PGI Tools User's Guide

Parallel Tools
for Scientists and Engineers

The Portland Group Compiler Technology
STMicroelectronics
9150 SW Pioneer Court, Suite H
Wilsonville, OR 97070
www.pgroup.com

While every precaution has been taken in the preparation of this document, The Portland Group™ Compiler Technology, Microelectronics makes no warranty for the use of its products and assumes no responsibility for any errors that may appear, or for damages resulting from the use of the information contained herein. The Portland Group™ Compiler Technology, Microelectronics retains the right to make changes to this information at any time, without notice. The software described in this document is distributed under license from The Portland Group™ Compiler Technology, STMicroelectronics and may be used or copied only in accordance with the terms of the license agreement. No part of this document may be reproduced or transmitted in any form or by any means, for any purpose other than the purchaser's personal use without the express written permission of The Portland Group™ Compiler Technology, STMicroelectronics

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this manual, The Portland Group™ Compiler Technology, Microelectronics was aware of a trademark claim. The designations have been printed in caps or initial caps.

CDK and *The Portland Group* are trademarks and *Cluster Development Kit, PGI, PGF90, PGHPF*, *PGF77*, *PGCC*, *PGPROF*, and *PGDBG* are registered trademarks of STMicroelectronics, Inc. Other brands and names are the property of their respective owners.

PGI Tools User's Guide
Copyright © 1998 - 2003 STMicroelectronics, Inc.
All rights reserved.
Printed in the United States of America

First Printing: Release 5.0, June 2003

Part Number: 2040-990-888-0603

Technical support: trs@pgroup.com
Sales: sales@pgroup.com
http://www.pgroup.com

Table of Contents

TABLE OF CONTENTS .... **III**

PREFACE **1**

AUDIENCE DESCRIPTION 1
COMPATIBILITY AND CONFORMANCE TO STANDARDS 1
ORGANIZATION 2
CONVENTIONS 3
RELATED PUBLICATIONS 3
SYSTEM REQUIREMENTS 4

THE *PGDBG* DEBUGGER **5**

1.1 DEFINITION OF TERMS 5
1.1.1 Compiler Options for Debugging *6*
1.2 INVOCATION AND INITIALIZATION 6
1.3 COMMAND-LINE ARGUMENTS 7
1.4 COMMAND LANGUAGE 8
1.4.1 Constants *8*
1.4.2 Symbols *8*
1.4.3 Scope Rules *9*
1.4.4 Register Symbols *9*
1.4.5 Source Code Locations *10*
1.4.6 Lexical Blocks *11*
1.4.7 Statements *12*
1.4.8 Events *12*
1.4.9 Expressions *15*
1.5 SIGNALS 17
1.5.1 Signals Used Internally by PGDBG *17*
1.6 DEBUGGING FORTRAN 18
1.6.1 Arrays *18*
1.6.2 Operators *18*
1.6.3 Name of Main Routine *18*
1.6.4 Fortran Common Blocks *18*
1.6.5 Nested Subroutines *19*
1.6.6 Fortran 90 Modules *20*
1.7 DEBUGGING *C++* 21
1.8 CORE FILES 22
1.9 *PGDBG* COMMANDS 22

1.9.1 Commands ... *22*
1.9.1.1 Process Control ... *22*
1.9.1.2 Process-Thread Sets ... *25*
1.9.1.3 Events ... *26*
1.9.1.4 Program Locations ... *33*
1.9.1.5 Printing and Setting Variables ... *35*
1.9.1.6 Symbols and Expressions ... *37*
1.9.1.7 Scope ... *39*
1.9.1.8 Register Access ... *40*
1.9.1.9 Memory Access ... *41*
1.9.1.10 Conversions ... *43*
1.9.1.11 Miscellaneous ... *44*
1.10 COMMANDS SUMMARY ... 49
1.10.1 Command Summary ... *50*
1.11 REGISTER SYMBOLS ... 57
1.11.1 X86 Register Symbols ... *58*
1.11.2 AMD64 Register Symbols ... *59*
1.12 X-WINDOWS GRAPHICAL USER INTERFACE ... 61
1.12.1 Main Window ... *62*
1.12.2 Disassembly Window ... *63*
1.12.3 Register Window ... *64*
1.12.4 Memory Window ... *65*
1.12.5 Custom Window ... *66*
1.12.5.1 Setting the Font ... *67*
1.13 *PGDBG*: PARALLEL DEBUG CAPABILITIES ... 68
1.13.1 OpenMP and Linuxthread Support ... *68*
1.13.2 MPI Support ... *69*
1.13.3 Process & Thread Control ... *69*
1.13.4 Graphical Presentation of Threads and Processes ... *69*
1.14 DEBUGGING PARALLEL PROGRAMS WITH PGDBG ... 70
1.14.1 Processes and Threads ... *70*
1.14.2 Thread-Parallel Debugging ... *71*
1.14.3 Graphical Features ... *72*
1.14.4 Process-Parallel Debugging ... *74*
1.15 THREAD-PARALLEL AND PROCESS-PARALLEL DEBUGGING ... 80
1.15.1 PGDBG Debug Modes and Process/Thread Identifiers ... *80*
1.15.2 Threads-only debugging ... *81*
1.15.3 Process-only debugging ... *81*
1.15.4 Multilevel debugging ... *82*
1.15.5 Process/Thread Sets ... *82*
1.15.6 P/t-set Notation ... *83*
1.15.7 Dynamic vs. Static P/t-sets ... *85*

1.15.8 Current vs. Prefix P/t-set *85*
1.15.9 P/t-set Commands *86*
1.15.10 Command Set *88*
1.15.11 Process and Thread Control *91*
1.15.12 Configurable Stop Mode *92*
1.15.13 Configurable Wait mode *93*
1.15.14 Status Messages *96*
1.15.15 The PGDBG Command Prompt *97*
1.15.16 Parallel Events *98*
1.15.17 Parallel Statements *101*
1.16 OPENMP DEBUGGING 102
1.16.1 Serial vs. Parallel Regions *103*
1.16.2 Disabling PGDBG's OpenMP Event Support *104*
1.17 MPI DEBUGGING 105
1.17.1 Process Control *105*
1.17.2 Process Synchronization *105*
1.17.3 MPI Message Queues *106*
1.17.4 MPI Groups *106*
1.17.5 MPI Listener Processes *106*
1.17.6 SSH and RSH *107*
1.18 LIMITATIONS 107
1.18.1 PGDBG Limitations—Parallel Debugging *107*
1.18.2 Other Limitations *108*
1.18.3 Private Variables *108*

THE *PGPROF* PROFILER 111

2.1 INTRODUCTION 111
2.1.1 Definition of Terms *112*
2.1.2 Compilation *112*
2.1.3 Program Execution *113*
2.1.4 Profiler Invocation and Initialization *113*
2.1.5 Virtual Timer *114*
2.1.6 Profile Data *115*
2.1.7 Caveats *116*
2.1.7.1 Clock Granularity *116*
2.1.7.2 Optimization *116*
2.2 X-WINDOWS GRAPHICAL USER INTERFACE 117
2.2.1 Command Line Switches and X-Windows Resources *117*
2.2.2 Using the PGPROF X-Windows GUI *119*
2.2.2.1 File Menu *120*
2.2.2.2 Options Menu *121*
2.2.2.3 Sort Menu and The Sort Option Box *121*

2.2.2.4 Select Menu and The Select Option Box *122*
2.2.2.5 Processes Menu *122*
2.2.2.6 SingleProcess Menu *123*
2.2.2.7 Threads Menu *123*
2.2.2.8 View Menu *124*
2.2.2.9 Help Menu *126*
2.3 COMMAND LANGUAGE 126
2.3.1 Command Usage *126*

INDEX 131

LIST OF TABLES

Table 1-1: *PGDBG* Operators 16

Table 1-2: Debugger Commands 49

Table 1-3: *PGDBG* Commands 50

Table 1-4: General Registers 58

Table 1-5: Floating-Point Registers 58

Table 1-6: Segment Registers 58

Table 1-7: Special Purpose Registers 59

Table 1-8: General Registers 59

Table 1-9: Floating-Point Registers 59

Table 1-10: Segment Registers 60

Table 1-11: Special Purpose Registers 60

Table 1-12: SSE Registers 60

Table 1-13: Thread State is Described using Color 74

Table 1-14: Process state is described using color 76

Table 1-15: MPI-CH Support 77

Table 1-16: The *PGDBG* Debug Modes 80

Table 1-17: P/t-set commands 86

Table 1-18: *PGDBG* Parallel Commands 88

Table 1-19: *PGDBG* Stop Modes 93

Table 1-20: *PGDBG* Wait Modes 94

Table 1-21: *PGDBG* Wait Behavior 95

Table 1-22: *PGDBG* Status Messages 97

LIST OF FIGURES

Figure 1-1: *PGDBG* Debugger Main Window 62

Figure 1-2: Disassembly Window 64

Figure 1-3: Register Window 65

Figure 1-4: Memory Window 66

Figure 1-5: Custom Window 67

Figure 1-6: *PGDBG* GUI Interface: PGI Workstation 73

Figure 1-7: *PGDBG* GUI Interface: Cluster Development Kit 79

Figure 2-1: Profiler Window 119

Figure 2-2: View Menu 125

Preface

This guide describes how to use The Portland Group Compiler Technology (PGI) Fortran, *C*, and *C++* debugger and profiler tools. In particular, these include the *PGPROF* profiler, and the *PGDBG* debugger. You can use the PGI compilers and tools to debug and profile serial (uni-processor) and parallel (multi-processor) applications for X86 and AMD64 processor-based systems.

Audience Description

This guide is intended for scientists and engineers using the PGI debugging and profiling tools. To use these tools, you should be aware of the role of high-level languages (e.g., Fortran, *C*, *C++*) and assembly-language in the software development process and should have some level of understanding of programming. The PGI tools are available on a variety of operating systems for the X86 and AMD64 hardware platforms. You need to be familiar with the basic commands available on your system.

Finally, your system needs to be running a properly installed and configured version of the compilers. For information on installing PGI tools, refer to the installation instructions.

Compatibility and Conformance to Standards

The PGI compilers run on a variety of systems and produce code that conforms to the ANSI standards for FORTRAN 77, Fortran 90, *C*, and *C++* and includes extensions from MIL-STD-1753, VAX/VMS Fortran, IBM/VS Fortran, SGI Fortran, Cray Fortran, and K&R *C*. *PGF77*, *PGF90, PGCC* ANSI *C,* and *C++* support parallelization extensions based on the OpenMP defacto standard. *PGHPF* supports data parallel extensions based on the High Performance Fortran (HPF) defacto standard. The PGI Fortran reference manuals describe Fortran statements and extensions as implemented in the PGI Fortran compilers.

For further information, refer to the following:

- *American National Standard Programming Language FORTRAN*, ANSI X3. -1978 (1978).
- *American National Standard Programming Language FORTRAN*, ANSI X3. -1991 (1991).
- *International Language Standard ISO Standard 1539-199 (E)*.

- *Fortran 90 Handbook*, Intertext-McGraw Hill, New York, NY, 1992.
- *High Performance Fortran Language Specification*, Revision 1.0, Rice University, Houston, Texas (1993), http://www.crpc.rice.edu/HPFF.
- *High Performance Fortran Language Specification*, Revision 2.0, Rice University, Houston, Texas (1997), http://www.crpc.rice.edu/HPFF.
- *OpenMP Fortran Application Program Interface*, Version 1.1, November 1999, http://www.openmp.org.
- *OpenMP C and C++ Application Program Interface*, Version 1.0, October 1998, 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.
- HPDF Standard (High Performance Debugging Forum) http://www.ptools.org/hpdf/draft/intro.html

Organization

This manual is divided into the following chapters:

Chapter 1 — *The PGDBG Debugger* describes the *PGDBG* symbolic debugger. *PGDBG* is a symbolic debugger for Fortran, *C*, *C++* and assembly language programs. Sections 1.1 through 1.13 describe *PGDBG* invocation, commands, signals, debugging Fortran and C++ using *PGDBG*, the *PGDBG* graphical user interface, and *PGDBG* parallel debugging capabilities.

1.14 Debugging Parallel Programs with PGDBG describes how to invoke the debugger for thread-parallel (SMP) debugging and for process-parallel (MPI) debugging.

1.15 Thread-parallel and Process-parallel Debugging describes how to name a single thread, how to group threads and processes into sets, and

how to apply PGDBG commands to groups of processes and threads.

1.16 OpenMP Debugging describes some debug situations within the context of a single process composed of many OpenMP threads.

1.17 MPI Debugging describes how *PGDBG* is used to debug parallel-distributed MPI programs and hybrid distributed SMP programs.

1.18 Limitations describe the limitations of the *PGDBG* tool.

Chapter 2 — *The PGPROF Profiler* describes the *PGPROF* Profiler. This tool analyzes data generated during execution of specially compiled *C*, *C*++, F77, F90 and HPF programs.

Conventions

This *User's Guide* uses the following conventions:

italic	is used for commands, filenames, directories, arguments, options and for emphasis.
`Constant Width`	is used in examples and for language statements in the text, including assembly language statements.
[*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 upper-case characters and a reduced point size.

Related Publications

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

- *PGF77 Reference User Manual* describes the FORTRAN 77 statements, data types, input/output format specifiers, and additional reference material.
- *PGHPF Reference Manual* describes the HPF statements, data types, input/output format specifiers, and additional reference material.
- *System V Application Binary Interface Processor Supplement* by AT&T UNIX System Laboratories, Inc. (Prentice Hall, Inc.).
- *FORTRAN 90 HANDBOOK,* Complete ANSI/ISO Reference (McGraw-Hill, 1992).
- *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 User's Guide*, *PGI Tools User's Guide*, *PGI 5.0 Release Notes*, FAQ, Tutorials
  http://www.pgroup.com/docs.htm
- MPI-CH
  http://www.netlib.org/
- OpenMP
  http://www.openmp.org/
- Ptools (Parallel Tools Consortium)
  http://www.ptools.org/
- HPDF (High Performance Debugging Forum) Standard
  http://www.ptools.org/hpdf/draft/intro.html

System Requirements

- PGI CDK 5.0, or WS 5.0
- Linux (See http://www.pgroup.com/faq/install.htm for supported releases)
- Intel X86 (and compatible), AMD Athlon, AMD64 processors

Chapter 1

The *PGDBG* Debugger

This chapter describes the *PGDBG* symbolic debugger. *PGDBG* is a symbolic debugger for Fortran, *C*, *C++* and assembly language programs. It allows you to control the execution of programs using breakpoints and single-stepping, and lets you check the state of a program by examining variables, memory locations, and registers. The following are *PGDBG* capabilities.

- Provides the capability to debug SMP Linux programs.
- Provides the capability to debug MPI programs on Linux clusters.
- Provides the capability to debug hybrid SMP/MPI programs on Linux clusters where each node contains multiple CPUs sharing memory but where each node has a separate memory from all other nodes.

1.1 Definition of Terms

Host — The system on which *PGDBG* executes. This will generally be the system where source and executable files reside, and where compilation is performed.

Target — A program being debugged.

Target Machine — The system on which a target runs. This may or may not be the same system as the host.

For an introduction to terminology used to describe parallel debugging, see Section *1.14.1 Processes and Threads*.

1.1.1 Compiler Options for Debugging

Use the *–g* compiler command line option to build programs for debugging. This option causes information about the symbols and source files in the program to be included in the executable file (this option also sets the optimization to level zero unless you specify *–O* on the command line). Programs built without *–g* can be debugged; however, information about types, local variables, arguments and source file line numbers is not available unless you specify *–g*.

When the *-g* compiler command line option is used, PGI compilers emit DWARF Version 2 debug information by default. To emit DWARF Version 1 debug information, specify the *-Mdwarf1* option with the *-g* option at the compiler command line.

1.2 Invocation and Initialization

PGDBG is invoked using the `pgdbg` command as follows:

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

where *`arguments`* may be any of the command-line arguments described in the following section, *Command-line Arguments*. See *1.14.4.1 Invoking PGDBG: MPI Debugging* for how to debug an MPI program.

The *`program`* is the name of the target program being debugged. The arguments *`arg1 arg2 … argn`* are the command-line arguments to the target program. The debugger begins by creating a symbol table for the program. Then the program is loaded into memory.

If an initialization file named `.pgdbgrc` exists in the current directory or in the home directory, it is opened and *PGDBG* executes the commands in the file. The initialization file is useful for defining common aliases, setting breakpoints and for other startup commands. 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 may execute the initialization file in the home directory, or execute *PGDBG* commands in any other file (for example in the file `.dbxinit` if you have a *dbx* debugger initialization file set up).

After processing the initialization file, *PGDBG* is ready to process commands. Normally, a session begins by setting one or more breakpoints, using the *break*, *stop* or *trace* commands, and then issuing a *run* command followed by *cont, step, trace* or *next.*

1.3 Command-Line Arguments

The `pgdbg` command accepts several command line arguments that must appear on the command line *before* the name of the program being debugged. The valid options are:

−dbx	Start the debugger in *dbx* mode.
−−program_args	*PGDBG* passes all arguments following this command line option to the program being debugged if an executable is included on the command line.
−s *startup*	The default startup file is ~/.pgdbgrc. The *–s* option specifies an alternate startup file *startup*.
−c "*command*"	Execute the debugger command *command* (*command* must be in double quotes) before executing the commands in the startup file.
−r	Run the debugger without first waiting for a command. If the program being debugged runs successfully, the debugger terminates. Otherwise, the debugger is invoked and stops when the trap occurs.
−text	Run the debugger using a command-line interface (CLI). The default is for the debugger to launch in graphical user interface (GUI) mode.
−tp k8-32	Debug a program running on an X86 target machine. This option is only necessary if the default PGDBG, determined by the PATH environment variable, is not capable of debugging X86 targeted programs.
−tp k8-64	Debug a program running on an AMD64 target machine. This option is only necessary if the default PGDBG, determined by the PATH environment variable, is not capable of debugging AMD64 targeted programs.

1.4 Command Language

User input is processed one line at a time. Each line must begin with the name of a command and its arguments, if any. The command language is composed of commands, constants, symbols, locations, expressions, and statements.

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.

There are two command modes: *pgi* and *dbx*. The *pgi* command mode maintains the original *PGDBG* command interface. In *dbx* mode, the debugger uses commands with a syntax compatible with the familiar *dbx* debugger. Both command sets are available in both command modes, however some commands have a slightly different syntax depending on the mode. The *pgienv* command allows you to change modes while running the debugger.

1.4.1 Constants

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

1.4.2 Symbols

PGDBG 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, subprograms (functions, and subroutines), types (including structure, union, pointer, array, and enumeration types), variables, and arguments. Symbol names are case-sensitive and must match the name as it appears in the object file.

The compilers add an underscore character, "_", to the beginning of each external identifier. On UNIX systems, the PGI Fortran compilers also add an underscore to the end of each external identifier. Therefore, if *PGDBG* is unable to locate a symbol as entered, it prepends an underscore and tries again. If that fails, it adds an underscore to the end of the name and tries again. If that fails, the leading underscore is stripped and the search is repeated. For example, if `cfunc` and `ffunc` are *C* and Fortran functions, respectively, then the names for the symbols in the object file are `_cfunc` and `_ffunc_`. *PGDBG* will accept `cfunc`, and `_cfunc` as names for `_cfunc`, and will accept `ffunc`, `_ffunc`, and `_ffunc_` as names for `_ffunc_`. Note however, that due to case-sensitivity, `FFUNC`, `_FFUNC`, etc. are not accepted as names for `_ffunc_`.

1.4.3 Scope Rules

Since several symbols may have the same name, scope rules are used to bind identifiers to symbols. *PGDBG* uses a notion of *search scope* for looking up identifiers. The *search scope* is a symbol which represents a function, a source file, or global scope. When the user enters a name, *PGDBG* 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 will be the same as the *current scope*, which is the function where execution is currently stopped. The current scope and the search scope are both set to the current function each time execution of the target program stops. However, the *enter* command changes the search scope.

A scope qualifier operator `@` allows selection of out-of-scope identifiers. For example, if `f` is a function 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, for example:

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

represents the variable `i` defined in file `xyz.c`.

1.4.4 Register Symbols

In order to provide access to the system registers, *PGDBG* builds symbols for them. Register names generally begin with `$` to avoid conflicts with program identifiers. Each register symbol has a type associated with it, and registers are treated like global variables of that type, except that their address may not be taken. See Section *1.10 Commands Summary* for a complete list of the register symbols.

1.4.5 Source Code Locations

Some commands need to reference 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.

Thus:

```
break 37
```

sets a breakpoint at line 37 of the current source file, and

```
break "xyz.c"@37
```

sets a breakpoint at line 37 of the source file `xyz.c`.

A range of lines is indicated using the range operator ":". Thus,

```
list 3:13
```

lists lines 3 through 13 of the current file, and

```
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 conversion commands *line*, and *addr* convert a constant to a line, or to an address respectively. For example:

```
{line 37}
```

means "line 37",

```
{addr 0x1000}
```

means "address 0x1000" , and

```
{addr {line 37}}
```

means "the address associated with line 37" , and

```
{line {addr 0x1000}}
```

means "the line associated with address `0x1000`".

1.4.6 Lexical Blocks

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

Below variable ***var*** 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***.

For Example:

```
lex.c:
main()
{
    int var = 0;
    {
        int var = 1;
        printf("var %d\n",var);
    }
    printf("var %d\n",var)
}
```

```
pgdbg> n
Stopped at 0x8048b10, function main, file
/home/pete/pgdbg/bugs/workon3/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
```

1.4.7 Statements

Although input is processed one line at a time, statement constructs allow multiple commands per line, and 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 by separating them with a semicolon. For example:

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

sets breakpoints in functions `main` and `xyz`, continues, and prints the new current location. Any value returned by the last statement on a line is printed.

Statements can be parallelized across multiple threads of execution. See Section *1.15.17 Parallel Statements* for details.

1.4.8 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.

The default action is to print the new value of the expression, and prompt the user for commands. By adding a location, or a condition, the event can be limited to a particular address or function, or may occur only when the condition is true. The action to be taken when an event occurs can be defined by specifying a command list.

PGDBG supports four 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*, *track* and *do*. The command *break* 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 *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.

The optional arguments bring flexibility to the event definition. They are:

`at` *`line`*	Event occurs at indicated line.
`at` *`addr`*	Event occurs at indicated address.
`in` *`function`*	Event occurs throughout indicated function.
`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. For example:

```
watch i at 37 if(y>1)
```

This event definition says that whenever execution is at line 37, and the value of `i` has changed since the last time execution was at line 37, and `y` is greater than 1, stop and print the new value of `i`.

```
do {print xyz} in f
```

This event definition says that at each line in the function `f` print the value of `xyz`.

```
break func1 if (i==37) do {print a[37]; stack}
```

This event definition says that each time the function `func1` is entered and `i` is equal to 37, then the value of `a[37]` should be printed, and a stack trace should be performed.

Event commands that do not explicitly define a location will occur at each source line in the program. For example:

```
do {where}
```

prints the current location at the start of each source line, and

```
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 to perform them requires single-stepping the target program (this may slow execution considerably). Restricting an event to a particular address causes minimal impact on program execution speed, while restricting an event to a single function causes execution to be slowed only when that function is executed.

PGDBG supports instruction level versions of several commands (for example *breaki*, *watchi*, *tracki*, and *doi*). The basic difference in the instruction version is that these commands will interpret integers as addresses rather than line numbers, and events will occur at each instruction rather than at each line.

When multiple events occur at the same location, all event actions will be 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:

```
break 37 do {continue}

break 37 do {print i}
```

This creates an ambiguous situation. It's not clear whether `i` should 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:

```
break 37
```

sets a breakpoint at line 37 in the current file.

```
track i
```

will track the value of whatever variable `i` is currently in scope. 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 Section *1.15.16 Parallel Events* for details.

1.4.9 Expressions

The debugger supports evaluation of expressions composed of constants, identifiers, and commands if they return values, and operators. Table 1-1 shows the *C* language operators that are supported. The operator precedence is the same as in the *C* language.

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

```
break {pc}+8
```

invokes the pc command to compute the current address, adds 8 to it, and sets a breakpoint at that address. Similarly, the following command compares the start address of the current function, with the start address of function `xyz`, and prints the value 1, if they are equal and 0 otherwise.

```
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 "->" .

PGDBG recognizes a range operator ":" which indicates array sub-ranges or source line ranges. For example,

```
print a[1:10]
```

prints elements 1 through 10 of the array `a,` and

```
list 5:10
```

lists source lines 5 through 10, and

```
list "xyz.c"@5:10
```

lists lines 5 through 10 in file `xyz.c`. The precedence of ':' is between '||' and '=' .

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. See Section *1.15.8 Current vs. Prefix P/t-set* for details.

Table 1-1: *PGDBG* Operators

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

1.5 Signals

PGDBG intercepts all signals sent to any of the threads in a multi-threaded program, and passes them on according to that signal's disposition maintained by *PGDBG* (see the *catch, ignore* commands).

If a thread runs into a busy loop or if the program runs into deadlock, control-C over the debugging command line to interrupt the threads. This causes SIGINT to be sent to all threads. By default *PGDBG* does not relay SIGINT to any of the threads, so in most cases program behavior is not affected.

Sending a SIGINT (control-C) to a program while it is in the middle of initializing its threads (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. When the number of threads employed by a program is large, thread initialization may take a while.

Sending SIGINT (control-C) to a running MPI program is not recommended. See Section *1.17.5 MPI Listener Processes* for details.

1.5.1 Signals Used Internally by *PGDBG*

SIGTRAP indicates a breakpoint has been hit. A message is displayed whenever a thread hits a breakpoint. SIGSTOP is used internally by *PGDBG*. Its use is mostly invisible to the user. Changing the disposition of these signals in *PGDBG* will result in undefined behavior.

Reserved Signals: On Linux, the thread library uses SIGRT1, SIGRT3 to communicate among threads internally. In the absence of real-time signals in the kernel, SIGUSR1, SIGUSR2 are used. Changing the disposition of these signals in *PGDBG* will result in undefined behavior.

1.6 Debugging Fortran

In order to create symbolic information for debugging, invoke your PGI Fortran compiler with the *–g* option. Fortran type declarations are printed using Fortran type names, not *C* type names. The only exception is Fortran character types, which are treated as arrays of *C* characters.

1.6.1 Arrays

Large arrays, arrays with lower dimensions, and adjustable arrays are all supported. Fortran array elements and ranges should be accessed using parentheses, rather than square brackets.

1.6.2 Operators

Only those operators that exist in the *C* language may be used in expressions. In particular `.eq.`, `.ne.`, and so forth are not supported. The analogous *C* operators `=`, `!=`, etc. must be used instead. Note that the precedence of operators matches the *C* language, which may in some cases be different than for Fortran.

1.6.3 Name of Main Routine

If a `PROGRAM` statement is used, the name of the main routine is the name in the program statement. Otherwise, the name of the main routine is `__unnamed_`. A function symbol named `_MAIN_` is defined with start address equal to the start of the main routine. As a result,

```
break MAIN
```

can always be used to set a breakpoint at the start of the main routine.

1.6.4 Fortran Common Blocks

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

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

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

then the entire common can be printed out using,

```
print xyz
```

and the individual elements can be accessed by name as in,

```
print a, b
```

1.6.5 Nested Subroutines

To reference a nested subroutine you must qualify its name with the name of its enclosing function using the scoping operator @.

For example:

```
subroutine subtest (ndim)
   integer(4), intent(in) :: ndim
   integer, dimension(ndim) :: ijk
   call subsubtest ()
   contains
       subroutine  subsubtest ()
       integer :: I
       i=9
       ijk(1) = 1
       end subroutine subsubtest
       subroutine  subsubtest2 ()
       ijk(1) = 1
       end subroutine subsubtest2
   end subroutine subtest
   program testscope
   integer(4), parameter :: ndim = 4
   call subtest (ndim)
   end program testscope
```

```
pgdbg> break subtest@subsubtest
breakpoint set at: subsubtest line: 8 in "ex.f90" address: 0x80494091
pgdbg> names subtest@subsubtest
i = 0
pgdbg> decls subtest@subsubtest
arguments:
```

```
variables:
integer*4 i;
pgdbg> whereis subsubtest
function:       "ex.f90"@subtest@subsubtest
```

1.6.6 Fortran 90 Modules

To access a member ***mm*** of a Fortran 90 module ***M*** you must qualify ***mm***

with ***M*** using the scoping operator **@**. If the current scope is ***M*** the qualification can be omitted.

For example:

```
module M
    implicit none
    real mm
    contains
    subroutine stub
    print *,mm
    end subroutine stub
    end module M

program test
    use M
    implicit none
    call stub()
    print *,mm
    end program test

pgdbg> Stopped at 0x80494e3, function MAIN, file M.f90, line 13
#13:       call stub()
pgdbg> which mm
"M.f90"@m@mm
pgdbg> print "M.f90"@m@mm
0
pgdbg> names m
mm = 0
stub = "M.f90"@m@stub
pgdbg> decls m
real*4 mm;
subroutine stub();
pgdbg> print m@mm
0
pgdbg> break stub
```

```
breakpoint set at: stub line:6 in "M.f90" address: 0x8049446      1
pgdbg> c
Stopped at 0x8049446, function stub, file M.f90, line 6
Warning: Source file M.f90 has been modified more recently than object
file
#6:           print *,mm
pgdbg> print mm
0
pgdbg>
```

1.7 Debugging *C++*

In order to create symbolic information for debugging, invoke your PGI *C++* compiler with the *–g* option.

Calling C++ Instance Methods

To call a *C++* instance method, the object must be explicitly passed as the first parameter to the call. For example, given the following definition of `class Person` and the appropriate implementation of its methods:

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

main(){
      Person * pierre;
      pierre =  new Person("Pierre");
      pierre.print();
    }
```

To call the instance method `print` on object `pierre`, use the following syntax:

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

Notice that `pierre` is explicitly passed into the method, and the class name must also be specified.

1.8 Core Files

Some implementations of *PGDBG* are capable of interpreting core files. The debugger invocation for these systems has been modified as follows:

```
pgdbg [-core corefile] program arg1 ... argn
```

Using the `-core` option an informational message is printed on invocation indicating that the core file is being read. Before any data from the core file can be accessed, you must use the **cont** command. The debugger will then indicate that it is stopped at the location of the violation that caused the core file to be generated. At this point, memory, registers and instruction space may be displayed just as if the program were active and breakpoints may be set.

While most execution related commands are ignored, the *run* command causes the program to be loaded and executed. Thereafter, *PGDBG* will behave just as if it had been invoked without the `-core` option.

1.9 *PGDBG* Commands

This section describes the *PGDBG* command set in detail. Section 1.9 Commands presents a table of all the debugger commands, with a summary of their syntax.

1.9.1 Commands

Command names may be abbreviated as indicated. Some commands accept a variety of arguments. Arguments contained in `[` and `]` are optional. Two or more arguments separated by `|` 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. Argument names appear in italics and are chosen to indicate what kind of argument is expected. For example:

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

indicates that the command *list* may be abbreviated to *lis*, and that it will accept either no argument, an integer count, a line range, a function name, or a line and a count.

1.9.1.1 Process Control

The following commands, together with the breakpoints described in the next section, let you control the execution of the target program. *PGDBG* allows you to easily group and control multiple threads and processes. See Section *1.15.11 Process and Thread Control* for more details.

c[ont]

Continue execution from the current location. This command may also be used to begin execution of the program at the beginning of the session.

de[bug]

Print the name and arguments of the program being debugged.

halt

Halt the running process or thread.

n[ext] [*count*]

Stop after executing one source line in the current function. This command steps over called functions. The *count* argument stops execution after executing *count* source lines. In a parallel region of code, *next* applies only to the currently active thread.

nexti [*count*]

Stop after executing one instruction in the current function. This command steps over called functions. The *count* argument stops execution after executing *count* instructions. In a parallel region of code, *nexti* applies only to the currently active thread.

proc [*number*]

Set the current thread to number. When issued with no argument, proc lists the current program location of the current thread of the current process. See Section *1.14.4 Process-Parallel Debugging* for how processes are numbered.

procs

Print the status of all live processes.

q[uit]

Terminate the debugging session.

rer[un]
rer[un] [*arg0 arg1 ... argn*] [< *inputfile*] [> *outputfile*]

Works like run except if no *args* are specified, none are used.

ru[n]
ru[n] [*arg0 arg1 ...argn*] [< *inputfile*] [> *outputfile*]

Execute program from the beginning. If arguments *arg0*, *arg1*, .. are specified, they are set up as the command line arguments of the program. Otherwise, the arguments for 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.

s[tep]
s[tep] *count*
s[tep] up

Stop after executing one source line. This command steps into called functions. The *count* argument, stops execution after executing *count* source lines. The `up` argument stops execution after stepping out of the current function. In a parallel region of code, *step* applies only to the currently active thread.

stepi
stepi *count*
stepi up

Stop after executing one instruction. This command steps into called functions. The *count* argument stops execution after executing *count* instructions. The `up` argument stops the execution after stepping out of the current function. In a parallel region of code, *stepi* applies only to the currently active thread.

stepo[ut]

Stop after returning to the caller of the current function. This command sets a breakpoint at the current return address, and does a *continue*. To work correctly, it must be possible to compute the value of the return address. Some functions, particularly terminal functions at higher optimization levels, may not set up a stack frame. Executing *stepout* from such a function will cause the breakpoint to be set in the caller of the most recent function that set up a stack frame. This command stops immediately upon return to the calling function. This means that the current location may not be the start of a source line because multiple function calls may occur on a single source line, and a user might want to stop after the first call. Users who want to step out of the current function and continue to the start of the next source line should simply follow *stepout* with *next*. In a parallel region of code, *stepout* applies only to the currently active thread.

sync

synci

Advance process/thread to specific program location, ignoring user defined events that fire.

thread [*number*]

Set the active thread to *number*. When issued with no argument, ***thread*** lists the current program location of the currently active thread. On Linux systems, *number* will be a 5-digit thread identifier.

threads

Print a status of all live threads. NCPUS+1 live threads will be listed, where NCPUS is the environment variable which specifies how many processes the program should use. One of the threads is a master thread that is not used in computations. It will always be listed as waiting at `_fini`.

wait

Return *PGDBG* prompt only after specific processes or threads stop.

1.9.1.2 Process-Thread Sets

The following commands deal with defining and managing process thread sets. See Section *1.15.9 P/t-set Commands* for a general discussion of process-thread sets.

defset

Assign a name to a process/thread set. Define a named set. This set can later be referred to by name. A list of named sets is stored by *PGDBG.*

focus

Set the target process/thread set for commands. Subsequent commands will be 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]` can not be removed

viewset

List the members of a process/thread set that currently exist as active threads

whichsets

List all defined p/t-sets to which the members of a process/thread set belongs

1.9.1.3 Events

The following commands deal with defining and managing events. See Section *1.4.8 Events* for a general discussion of events, and the optional arguments.

b[reak]
b[reak] *line* [if (*condition*)] [do {*commands*}]
b[reak] *func* [if (*condition*)] [do {*commands*}]

If no argument is specified, print the current breakpoints. Otherwise, set a breakpoint at the indicated line or function. If a function is specified, and the function was compiled for debugging, then the breakpoint is set at the start of the first statement in the function, that is, after the function's prologue code. If the function was not compiled for debugging, then the breakpoint is set at the first instruction of the function, 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* evaluates true. If *do* is specified with a *command* or several *commands* as an argument, the command or commands are executed when the breakpoint occurs.

The following examples set breakpoints at line 37 in the current file, line 37 in file `xyz.c`, the first executable line of function `main`, address `0xf0400608`, the current line, and the current address, respectively.

```
break 37
    break "xyz.c"@37
    break main
    break {addr 0xf0400608}
    break {line}
    break {pc}
```

More sophisticated examples include:

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

This command stops when function *xyz* is entered only if the argument *n* is greater than 10.

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

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

breaki
breaki *func* [if (*condition*)] [do {*commands*}]
breaki *addr* [if (*condition*)] [do {*commands*}]

Set a breakpoint at the indicated address or function. If a function is specified, the breakpoint is set at the first address of the function. This means that when the program stops at this breakpoint the prologue code which sets up the stack frame will not yet have been executed, and hence, values of stack arguments will not be correct. Integer constants are interpreted as addresses. To specify a line, use the *line* command to convert the constant to a line number, or use the *break* command.

The *if*, and *do* arguments are interpreted as in the *break* command. The next examples set breakpoints at address `0xf0400608`, line 37 in the current file, line 37 in file `xyz.c`, the first executable address of function `main`, the current line, and the current address, respectively:

```
breaki 0xf0400608
    breaki {line 37}
    breaki "xyz.c"@37
    breaki main
    breaki {line}
    breaki {pc}
```

Similarly,

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

stops and prints the new value of `n` at address `0x6480` only if `n` is greater than 3.

breaks

Display all the existing breakpoints.

catch
catch [*sig:sig*]
catch [*sig* [, *sig*...]]

With no arguments, print the list of signals being caught. With the `:` argument, catch the specified range of signals. With a list, trap signals with the specified number.

clear
clear all
clear *func*
clear *line*
clear addr {*addr*}

Clear all breakpoints at current location. Clear all breakpoints. Clear all breakpoints from first statement in the specified function *func.* Clear breakpoints from line number *line.* Clear breakpoints from the address *addr*.

del[ete] *event-number*
del[ete] 0
del[ete] all
del[ete] *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 a comma.

disab[le] *event-number*
disab[le] all

Disable the indicated event *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.

do {*commands*} [if (*condition*)]
do {*commands*} at *line* [if (*condition*)]
do {*commands*} in *func* [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 The *at* argument with a *line* specifies the commands to be executed each time that line is reached. The *in* argument with a func specifies the commands are executed at each line in the function. The *if* option has the same meaning as in *watch*. If a condition is specified, the *do* commands are executed only when *condition* is true.

doi {*commands*} [if (*condition*)]
doi {*commands*} at *addr* [if (*condition*)]
doi {*commands*} in *func* [if (*condition*)]

Define a *doi* event. This command is similar to *watchi* except that instead of defining an expression, it defines a list of commands to be executed. If an addr is specified, the commands are executed each time that address is reached. If a function func is specified, the commands are

executed at each line in the function. If neither is specified, the commands are executed at each line in the program. The *if* option has the same meaning as in *do* above

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

Enable the indicated event *event-number*, or all events.

hwatch *addr* [if (*condition*)] [do {*commands*}]

Define a hardware watchpoint. This command uses hardware support to create a watchpoint for a particular address. 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.

If an *if* option is specified, the event will cause no action unless the expression is true. If a *do* option is specified then the commands will be executed when the event occurs.

hwatchr[ead] *addr* [if (*condition*)] [do {*commands*}]

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

hwatchb[oth] *addr* [if (*condition*)] [do {*commands*}]

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

ignore
ignore[*sig:sig*]
ignore [*sig* [, *sig*...]]

With no arguments, print the list of signals being ignored. With the `:` argument, ignore the specified range of signals. With a list, ignore signals with the specified number.

stat[us]

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

stop *var*
stop at *line* [if (*condition*)][do {*commands*}]
stop in *func* [if (*condition*)][do {*commands*}]
stop if (*condition*)

Set a breakpoint at the indicated function or line. Break when the value of the indicated variable *var* changes. The *at* keyword and a number specifies a line number. The *in* keyword and a function name specifies the first statement of the specified function. With the *if* keyword, the debugger stops when the condition *condition* is true.

stopi *var*
stopi at *address* [if (*condition*)][do {*commands*}]
stopi in *func* [if (*condition*)][do {*commands*}]
stopi if (*condition*)

Set a breakpoint at the indicated address or function. Break when the value of the indicated variable var changes. The *at* keyword and a number specifies an address to stop at. The *in* keyword and a function name specifies the first address of the specified function to stop at. With the *if* keyword, the debugger stops when condition is true.

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 a new value is printed.

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

Define an instruction level track event. This command is equivalent to *watchi* except that execution resumes after the new value is printed.

trace *var* [if (*condition*)][do {*commands*}]
trace *func* [if (*condition*)][do {*commands*}]
trace at *line* [if (*condition*)][do {*commands*}]
trace in *func* [if (*condition*)][do {*commands*}]

Activate source line tracing when var changes. Activate source line tracing and trace when in function func. With *at*, activate source line tracing to display the specified line each time it is executed. With **in**, activate source line tracing to display the specified each source line when in the specified function. If condition is specified, trace is on only if the condition evaluates to true. The *do* keyword defines a list of commands to execute at each trace point. Use the command *pgienv speed secs* to set the time in seconds between trace points. Use the *clear* command to remove tracing for a line or function.

tracei *var* [if (*condition*)][do {*commands*}]
tracei *func* [if (*condition*)][do {*commands*}]
tracei at *addr* [if (*condition*)][do {*commands*}]
tracei in *func* [if (*condition*)][do {*commands*}]

Activate instruction tracing when var changes. Activate instruction tracing when in function func. With *at*, activate tracing to display the specified line each time it is executed. With the *in* keyword, display instructions while in the specified function. Use the command *pgienv speed secs to* set the time in seconds between trace points. Use the *clear* command to remove tracing for a line or function.

unb[reak] *line*
unb[reak] *func*
unb[reak] all

Remove a breakpoint from the statement line. Remove a breakpoint from the function *func*. Remove all breakpoints.

unbreaki *addr*
unbreaki *func*
unbreaki all

Remove a breakpoint from the address *addr*. Remove a breakpoint from the function *func*. Remove all breakpoints.

wa[tch] *expression*
wa[tch] *expression* [if (*condition*)][do {*commands*}]
wa[tch] *expression* at *line* [if (*condition*)][do {*commands*}]
wa[tch] *expression* in *func* [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. If a particular *line* is specified, the expression is only evaluated at that line. If a function *func* is specified, the expression is evaluated at each line in the function. If no location is specified, the expression will be 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, they are executed whenever the expression is evaluated and the value changes.

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

Note: *Using watchpoints indiscriminately can dramatically slow program execution.*

Using the *at* and *in* options 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
```

will barely slow program execution at all, while

```
watch i
```

will slow execution considerably.

watchi *expression*
watchi *expression* [if(*condition*)][do {*commands*}]
watchi *expression* at *addr* [if(*condition*)][do {*commands*}]
watchi *expression* in *func* [if(*condition*)][do {*commands*}]

Define an instruction level watch event. This is just like the *watch* command except that the *at* option interprets integers as addresses rather than line numbers, and the expression is evaluated at every instruction instead of at every line.

This command is useful if line number information is limited. It causes programs to execute more slowly than *watch*.

when do {*commands*} [if (*condition*)]
when at *line* do {*commands*} [if (*condition*)]
when in *func* do {*commands*} [if (*condition*)]

Execute command at every line in the program. Execute commands at specified line in the program. Execute command in the specified function. If the optional *condition* is specified, commands are executed only when the expression evaluates to true.

wheni do {*commands*} [if (*condition*)]
wheni at *addr* do {*commands*} [if (*condition*)]
wheni in *func* do {*commands*} [if (*condition*)]

Execute *commands* at each address in the program. If an *addr* is specified, the commands are executed each time the address is reached. If a function *func* is specified, the commands are executed at each line in the function. If the optional *condition* is specified, commands are executed whenever the expression is evaluated true.

Events can be parallelized across multiple threads of execution. See Section *1.15.16 Parallel*

Events for details.

1.9.1.4 Program Locations

This section describes PGDBG program locations commands.

arri[ve]

Print location information and update GUI markers for the current location.

cd [*dir*]

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

dis[asm]
dis[asm] *count*
dis[asm] *lo*:*hi*
dis[asm] *func*
dis[asm] *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 is given, disassemble the memory in the range. If a function name is given, disassemble the entire function. If the function was compiled for debug, and source code is available, the source code will be interleaved with the disassembly. If an address and a count are given, disassemble *count* instructions starting at address *addr*.

edit
edit *filename*
edit *func*

If no argument is supplied, edit the current file starting at the current location. With a *filename* argument, edit the specified file *filename*. With the func argument, edit the file containing function *func*. This command uses the editor specified by the environment variable $EDITOR.

file [*filename*]

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

lines *function*

Print the lines table for the specified function.

lis[t]
lis[t] *count*
lis[t] *line*,*num*
lis[t] *lo*:*hi*
lis[t] *function*

With no argument, list 10 lines centered about the current source line. If a count is given, list *count* lines centered about the source line. If a line and count are given, list *number* lines starting at line number *line*. For the dbx environment, this option lists lines from *start* to *number*. If a line range is given, list the indicated source lines in the current source file (this option is not valid in the dbx environment). If a function name is given, list the source code for the indicated function.

pwd

Print the current working directory.

stack[trace] [*count*]

Print a stacktrace. For each live function print the function name, source file, line number, current address. This command also prints the names and values of the arguments, if available. If a count is specified, display a maximum of count stack frames.

stackd[ump] [*count*]

Print a formatted dump of the stack. This command displays a hex dump of the stack frame for each live function. This command is a machine-level version of the *stacktrace* command. If a count is specified, display a maximum of count stack frames.

w[here] [*count*]

Print the address, function, source file and line number for the current location. If *count* is specified, print a maximum of *count* live functions on the stack.

/
/ [*string*] /

Search forward for a string *string* of characters in the current file. With just /, search for the next occurrence of *string* in the current file.

?

?[*string*] ?

Search backward for a string *string* of characters in the current file. With just ?, search for the previous occurrence of *string* in the current file.

1.9.1.5 Printing and Setting Variables

This section describes *PGDBG* commands used for printing and setting variables.

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 character string.

Character string constants print out literally. For example:

```
pgdbg> print "The value of i is ", i
       The value of i is 37
```

The array sub-range operator **:** prints a range of an array. The following example prints elements 0 through 10 of the array `a`:

```
print a[0:10]
```

printf "*format_string*", *expr*,...*expr*

Print expressions in the format indicated by the format string. Behaves like the *C* library function *printf*. For example:

```
pgdbg> printf "f[%d]=%G",i,f[i]
       f[3]=3.14
```

The *pgienv* command with the *stringlen* argument sets the maximum number of characters that will print with a *print* command. For example, the `char` declaration below:

```
char *c="a whole bunch of chars over 1000 chars long....";
```

A `print c` command will only print the first 512 (or *stringlen*) bytes. Normally, the printing occurs until a `NULL` is reached, but without some limit, the printing may never end.

asc[ii] *exp* [,...*exp*]

Evaluate and print as an ascii character. Control characters are prefixed with the '^' character; that is, .3 prints as ^c. Otherwise, values that can not be printed as characters are printed as integer values prefixed by `\'. for example,. 250 prints as \250.

bin *exp* [,...*exp*]

Evaluate and print the expressions. Integer values are printed in binary.

dec *exp* [,...*exp*]

Evaluate and print the expressions. Integer values are printed in decimal.

display
display *exp* [,...*exp*]

Without arguments, list the expressions set to display at breakpoints. With an argument or several arguments, print expression *exp* at every breakpoint. See the description for *undisplay*.

hex *exp* [,...*exp*]

Evaluate and print the expressions. Integer values are printed in hex.

oct *exp* [,...*exp*]

Evaluate and print the expressions. Integer values are printed in octal.

set *var=expression*

Set variable *var* to the value of *expression*.

str[ing] *exp* [,...*exp*]

For each expression, evaluate, treat the result as a character pointer, and fetch and print a null terminated string from that address. This command will fetch a maximum of 70 characters.

undisplay 0
undisplay all
undisplay *exp* [,...*exp*]

Remove all expressions being printed at breakpoints. With an argument or several arguments,

remove the expression *exp* from the list of display expressions.

1.9.1.6 Symbols and Expressions

This section describes the commands that deal with symbols and expressions.

as[sign] *var* = *exp*

Assign the value of the expression *exp* to the specified variable *var*.

call *func* [(*exp*,...)]

Call the named function. *C* argument passing conventions are used. Breakpoints encountered during execution of the function are ignored. If a signal is caught during execution of the function, execution will stop, but continued execution may produce unpredictable results. The return value, is assumed to be an integer, and is returned by this command. Fortran functions and subroutines can be called, but the argument values will be passed according to *C* conventions.

decl[aration] *name*

Print the declaration for the symbol, based on the type of the symbol in the symbol table. The symbol must be a variable, argument, enumeration constant, function, a structure, union, enum, or a typedef tag. For example, given declarations:

```
int i, iar[10];
    struct abc {int a; char b[4]; struct abc *c;}val;
```

The commands,

```
decl I
    decl iar
    decl val
    decl abc
```

will respectively print out as

```
int i

int iar[10]

struct abc val

struct abc {
       int a;
```

```
        char b[4];
        struct abc *c;
    };
```

entr[y]
entr[y] *func*

Return the address of the first executable statement in the program or specified function. This is the first address after the function's prologue code.

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 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.

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 of an assignment statement. The type of the expression may be any scalar, pointer, structure, or function type.

siz[eof] *name*

Return the size, in bytes, of the variable type *name*.

type *expr*

Return the type of the expression. The expression may contain structure reference operators (`.` , and ->), dereference (*), and array index ([ ]) expressions. For example, given declarations shown previously, the commands:

```
type I
    type iar
    type val
    type val.a
    type val.abc->b[2]
```

produce the following output:

```
int

int [10]

struct abc
```

```
int
char
```

whatis
whatis *name*

With no arguments, print the declaration for the current function. With argument name, print the declaration for the symbol *name*.

1.9.1.7 Scope

The following commands deal with program scope. See Section *1.4.3 Scope Rules* for a discussion of scope meaning and conventions.

decls
decls *func*
decls "*sourcefile*"
decls {global}

Print the declarations of all identifiers defined in the indicated scope. If no scope is given, print the declarations for global scope.

down [*number*]

Enter scope of function down one level or *number* levels on the call stack.

en[ter]
en[ter] *func*
en[ter] "*sourcefile*"
en[ter] {global}

Set the search scope to be the indicated symbol, which may be a function, source file or global. If no scope is specified use the search scope. The default *enter* with no argument is *enter global*.

files

Return the list of the files that make up the object file.

glob[al]

Return a symbol representing global scope. This command can also be used with the scope operator @ to specify symbols at global scope.

names
names *func*
names "*sourcefile*"
names {global}

Print the names of all identifiers defined in the indicated scope. If no scope is specified, use the search scope.

sco[pe]

Return a symbol for the search scope. The search scope is set to the current function each time program execution stops. It may also be set using the *enter* command. The search scope is always searched first for symbols.

up [*number*]

Enter scope of function up one level or *number* levels on the call stack.

whereis *name*

Print all declarations for *name*.

which *name*

Print full scope qualification of symbol *name*.

1.9.1.8 Register Access

System registers can be accessed by name. See Section *1.4.4 Register Symbols* for the complete set of registers. A few commands exist to access common registers.

fp

Return the current value of the frame pointer.

pc

Return the current program address.

regs

Print a formatted display of the names and values of the integer, float, and double registers.

ret[addr]

Return the current return address.

sp

Return the current value of the stack pointer.

1.9.1.9 Memory Access

The following commands display the contents of arbitrary memory locations.

cr[ead]*addr*

Fetch and return an 8-bit signed integer (character) from the specified address.

dr[ead]*addr*

Fetch and return a 64 bit double from the specified address.

du[mp] *address*, *count*, "*format-string*"

This command dumps a region of memory according to a *printf*-like format 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 *printf*. Format specifiers are preceded by `%`.

The meaning of the recognized format descriptors is as follows:

`%d, %D, %o, %O, %x, %X, %u, %U`
Fetch and print integral values as decimal, octal, hex, or unsigned. Default size is 32 bits. The size of the item read can be modified by either inserting 'h', or 'l' before the format character to indicate half (16) bits or long (32 bits). Alternatively, a 1, 2, or 4 after the format character can be used to specify the number of bytes to read.

`%c`
Fetch and print a character.

`%f, %F, %e, %E, %g, %G`
Fetch and print a *float* (lower case) or double (upper case) value using *printf* `f`, `e`, or `g` format.

`%s`
Fetch and print a null terminated string.

`%p<format-chars>`
Interpret the next four bytes as a pointer to an item specified by the following format characters. The pointed-to item is fetched and displayed. Examples:

`%px`
pointer to hex int.

`%ps`
pointer to string.

`%pps`
pointer to pointer to string.

`%i`
Fetch an instruction and disassemble it.

`%w, %W`
Display address about to be dumped.

`%z<n>, %Z<n>, %z<-n>, %Z<-n>`
Display nothing but advance or decrement current address by *n* bytes.

`%a<n>, %A<n>`
Display nothing but advance current address as needed to align modulo *n*.

fr[ead]*addr*

Fetch and return a 32 bit float from the specified address.

ir[ead] *addr*

Fetch and return a 32-bit signed integer from the specified address.

sr[ead]*addr*

Fetch and return a 16-bit signed integer from the specified address.

1.9.1.10 Conversions

The commands in this section are useful for converting between different kinds of values. These commands accept a variety of kinds of arguments, and return a value of particular kind.

ad[dr]
ad[dr] *n*
ad[dr] *line*
ad[dr] *func*
ad[dr] *var*
ad[dr] *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 function is given, return the first address of the function.
- If a variable or argument is given, return the address where that variable or argument is stored.

For example:

```
breaki {line {addr 0x22f0}}
```

func[tion]
func[tion] *addr*
func[tion] *line*

Return a function symbol. If no argument is specified, return the current function. If an address is given, return the function containing that address. An integer argument is interpreted as an address. If a line is given, return the function containing that line.

lin[e]
lin[e] *n*
lin[e] *func*

lin[e] *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 function *func* is given, return the first line of the function. If an address *addr* is given, return the line containing that address.

For example, the following command returns the line number of the specified address:

```
line {addr 0x22f0}
```

1.9.1.11 Miscellaneous

The following commands make using the debugger easier.

al[ias]
al[ias] *name*
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 a name and string, are given, make name an alias for string. Subsequently, whenever name is encountered it will be replaced by string. Although string may be an arbitrary string, name must not contain any blanks.

For example:

```
alias xyz print "x= ",x,"y= ",y,"z= ",z; cont
```

creates an alias for `xyz`. Now whenever `xyz` is typed, *PGDBG* will respond as though the following command was typed:

```
print "x= ",x,"y= ",y,"z= ",z; cont
```

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 exists so that users can debug programs even when some or all of the program source files are in a directory other than the user's current directory. For example:

```
dir morestuff
```

adds the directory *morestuff* to the list of directories to be searched. Now, source files stored in

morestuff are accessible to *PGDBG*.

If the first character in pathname is ~, it will be substituted by $HOME.

help [*command*]

If no argument is specified, print a brief summary of all the commands. If a command name is specified, print more detailed information about the use of that command.

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 command -num from the most current command
!string [modifier]	Execute the most recent command starting with string
!?string? [modifier]	Execute the most recent command containing string
^	Quick history command substitution ^old^new^<modifier> this is equivalent to !:s/old/new/

The history modifiers may be**:**

`:s/`*`old`*`/`*`new`*`/`	Substitute the value *new* for the value *old*.
`:p`	Print but do not execute the command.

The command `pgienv history off` tells the debugger not to display the history record number. The command `pgienv history on` tells the debugger to display the history record number.

language

Print the name of the language of the current file.

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.

nop[rint] *exp*

Evaluate the expression but do not print the result.

pgienv [*command*]

Define the debugger environment. With no arguments, display the debugger settings.

help *pgienv*	Provide help on pgienv
[*pgi*]env	Define the debugger environment
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 command
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
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 OpenMP debug support.
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 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	For each process, debugger sets thread stopping mode to 'sync' in serial

auto	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 immediatly; no wait for running threads
pgienv threadwait any	prompt available when at least a single thread stops
pgienv threadwait all	prompt available only after all threads have stopped
pgienv procwait none	prompt available immediately; no 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
pgienv verbose <*bitmask*>	Choose which debug status messages to report. Accepts an integer valued bit mask of the following values: o 0x1 - Standard messaging (default). Report status information on current process/thread only. o 0x2 - Thread messaging. Report status information on all threads of (current) processes. o 0x4 - Process messaging. Report status information on all processes. o 0x8 - SMP messaging (default). Report SMP events. o 0x16 - Parallel messaging (default). Report parallel events. o 0x32 - Symbolic debug information. Report any errors encountered while processing symbolic debug information (e.g. STABS, DWARF). Pass 0x0 to disable all messages.

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. With the num argument, re-execute the command number *num*, or with -*num*, the last *num* commands. With the first and last arguments, re-execute commands number *first* to *last* (optionally *n* times).

scr[ipt] *filename*

Open the indicated file and execute the contents as though they were entered as commands. If you use ~ before the filename, this is expanded to the value of `$HOME`.

setenv *name*
setenv *name value*

Print value of environment variable *name*. With a specified *value*, set name to *value*.

shell [*arg0*, *arg1,... argn*]

Fork a shell (defined by `$SHELL`) and give it the indicated arguments (the default shell is `sh`). If no arguments are specified, an interactive shell is invoked, and executes until a "^D" is entered.

sle[ep] [*time*]

Pause for *time* seconds or one second if no time is specified.

sou[rce] *filename*

Open the indicated file and execute the contents as though they were entered as commands. If you use ~ before the filename, this is expanded to the value of `$HOME`.

unal[ias] *name*

Remove the alias definition for name, if one exists.

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 ~, it will be substituted by `$HOME`.

1.10 Commands Summary

This section contains a brief summary of the *PGDBG* debugger commands. For more detailed information on a command, select the hyperlink under the selection category.

Table 1-2: Debugger Commands

addr	next	undefset
assign	nexti	undisplay
catch	pgienv	up
cd	pwd	use
clear	proc [number]	viewset
debug	procs	wait
defset	repeat	watchi
delete	set	whatis
display	setenv	when
down	shell	wheni
dump	sizeof	where
edit	source	whereis
entry	step	which
file	stepi	whichsets
files	stop	/
focus	sync	?
halt	synci	!
history	stopi	^
ignore	thread	
language	threads	
lines	trace	
list	tracei	

1.10.1 Command Summary

For more detailed information on a command, select the hyperlink under the Section category.

Table 1-3: *PGDBG* Commands

Name	Arguments	Section
arri[ve]		1.9.1.4 Program Locations
ad[dr]	[*n* | *line* | *func* | *var* | *arg*]	1.9.1.10 Conversions
al[ias]	[*name* [*string*]]	1.9.1.11 Miscellaneous
asc[ii]	*exp* [,...*exp*]	1.9.1.5 Printing and Setting Variables
as[sign]	*var=exp*	1.9.1.6 Symbols and Expressions
bin	*exp* [,...*exp*]	1.9.1.5 Printing and Setting Variables
b[reak]	[*line* | *func*] [if (*condition*)] [do {*commands*}]	1.9.1.3 Events
breaki	[*addr* | *func*] [if (*condition*)] [do {*commands*}]	1.9.1.3 Events
breaks		1.9.1.3 Events
call	*func* [(*exp*,...)]	1.9.1.6 Symbols and Expressions
catch	[*number* [,*number...*]]	1.9.1.3 Events
cd	[*dir*]	1.9.1.4 Program Locations
clear	[all | *func* | *line* | addr {*addr*}]	1.9.1.3 Events
c[ont]		1.9.1.1 Process Control

Name	Arguments	Section
cr[ead]	*addr*	1.9.1.9 Memory Access
de[bug]		1.9.1.1 Process Control
dec	*exp* [,...*exp*]	1.9.1.5 Printing and Setting Variables
decl[aration]	*name*	1.9.1.6 Symbols and Expressions
decls	[*func* | "*sourcefile*" | {global}]	1.9.1.7 Scope
del[ete]	*event-number* | all | 0 | *event-number* [,.*event-number.*]	1.9.1.3 Events
dir[ectory]	[*pathname*]	1.9.1.11 Miscellaneous
dis[asm]	[*count* | *lo*:*hi* | *func* | *addr*, *count*]	1.9.1.4 Program Locations
disab[le]	*event-number* | all	1.9.1.3 Events
display	*exp* [,...*exp*]	1.9.1.5 Printing and Setting Variables
do	{*commands*} [at *line* | in *func*] [if (*condition*)]	1.9.1.3 Events
doi	{*commands*} [at *addr* | in *func*] [if (*condition*)]	1.9.1.3 Events
down		1.9.1.7 Scope
defset	*name* [p/t-set]	1.9.1.2 Process-Thread Sets
dr[ead]	*addr*	1.9.1.9 Memory Access
du[mp]	*address*, *count*, "format-string"	1.9.1.9 Memory Access
edit	[*filename* | *func*]	1.9.1.4 Program Locations

Name	Arguments	Section
enab[le]	*event-number* | all	1.9.1.3 Events
en[ter]	*func* | "*sourcefile*" | {global}	1.9.1.7 Scope
entr[y]	*func*	1.9.1.6 Symbols and Expressions
fil[e]		1.9.1.4 Program Locations
files		1.9.1.7 Scope
focus	[p/t-set]	1.9.1.2 Process-Thread Sets
fp		1.9.1.8 Register Access
fr[ead]	*addr*	1.9.1.9 Memory Access
func[tion]	[*addr* | *line*]	1.9.1.10 Conversions
glob[al]		1.15.10.3 Global Commands
halt	[*command*]	1.9.1.1 Process Control
he[lp]		1.9.1.11 Miscellaneous
hex	*exp* [,...*exp*]	1.9.1.5 Printing and Setting Variables
hi[story]	[*num*]	1.9.1.11 Miscellaneous
hwatch	*addr* [if (*condition*)] [do {*commands*}]	1.9.1.3 Events
hwatchb[oth]	*addr* [if (*condition*)] [do {*commands*}]	1.9.1.3 Events
hwatchr[ead]	*addr* [if (*condition*)] [do {*commands*}]	1.9.1.3 Events
ignore	[*number* [,*number*...]]	1.9.1.3 Events
ir[ead]	*addr*	1.9.1.9 Memory Access

Name	Arguments	Section
language		1.9.1.11 Miscellaneous
lin[e]	[*n* | *func* | *addr*]	1.9.1.10 Conversions
lines	*function*	1.9.1.4 Program Locations
lis[t]	[*count* | *line,count* | *lo*:*hi* | *function*]	1.9.1.4 Program Locations
log	*filename*	1.9.1.11 Miscellaneous
lv[al]	*exp*	1.9.1.6 Symbols and Expressions
names	[*func* | "*sourcefile*" | {global}]	1.9.1.7 Scope
n[ext]	[*count*]	1.9.1.1 Process Control
nexti	[*count*]	1.9.1.1 Process Control
nop[rint]	*exp*	1.9.1.11 Miscellaneous
oct	*exp* [,...*exp*]	1.9.1.5 Printing and Setting Variables
pc		1.9.1.8 Register Access
pgienv	[*command*]	1.9.1.11 Miscellaneous
p[rint]	*exp1* [,...*expn*]	1.9.1.5 Printing and Setting Variables
printf	"*format_string*", *expr*,...*expr*	1.9.1.5 Printing and Setting Variables
proc	[*number*]	1.9.1.1 Process Control
procs		1.9.1.1 Process Control
pwd		1.9.1.4 Program Locations

<table>
<tr><th>Name</th><th>Arguments</th><th>Section</th></tr>
<tr><td>q[uit]</td><td></td><td>1.9.1.1 Process Control</td></tr>
<tr><td>regs</td><td></td><td>1.9.1.8 Register Access</td></tr>
<tr><td>rep[eat]</td><td>[*first*, *last*] | [*first*: *last*:*n*] | [*num*] | [-*num*]</td><td>1.9.1.11 Miscellaneous</td></tr>
<tr><td>rer[un]</td><td>[*arg0 arg1* ... *argn*] [< *inputfile*] [> *outputfile*]</td><td>1.9.1.1 Process Control</td></tr>
<tr><td>ret[addr]</td><td></td><td>1.9.1.8 Register Access</td></tr>
<tr><td>ru[n]</td><td>[*arg0 arg1* ... *argn*] [< *inputfile*] [> *outputfile*]</td><td>1.9.1.1 Process Control</td></tr>
<tr><td>rv[al]</td><td>*expr*</td><td>1.9.1.6 Symbols and Expressions</td></tr>
<tr><td>sco[pe]</td><td></td><td>1.9.1.7 Scope</td></tr>
<tr><td>scr[ipt]</td><td>*filename*</td><td>1.9.1.11 Miscellaneous</td></tr>
<tr><td>set</td><td>*var* = *ep*</td><td>1.9.1.6 Symbols and Expressions</td></tr>
<tr><td>setenv</td><td>*name* | *name value*</td><td>1.9.1.11 Miscellaneous</td></tr>
<tr><td>sh[ell]</td><td>*arg0* [... *argn*]</td><td>1.9.1.11 Miscellaneous</td></tr>
<tr><td>siz[eof]</td><td>*name*</td><td>1.9.1.6 Symbols and Expressions</td></tr>
<tr><td>sle[ep]</td><td>*time*</td><td>1.9.1.11 Miscellaneous</td></tr>
<tr><td>source</td><td>*filename*</td><td>1.9.1.11 Miscellaneous</td></tr>
<tr><td>sp</td><td></td><td>1.9.1.8 Register Access</td></tr>
<tr><td>sr[ead]</td><td>*addr*</td><td>1.9.1.9 Memory Access</td></tr>
<tr><td>stackd[ump]</td><td>[*count*]</td><td>1.9.1.4 Program Locations</td></tr>
</table>

Name	Arguments	Section
stack[trace]	[*count*]	1.9.1.4 Program Locations
stat[us]		1.9.1.3 Events
s[tep]	[*count*] [up]	1.9.1.1 Process Control
stepi	[*count*] [up]	1.9.1.1 Process Control
stepo[ut]		1.9.1.1 Process Control
stop	[at *line* | in *func*] [*var*] [if (*condition*)] [do {*commands*}]	1.9.1.3 Events
stopi	[at *addr* | in *func*] [*var*] [if (*condition*)] [do {*commands*}]	1.9.1.3 Events
sync	[*func* | *line*]	1.9.1.1 Process Control
synci	[*func* | *addr*]	1.9.1.1 Process Control
str[ing]	*exp* [,...*exp*]	1.9.1.5 Printing and Setting Variables
thread	*number*	1.9.1.1 Process Control
threads		1.9.1.1 Process Control
track	*expression* [at *line* | in *func*] [if (*condition*)] [do {*commands*}]	1.9.1.3 Events
tracki	*expression* [at *addr* | in *func*] [if (*condition*)] [do {*commands*}]	1.9.1.3 Events
trace	[at *line* | in *func*] [*var*| *func*] [if (*condition*)] do {*commands*}	1.9.1.3 Events
tracei	[at *addr* | in *func*] [*var*] [if (*condition*)] do {*commands*}	1.9.1.3 Events

Name	Arguments	Section
type	*expr*	1.9.1.6 Symbols and Expressions
unal[ias]	*name*	1.9.1.11 Miscellaneous
undefset	[*name* | *-all*]	1.9.1.2 Process-Thread Sets
undisplay	[all | 0 | *exp*]	1.9.1.5 Printing and Setting Variables
unb[reak]	*line* | *func* | all	1.9.1.3 Events
unbreaki	*addr* | *func* | all	1.9.1.3 Events
up		1.9.1.7 Scope
use	[*dir*]	1.9.1.11 Miscellaneous
viewset	*name*	1.9.1.2 Process-Thread Sets
wait	[any | all | none]	1.9.1.1 Process Control
wa[tch]	*expression* [at *line* | in *func*] [if (*condition*)] [do {*commands*}]	1.9.1.3 Events
watchi	*expression* [at *addr* | in *func*] [if(*condition*)] [do {*commands*}]	1.9.1.3 Events
whatis	[*name*]	1.9.1.6 Symbols and Expressions
when	[at *line* | in *func*] [if (*condition*)] do {*commands*}	1.9.1.3 Events
wheni	[at *addr* | in *func*] [if(*condition*)] do {*commands*}	1.9.1.3 Events
w[here]	[*count*]	1.9.1.4 Program Locations
whereis	*name*	1.9.1.7 Scope

Name	Arguments	Section
whichsets	[p/t-set]	1.9.1.2 Process-Thread Sets
which	*name*	1.9.1.7 Scope
/	/ [*string*] /	1.9.1.4 Program Locations
?	**?**[*string*] **?**	1.9.1.4 Program Locations
!	History modification	1.9.1.11 Miscellaneous
^	History modification	1.9.1.11 Miscellaneous

1.11 Register Symbols

This section describes the register symbols defined for X86 processors, and AMD64 processors operating in compatibility or legacy mode.

1.11.1 X86 Register Symbols

This section describes the X86 register symbols.

Table 1-4: General Registers

Name	Type	Description
`$edi`	`unsigned`	General purpose
`$esi`	`unsigned`	General purpose
`$eax`	`unsigned`	General purpose
`$ebx`	`unsigned`	General purpose
`$ecx`	`unsigned`	General purpose
`$edx`	`unsigned`	General purpose

Table 1-5: Floating-Point Registers

Name	Type	Description
`$d0 - $d7`	`64-bit IEEE`	Floating-point

Table 1-6: Segment Registers

Name	Type	Description
`$gs`	`16-bit unsigned`	Segment register
`$fs`	`16-bit unsigned`	Segment register
`$es`	`16-bit unsigned`	Segment register
`$ds`	`16-bit unsigned`	Segment register
`$ss`	`16-bit unsigned`	Segment register
`$cs`	`16-bit unsigned`	Segment register

Table 1-7: Special Purpose Registers

Name	Type	Description
`$ebp`	`32-bit unsigned`	Frame pointer
`$efl`	`32-bit unsigned`	Flags register
`$eip`	`32-bit unsigned`	Instruction pointer
`$esp`	`32-bit unsigned`	Privileged-mode stack pointer
`$uesp`	`32-bit unsigned`	User-mode stack pointer

1.11.2 AMD64 Register Symbols

This section describes the register symbols defined for AMD64 processors operating in 64-bit mode.

Table 1-8: General Registers

Name	Type	Description
`$r8 - $r15`	`64-bit unsigned`	General purpose
`$rdi`	`64-bit unsigned`	General purpose
`$rsi`	`64-bit unsigned`	General purpose
`$rax`	`64-bit unsigned`	General purpose
`$rbx`	`64-bit unsigned`	General purpose
`$rcx`	`64-bit unsigned`	General purpose
`$rdx`	`64-bit unsigned`	General purpose

Table 1-9: Floating-Point Registers

Name	Type	Description
`$d0 - $d7`	`64-bit IEEE`	Floating-point

Table 1-10: Segment Registers

Name	Type	Description
$gs	16-bit unsigned	Segment register
$fs	16-bit unsigned	Segment register
$es	16-bit unsigned	Segment register
$ds	16-bit unsigned	Segment register
$ss	16-bit unsigned	Segment register
$cs	16-bit unsigned	Segment register

Table 1-11: Special Purpose Registers

Name	Type	Description
$ebp	64-bit unsigned	Frame pointer
$rip	64-bit unsigned	Instruction pointer
$rsp	64-bit unsigned	Stack pointer
$eflags	64-bit unsigned	Flags register

Table 1-12: SSE Registers

Name	Type	Description
$mxcsr and status register	64-bit unsigned	SIMD floating-point control
$xmm0 - $xmm15	Packed 4x32-bit IEEE	SSE floating-point registers

1.11.3 SSE Register Symbols

On AMD64, Pentium III, and compatible processors, an additional set of SSE (streaming SIMD enhancements) registers and a SIMD floating-point control and status register are available.

Each SSE register contains four IEEE 745 compliant 32-bit single-precision floating-point values. The *PGDBG* regs command reports these values individually in both hexadecimal and floating-point format. PGDBG provides syntax to refer to these values individually, as members of a range,

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:

```
Bits:   127        96 95        65  63         32 31          0
```

$xmm0(3)	$xmm0(2)	$xmm0(1)	$xmm0(0)
$xmm1(3)	$xmm1(2)	$xmm1(1)	$xmm1(0)
$xmm7(3)	$xmm7(2)	$xmm7(1)	$xmm7(0)

To access a `$xmm0(3)`, the 32-bit single-precision floating point value that occupies bits 96 – 127 of SSE register 0, use the following *PGDBG* command:

```
pgdbg> print $xmm0(3)
```

To set `$xmm2(0)` to the value of `$xmm3(2)`, use the following *PGDBG* command:

```
pgdbg> set $xmm2(3) = $xmm3(2)
```

You can also subscript SSE registers with range expressions to specify runs of consecutive component values, and access an SSE register as a whole. For example, the following are legal *PGDBG* commands:

```
pgdbg> set $xmm0(0:1) = $xmm1(2:3)
pgdbg> set $xmm6 = 1.0/3.0
```

The first command above initializes elements 0 and 1 of `$xmm0` to the values in elements 2 and 3 respectively in `$xmm1`. The second command above initializes all four elements of `$xmm6` to the constant `1.0/3.0` evaluated as a 32-bit floating-point constant.

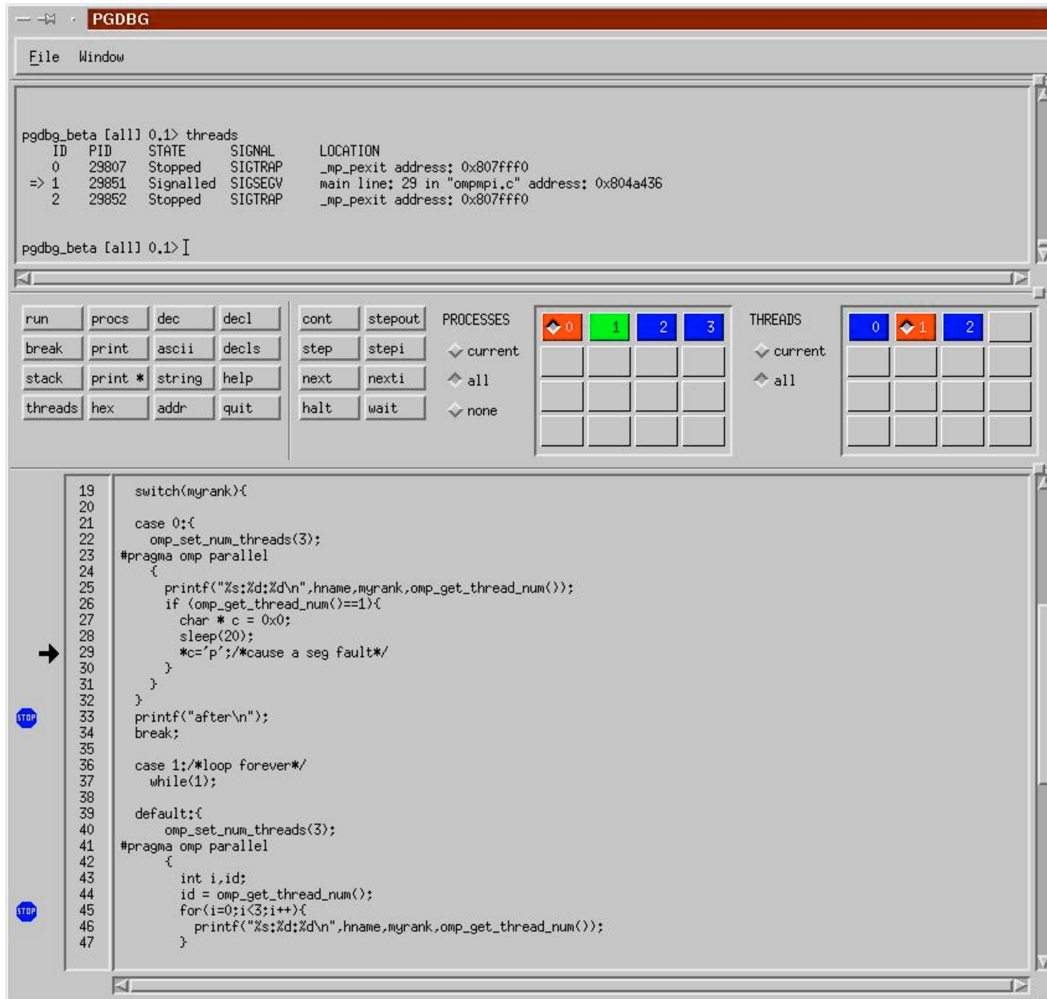
1.12 X-Windows Graphical User Interface

The *PGDBG* X-Windows Graphical User Interface (GUI) is invoked on UNIX systems by default using the command `pgdbg`. The GUI runs as a separate process and communicates with `pgdbg`. There may be minor variations in the GUI from host to host, depending on the type of monitor available, the settings for various defaults and the window manager used. The basic interface across all systems remains the same with the exception of the differences tied to the display characteristics and the window manager used.

1.12.1 Main Window

Figure 1-1 shows the main window of `pgdbg`. This window appears when `pgdbg` starts and remains throughout the debug session. If the debugger is licensed as a component of the CDK (PGI Cluster Development Kit), a process grid is employed by the GUI, otherwise only a thread grid will appear.

Figure 1-1: PGDBG Debugger Main Window



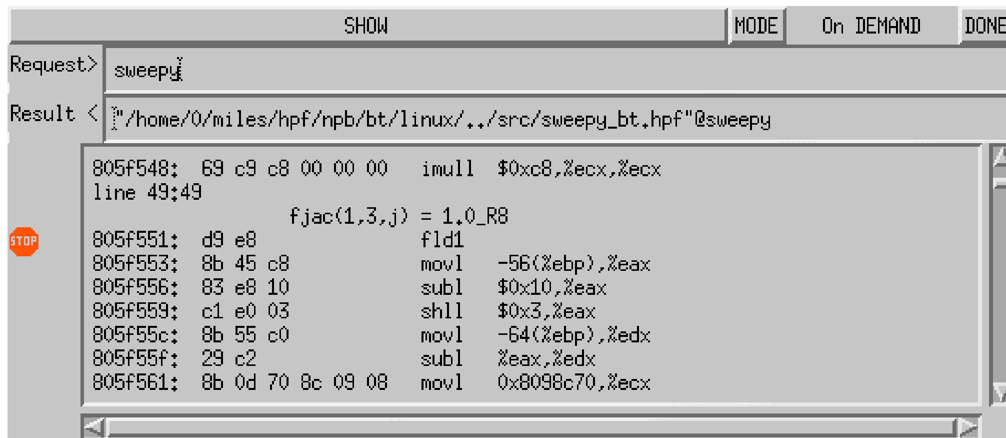
The components of the main window are:

- *Dialog Window* – This window supports a dialog with the debugger. Commands entered in this window are executed, and the results are displayed.
- *Button Panel* – This window displays buttons that can be clicked with the mouse as an alternative to typing commands. Many buttons, such as the `print` button, will pass selected text to the debugger as an argument to the command.
- *Source Window* – This window displays the source code for the current location. The current location is marked by a footprint icon. Breakpoints may be set at any source line by clicking the left mouse button in the margin to the left of the source line. The breakpoints are marked by stop sign icons. An existing breakpoint may be cleared by clicking the left mouse button on the stop sign icon.
- *Process/Threads Grid* – The *PGDBG* GUI lists all active processes in a process grid. Each element of the process grid is labeled with a process ID and represents a single process. Each element is a button that can be pushed to select a particular process as the current process. A diamond indicates the current process. The thread grid depicts the threads of the current process. When the current process is changed, the thread grid is refreshed to describe the threads of the (new) current process, and the current thread is set to be the current thread of that process. *PGDBG* displays the program context of the current thread (source position, registers, disassembly, etc.).

1.12.2 Disassembly Window

The figure below shows the disassembly window of `pgdbg`. It is useful for debugging code at the assembly code level. It is invoked by selecting `DISASM` in the main window's `Window` pulldown menu. By default, the current function is disassembled, and the current position is always displayed and marked with the footprint icon.

Figure 1-2: Disassembly Window



The components of the disassembly window are:

- *Control Panel* – The items in the control panel allow selection of what memory is to be disassembled, and whether the display is to be updated automatically or on demand. A region other than the current function can be displayed by placing a function name or address range in the request field and clicking the `SHOW` button. The mode selection controls whether the display is updated for each new location, or whether the display is only updated on demand.
- *Disassembled Memory* – This window displays a range of memory in disassembly format. Each instruction is preceded by its address. Breakpoints may be set at any instruction by clicking the left mouse button in the margin to the left of the instruction. If a function is being disassembled, the source code for the function is interleaved with the disassembled instructions.

1.12.3 Register Window

The register window displays the value of the processor's registers. It is invoked by selecting `REGISTER` in the main window's `Window` pulldown menu.

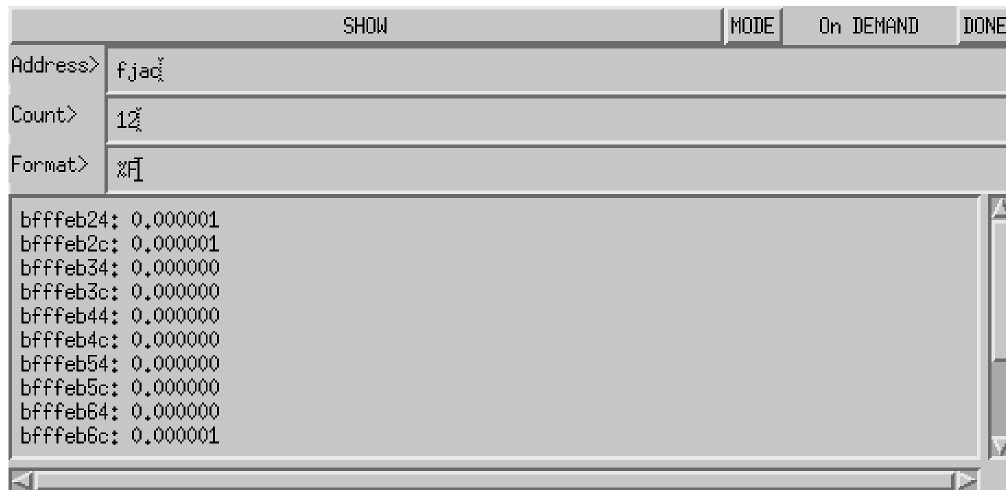
Figure 1-3: Register Window

- *Control Panel* – The display is updated each time the `SHOW` button is clicked. The mode item controls whether the display is updated automatically for each new location or on demand. The default is on demand.

1.12.4 Memory Window

The memory window displays a region of memory in a *printf*-like format descriptor. It is essentially a graphical interface to the `pgdbg` *dump* command (see Section *1.14.3 Graphical Features*). It is invoked by selecting `MEMORY` in the main window's `Window` pull-down menu.

Figure 1-4: Memory Window

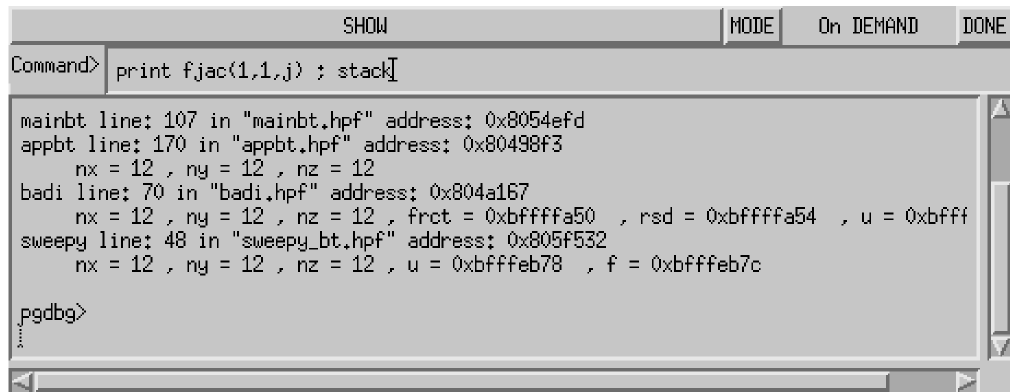


- *Control Panel* – The `Address`, `Count`, and `Format` fields correspond to the three arguments to the `pgdbg` *dump* command. They specify the start address, repeat count, and dump format. The display is updated each time the `SHOW` button is clicked. The mode item controls whether the display is updated automatically for each new location or on demand. The default is on demand.
- *Memory Display* – The display is simply the result of the `pgdbg` *dump* command. Starting at `Address`, `Count` data items are read from memory according to the descriptor in the `Format` field. Each line of output begins with the address being displayed in hexadecimal format. This window is scrollable so that large regions of memory may be examined.

1.12.5 Custom Window

The custom window is useful for repeatedly executing a sequence of debugger commands. The commands, entered in the control panel, can be executed at the click of a button or automatically at each new location.

Figure 1-5: Custom Window



- *Control Panel* – Each time the `SHOW` button is clicked, the text string in the command field is executed and the results are displayed. If the `MODE` is set to "on new location", then the command is executed at each new location.
- *Results Display* – This display shows the results of the executed command. The display is cleared prior to executing the command field. This window is scrollable so that all the results can be viewed.

1.12.5.1 Setting the Font

Use the ***xlsfonts*** command to list all fonts installed on your system, then choose one you like. For this example, we choose a *sony* font that is completely specified by the following string:

-sony-fixed-medium-r-normal--24-230-75-75-c-120-iso8859-1

There are two ways to set the font that your *PGDBG* GUI uses.

1. Use your .Xresources file:

Xpgdbg*font : <chosen font>
pgdbg*font : <chosen font>

For example:

pgdbg*font : -sony-fixed-medium-r-normal--24-230-75-75-c-120-iso8859-1

You will have to merge these changes into your X environment for them to take effect. You can use the following command:

% xrdb -merge $HOME/.Xresources

2. Use the command line options : ***-fn <font>***. For example:

% pgdbg -fn -sony-fixed-medium-r-normal--0-0-100-100-c-0-jisx0201.1976-0...

1.13 *PGDBG*: Parallel Debug Capabilities

This section describes the parallel debug capabilities of *PGDBG*. *PGDBG* is a distributed SMP debugger is capable of debugging parallel-distributed MPI programs, thread-parallel SMP OpenMP (and Linuxthread) programs, and hybrid distributed SMP programs.

See http://www.pgroup.com/docs.htm for the most recent documentation. This material is also available in $PGI/docs/index.htm. See http://www.pgroup.com/faq/index.htm for an online FAQ.

1.13.1 OpenMP and Linuxthread Support

- Automatic thread detection and attach
- Full thread control in parallel regions
- Thread grouping
- Threads presented by their OpenMP logical thread number
- Line level debugging preserved when thread
    - Enters a parallel region
    - Enters a serial region
    - Hits an OpenMP barrier
    - Hits an OpenMP synchronize statement
    - Enters an OpenMP sections program section
- Informative messages regarding thread state and location

1.13.2 MPI Support

- Automatic process detection and attach
- Informative messages regarding process state and location
- Process grouping
- Processes presented by their global rank in COMMWORLD

1.13.3 Process & Thread Control

- Concise control of groups of processes/threads
- Thread and process synchronization
- Configurable thread and process stop mode
- Configurable wait mode
- Serial, process-only, threads-only, and multilevel debug modes

1.13.4 Graphical Presentation of Threads and Processes

- Process grid
- Thread grid
- Graphical grouping logic
- Color depiction of whole program execution state
- New window dedicated to printing program output, and accepting program input
- Thread sub-window. Lists each thread by its logical CPU ID. Displays for each thread its state and stop location. Threads are grouped by parent process.
- Program I/O sub-window. Pops up automatically when program prints to stdout. The program I/O sub-window can also be raised from the Window menu.
- Output written to *stdout* by the process being debugged is no longer block buffered.
- Process grid. Displays each process as a color-coded button in a grid. Click on a grid element to refresh the GUI in the scope of that process. Each grid element is numbered with the process's logical ID.
- Process grouping. Control processes in groups

- Thread grid. Displays each thread as a color-coded button in a grid. Click on a grid element to refresh the GUI in the scope of that thread. Each grid element is numbered with the thread's logical CPU ID.

1.14 Debugging Parallel Programs with PGDBG

This section describes how to invoke the debugger for thread-parallel (SMP) debugging and for process-parallel (MPI) debugging. It provides some important definitions and background information on how *PGDBG* represents processes and threads.

1.14.1 Processes and Threads

An active process is made up of one or more active threads of execution. In the context of a process-parallel program, a process is an MPI process composed of one thread of execution. In the context of a thread-parallel program, a thread is an OpenMP or Linux Pthread SMP thread. *PGDBG* is capable of debugging hybrid process-parallel/thread-parallel programs where the program employs multiple SMP processes.

PGDBG assigns a thread ID to each thread. *PGDBG* uses a thread's OpenMP ID when debugging an OpenMP program; zero based, incrementing in order of thread creation otherwise. Thread ID's are unique within the context of a single process only.

PGDBG assigns a process ID to each process. *PGDBG* uses a process' MPI rank (in communicator COMMWORLD) when debugging an MPI program; zero based, incrementing in order of process creation otherwise. Process ID's are unique across all active processes.

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 having thread ID 4 and the parent process having process ID 1.

An OpenMP program (thread-parallel only) 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. See Section *1.15.2 Threads-only debugging.*

An MPI program (non-SMP) 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. See Section *1.15.3 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. See Section *1.15.4 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 unnecessary but optional.

1.14.2 Thread-Parallel Debugging

PGDBG automatically attaches to new threads as they are created during program execution. *PGDBG* describes when a new thread is created; the thread ID of each new thread is printed.

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

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

During a debug session, at any one time, *PGDBG* operates in the context of a single thread, the current thread. The current thread is chosen by using the *thread* command when the debugger is operating in text mode (invoked with the `-text` option), or by clicking in the thread grid when the GUI interface is in use (the default). See Section *1.15.10.2 Thread Level Commands.*

The *threads* command lists all threads currently employed by an active program. The *threads* command displays for each thread its unique thread ID, system ID (Linux process ID), execution state (running, stopped, signaled, exited, or killed), signal information and reason for stopping, and the current location (if stopped or signaled). The arrow indicates the current thread. The process ID of the parent is printed in the top left corner. The *thread* command changes the current thread.

```
pgdbg [all] 2> thread 3

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
```

1.14.2.1 Invoking PGDBG: OpenMP, Linux Pthread Debugging

Use the following to invoke *PGDBG*, OpenMP, Linux Pthread debugging using text or GUI mode:

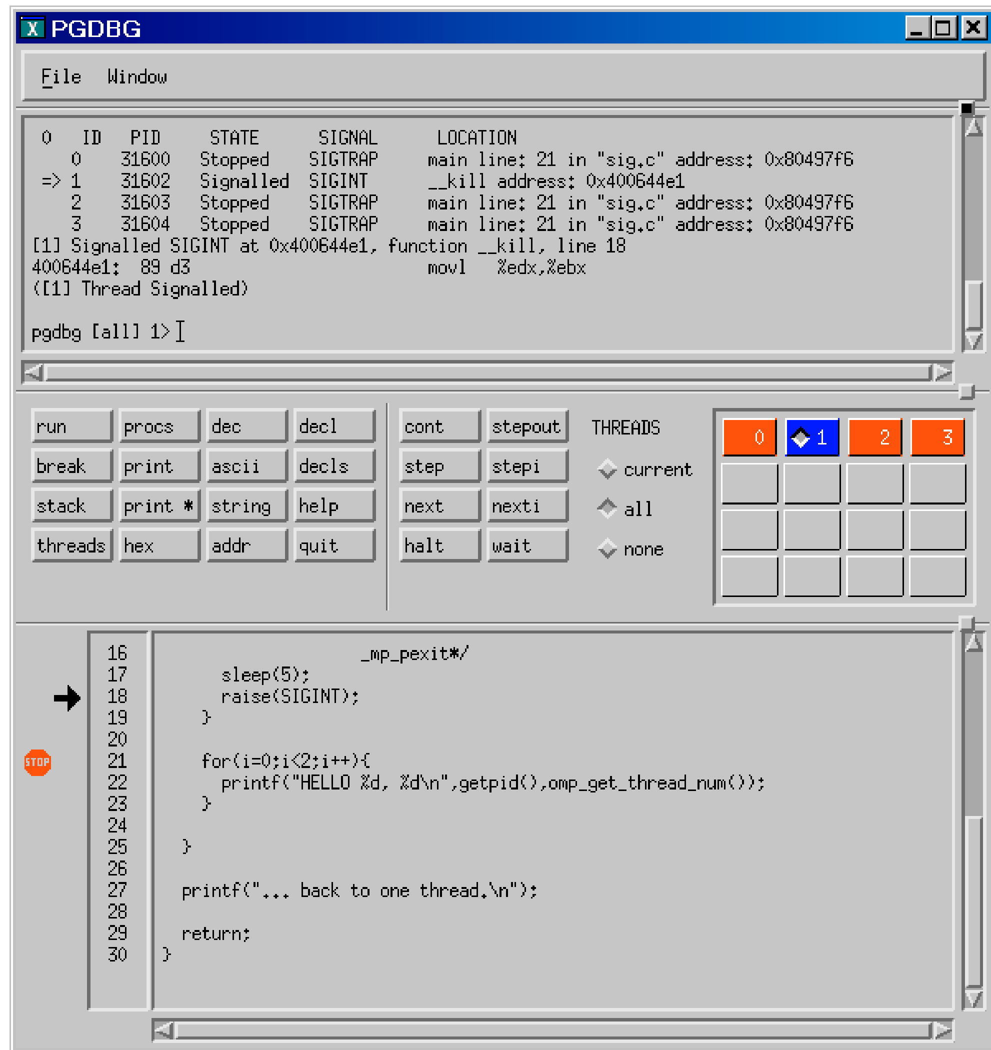
```
GUI mode:

    %pgdbg <executable> <args>,...<args>


TEXT mode:

    %pgdbg -text <executable> <args>,...<args>
```

See the online tutorial at http://www.pgroup.com/doc/index.htm to get started.

1.14.3 Graphical Features

The *PGDBG* Graphical User Interface (GUI) lists all active threads in a thread grid. Each element of the thread grid is labeled with a thread ID and represents a single thread. Each element is a button that can be pushed to select a particular thread as the current thread. The *PGDBG* GUI displays the program context of the current thread.

Figure 1-6: *PGDBG* GUI Interface: PGI Workstation



Each button in the thread grid is color coded to depict the execution state of the underlying thread.

Table 1-13: Thread State is Described using Color

Option	Description
Stopped	Red
Signaled	Blue
Running	Green
Exited	Black
Killed	Black

1.14.4 Process-Parallel Debugging

PGDBG automatically attaches to new MPI processes as they are created by a running MPI program. *PGDBG* must be invoked via the MPIRUN script. Use the MPIRUN *-dbg* option to specify which debugger to use. To choose *PGDBG*, use *-dbg=pgdbg* before the executable name (this is not a program argument). *PGDBG* must be installed on your system and your PGI environment variable set appropriately, and added to your PATH.

PGDBG displays an informational message as it attaches to the freshly created processes.

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

The MPI global rank is printed with the message. Use the *procs* command to list the host and the PID of each process by rank. The current process is marked with an arrow. To change the current process by process ID, use the *proc* command.

```
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
```

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

The prompt displays the current process and the current thread. The current process above has been changed to process 1, and the current thread of process 1 is 0. This is written as 1.0. See Section *1.15.15 The PGDBG Command Prompt* for a complete description of the prompt format.

The following rules apply during a *PGDBG* debug session:

- At any one time, *PGDBG* operates in the context of a single process, the current process.
- Each active process has a thread set of size >=1.
- The current thread is a member of the thread set of the current process.

A license file distributed with *PGDBG* that restricts *PGDBG* to debugging a total of 64 threads. *Workstation* and *CDK* license files may further restrict the number of threads that *PGDBG* is eligible to debug. *PGDBG* will use the *Workstation* or *CDK* license files to determine the number of threads it is able to debug.

With its 64 thread limit, PGDBG is capable of debugging a 16 node cluster with 4 CPUs on each node or a 32 node cluster with 2 CPUs on each node or any combination of threads that add up to 64.

Use the *proc* command to change the current process. Those *PGDBG* commands that refer to program scope execute off of the current scope of the current thread by default. The current thread must be stopped in order to read from its memory space. See Section *1.15.10.2 Thread Level Commands* for a description and list of these context sensitive commands.

To list all active processes, use the *procs* command. The *procs* command lists all active processes by process ID (MPI rank where applicable). Listed for each process: the system ID of the initial thread, process execution state, number of active threads, and host name. The initial process is run locally; ‘local’ describes the host the debugger is running on. The execution state of a process is described in terms of the execution state of its component threads:

Table 1-14: Process state is described using color

Process state	Description	Color
Stopped	If all threads are stopped at breakpoints, or where directed to stop by PGDBG	Red
Signaled	If at least one thread is stopped on an interesting signal (as described by *catch*)	Blue
Running	If at least one thread is running	Green
Exited or Killed	If all threads have been killed or exited	Black

1.14.4.1 Invoking *PGDBG*: MPI Debugging

To debug an MPI program, *PGDBG* is invoked via MPIRUN. MPIRUN sets a breakpoint at `main` and starts the program running under the control of *PGDBG*. When the initial process hits `main` no other MPI processes are active. The non-initial MPI processes are created when the process calls `MPI_Init`.

A Fortran MPI program stops at `main` initially instead of `MAIN`. You must *step* into `MAIN`.

GUI mode:

```
%mpirun -np 4 -dbg=pgdbg <executable> <args>,...<args>
```

TEXT mode:

```
%unsetenv DISPLAY
%mpirun -np 4 -dbg=pgdbg <executable> <args>,...<args>
```

An MPI debug session starts with the initial process stopped at `main`. Set a breakpoint at a program location after the return of `MPI_Init` to stop all processes there. If debugging Fortran, *step* into the `MAIN` program.

See the online tutorial at http://www.pgroup.com/doc/index.htm to get started.

1.14.4.2 MPI-CH Support

PGDBG supports redirecting `stdin`, `stdout`, and `stderr` with the following MPI-CH switches:

Table 1-15: MPI-CH Support

Command	Output
`-stdout  <file>`	Redirect standard output to `<file>`
`-stdin   <file>`	Redirect standard input from `<file>`
`-stederr <file>`	Redirect standard error to `<file>`

PGDBG also provides support for the following MPI-CH switches:

Command	Output
`-nolocal`	PGDBG runs locally, but no MPI processes run locally
`-all-local`	PGDBG runs locally, all MPI processes run locally

If you are using your own version of MPI-CH, see our online FAQ for how to integrate the MPIRUN scripts with PGDBG.

When *PGDBG* is invoked via MPIRUN the following PGDBG command line arguments are not accessible. A possible workaround is listed for each.

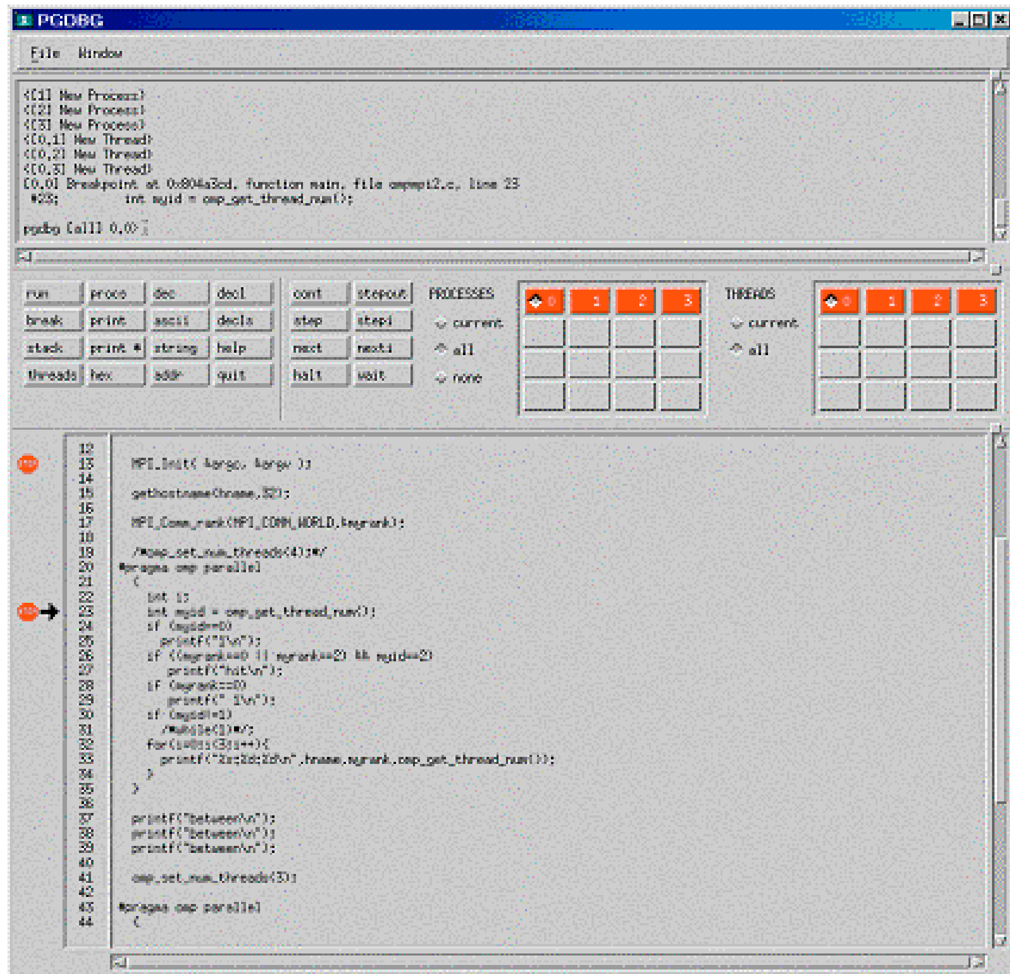
Argument	Workaround
`-dbx`	Include `'pgienv dbx on'` in *.pgdbgrc* file
`-s startup`	Use *.pgdbgrc* default script file and the *script* command.
`-c "command"`	Use *.pgdbgrc* default script file and the *script* command.

-text	Clear your DISPLAY environment variable before invoking MPIRUN
-t <target>	Add to the beginning of the PATH environment variable a path to the appropriate *PGDBG*.

1.14.4.3 Graphical Features

The *PGDBG* GUI lists all active processes in a process grid. Each element of the process grid is labeled with a process ID and represents a single process. Each element is a button that can be pushed to select a particular process as the current process. A diamond indicates the current process. The thread grid depicts the threads of the current process. When the current process is changed, the thread grid is refreshed to describe the threads of the (new) current process, and the current thread is set to be the current thread of that process. *PGDBG* displays the program context of the current thread (source position, registers, disassembly, etc.).

Figure 1-7: *PGDBG* GUI Interface: Cluster Development Kit



1.15 Thread-parallel and Process-parallel Debugging

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

1.15.1 PGDBG Debug Modes and Process/Thread Identifiers

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

Table 1-16: The *PGDBG* 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

PGDBG starts out 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.

If *PGDBG* is licensed as a *Workstation* product, it operates in Threads-only mode by default (however multilevel notation is always valid).

If PGDBG is licensed as a *CDK* product, it operates in process-only mode by default.

The *PGDBG* prompt displays the current thread according to the current debug mode. See *1.15.15 The PGDBG Command Prompt* for a description of the *PGDBG* prompt.

The *pgienv* command is used to change debug modes manually.

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

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

1.15.2 Threads-only debugging

Enter threads-only mode to debug a program with a single SMP process. As a convenience the process ID portion can be omitted. PGDBG automatically enters threads-only debug mode from serial debug mode when it attaches to SMP threads.

Example 1-1: Thread IDs in threads-only debug mode

`1`	Thread 1 of all processes (*.1)
`*`	All threads of all processes (*. *)
`0.7`	Thread 7 of process 0 (Multilevel thread names valid in threads-only debug mode)

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

1.15.3 Process-only debugging

Enter process-only mode to debug a program with non-SMP nodes. As a convenience, the thread ID portion can be omitted. *PGDBG* automatically enters process-only debug mode from serial debug mode when the target program returns from `MPI_Init`.

Example 1-2: 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 thread names are valid in this mode)

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

on context.

1.15.4 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. *PGDBG* changes automatically to multilevel debug mode from process-only debug mode or threads only-debug mode when at least one MPI process spawns SMP threads.

Example 1-3: Thread IDs in multilevel debug mode

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

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

1.15.5 Process/Thread Sets

A process/thread set (*p/t-set*) is used to restrict a debugger command to apply to just 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 (described below) to define a p/t-set.

The *current p/t-set* can be set using the *focus* command, which establishes the default p/t-set for cases where no p/t-set prefix is specified. This begins as the debugger-defined set `[all]`, which describes all threads of all processes.

P/t-set notation can be used to prefix a debugger command. This overrides the current p/t-set defining the target threads to be those threads described by the *prefix p/t-set*.

The *target p/t-set* is defined then to be the prefix p/t-set if present, it is the current p/t-set otherwise.

- Use *defset* to define a named or user-defined p/t-set.
- Use *viewset* and *whichsets* to inspect the active members described by a particular p/t-set.

The target p/t-set determines which threads are affected by a PGDBG command.

1.15.6 P/t-set Notation

The following set of rules describes how to use and construct process/thread sets (*p/t-sets*).

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

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

p/t-id :
    {integer|*}.{integer|*}
```

Optional notation when processes-only debugging or threads-only debugging is in effect (see pgienv command).

{integer|*}

```
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}]
```

Example 1-4: P/t-sets in threads-only debug

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

Example 1-5: P/t-sets in process-only debug

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

Example 1-6: 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

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

1.15.7 Dynamic vs. Static P/t-sets

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

Example 1-7: Defining a dynamic p/t-set

`defset clients [*.1:3]`	Defines a named set `clients` whose members are threads 1, 2, and 3 of all processes that are currently active when `clients` is used. Membership in `clients` changes as processes are created and destroyed.

The members of a *static p/t-set* are those threads described by the p/t-set at the time that p/t-set is defined. Use a `!` to specify a static set. Membership in a static set is fixed at definition time.

Example 1-8: Defining a Static p/t-set

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

1.15.8 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 (depending on debug mode). A p/t-set can be used to prefix a command to override 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. See Section *1.15.15 The PGDBG Command Prompt* for a description of the PGDBG prompt.

- The target p/t-set is the current p/t-set:

  ```
  pgdbg [all] 0.0> cont
  ```

  ```
  Continue all threads in all processes
  ```

- The target p/t-set is the prefix p/t-set:

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

Above, the current p/t-set is the debugger-defined set `[all]` in both cases. In the first case, `[all]` is the target p/t-set. In the second case, the prefix set overrides `[all]` as the target p/t-set. The *continue* command is applied to all active threads in the target p/t-set. Using a prefix p/t-set does not change the current p/t-set.

1.15.9 P/t-set Commands

The following commands can be used to collect threads into logical groups.

- *defset* and *undefset* can be used to manage a list of named p/t-sets.
- *focus* is used to set the current p/t-set.
- *viewset* is used to view the active members described by a particular p/t-set.
- *whichsets* is used to describe the p/t-sets to which a particular process/thread belongs.

Table 1-17: P/t-set commands

Command	Description
`focus`	Set the target process/thread set for commands. Subsequent commands will be applied to the members of this set by default
`defset`	Assign a name to a process/thread set. Define a named set. This set can later be referred to by name. A list of named sets is stored by PGDBG
`undefset`	'Undefine' a previously defined process/thread set. The set is removed from the list. The debugger-defined p/t-set `[all]` can not be removed
`viewset`	List the members of a process/thread set that currently exist as active threads
`whichsets`	List all defined p/t-sets to which the members of a process/thread set belongs

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

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

The p/t-set `initial` is defined to contain only thread 0. We focus on `initial` and advance the thread. *Focus* sets the current p/t-set. Because we are not using a prefix p/t-set, the target p/t-set is the current p/t-set which is `initial`.

The *whichsets* command above shows us that thread 0 is a member of two defined p/t-sets. The *viewset* command displays all threads that are active and are members of defined p/t-sets. The `'pgienv verbose'` command can be used to turn on verbose messaging, displaying the stop location of each thread as it stops.

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

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

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

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

```
pgdbg [all] 0>
```

1.15.10 Command Set

For the purpose of parallel debugging, the *PGDBG* 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 are parallelized. *Global* commands are not.

Table 1-18: *PGDBG* 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. Ignores current p/t-set
Global Commands	Non-parallel commands

1.15.10.1 Process Level Commands

The *process level commands* are the *PGDBG* control commands.

The *PGDBG* 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. If a target p/t set does not exist, the current p/t-set is the prefix.

cont	next	nexti	step	stepi
stepout	sync	synci	halt	wait

Example:

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

The *next* command is applied to threads 1 and 2 of process 0.

Example:

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

This demonstrates the use of a prefix p/t-set. The *next* command is applied to thread 3 of process 0 only.

1.15.10.2 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 if not prefix p/t set exists. The *thread level* commands are:

```
set          assign       pc          sp
fp           retaddr      regs        line
func         lines        addr        entry
decl         whatis       rval        lval
sizeof       iread        cread       sread
fread        dread        print       hex
dec          oct          bin         ascii
string       disasm       dump        pf
noprint      where        stack       break*
stackdump    scope        watch       track
break        do           watchi      tracki
doi          hwatch
```

* 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 occurs 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.

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

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> [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
```

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

1.15.10.3 Global Commands

The rest of the *PGDBG* commands ignore threads and processes, or are defined globally for all threads across all processes. The current p/t-set and a prefix p/t-set are ignored.

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

```
debug        run          rerun          threads
procs        proc         thread         call
unbreak      delete       disable        enable
arrive       wait         breaks         status
help         script       log            shell
alias        unalias      directory      repeat
pgienv       files        funcs          source
use          cd           pwd            whereis
edit         /            ?              history
catch        ignore       quit           focus
defset       undefset     viewset        whichsets
display
```

1.15.11 Process and Thread Control

PGDBG supports thread and process control (‘*step*ping’, ‘*next*ing', '*cont*inuing' ...) everywhere in the program. Threads and processes can be advanced in groups anywhere in the program.

The *PGDBG* control commands are:

```
cont,    step,  stepi,  next,  nexti,
stepout,  halt,  wait,   sync,  synci
```

To describe those threads you wish to advance, 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. So if the current thread '*next*'s over an `_mp_init` call (at the beginning of every OpenMP parallel region), then all threads created by `_mp_init` will '*next*' into the parallel region.

A process inherits the control operation of the current process when it is created. So if the current process is '*cont*inuing' out of a call to `MPI_Init`, the new process will do the same.

The *PGDBG* GUI supports process/thread selection via the use of the thread grid and the process grid. To change the current process/thread, click on the corresponding button in the grid. Changing processes updates the thread grid (if present) to display the threads of the new current process. See Section *1.12.1 Main Window* for sample of the graphical user interface.

Accompanying each grid is a set of toggle buttons with the labels *'all'* or *'current'*. These buttons can be used to construct a prefix p/t-set for the next command. The toggle buttons apply to the *'cont'*, *'stepout'*, *'next'*, *'nexti'*, *'step'*, *'stepi'*, *'halt'*, and *'wait'* buttons only.

A process grid is part of the GUI if *PGDBG* is licensed as part of the *CDK*. Selecting *'all'* processes translates to a `'*'` for processes in the prefix set. Selecting *'current'* places the process ID of the current process into the prefix set.

A thread grid is part of the GUI if *PGDBG* is licensed as part of the *CDK* or as part of the *Workstation*. Selecting *'all'* threads translates to a `'*'` for threads in the prefix set. Selecting *'current'* places the thread ID of the current thread into the prefix set.

Example:

'all' process and *'all'* threads constructs a [*.*] p/t-set prefix.

'*current*' process and *'all'* threads constructs a [0.*] p/-set prefix if process 0 is the current process for example.

At least one of the grids will also have a *'none'* button in the toggle set. Selecting none disables automatic p/t-set prefixing by the GUI causing the target p/t-set to be the current p/t-set (since there is no prefix override).

1.15.12 Configurable Stop Mode

PGDBG lets you configure how threads and processes stop in relation to one another. PGDBG defines two new `pgienv` environment variables, `threadstop` and `procstop`, for this purpose. *PGDBG* defines two *stop modes*, synchronous (`sync`) and asynchronous (`async`).

Table 1-19: *PGDBG* 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]
```

PGDBG defines the default to be asynchronous for both thread and process stop modes. When debugging an OpenMP program, *PGDBG* automatically enters synchronous thread stop mode in serial regions, and asynchronous thread stop mode in parallel regions. Synchronous stopping is useful for debugging critical regions. See Section *1.16 OpenMP* Debugging*,* for details.

The `pgienv` environment variable `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.

1.15.13 Configurable Wait mode

Wait mode describes when *PGDBG* 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 threads/processes 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. The

PGDBG prompt will 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 will bring up a prompt if it is available. The availability of the prompt is determined by the current wait mode and any pending wait commands (described below).

PGDBG accepts a compound statement at each prompt. Each compound statement is a bundle of commands, which are processed in order at once. The wait mode describes when to accept the next compound statement. *PGDBG* supports three wait modes:

Table 1-20: *PGDBG* Wait Modes

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

- *Thread wait mode* describes which threads PGDBG waits for before accepting a next command.
- *Process wait mode* describes which processes PGDBG waits for before accepting a next command.

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

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

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.

In TEXT mode PGDBG defaults to

```
threadwait  all
procwait    any
```

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

In GUI mode:

```
threadwait none
procwait   none
```

Setting the wait mode may be necessary when invoking the debugger using the **`-s`** (script file) option in GUI mode (to ensure that the necessary threads are stopped before the next command is processed if necessary).

PGDBG 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. The *`wait`* command can be used to insert wait points between the commands of a compound statement.

The `threadwait` and `procwait` environment variables can be used to configure the behavior of `wait` (see *`pgienv`*).

Table 1-21: *PGDBG* Wait Behavior

The following table describes the behavior of wait. In this example:

- S is the target p/t-set
- P is the set of all processes described by S and p is a process
- T is the set of all threads described by S and t is a thread

Command	threadwait	procwait	Wait set
`wait`	all	all	Wait for T
`wait`	all	none|any	Wait for all threads in at least one p in P
`wait`	none|any	all	Wait for T
`wait`	none|any	none|any	Wait for all t in T for at least one p in P

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

1.15.14 Status Messages

Use the `pgienv` command to enable/disable various status messages. This can be useful in text mode in the absence of the graphical aids provided by the GUI.

```
pgienv verbose <bitmask>
```

Choose the debug status messages that are reported by *PGDBG*. The tool accepts an integer valued bit mask of the values described in the following table.

Table 1-22: *PGDBG* Status Messages

Thread	Format	Information
0x1	Standard	Report status information on current process/thread only. A message is printed only when the current thread stops. Also report when threads and processes are created and destroyed. Standard messaging cannot be disabled. (default)
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/exits a parallel region, or when the threads synchronize.
0x16	Parallel	Report process-parallel events (default). Currently unused.
0x32	Symbolic debug information	Report any errors encountered while processing symbolic debug information (e.g. STABS, DWARF).

1.15.15 The PGDBG Command Prompt

The *PGDBG* command prompt reflects the current debug mode (See Section *1.15.1 PGDBG Debug Modes and Process/Thread Identifiers).*

In serial debug mode, the *PGDBG* prompt looks like this:

```
pgdbg>
```

In threads-only debug mode, *PGDBG* displays the current p/t-set followed by the ID of the current thread.

```
pgdbg [all] 0>
```

```
Current thread is 0
```

In process-only debug mode, *PGDBG* displays the current p/t-set followed by the ID of the current process.

```
pgdbg [all] 0>
```

```
Current process is 0
```

In multilevel debug mode, *PGDBG* displays the current p/t-set 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.

See Section *1.15.1 PGDBG Debug Modes and Process/Thread Identifiers* for a description of the PGDBG debug modes.

1.15.16 Parallel Events

This section describes how to use a p/t-set to define an event across multiple threads and processes. *1.9.1.3 Events,* such as breakpoints and watchpoints, are user-defined events. User defined events are Thread Level commands (See Section *1.15.10.2 Thread Level Commands* for details).

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.

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 on threads 1,2,3 of
   process 0 only.
```

All other user events by default are set for the current thread only. A prefix p/t-set can be used to set user events on specific processes and threads.

Example:

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

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

```
i sets a data breakpoint for glob on thread 0 only.  ii sets a data
   breakpoint for glob on all threads that are currently active.
```

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

Many events contain 'if' and 'do' clauses.

Example:

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

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 on threads 1,2,3 of
   process 0 only.
```

All other user events by default are set for the current thread only. A prefix p/t-set can be used to set user events on specific processes and threads.

Example:

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

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

```
i sets a data breakpoint for glob on thread 0 only.  ii sets a data
   breakpoint for glob on all threads that are currently active.
```

When a process or thread is created, it inherits all of the breakpoints defined for it thus far. All other events must be defined after the process/thread is created. All processes must be stopped to

add, enable, or disable a user event.

Many events contain 'if' and 'do' clauses.

Example:

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

The breakpoint will fire only if `glob` is non-zero. The 'do' clause is executed if the breakpoint fires. The 'if' clause and the 'do' clause execute in the context of a single thread. The conditional in the 'if' and the body of the 'do' execute off of a single (same) thread; the thread that triggered the event. Think of the above definition as:

```
[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.

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.

```
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.

Example:

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

This is illegal. The body of a 'do' statement cannot be parallelized.

1.15.17 Parallel Statements

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

1.15.17.1 Parallel Compound/Block Statements

Example:

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

i. and ii. are equivalent. Each command in a compound statement is executed in order. The target p/t-set is broadcast to all statements. Use the *wait* command if subsequent commands require threads to be stopped (the *print* command above). The commands in a compound statement are executed together in order.

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

1.15.17.2 Parallel If, Else Statements

This section describes parallel 'if' and 'else' statements.

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

A prefix p/t-set parallelizes an 'if' statement. An 'if' statement executes in the context of the current thread by default. The above example is equivalent to:

```
[*] if (i==1) ==> [s]
    [s]break func; [s]c; [s]wait;
else ==> [s']
    [s']sync func2
```

Where [s] is the subset of [*] for which (i==1), and [s'] is the subset of [*] for which (i!=1).

1.15.17.3 Parallel While Statements

This section describes parallel 'while' statements.

Example:

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

A prefix p/t-set parallelizes a 'while' statement. A 'while' statement executes in the context of the current thread by default. The above example is equivalent to:

```
[*] ==> [s]
while(|[s]|){
[s] if (i<10) ==> [s]
    [s]n; [s]wait; [s]print i;
}
```

Where [s] is the subset of [*] for which (i<10). The 'while' statement terminates when [s] is the empty set (or a 'return') statement is executed in the body of the 'while'.

1.15.17.4 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 will return last value evaluated.

1.16 OpenMP Debugging

An attempt is made by *PGDBG* to preserve line level debugging and to help make debugging OpenMP programs more intuitive. *PGDBG* preserves line level debugging across OpenMP threads in the following situations:

- Entrance to parallel region
- Exit parallel region
- Nested parallel regions synchronization points
- Critical and exclusive sections
- Parallel sections

Threads may be *held* and others advanced automatically to negotiate OpenMP program constructs like *synchronization* points and *critical sections*. *PGDBG* also sets the thread stop mode to synchronous when a program runs to a serial region, and asynchronous when a program runs to a parallel region. These features are an attempt to make debugging an OpenMP program easier, and are included by default. (See Section *1.16.2 Disabling PGDBG's OpenMP Event Support* to turn off this behavior). *PGDBG* assumes a legal OpenMP target program.

A control command applied to a running process will only be applied to the stopped threads of that process, and is ignored by its running threads. Threads held by the *PGDBG* OpenMP event handler will also ignore the control command in most situations. For a general introduction to thread-parallel debugging, see sections *1.14.2 Thread-Parallel Debugging* and *1.14.3 Graphical Features* for instructions on using the GUI for thread-parallel debugging. See the online tutorial at http://www.pgroup.com/doc/index.htm to get started using OpenMP Debugging.

1.16.1 Serial vs. Parallel Regions

The initial thread is the thread with OpenMP ID 0. Conceptually, the initial thread is the only thread that is well defined (for the purpose of doing useful work) in a serial region of code. All threads are well defined in a parallel region of code. When the initial thread is in a serial region, the non-initial threads are busy waiting at the end of the last parallel region, waiting to be called down to do some work in the next parallel region. All threads enter the (next) parallel region only when the first thread has entered the parallel region, (i.e., the initial thread is not in a serial region.)

PGDBG source line level debugging operations (*next*, *step*,...) are not well defined for non-initial threads in serial regions since these threads are stuck in a busy loop, which is not compiled to include source line information. The instruction level *PGDBG* control commands (*nexti*, *stepi*, ...) are well defined if you want to advance through the described wait loop at the assembly level.

For example, if *next* is applied to a single thread with OpenMP ID 3 (non-initial thread) in a serial region and all other threads are stopped, then *next* will never complete. By definition, *next* returns only when the thread hits a source line. However, thread 3 running by itself in a serial region will never hit a text address that corresponds to a source line since it is waiting in a busy loop for work to do in the next parallel region. The program will not run to the next parallel region unless thread 0 is run. The busy loop is not compiled with debug information, and is linked into your program when you compile with *-mp* or *-Mconcur*.

Control commands should only be applied to the initial thread in serial regions of code. Synchronous thread stop mode can also be used, however '*step*ping' and '*next*ing' all threads in a serial region may slow down the debugger.

1.16.2 Disabling PGDBG's OpenMP Event Support

PGDBG provides explicit support for OpenMP events. OpenMP events are points in a well-defined OpenMP program where one thread depends on the program location of another thread for it to continue (a barrier for example). *PGDBG*'s support for OpenMP events can be disabled using the `omp` *pgienv* environment variable.

```
pgienv omp [on|off]
```

If *PGDBG*'s OpenMP Event support is disabled, ('`pgienv omp off`') it is recommended that you use the following *pgienv* settings:

```
pgienv threadstopconfig user

pgienv threadstop async

pgienv threadwait none
```

These settings allow threads to stop independently, while allowing commands to be entered while threads are running ('`threadwait none`'). This is necessary to start (*cont*, *next*, ...) and stop (*halt*) threads while some are spinning in OpenMP wait loops (barriers etc.).

Note: OpenMP thread-parallel programs running under the control of PGDBG will run faster with the OpenMP event handler disabled.

1.17 MPI Debugging

1.17.1 Process Control

PGDBG is capable of debugging parallel-distributed MPI programs and hybrid distributed SMP programs. *PGDBG* is invoked via *MPIRUN* and automatically attaches to each MPI process as it is created.

See Section *1.14.4 Process-Parallel Debugging* to get started.

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

- Use p/t-sets to focus on a set of processes. Mind process dependencies.
- In order for a process to receive a message, the sender must be allowed to run.
- Process synchronization points, such as `MPI_Barrier`, will not return until all processes have hit the sync point.
- `MPI_Finalize` will not return for Process 0 until Process 1..n-1 exit.

A control command (*cont*, *step*,...) can be applied 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. Those threads that are held by the OpenMP event handler will also ignore the control command in most situations.

PGDBG automatically switches to process wait mode `none ('pgienv procwait none')` as soon as it attaches to its first MPI process. See the *pgienv* command and Section *1.17.5 MPI* Listener Processes for details.

Use the *run* command to rerun an MPI program. The *rerun* command is not useful for debugging MPI programs since MPIRUN passes arguments to the program that must be included.

1.17.2 Process Synchronization

Use the *PGDBG sync* command to synchronize a set of processes to a particular point in the program.

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

This command runs all processes to MPI_Finalize.

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

This command runs process 0 and process 1 to MPI_Finalize.

A synchronize command will only successfully *sync* the target processes if the *sync* address is well defined for each member of the target process set, and all process dependencies are satisfied (otherwise the member could wait forever for a message for example). The debugger cannot predict if a text address is in the path of an executing process.

1.17.3 MPI Message Queues

PGDBG currently does not support MPI message queue dumping. One way to inspect the MPI message queues is to compile your MPI-CH distribution with *PGCC* with the option **-g** to include debug information. Then inspect the contents of each queue by variable name. See the online FAQ at http://www.pgroup.com/faq/index.htm for details.

1.17.4 MPI Groups

PGDBG identifies each process by its COMMWORLD rank. In general, *PGDBG* currently ignores MPI groups.

1.17.5 MPI Listener Processes

Entering Control-C (^C) from the *PGDBG* command line can be used to halt all running processes. However, this is not the preferred method to use while debugging an MPI program. Entering ^C at the command line, sends a SIGINT signal to the debugger's children. This signal is never received by the MPI processes listed by the *procs* command (i.e., the initial and attached processes), SIGINT is intercepted in each case by *PGDBG*. *PGDBG* does not attach to the MPI listener processes that pair each MPI process. These processes handle IO requests among other things. As a result, a ^C from the command line will kill these processes resulting in undefined program behavior.

It is for this reason, that *PGDBG* automatically switches to process wait mode `none ('pgienv procwait none')` as soon as it attaches to its first MPI process. This allows the use of the *halt* command to stop running processes, without the use of ^C. The setting of `'pgienv procwait none'` allows commands to be entered while there are running processes.

Note: *halt* cannot interrupt a *wait* by definition of *wait*. ^C must be used for this, or careful use of *wait*.

1.17.6 SSH and RSH

PGDBG supports *ssh* as well as *rsh*. The environment variable **PGRSH**, should be set to *ssh* or *rsh*, to indicate the communication method needed. The default is *rsh*.

1.18 Limitations

1.18.1 *PGDBG* Limitations—Parallel Debugging

This sections describes limitations in *PGDBG's* parallel debug support:

- Versions of Linux must be identical on all nodes of cluster.
- *PGDBG* assumes that all processes share the same process image (homogeneous loads, and memory map).
- *PGDBG* recognizes shared objects on the initial node only.
- Process attach is not supported. Initial process is forked from debugger.
- Fortran thread private variables are not available in *PGDBG*.
- The values of thread private variables are available at the first instruction of each source line only (line level debugging only).
- If you are using RH 6.2, you will need to install a patch to use hardware watchpoints. For more details, see the FAQ at http://www.pgroup.com/faq/index.htm.

1.18.2 Other Limitations

This section describes system limitations and other behavior.

- Stdout is block buffered by all non-initial MPI processes.
- Calling omp_set_num_threads(n) from program may cause all non-initial threads to exit before creating n-1 new threads. This is expected behavior.
- Calling omp_set_num_threads(n) from hybrid OpenMP/MPI program may employ the use of the SUGHUP signal off of `MPI_Finalize()`. This is expected behavior.
- Some IO routines use semaphores and locking. It is therefore possible to hang some threads off of a *stdio* routine indefinitely. When your program appears to hang, it is best to continue all threads to a breakpoint outside of the *stdio* routine.
- There is a socket limit imposed by the Linux system. *PGDBG* uses sockets to communicate with thin debug servers running on each node of the cluster. If the system runs out of sockets, the following message will be printed after a one minute timeout. *PGDBG* will then ignore the rest of the (unattached) processes. Those ignored processes will never return from `MPI_Init`.

```
poll: protocol failure in circuit setup
- accept: Bad file descriptor
ERROR: unable to attach to (PID 2763, HOST
red2.wil.st.com)
[New Process (PID 26200, HOST red2.wil.st.com) IGNORED]
```

1.18.3 Private Variables

PGDBG understands private variables with some restrictions. In particular, inspecting private variables while debugging FORTRAN programs is not supported.

Private variables in *C* must be declared in the enclosing lexical block of the parallel region in order for them to be visible using *PGDBG*.

For example:

```
{
    #pragma omp parallel
```

```
        {
            int i;
            ...
            /* i is private to 'this' thread */
            ...
        }
    }
```

In the above case, i would be visible inside *PGDBG* for each thread. However, in the following example, i is not visible inside *PGDBG*:

```
    {
        int i;
        #pragma omp parallel private(i)
        {
            ...
            /* i is private to 'this' thread
               but not visible within PGDBG */
            ...
        }
    }
```

A private variable of a Thread A is accessed by switching the current thread to A, and by using the name (qualified if necessary) of the private variable.

Chapter 2

The *PGPROF* Profiler

This chapter introduces the *PGPROF* profiler. The profiler is a tool that analyzes data generated during execution of specially compiled *C*, *C++*, F77, F90 and HPF programs. The *PGPROF* profiler lets you discover which functions and lines were executed as well as how often they were executed and how much of the total time they consumed.

The *PGPROF* profiler also allows you to profile multi-process HPF or MPI programs, multi-threaded SMP programs (OpenMP or programs compiled with *-Mconcur*), or hybrid multi-process programs employing multiple processes with multiple SMP threads for each process. The multi-process information lets you select combined minimum and maximum process data, or select process data on a process-by-process basis. Multi-threaded information can be queried in the same way as on a per-process basis. This information can be used to identify communications patterns, and identify the portions of a program that will benefit the most from performance tuning.

2.1 Introduction

Profiling is a three-step process:

Compilation — Compiler switches cause special profiling calls to be inserted in the code and data collection libraries to be linked in.

Execution — The profiled program is invoked normally, but collects call counts and timing data during execution. When the program terminates, a profile data file is generated (*pgprof.out*).

Analysis — The *PGPROF* tool interprets the *pgprof.out* file to display the profile data and associated source files. The profiler supports function level and line level data collection modes. The next section provides definitions for these data collection modes.

2.1.1 Definition of Terms

Function Level Profiling
: Is the strategy of collecting call counts and execution times on a per function basis.

Line Level Profiling
: Execution counts and times within each function are collected in addition to function level data. *Line Level* is somewhat of a misnomer because the granularity ranges from data for individual statements to data for large blocks of code, depending on the optimization level. At optimization level 0, the profiling is truly line level.

Basic Block
: At optimization levels above 0, code is broken into basic blocks, which are groups of sequential statements without any conditional or looping controls. Line level profile data is collected on basic blocks rather than individual statements at these optimization levels.

Virtual Timer
: A statistical method for collecting time information by directly reading a timer which is being incremented at a known rate on a processor by processor basis.

Data Set
: A profile data file is considered to be a data set.

Host
: The system on which the *PGPROF* tool executes. This will generally be the system where source and executable files reside, and where compilation is performed.

Target Machine
: The system on which a profiled program runs. This may or may not be the same system as the host.

GUI
: Graphical User Interface. A set of windows, and associated menus, buttons, scrollbars, etc., that can be used to control the profiler and display the profile data.

2.1.2 Compilation

The following list shows driver switches that cause profile data collection calls to be inserted and libraries to be linked in the executable file:

–Mprof=func
: insert calls to produce a *pgprof.out* file for function level data.

–Mprof=lines
: insert calls to produce a *pgprof.out* file which contains both function and line level data.

-Mprof=mpi Link in MPI profile library which intercepts MPI calls in order to record message sizes and to count message sends and receives. Both line-level and function-level profiling are valid with this switch. For example: *-Mprof=mpi,func*

2.1.3 Program Execution

Once a program is compiled for profiling, it needs to be executed. The profiled program is invoked normally, but while running it collects call counts and/or time data. When the program terminates, a profile data file called *pgprof.out* is generated.

To profile an MPI program, use *mpirun* to execute the program which was compiled and linked with the *–Mprof=mpi* switch. A separate data file is generated for each non-initial MPI process. The *pgprof.out* file acts as the "root" profile data file. It contains profile information on the initial MPI process and points to the separate data files for the rest of the processes involved in the profiling run.

The pgmerge utility can be used to merge all data files produced by an MPI program into a single file, *pgprof.out*. The other data files are removed. Data files, unless renamed, are written over by subsequent invocations of *PGPROF*.

2.1.4 Profiler Invocation and Initialization

Running the *PGPROF* profiler allows the profile data produced during the execution phase to be analyzed and initializes the profiler.

The *PGPROF* profiler is invoked as follows:

```
% pgprof [options] [-I srcdir] [datafile]
```

If invoked without any options or arguments, the *PGPROF* profiler looks for the *pgprof.out* data file and the program source files in the current directory. The program executable name, as specified when the program was run, is usually stored in the profile data file. If all program-related activity occurs in a single directory, the *PGPROF* profiler needs no arguments.

If present, the arguments are interpreted as follows:

–s Read commands from standard input. On hosts that have a GUI, this causes *PGPROF* to operate in a non-graphical mode. This is useful if input is being redirected from a file or if the user is remotely logged in to the host

system.

–Isrcdir Add a directory to the source file search path. The *PGPROF* profiler will always look for a program source file in the current directory first. The *–I* option can be used multiple times to append additional directories to the search path. Directories will be searched in the order specified. It is acceptable to leave white space between the *–I* and the *srcdir* arguments.

datafile A single datafile name may be specified on the command line. Specifying a data file that has been generated for a non-initial MPI process is not recommended. Using *PGPROF*, you can inspect profile information on a subset of processes (if you so choose.)

An initialization file named *.pgprofrc* may be placed in the current directory. The data in this file will be interpreted as command line arguments, with any number of arguments per line. A word beginning with # is a comment and causes the rest of the line to be ignored. A typical use of this file would be to specify multiple source directories. The *.pgprofrc* file is read after the command line arguments have been processed. Any arguments provided on the invocation line will override conflicting arguments found in the *.pgprofrc* file.

2.1.5 Virtual Timer

This data collection method employs a single timer, that starts at zero (0) and is incremented at a fixed rate while the active program is being profiled. For multiprocessor programs, there is a timer on each processor, and the profiler's summary data (minimum, maximum and per processor) is based on each processor's time to run a function. How the timer is incremented and at what frequency depends on the target machine. The timer is read from within the data collection functions and is used to accumulate COST and TIME values for each line, function, and the total execution time. The line level data is based on source lines; however, in some cases, there may be multiple statements on a line and the profiler will show data for each statement.

Note: *Due to the timing mechanism used by the profiler to gather data, information provided for longer running functions will be more accurate than for functions that only execute for a short percentage of the timer's granularity. Refer to the list of Caveats for more profiler limitations.*

2.1.6 Profile Data

The following statistics are collected and may be displayed by the *PGPROF* profiler.

BYTES — For HPF and MPI profiles only. This is the number of message bytes sent and received by the function or line.

BYTES RECEIVED — For HPF and MPI profiles only. This is the number of bytes received by the function or line in a data transfer.

BYTES SENT — For HPF and MPI profiles only. This is the number of bytes sent by the function or line.

CALLS — This is the number of times a function is called.

COST — This is the sum of the differences between the timer value entering and exiting a function. This includes time spent on behalf of the current function in all children whether profiled or not.

COUNT — This is the number of times a line or function is executed.

COVERAGE — This is the percentage of lines in a function that were executed at least once.

LINE NUMBER — For line mode, this is the line number for that line. For function mode, this is the line number of the first line of the function.

MESSAGES — For HPF and MPI profiles only. This is the number of messages sent and received by the function or line.

RECEIVES — For HPF and MPI profiles only. This is the number of messages received by the function or line.

SENDS — For HPF and MPI profiles only. This is the number of messages sent by the function or line.

STMT ON LINE — For programs with multiple statements on a line, data is collected and displayed for each statement individually.

TIME — This is only the time spent within the function or executing the line. The TIME does not include time spent in functions called from this function or line. TIME may be displayed in seconds or as a percent of the total time.

TIME PER CALL This is the TIME for a function divided by the CALLS to that function. TIME PER CALL is displayed in milliseconds.

The data provided by virtual timer profiling-based collection allows you to analyze relationships between functions and between processors.

2.1.7 Caveats

Collecting performance data for programs running on high-speed processors and parallel processors is a difficult task. There is no ideal solution. Since programs running on these processors tend to operate within large internal caches, external hardware cannot be used to monitor their behavior. The only other way to collect data is to alter the program itself, which is how this profiling process works. Unfortunately, it is impossible to do this without affecting the temporal behavior of the program. Every effort has been made to strike a balance between intrusion and utility, and to avoid generating misleading or incomprehensible data. It would, however, be unwise to assume the data is beyond question.

2.1.7.1 Clock Granularity

Many target machines provide a clock resolution of only 20 to 100 ticks per second. Under these circumstances a function must consume at least a few seconds of CPU time to generate meaningful line level times.

2.1.7.2 Optimization

At higher optimization levels, and especially with highly vectorized code, significant code reorganization may have occurred within functions. Most line profilers deal with this problem by disallowing profiling above optimization level 0. The *PGPROF* profiler allows line profiling at any optimization level, and significant effort was expended on associating the line level data with the source in a rational manner and avoiding unnecessary intrusion. Despite this effort, the correlation between source and data may at times appear inconsistent. Compiling at a lower optimization level or examining the assembly language source may be necessary to interpret the data in these cases.

2.2 X-Windows Graphical User Interface

The *PGPROF* X-Windows Graphical User Interface (GUI) is invoked using the command `pgprof`. This chapter describes how to use the profiler with the GUI on systems where it's supported. There may be minor variations in the GUI from host to host, depending on the type of monitor available, the settings for various defaults and the window manager used. Some monitors do not support the color features available with the *PGPROF* GUI. The basic interface across all systems remains the same, as described in this chapter, with the exception of the differences tied to the display characteristics and the window manager used.

There are two major advantages provided by the *PGPROF* GUI.

Source Interaction
: The *PGPROF* GUI lets you view the program source for any known function in the line profiler window whether or not line level profile data is available simply by selecting the function name. Since interpreting profile data usually involves correlating the program source and the data, the source interaction provided by the GUI greatly reduces the time spent interpreting data. The GUI allows you to easily compare data on a per processor basis, and identify problem areas of code based on processor execution time differences for functions or lines.

Graphical Display of Data
: It is often difficult to visualize the relationships between the various percentages and execution counts. The GUI allows bar graphs to be displayed which graphically represent these relationships. This makes it much easier to locate the 'hot spots' while scrolling through the data for a large program.

2.2.1 Command Line Switches and X-Windows Resources

PGPROF command line switches may be used to control some features of the GUI. These command line switches may be used when the *PGPROF* profiler is invoked.

–bg <color> sets the display background color to *color*; the default is set by the Motif libraries. For example: `–bg blue`

–fg <color> sets the display foreground color to *color*; the default is set by the Motif libraries. For example: `–fg black`

–bar <num> sets the width of bar graphs to *num*, in number of characters.

–source <num> sets the number of characters of the source program to display for line level data to *num*.

–file <num> sets the number of characters of the filename to display to *num*.

–high <color> See definition below:
–medium <color>
–low <color>
–verylow <color>

Bar graphs are divided into four groups by length, at 25%, 50% and 75% of the longest bar. These bar coloring options (*–high*, *–medium*, *–low* and *–verylow*) let you set the color to use for these four bar groups.

–title <string> sets the window title to *string*.

Normal X-windows switches may also be used, such as *–display* and *–geometry*.

In addition to normal X-windows resources, *PGPROF* uses the following resources, which can be set with the `xrdb` command.

`pgprof.bar: num` equivalent to –bar num

`pgprof.source: num` equivalent to –source num

`pgprof.filename: num` equivalent to –file num

`pgprof.high: color` equivalent to –high color

`pgprof.medium: color` equivalent to –medium color

`pgprof.low: color` equivalent to –low color

`pgprof.verylow: color` equivalent to –verylow color

`pgprof.foreground: color`
equivalent to –fg color

`pgprof.background: color`
equivalent to –bg color

`pgprof.browser: path` sets the path to the web browser used to browse the help page.

`pgprof.helppage: http-address`
the http-address of the *PGPROF* HTML help page.

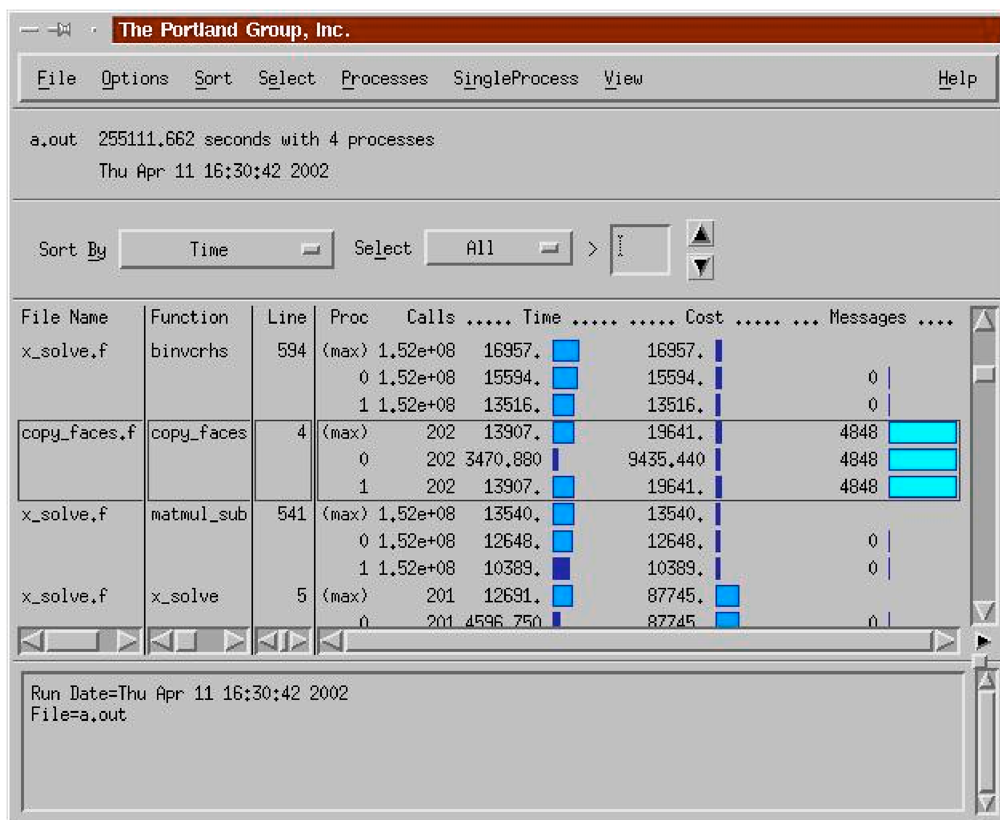
`pgprof.browserdirect:` *`command`*

a *printf* format string used to construct the arguments to the browser; use %s (percent signs) where the http-address should appear.

2.2.2 Using the *PGPROF* X-Windows GUI

The profiler window is divided into five areas from top to bottom, as follows: the Menu Bar area, the Title area, the Sort/Select area, the Display area and the Messages area. The illustration in figure 7-1 depicts a function-level profile window with two processes.

Figure 2-1: Profiler Window



The Menu Bar contains File, Options, Sort, Select, Processes (or Threads), View, and Help menus. The Menu Bar optionally contains a SingleProcess menu. Any of these menus can be selected with a mouse click or by keyboard shortcuts. For example, use Alt+F for File. All menus have tear-off mode enabled. This is performed by clicking on the dashed line on the top of each menu.

- The **Title** area displays the name of the executable as well as the date and time the executable was created. To the right, it also displays the total execution time of the run, the number of processes or processors it used and the date and time of the run.
- The **Sort/Select** area allows you to re-sort the functions or select subsets of the functions based on time, cost, coverage, or other properties; line mode windows do not have a Sort/Select area.
- The **Messages** area contains a scrollable display with informational messages from the *PGPROF* profiler.

2.2.2.1 File Menu

The **`File`** menu permits the following actions:

`Open` <Ctrl+O> Opens a file-selection window to allow you to select a new profiler output file to display. The new data is displayed in a new window.

`Merge` Opens a file-selection window to allow you to select a profiler output file to merge with the current file. The merged execution times are added for each function and line. The merged file must have been created with the same program and executable.

`Print` Sends the data in the **Display** area to a printer.

`Print to File`
Prints the data in the **Display** area to a file.

`Append to File`
Appends the data in the **Display** area to the file most recently created by the **`Print to File`** action.

`Close` <Ctrl+C>
Closes the current *PGPROF* window.

`Quit` <Ctrl+Q>
Closes all *PGPROF* windows and exits the application.

2.2.2.2 Options Menu

The **Options** menu controls the following options:

Printer Options
: Allows you to select the printer command used for the **Print** action. The default is "lpr".

Help Options
: Allows you to change the path to the browser and help page used for interactive help.

Source Directory
: Allows you to add another directory to the search path for source files.

2.2.2.3 Sort Menu and The Sort Option Box

The **Sort** menu and the **Sort** option box in the Sort/Select area allow you to sort the functions by any of several keys. The **Sort** menu and **Sort** option box give the same functionality. The **Sort** menu is not available for line-level windows. The sort keys are:

Name function name (alphabetical)

File Name name of the source file (alphabetical).

Calls number of calls to this function (numerical).

Time execution time spent in this function.

Cost execution time spent in this function and all functions called from this function.

Coverage percentage of lines in this function that were executed.

Time/Call ratio of Time and Calls.

Messages for HPF and MPI profiles, the number of messages sent and received.

Messages Sent
: for HPF and MPI profiles, the number of messages sent.

Messages Received
: for HPF and MPI profiles, the number of messages received.

Bytes for HPF and MPI profiles, the length of all messages sent and received in

bytes.

Bytes Sent for HPF and MPI profiles, the length of all messages sent in bytes.

Bytes Received
for HPF and MPI profiles, the length of all messages received in bytes.

2.2.2.4 Select Menu and The Select Option Box

The **Select** menu and the **Select** option box in the Sort/Select area allow you to select a subset of the functions by any one of several properties. The **Select** menu and **Select** option box give the same functionality. The **Select** menu is not available for line-level windows. The selection options are:

All all functions are displayed.

Calls only functions with more than **N** calls are displayed. Setting the value N is described at the end of this subsection.

Time only functions taking more than **N**% of the total execution time are displayed. Setting the value **N** is described at the end of this subsection.

Coverage only functions with coverage less than **N**% coverage are displayed. Setting the value N is described at the end of this subsection.

Executed only functions that were actually executed are displayed.

Unexecuted only functions that were never called are displayed.

The value **N** used in the description of **Calls**, **Time** and **Coverage** above can be set by typing into the text window in the **Sort/Select** area or by clicking on the up/down arrows next to that text window.

2.2.2.5 Processes Menu

For HPF and MPI profiles, the Processes menu allows you to choose which processor data to display. One or more options may be selected. The options are:

Maximum displays the maximum value (time, cost, calls, count, etc.) from among all processes.

Average displays the average value of all processes.

Minimum displays the minimum value from among all processes.

Sum displays the sum of values for all processes.

All displays data for each processor for each function or lines displayed. One line is used for each processor.

***Note**: if many processes were used, the display can be quite long.*

Individual opens a selection window allowing you to select individual processor data to display.

None turns off individual processor displays. Note that only one of All, Individual and None may be selected.

2.2.2.6 SingleProcess Menu

The Menu Bar includes a SingleProcess menu when PGPROF detects a process that employed more than one thread of execution during a profiling run. SingleProcess lists all processes that participated in the run of the program. Selecting a process from the SingleProcess list spawns a new window, which displays the function-level profile data for that process and each of its SMP threads. The new window has the same format and function as the initial profile window except that it focuses on the selected process only and contains a Threads menu to view the profile data of particular threads.

2.2.2.7 Threads Menu

The Menu Bar includes a Threads menu instead of a Processes menu inside of a process window that is spawned by selecting from the SingleProcess menu. The Threads menu allows you to choose which thread data to display for the selected process. One or more options may be selected. The options are:

Maximum displays the maximum value (time, cost, calls, count, etc.) from among all threads for a given process.

Average displays the average value of all threads for a given process.

Minimum displays the minimum value from among all threads for a given process.

Sum displays the sum of values for all threads for a given process.

All displays data for each thread for each function or line for a given process. One line is used for each thread.

Note: *If many threads were used, the display can be quite long.*

Individual opens a selection window allowing you to select individual thread data to display for a given process.

None turns off individual thread displays. Note that only one of **All**, **Individual** and **None** may be selected.

2.2.2.8 View Menu

The View menu lets you select which data to display. The data that may be viewed for functions include:

Filename name of the source file containing the function.

Line Number line number where the function starts in the source file.

Name name of the function.

Processor for HPF and MPI profiles, the processor number to which this data line corresponds, or the string “max”, “avg”, “min” or “sum”.

Calls number of calls to the function. This may be displayed numerically or as a bar chart.

Time time spent in this function. This may be displayed numerically in seconds or as a percent of total time; or it may be displayed as a bar chart. It may also be displayed as Time Per Call numerically in milliseconds or as a bar chart.

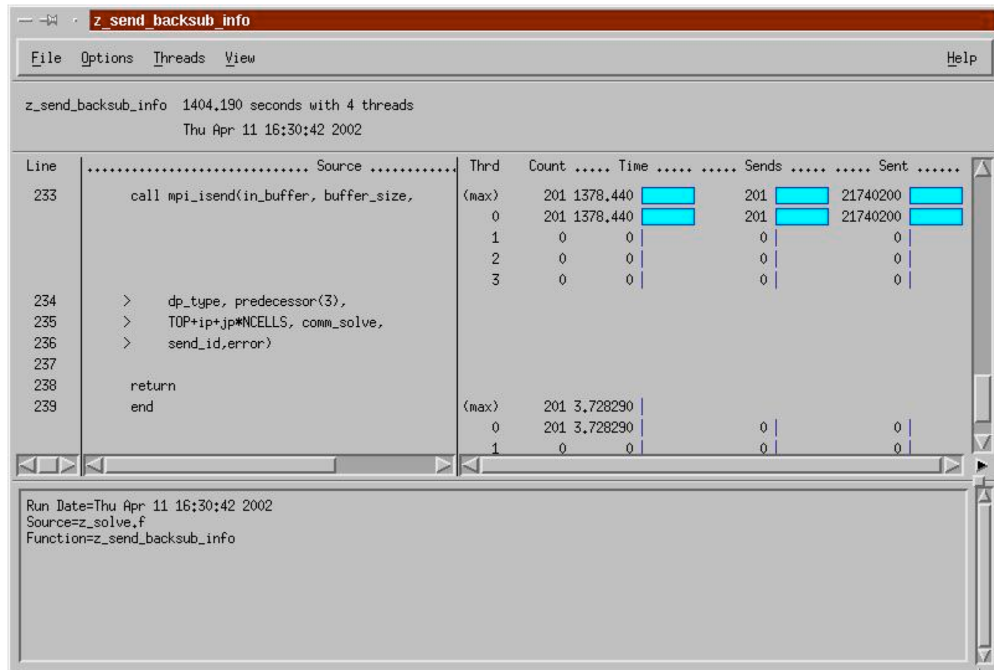
Cost time spent in this function and all functions called from this function. This may be displayed numerically in seconds or as a percent of total time; or it may be displayed as a bar chart.

Coverage number of lines that were actually executed. This may be displayed numerically as a line count or as a percent of actual coverage; or it may be displayed as a bar chart.

Messages for HPF and MPI profiles, the number of messages total, or sent, or received; all either numerically or as a bar chart. Additionally, messages that were executed on the same processor as **copies** may be displayed numerically or as bar charts.

Bytes for HPF and MPI profiles, the total length of all messages in bytes, or messages sent, or messages received; all either numerically or as bar charts. Additionally, the bytes count for messages that were executed on the same processor as copies may be displayed.

Figure 2-2: View Menu



The illustration above shows an individual source line window. Selecting a function name from the function-level profile window and invoking it, usually by double-clicking, will cause a line-level source window to be displayed. The data that may be viewed for individual source lines is:

Line Number line number in the file.

Stmt/on/Line

for programs with multiple statements on one line.

Source the program source text.

Processor for HPF and MPI profiles, the processor number to which this data line corresponds, or the string “max”, “avg”, “min”, “sum”.

Counts the number of times this line was executed. This may be displayed

numerically or as a bar chart.

Time the time spent executing this line. The Seconds may be displayed numerically, as a percent of total time, or as a bar chart. Alternately, Time per Count may be displayed numerically in milliseconds or as a bar chart.

Cost time spent executing this line and all functions called from this line. This may be displayed in Seconds, as a Percent of Cost or as a Bar Chart.

Messages for HPF and MPI profiles, the number of messages total, or sent, or received; all either numerically or as a bar chart. Additionally, messages that were executed on the same processor as **copies** may be displayed numerically or as bar charts.

Bytes for HPF and MPI profiles, the total length of all messages in bytes, or messages sent, or messages received. This may be displayed either numerically or as bar charts. Additionally, the bytes count for messages that were executed on the same processor as copies may be displayed.

2.2.2.9 Help Menu

The Help menu has two options:

About this option opens a window giving version information about *PGPROF*.

Index this option starts up a WWW browser to interactively browse the *PGPROF* help page.

2.3 Command Language

The interface for non-GUI versions of the *PGPROF* profiler is a simple command language. This command language is available in GUI versions of the profiler using the *–s* option. The language is composed of commands and arguments separated by white space. A `pgprof>` prompt is issued unless input is being redirected.

2.3.1 Command Usage

This section describes the profiler's command set. Command names are printed in bold and may be abbreviated as indicated. Arguments contained in [and] are optional. Separating two or more arguments by | indicates that any one is acceptable. Argument names in *italics* are chosen to

indicate what kind of argument is expected. Argument names which are not in *italics* are keywords and should be entered as they appear.

d[*isplay*] [*display options*] | all | none

Specify display information. This includes information on minimum values, maximum values, average values, or per processor data.

he[lp] [*command*]

Provide brief command synopsis. If the *command* argument is present only information for that command will be displayed. The character "?" may be used as an alias for *help*.

h[istory] [*size*]

Display the history list, which stores previous commands in a manner similar to that available with csh or dbx . The optional *size* argument specifies the number of lines to store in the history list.

l[ines] function [[>] *filename*]

Print (display) the line level data together with the source for the specified *function*. If the *filename* argument is present the output will be placed in the named file. The '>' means redirect output, and is optional.

lo[ad] [*datafile*]

Load a new dataset. With no arguments reloads the current dataset. A single argument is interpreted as a new data file. With two arguments, the first is interpreted as the program and the second as the data file.

m[erge] datafile

Merge the profile data from the named *datafile* into the current loaded dataset. The *datafile* must be in standard *pgprof.out* format, and must have been generated by the same executable file as the original dataset (no datafiles are modified.)

pro[cess] processor_num

For HPF profiles, specify the processor number of the data to display.

p[rint] [[>] *filename*]

Print (display) the currently selected function data. If the *filename* argument is present the output will be placed in the named file. The '>' means redirect output, and is optional.

q[uit]

Exit the profiler.

sel[ect] `coverage | covered | uncovered | all` [[<] *cutoff*]

This is the coverage mode variant of the *select* command. The cutoff value is interpreted as a percentage and is only applicable to the *coverage* option. The '<' means less than, and is optional. The default is `coverage` < 100%.

sel[ect] `calls | time/call | time | cost | all` [[>] *cutoff*]

You can choose to display data for a selected subset of the functions. This command allows you to set the selection key and establish a cutoff percentage or value. The cutoff value must be a positive integer, and for time related fields is interpreted as a percentage. The '>' means greater than, and is optional. The default is `time` > 1%.

si[ngleprocessl] process_num

For multiptocess profiles, focus on a single process.

sh[ell] arg1, arg2, argn...

For a shell using the given arguments.

so[rt] [by] `calls | time/call | time | cost | name`

(Profile Mode) Function level data is displayed as a sorted list. This command establishes the basis for sorting. The default is `time`.

so[rt] [by] `coverage | name`

This is the coverage mode variant of the *sort* command. The default is `coverage`, which causes the functions to be sorted based on percentage of lines covered, in ascending order.

src[dir] directory

Add the named *directory* to the source file search path. This is useful if you neglected to specify source directories at invocation.

s[tat] [no]min|[no]avg|[no]max|[no]proc|[no]all]

Set which HPF fields to display or do not display with the no versions.

th[read] thread_num.

Specify a thread for a multithreaded process profile.

t[imes] `raw | pct`

Specify whether time related values should be displayed as raw numbers or as percentages. The default is `pct`. This command does not exist in coverage mode.

!! repeat previous command.

! *num* repeat previous command numbered *num* in the history list.

!-*num* repeat the *num*-th previous command numbered *num* in the history list.

! *string* repeat the most recent previous command starting with *string* from the history list.

Index

Audience Description 1
Clock Granularity 116
Command Set 88
Configurable Wait mode 93
Conformance to Standards 1
Conventions 3
Debug Modes 80, 97, 98
Debugging Parallel Programs with PGDBG 70
Disabling OpenMP Event Support 104
Dynamic vs. Static P/t-sets 85
Global Commands 90
HPF 1
Levical blocks 11
Manual organization 2
MPI
- debugging 76, 105
- groups 106
- *listener process* 17, 105, 106
- message queues 106
- MPI-CH Support 77
- support 69

Multilevel debugging 82
OpenMP 1
- Debugging 102
- debugging serial vs. parallel region 103
- disabling event support 104
- OpenMP and Linuxthread Support 68

P/t-set
- commands 86
- notation 83

Parallel Statements 101
- parallel compound/block statements 101
- parallel if, else statements 101
- parallel while statements 102
- return statements 102

PGDBG
- C++ debugging 21
- commands 8, 22, 50, 88
- core files 22
- debug modes 80, 97, 98
- debugger 5
- debugging parallel programs 70
- dynamic vs. static P/t-sets 85
- events 12
- expressions 15
- Fortran arrays 18
- Fortran common 18
- Fortran debugging 18
- global commands 90
- Graphical presentation of threads and processes 69
- graphical user interface 61
- initialization files 6
- invocation 6
- Limitations 107
- MPI debugging 76
- MPI support 69
- MPI-CH support 77
- multilevel debugging 82
- name of main routine 18
- operators 18
- P/t-set Commands 86
- P/t-set notation 83
- Parallel Debug Capabilities 68
- parallel debugging 80
- process and thread-control 69
- process level commands 88
- process parallel debugging 74

processes and threads 70, 91
Process-only debugging 81
register symbols 9
scope rules 9
source code locations 10
starting up 6
statements 12
status messages 97
thread level commands 89
thread parallel debugging 71
Threads-only debugging 81
user defined events 98
wait behavior 95
wait modes 94

PGDBG Commands
- *command prompt* 75, 80, 85, 97
- conversions 43
- events 26
- memory access 41
- miscellaneous 44
- printing and setting variables 35
- process control 22
- program locations 33
- register access 40
- scope 39
- symbols and expressions 37

PGPROF
- commands 126
- file menu 120
- graphical user interface 117
- help menu 126
- invocation 113
- optimization 116
- options menu 121
- overview 111
- processes menu 122
- profile data 115
- profiler window 120
- select menu 122
- single process menu 123
- sort menu 121
- threads menu 123
- view menu 124
- X-Windows resources 117

PGPROF Command-line options
- -I 114
- -s 113

PGPROF Commands
- *help* 127
- *lines* 127
- *load* 127
- *merge* 127
- *print* 127
- *quit* 127
- *select* 128
- *sort* 128
- *srcdir* 128
- *stat* 128
- *times* 128

Process
- process control 105
- process level commands 88
- process-only debugging 81
- synchronization 105

Processes and Threads 70
- sets 82

Process-Parallel Debugging 74

Profiling
- basic block 112
- command-level interface 126
- compilation 112
- coverage 112
- data set 112
- function level 112
- host 112
- line level 112
- optimization 116
- PGPROF 111
- sampling 112
- target machine 112
- virtual timer 112, 114

Related Publications 3
Serial vs. Parallel Regions 103
Status Messages 96
Synchronization 105
System Requirements 4

Threads
- thread level commands 89
- thread-parallel debugging 71, 80
- threads-only debugging 81

Virtual timer 114