replay debug sessions.

13.13.7. noprint

```
nop[rint] exp
```

Evaluate the expression but do not print the result.

13.13.8. pgienv

```
pgienv [command]
```

Define the debugger environment. With no arguments, display the debugger settings.

Table 15 pgienv Commands

Use this command...	To do this...
help pgienv	Provide help on pgienv
pgienv	Display the debugger settings
pgienv dbx on	Set the debugger to use dbx style commands
pgienv dbx off	Set the debugger to use PGI style commands
pgienv history on	Display the history record number with prompt
pgienv history off	Do not display the history number with prompt
pgienv exe none	Ignore executable's symbolic debug information
pgienv exe symtab	Digest executable's native symbol table (typeless)
pgienv exe demand	Digest executable's symbolic debug information incrementally on demand
pgienv exe force	Digest executable's symbolic debug information when executable is loaded
pgienv solibs none	Ignore symbolic debug information from shared libraries
pgienv solibs symtab	Digest native symbol table (typeless) from each shared library

Use this command...	To do this...
pgienv solibs demand	Digest symbolic debug information from shared libraries incrementally on demand
pgienv solibs force	Digest symbolic debug information from each shared library at load time
pgienv mode serial	Single thread of execution (implicit use of p/t-sets)
pgienv mode thread	Debug multiple threads (condensed p/t-set syntax)
pgienv mode process	Debug multiple processes (condensed p/t-set syntax)
pgienv mode multilevel	Debug multiple processes and multiple threads
pgienv omp [on|off]	Enable/Disable the PGI debugger's OpenMP event handler. This option is disabled by default. The OpenMP event handler, when enabled, sets breakpoints at the beginning and end of each parallel region. Breakpoints are also set at each thread synchronization point. The handler coordinates threads across parallel constructs to maintain source level debugging. This option, when enabled, may significantly slow down program performance. Enabling this option is recommended for localized debugging of a particular parallel region only.
pgienv prompt <name>	Set the command-line prompt to <name>
pgienv promptlen <num>	Set maximum size of p/t-set portion of prompt
pgienv speed <secs>	Set the time in seconds <secs> between trace points
pgienv stringlen <num>	Set the maximum # of chars printed for ‘´char *'s’
pgienv termwidth <num>	Set the character width of the display terminal.
pgienv logfile <name>	Close logfile (if any) and open new logfile <name>
pgienv threadstop sync	When one thread stops, the rest are halted in place
pgienv threadstop async	Threads stop independently (asynchronously)
pgienv procstop sync	When one process stops, the rest are halted in place
pgienv procstop async	Processes stop independently (asynchronously)
pgienv threadstopconfig auto	For each process, debugger sets thread stopping mode to 'sync' in serial regions, and 'async' in parallel regions
pgienv threadstopconfig user	Thread stopping mode is user defined and remains unchanged by the debugger.
pgienv procstopconfig auto	Not currently used.
pgienv procstopconfig user	Process stop mode is user defined and remains unchanged by the debugger.
pgienv threadwait none	Prompt available immediately; do not wait for running threads
pgienv threadwait any	Prompt available when at least one thread stops
pgienv threadwait all	Prompt available only after all threads have stopped
pgienv procwait none	Prompt available immediately; do not wait for running processes
pgienv procwait any	Prompt available when at least a single process stops
pgienv procwait all	Prompt available only after all processes have stopped

Use this command...	To do this...
pgienv threadwaitconfig auto	For each process, the debugger sets the thread wait mode to ‘all’ in serial regions and ‘none’ in parallel regions. (default)
pgienv threadwaitconfig user	The thread wait mode is user-defined and remains unchanged by the debugger.
pgienv mqslib default	Set MPI message queue debug library by inspecting executable.
pgienv mqslib <path>	Determine MPI message queue debug library to <path>.
pgienv verbose <bitmask>	Choose which debug status messages to report. Accepts an integer valued bit mask of the following values: ▸ 0x0 - Disable all messages. ▸ 0x1 - Standard messaging (default). Report status information on current process/thread only. ▸ 0x2 - Thread messaging. Report status information on all threads of (current) processes. ▸ 0x4 - Process messaging. Report status information on all processes. ▸ 0x8 - OpenMP messaging (default). Report OpenMP events. ▸ 0x10 - Parallel messaging (default). Report parallel events. ▸ 0x20 - Symbolic debug information. Report any errors encountered while processing symbolic debug information. ▸ Pass 0x0 to disable all messages.

13.13.9. repeat

```
rep[eat] [first, last]
rep[eat] [first:last:n]
rep[eat] [num ]
rep[eat] [-num ]
```

Repeat the execution of one or more previous history list commands. Use the *num* argument to re-execute the last *num* commands. With the *first* and *last* arguments, re-execute commands number *first* to *last* (optionally *n* times).

13.13.10. script

```
scr[ipt] filename
```

Open the indicated file and execute the contents as though they were entered as commands. Use ~ before the filename in place of the environment variable $HOME.

13.13.11. setenv

```
setenv name |  name value
```

Print the value of the environment variable *name*. With a specified *value*, set *name* to *value*.

13.13.12. shell

```
shell [arg0, arg1,... argn]
```

Fork a shell and give it the indicated arguments. The default shell type is sh or defined by $SHELL. If no arguments are specified, an interactive shell is invoked, and executes until a Ctrl+D is entered.

13.13.13. sleep

```
sle[ep] [time]
```

Pause for one second or *time* seconds.

13.13.14. source

```
sou[rce] filename
```

Open the indicated file and execute the contents as though they were entered as commands. Use ~ before the filename in place of the environment variable $HOME.

13.13.15. unalias

```
unal[ias] name
```

Remove the alias definition for *name*, if one exists.

13.13.16. use

```
use [dir]
```

Print the current list of directories or add *dir* to the list of directories to search. The character ~ or environment variable $HOME can be used interchangeably.

Chapter 14. CONTACT INFORMATION

You can contact PGI at:

20400 NW Amberwood Drive Suite 100
Beaverton, OR 97006

Or electronically using any of the following means:

Fax: +1-503-682-2637
Sales: sales@pgroup.com
WWW: http://www.pgroup.com

The PGI User Forum is monitored by members of the PGI engineering and support teams as well as other PGI customers. The forum newsgroups may contain answers to commonly asked questions. Log in to the PGI website to access the forum:

http://www.pgroup.com/userforum/index.php

Many questions and problems can be resolved by following instructions and the information available at our frequently asked questions (FAQ) site:

http://www.pgroup.com/support/faq.htm

Submit technical support requests through the online form at:

https://www.pgroup.com/support/support_request.php

PGI documentation is available at http://www.pgroup.com/resources/docs.htm.