IBM Parallel Environment for AIX: Operation and Use, Volume 2 Tools Reference Version 2, Release 3

Document Number SC28-1980-01

Note: Before using this information and the product it supports, be sure to read the general information under “Notices” on page vii.

| **Second Edition (August 1997)**

| This edition applies to Version 2, Release 3, Modification 0 of the IBM Parallel Environment for AIX (5765-543), and to all subsequent releases and modifications until otherwise indicated in new editions.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

IBM welcomes your comments. A form for your comments appears at the back of this publication. If the form has been removed, address your comments to:

IBM Corporation, Department 55JA, Mail Station P384
522 South Road
Poughkeepsie, NY 12601-5400
United States of America

FAX (United States and Canada: 1+914+432-9405
FAX (Other Countries)
Your International Access Code)+1+914+432-9405

IBMLink (United States customers only): KGNVMC(MHVRCFS)
IBM Mail Exchange: USIB6TC9 at IBMMAIL
Internet e-mail: mhvrcf@vnet.ibm.com
World Wide Web: http://www.rs6000.ibm.com (select Parallel Computing)

If you would like a reply, be sure to include your name, address, telephone number, or FAX number.

Make sure to include the following in your comment or note:

- Title and order number of this book
- Page number or topic related to your comment

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

| Copyright International Business Machines Corporation 1997. All rights reserved. Note to U.S. Government Users — Documentation related to restricted rights — Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

Contents

Notices . . . vii
Trademarks . . . vii

About This Book . . . ix
Who Should Use This Book . . . ix
How This Book is Organized . . . ix
Overview of Contents . . . ix
Typographic Conventions . . . x
Related Publications . . . xi
IBM Parallel Environment for AIX Publications . . . xi
Related IBM Publications . . . xi
Related Non-IBM Publications . . . xii
National Language Support . . . xii
Accessing Online Information . . . xii
Obtaining Documentation . . . xiii

Chapter 1. Using the pdbx Debugger . . . 1
pdbx Subcommands . . . 1
Starting the pdbx Debugger . . . 3
Normal Mode . . . 4
Attach Mode . . . 6
Attach Screen . . . 7
Loading the Partition with the Load Subcommand . . . 8
Displaying Tasks and their States . . . 9
Grouping Tasks . . . 10
Setting Command Context . . . 14
Context Switch when Blocked . . . 16
Controlling Program Execution . . . 18
Setting Breakpoints . . . 18
Interrupting Tasks . . . 20
Setting Tracepoints . . . 20
Specifying Variables On the Trace and Stop subcommands . . . 21
Deleting pdbx Events . . . 22
Checking Event Status . . . 23
Unhooking and Hooking Tasks . . . 24
Examining Program Data . . . 24
Viewing Program Call Stacks . . . 24
Viewing Program Variables . . . 25
Displaying Source . . . 28
Other Key Features . . . 29
Accessing Help for pdbx Subcommands . . . 29
Accessing Help for dbx Subcommands . . . 29
Creating, Removing, and Listing Command Aliases . . . 30
Reading Subcommands From a Command File . . . 31
Specifying Expressions . . . 31
Other Important Notes on pdbx . . . 33
Initial Breakpoint . . . 33
Overloaded Symbols . . . 33
Exiting pdbx . . . 34

© Copyright IBM Corp. 1996, 1997

Chapter 2. Using the pedb Debugger 35
Starting the pedb Debugger 35
Normal Mode 36
Attach Mode 37
Attach Window 38
The pedb Main Window 40
Loading the Partition from the Load Executables Window 42
The pedb Window with a Partition Loaded 44
Setting the Context 46
Creating and Deleting Task Groups 47
Task Status Information 50
Controlling Program Execution 51
Threaded Programs 53
Setting Breakpoints 53
Stepping Execution 57
Tracing Program Execution 59
Unhooking Tasks 62
Examining Program Data 62
Data, Stack, Threads, and Break/Trace Information 62
Locating Breakpoint in Source 63
Displaying Local Variables Within the Program Stack 63
Displaying Local Variables and Program Stack Within a Thread 63
Understanding Data Types 64
Displaying Threads Information 67
Threaded MPI Library 70
Data Display Techniques 70
Viewing the Contents of an Array 73
Exporting Array Information to File 75
Visualizing Program Arrays 80
Source Code Control 84
Other Key Features 89
Debugging Programs Using Multiple Views 89
Getting Help 89
Customizing pedb Resources 90
Leaving pedb 90

Chapter 3. Profiling Parallel Programs with Xprofiler 93
Before You Begin 93
About Xprofiler 93
Requirements and Limitations 93
Xprofiler versus gprof 94
Compiling Applications to be Profiled 94
Starting Xprofiler 95
Xprofiler Command Line Options 95
Loading Files from the Xprofiler GUI 98
Setting the File Search Sequence 105
Understanding the Xprofiler Display 106
The Xprofiler Main Window 106
Using the Xprofiler Graphical User Interface 111
Using the Dialog Window Buttons 112
Using the Search Engine 112
Using the Save Dialog Windows 112
Using the Dialog Window Filters 113
Using the Radio/Toggle Buttons and Sliders 113

Manipulating the Function Call Tree . . . 115
Zooming In on the Function Call Tree . . . 115
Other Viewing Options . . . 119
Filtering What You See . . . 120
Clustering Libraries Together . . . 125
Locating Specific Objects in the Function Call Tree . . . 129
Getting Performance Data for Your Application . . . 130
Getting Basic Data . . . 131
Getting Detailed Data via Reports . . . 135
Looking at Source Code . . . 144
Saving Screen Images of Profiled Data . . . 148

Chapter 4. Visualizing Program and System Performance . . . 151
Starting the Visualization Tool . . . 155
Opening and Closing Views . . . 156
Using VT for Trace Visualization . . . 159
Types of Trace Records . . . 159
Trace Record Timestamps . . . 160
Generating Trace Files . . . 161
Using VT to Play Trace Files . . . 167
Using VT for Performance Monitoring . . . 182
The Performance Monitor Window . . . 182
Adjusting a View's Time Resolution and Colors . . . 187
Adjusting View Colors . . . 188
Saving and Loading a VT Configuration File . . . 190
View Descriptions . . . 191
Communication/Program Views . . . 192
Computation Views . . . 204
Disk Views . . . 216
Network Views . . . 222
System Views . . . 226

Appendix A. Parallel Environment Tools Commands . . . 235
pdbx . . . 236
pedb . . . 273
vt . . . 275
xprofiler . . . 278
. . . 281

Appendix B. Command Line Flags for Normal or Attach Mode . . . 283

Appendix C. Exporting Arrays to Hierarchical Data Format (HDF) . . . 285

Appendix D. Visualization Customization and Data Explorer Samples . 287

Appendix E. Customizing Tool Resources . . . 289
Xprofiler Resource Variables . . . 290
Controlling Fonts . . . 290
Controlling the Appearance of the Xprofiler Main Window . . . 290
Controlling Variables Related to the File Menu . . . 291
Controlling Variables Related to the View Menu . . . 294
Controlling Variables Related to the Filter Menu . . . 295

Appendix F. Profiling Programs with the AIX prof and gprof Commands 297

Glossary of Terms and Abbreviations 299

Index 309

Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, are the user's responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
500 Columbus Avenue
Thornwood, NY 10594
USA

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Mail Station P300
522 South Road
Poughkeepsie, NY 12601-5400
USA
Attention: Information Request

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

AIX
AIX/6000
IBM
LoadLeveler
Micro Channel
RISC System/6000
RS/6000
Scalable POWERparallel Systems
SP

Microsoft, Windows, and the Windows 95 logo are trademarks or registered trademarks of Microsoft Corporation.

PostScript is a trademark of Adobe Systems, Incorporated.

Motif is a trademark of Open Software Foundation.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

Other company, product, and service names, which may be denoted by a double asterisk (**), may be trademarks or service marks of others.

About This Book

This book describes the facilities and tools for the IBM Parallel Environment (PE) for AIX program product and how to use them to analyze, debug, and tune parallel programs. It describes the various PE tools for debugging parallel programs, visualizing a program's performance, and tuning parallel programs for their most efficient performance.

This book concentrates on the actual commands, graphical user interfaces, and use of these tools as opposed to the writing of parallel programs. For this reason, you should use this book in conjunction with *IBM Parallel Environment for AIX: MPI Programming and Subroutine Reference*, (GC23-3894) and *IBM Parallel Environment for AIX: MPL Programming and Subroutine Reference*, (GC23-3893).

This book assumes that AIX Version 4.2.1, X-Windows**, and the PE software are already installed. It also assumes that you have been authorized to run the Parallel Operating Environment (POE). The PE software is designed to run on an IBM RS/6000 SP, an RS/6000 network cluster, or on a mixed system where additional RS/6000 processors supplement an SP system. For complete information on installing the PE software and setting up users, see *IBM Parallel Environment for AIX: Installation*, (GC23-3892). Also, see the appropriate AIX 4.2.1 documentation listed under “Related Publications” on page xi. For information on POE and executing parallel programs, see *IBM Parallel Environment for AIX: Operation and Use, Volume 1, Using the Parallel Operating Environment* and *IBM Parallel Environment for AIX: Hitchhiker's Guide*.

Who Should Use This Book

This book is designed primarily for end users and application developers. It is also intended for those who run parallel programs, and some of the information and tools covered should interest system administrators. Readers should have some experience with graphical user interface concepts such as windows, pull-down menus, and menu bars. They should also have knowledge of the AIX operating system and the X-Window system. Where necessary, this book provides some background information relating to these areas. More commonly, this book refers you to the appropriate documentation.

How This Book is Organized

Overview of Contents

This book contains the following information:

- Chapter 1, “Using the pdbx Debugger” on page 1 describes the Parallel Environment's command line debugger – **pdbx**. This tool uses a line-oriented interface, allowing you to invoke a parallel program from an ASCII terminal.
- Chapter 2, “Using the pedb Debugger” on page 35 describes the other Parallel Environment debugger – **pedb**. This tool uses an X-windows interface for interactive debugging.
- Chapter 3, “Profiling Parallel Programs with Xprofiler” on page 93 describes how to profile your programs with the Parallel Environment's Xprofiler.

© Copyright IBM Corp. 1996, 1997

- Chapter 4, “Visualizing Program and System Performance” on page 151 describes the PE Visualization Tool (VT). Intended for application developers, this chapter describes how you can play back statistical and event records – called *trace records* – generated during a program's execution. It also describes how you can use VT to monitor the operational status and activity of each of the processor nodes.
- Appendix A, “Parallel Environment Tools Commands” on page 235 contains the manual pages for the PE commands discussed throughout this book.
- Appendix B, “Command Line Flags for Normal or Attach Mode” on page 283 shows the command line flags for **pedb** debugging in normal or attach mode.
- Appendix C, “Exporting Arrays to Hierarchical Data Format (HDF)” on page 285 describes additional information about the format of the data used for the **pedb** export feature.
- Appendix D, “Visualization Customization and Data Explorer Samples” on page 287 shows additional information on the Data Explorer samples that are included as a set of pre-packaged interfaces for the visualization of program data.
- Appendix E, “Customizing Tool Resources” on page 289 describes how to customize X-Windows resources for PE tools.
- Appendix F, “Profiling Programs with the AIX prof and gprof Commands” on page 297 describes how to use the AIX profilers **prof** and **gprof** to profile parallel programs.

Typographic Conventions

This book uses the following typographic conventions:

Type Style	Used For
bold	**Bold** words or characters represent system elements that you must use literally, such as command names, flag names, and path names. **Bold** words also indicate the first use of a term included in the glossary.
italic	*Italic* words or characters represent variable values that you must supply. *Italics* are also used for book titles and for general emphasis in text.
`Constant width`	Examples and information that the system displays appear in `constant width` typeface.

In addition to the highlighting conventions, this manual uses the following conventions when describing how to perform tasks. User actions appear in uppercase boldface type. For example, if the action is to enter the **smit** command, this manual presents the instruction as:

ENTER smit

The symbol “●” indicates the system response to an action. So the system's response to entering the **smit** command would read:

● The System Management Interface Tool's main window opens.

Related Publications

IBM Parallel Environment for AIX Publications

- *IBM Parallel Environment for AIX: General Information*, GC23-3906
- *IBM Parallel Environment for AIX: Hitchhiker's Guide*, GC23-3895
- *IBM Parallel Environment for AIX: Installation*, GC28-1981
- *IBM Parallel Environment for AIX: Operation and Use, Volume 1, Using the Parallel Operating Environment*, SC28-1979
- *IBM Parallel Environment for AIX: MPI Programming and Subroutine Reference*, GC23-3894
- *IBM Parallel Environment for AIX: MPL Programming and Subroutine Reference*, GC23-3893
- *IBM Parallel Environment for AIX: Messages*, GC28-1982
- *IBM Parallel Environment for AIX: Licensed Program Specification*, GC23-3896

As an alternative to ordering the individual books, you can use SBOF-8588 to order the entire IBM Parallel Environment for AIX library.

Related IBM Publications

- *IBM AIX Version 4 Getting Started*, SC23-2527
- *IBM AIX General Concepts and Procedures for RS/6000*, GC23-2202
- *IBM AIX Version 4 Files Reference*, SC23-2512
- *IBM AIX Version 4 System Management Guide: Communications and Networks*, SC23-2526
- *IBM AIX Version 4.1 Installation Guide*, SC23-2550
- *IBM AIX Version 4.2 Installation Guide*, SC23-1924
- *IBM AIX Version 4 Commands Reference*, SBOF-1851 (all volumes)
- *IBM AIX Versions 3.2 and 4 Performance Tuning Guide*, SC23-2365
- *IBM AIX Version 4 Messages Guide and Reference*, SC23-2641
- *IBM AIX Version 4.1 Network Installation Management Guide and Reference*, SC23-2627
- *IBM AIX Version 4.2 Network Installation Management Guide and Reference*, SC23-1926
- *IBM AIX Version 4 System Management Guide: Operating System and Devices*, SC23-2525
- *IBM AIX Version 4 General Programming Concepts: Writing and Debugging Programs*, SC23-2533
- *IBM AIX Version 4 Communications Programming Concepts*, SC23-2610
- *Diskless Workstation Management Guide*, SC23-2433
- *C++ for AIX/6000: Language Reference*, SC09-1606
- *C++ for AIX/6000: Standard Class Library Reference*, SC09-1604

- *C++ for AIX/6000: User's Guide*, SC09-1605
- *IBM Performance Toolbox 1.2 and 2 for AIX: Guide and Reference*, SC23-2625

Related Non-IBM Publications

- Almasi, G., Gottlieb, A., *Highly Parallel Computing*, Benjamin-Cummings Publishing Company, Inc., 1989.
- Gropp, W., Lusk, E., Skjellum, A., *Using MPI*, The MIT Press, 1994.
- Message Passing Interface Forum, MPI: *A Message-Passing Interface Standard*. Version 1.1, University of Tennessee, Knoxville, Tennessee, June 6, 1995.
- Foster, I., *Designing and Building Parallel Programs*. Addison-Wesley, 1995.
- Pfister, Gregory, F., *In Search of Clusters*. Prentice Hall, 1995.

National Language Support

For National Language Support (NLS), all PE components and tools display messages located in externalized message catalogs. English versions of the message catalogs are shipped with the PE program product, but your site may be using its own translated message catalogs. The AIX environment variable **NLSPATH** is used by the various PE components to find the appropriate message catalog. **NLSPATH** specifies a list of directories to search for message catalogs. The directories are searched, in the order listed, to locate the message catalog. In resolving the path to the message catalog, **NLSPATH** is affected by the values of the environment variables **LC_MESSAGES** and **LANG**. If you get an error saying that a message catalog is not found, and want the default message catalog:

ENTER **export NLSPATH=/usr/lib/nls/msg/%L/%N**

export LANG=C

The PE message catalogs are in English, and are located in the following directories:

/usr/lib/nls/msg/C
/usr/lib/nls/msg/En_US
/usr/lib/nls/msg/en_US

If your site is using its own translations of the message catalogs, consult your system administrator for the appropriate value of **NLSPATH** or **LANG**. For additional information on NLS and message catalogs, see *IBM Parallel Environment for AIX: Messages* and *IBM AIX Version 4 General Programming Concepts: Writing and Debugging Programs*.

Accessing Online Information

The PE program product provides manual pages as well as online versions of the publications that can be viewed using an HTML browser (such as Netscape, WebExplorer, or Mosaic).

To access a list of manual pages for PE:

ENTER **man pe**

Note: In order to use the PE MPI subroutine manual pages or access the PE online (HTML) publications, the **ppe.pedocs** fileset must first be installed. To view the PE online publications, you need the following:

- Access to a common HTML document browser (such as Netscape, WebExplorer, or Mosaic)
- The location of the HTML index file, provided with the **ppe.pedocs** fileset. Contact your system administrator or installer for this location.

Obtaining Documentation

This book is available in PostScript format. You can obtain the PostScript files for the product documentation by:

- Downloading them from the **lscftp.pok.ibm.com** anonymous FTP server. The files for *PE_prod_name* can be found in the directory called /**pub**/**pps**/**PE**.
- Printing the PostScript documentation files that are shipped on the product media.

You can also view or download this book from the IBM RS/6000 web site, whose current location is listed in the PE filesets' README files.

Chapter 1. Using the pdbx Debugger

This chapter describes the **pdbx** debugger. This debugger extends the **dbx** debugger's line-oriented interface and subcommands. Some of these subcommands, however, have been modified for use on parallel programs. The **pdbx** debugger is a POE application with some modifications on the *home node* to provide a user interface.

Before invoking a parallel program using **pdbx** for interactive debugging, you first need to compile the program and set up the execution environment. See *IBM Parallel Environment for AIX: Operation and Use, Volume 1, Using the Parallel Operating Environment* for more information on the following:

- Compiling the program. Be sure to specify the **-g** flag when compiling the program. This produces an object file with symbol table references needed for symbolic debugging. It is also advisable to not use the optimization option, **-O**. Using the debugger on optimized code may produce inconsistent and erroneous results. For more information on the **-g** and **-O** compiler options, refer to their use on other compiler commands such as **cc** and **xlf**. These compiler commands are described in *IBM AIX Version 4 Commands Reference*, or your online manual pages.
- Copying files to individual nodes. Like **poe**, **pdbx** requires that your application program be available to run on each node in your partition. To support source level debugging, **pdbx** requires the source files to be available as well. You will generally use the same mechanism to make the source files accessible as you used for the application program.
- Setting up the execution environment.

As you read these steps, keep in mind that **pdbx** accepts almost all the option flags that **poe** accepts, and responds to the same environment variables.

Also, throughout this book, keep in mind the following information.

The RS/6000 processors of your system are called *processor nodes*. A parallel program executes as a number of individual, but related, *parallel tasks* on a number of your system's processor nodes. The group of parallel tasks is called a *partition*. The processor nodes are connected on the same network, so the parallel tasks of your partition can communicate to exchange data or synchronize execution.

pdbx Subcommands

Table 1 on page 2 outlines the **pdbx** subcommands described in this chapter. Complete syntax information for all these subcommands is also provided under the entry for the **pdbx** command in Appendix A, “Parallel Environment Tools Commands” on page 235.

The debugger supports most of the familiar **dbx** subcommands, as well as some additional **pdbx** subcommands. In **pdbx**, *command context* refers to a setting that controls which task(s) receive the subcommands entered at the **pdbx** command prompt.

© Copyright IBM Corp. 1996, 1997

pdbx subcommands can either be *context sensitive* or *context insensitive*. The debugger directs context sensitive subcommands to just the tasks in the current command context. Command context has no bearing on context insensitive commands, which control overall debugger behavior, and are generally processed on the home node only. These include subcommands for setting help and other information, and ending a **pdbx** session.

You can set the command context on a single task or a group of tasks as described in "Setting Command Context" on page 14.

Table 1 (Page 1 of 2). pdbx Subcommands

Context Insensitive pdbx Subcommands		
This subcommand:	**Is used to:**	**For more information see:**
alias [alias_name string]	Set or display aliases.	"Creating, Removing, and Listing Command Aliases" on page 30
attach <[all | task_list]>	Attach the debugger to some or all the tasks of a given **poe** job.	"Attach Mode" on page 6
detach	Detach **pdbx** from all tasks that were attached. This subcommand causes the debugger to exit but leaves the **poe** application running.	"Exiting pdbx" on page 34
dhelp [dbx_command]	Display a brief list of **dbx** commands or help information about them.	"Accessing Help for dbx Subcommands" on page 29
group <action> [group_name] [task_list]	Manipulate groups. The actions are **add**, **change**, **delete**, and **list**. To indicate a range of tasks, enter the first and last task numbers, separated by a colon or dash. To indicate individual tasks, enter the numbers, separated by a space or comma.	"Grouping Tasks" on page 10
help [subject]	Display a list of **pdbx** commands and topics or help information about them.	"Accessing Help for pdbx Subcommands" on page 29
on <[group | task]> [command]	Set the command context used to direct subsequent commands to a specific task or group of tasks. This subcommand can also be used to deviate from the command context for a single command without changing the current command context.	"Setting the Current Command Context" on page 14
quit	End a **pdbx** session.	"Exiting pdbx" on page 34
source <cmd_file>	Execute **pdbx** subcommands from a specified file. **Note:** The file may contain context sensitive commands.	"Reading Subcommands From a Command File" on page 31
tasks [long]	Display information about all the tasks in the partition.	"Displaying Tasks and their States" on page 9
unalias alias_name	Remove a command alias specified by the **alias** subcommand.	"Creating, Removing, and Listing Command Aliases" on page 30
Context Sensitive pdbx Subcommands		
This Subcommand:	**Is used to:**	**For more information see:**
delete <[event_list | * | all]>	Remove breakpoints and tracepoints set by the **stop** and **trace** subcommands. To indicate a range of events, enter the first and last event numbers, separated by a colon or a dash. To indicate individual events, enter the number(s), separated by a space or comma.	"Deleting pdbx Events" on page 22

Table 1 (Page 2 of 2). pdbx Subcommands

dbx <dbx_command>	Issue a **dbx** subcommand directly to the **dbx** sessions running on the remote nodes. This subcommand is not intended for casual use. It must be used with caution, because it circumvents the **pdbx** server which normally manages communication between the user and the remote **dbx** sessions. It enables experienced **dbx** users to communicate directly with remote **dbx** sessions, but can cause problems as **pdbx** will have no knowledge of the communication that transpired. **Note:** In addition to the **pdbx** subcommands shown in this table, you can use most of the **dbx** subcommands. The **dbx** subcommands are all context sensitive. The only **dbx** subcommands that you cannot use are **clear**, **detach**, **edit**, **multproc**, **prompt**, **run**, **rerun**, **screen**, and the **sh** subcommand with no arguments.	the online PE manual page for **pdbx**. This manual page also appears in Appendix A, "Parallel Environment Tools Commands" on page 235.
hook	Regain control over an unhooked task.	"Unhooking and Hooking Tasks" on page 24
list [line_number | line_number, line_number | procedure]	Display lines of the current source file, or of a procedure.	"Displaying Source" on page 28
load <program> [program_arguments]	Load a program on each node in the current context. This can only be issued once per task per **pdbx** session. **pdbx** will look for the program in the current directory unless a relative or absolute pathname is specified.	"Loading the Partition with the Load Subcommand" on page 8
print <[expression | procedure]>	Print the value of an expression, or run a procedure and print the return code of that procedure.	"Viewing Program Variables" on page 25
status [all]	Display a list of breakpoints and tracepoints set by the **stop** and **trace** subcommands in the current context. If "all" is specified, all events, regardless of context are shown.	"Checking Event Status" on page 23
stop	Set a breakpoint for tasks in the current context. Breakpoints are stopping places in your program that halt execution.	"Setting Breakpoints" on page 18
trace	Set a tracepoint for tasks in the current context. Tracepoints are places in your program that, when reached during execution, cause the debugger to print information about the state of the program.	"Setting Tracepoints" on page 20
unhook	Unhook a task or group of tasks. Unhooking allows the task(s) to run without intervention from the debugger.	"Unhooking and Hooking Tasks" on page 24
where	Display a list of active procedures and functions.	"Viewing Program Call Stacks" on page 24
<**Ctrl-c**>	Regain debugger control when some tasks in the current context are running. This causes a **pdbx** subset prompt to be displayed, which allows a subset of the **pdbx** function to be performed.	"Context Switch when Blocked" on page 16

Starting the pdbx Debugger

You can start the **pdbx** debugger in either *normal* mode or *attach* mode. In normal mode your program runs under the control of the debugger. In attach mode you attach to a program that is already running. Certain options and functions are only available in one of the two modes. Since **pdbx** is a source code debugger, some files need to be compiled with the **-g** option so that the compiler provides debug symbols, source line numbers, and data type information.

When the application is started using **pdbx** in normal mode, debugger control of the application is given to the user by default at the first executable source line within the main routine. This is function *main* in C code or the the routine defined by the *program* statement in Fortran. In Fortran, if there is no *program* statement, the program name defaults to *main*. If the file containing the main routine is not compiled with **-g** the debugger will exit. The environment variable **MP_DEBUG_INITIAL_STOP** can be set before starting the debugger to manually set an alternate file name and source line where the user initially receives debugger control of the application. Refer to the appendix on POE environment variables and command-line flags in *IBM Parallel Environment for AIX: Operation and Use, Volume 1, Using the Parallel Operating Environment*.

Normal Mode

The way you start the debugger in normal mode depends on whether the program(s) you are debugging follow the SPMD (Single Program Multiple Data) or MPMD (Multiple Program Multiple Data) model of parallel programming. In the SPMD model, the same program runs on each of the nodes in your partition. In the MPMD model, different programs can run on the nodes of your partition.

If you are debugging an SPMD program, you can enter its name on the **pdbx** command line. It will be loaded on all the nodes of your partition automatically. If you are debugging an MPMD program, you will load the tasks of your partition after the debugger is started. **pdbx** will look for the program in the current directory unless a relative or absolute pathname is specified.

ENTER **pdbx** [*program* [*program_options*]] [*poe options*] [**-c** *command_file*] [**-d** *nesting_depth*] [**-l** *directory* [**-l** *directory*]...] [**-F**] [**-x**]

- This starts **pdbx**. If you specified a *program*, it is loaded on each node of your partition and you see the message:

  ```
  0031-504  Partition loaded ...
  ```

  You will then see the pdbx prompt:

  ```
  pdbx(all)
  ```

  The prompt shows the command context `all`. For more information see "Setting Command Context" on page 14.

ENTER **pdbx -a** *poe process id* [*limited poe options*] [**-c** *command_file*] [**-d** *nesting_depth*] [**-l** *directory* [**-l** *directory*]...] [**-F**] [**-x**]

- This starts **pdbx** in attach mode. See "Attach Mode" on page 6 for more information.

ENTER **pdbx -h**

- This writes the **pdbx** usage to STDERR. It includes **pdbx** command line syntax and a description of **pdbx** options.

The options you specify with the **pdbx** command can be program options, POE options, or **pdbx** options listed in Table 2 on page 5. Program options are those that your application program will understand.

You can use the same command-line flags on the **pdbx** command as you use when invoking a parallel program using the **poe** command. For example, you can

override the **MP_PROCS** variable by specifying the number of processes with the **-procs** flag. Or you could use the **-hostfile** flag to specify the name of a host list file. For more information on the POE command-line flags, see *IBM Parallel Environment for AIX: Operation and Use, Volume 1, Using the Parallel Operating Environment*.

Note: **poe** uses the **PATH** environment variable to find the program, while **pdbx** does not.

After **pdbx** initializes, the **pdbx** command prompt displays to indicate that **pdbx** is ready for a command.

Table 2 (Page 1 of 2). Debugger Option Flags (pdbx)

Use this flag:	To:	For example:
-a	Attach to a running **poe** job by specifying its process id. This must be executed from the node where the **poe** job was initiated. When using the debugger in attach mode there are some debugger command line arguments that should not be used. In general, any arguments that control how the partition is set up or specify application names and arguments should not be used.	To attach the **pdbx** debugger to an already running **poe** job. **ENTER** **pdbx -a** <*poe_process_id*>
-c	Read **pdbx** startup commands from the specified *commands_file*. The commands stored in the specified file are executed before command input is accepted from the keyboard. If the **-c** flag is not used, the **pdbx** debug program attempts to read startup commands from the file *.pdbxinit*. To find this file, it first looks in the current directory, and then in the user's home directory. In a **pdbx** session, you can also read commands from a file using the **source** subcommand. "Reading Subcommands From a Command File" on page 31 describes how to use this subcommand in **pdbx**.	To start the **pdbx** debugger and read startup commands from a file called *start.cmd*: **ENTER** **pdbx -c** *start.cmd*
-d	Set the limit for the nesting of program blocks. The default nesting depth limit is 25. This flag is passed to **dbx** unmodified.	To specify a nesting depth limit: **ENTER** **pdbx -d** *nesting.depth*
-F	This flag can be used to turn off *lazy reading* mode. Turning lazy reading mode off forces the remote **dbx** sessions to read all symbol table information at startup time. By default, lazy reading mode is on. Lazy reading mode is useful when debugging large executable files, or when paging space is low. With lazy reading mode on, only the required symbol table information is read upon initialization of the remote **dbx** sessions. Because all symbol table information is not read at **dbx** startup time when in lazy reading mode, local variable and related type information will not be initially available for functions defined in other files. The effect of this can be seen with the **whereis** command, where instances of the specified local variable may not be found until the other files containing these instances are somehow referenced.	To start the **pdbx** debugger and read all symbol table information: **ENTER** **pdbx -F**
-h	Write the **pdbx** usage to STDERR then exit. This includes **pdbx** command line syntax and a description of **pdbx** options.	**ENTER** **pdbx -h**
-I (upper case i)	Specify a directory to be searched for an executable's source files. This flag must be specified multiple times to set multiple paths. (Once **pdbx** is running, this list can be overridden on a group or single node basis with the **use** command.)	To add *directory1* to the list of directories to be searched when starting the **pdbx** debugger: **ENTER** **pdbx -I** *dir1* You can add as many directories as you like to the directory list in this way. For example, to add two directories: **ENTER** **pdbx -I** *dir1* **-I** *dir2*

Table 2 (Page 2 of 2). Debugger Option Flags (pdbx)		
Use this flag:	**To:**	**For example:**
-x	Prevent the **dbx** command from stripping _ (trailing underscore) characters from symbols originating in Fortran source code. This flag allows **dbx** to distinguish between symbols which are identical except for an underscore character, such as xxx and xxx_.	To prevent trailing underscores from being stripped from symbols in Fortran source code: **ENTER pdbx -x**

These **pdbx** flags are closely tied to the flags supported by **dbx**. For more information on the option flags described in this table, refer to their use with **dbx** as described in *IBM AIX Version 4 Commands Reference* and *IBM AIX Version 4 General Programming Concepts: Writing and Debugging Programs*.

For a listing of **pdbx** subcommands, you can also refer to its online manual page. This manual page also appears in Appendix A, “Parallel Environment Tools Commands” on page 235.

Attach Mode

The **pdbx** debugger provides an attach feature, which allows you to attach the debugger to a parallel application that is currently executing. This feature is typically used to debug large, long running, or apparently “hung” applications. The debugger attaches to any subset of tasks without restarting the executing parallel program.

Parallel applications run on the partition managed by **poe**. For attach mode, you must specify the appropriate process identifier (PID) of the **poe** job, so the debugger can attach to the correct application processes contained in that particular job. To get the PID of the **poe** job, enter the following command on the node where **poe** was started:

```
$ ps -ef | grep poe
```

You initiate attach mode by invoking **pdbx** with the **-a** flag and the PID of the appropriate **poe** process:

```
$ pdbx -a <poe PID>
```

For example, if the process id of the **poe** process is `12345` then the command would be:

```
$ pdbx -a 12345
```

After you invoke the debugger in attach mode, it displays a list of tasks you can choose. The paging tool used to display the menu will default to **pg -e** unless another pager is specified by the **PAGER** environment variable.

pdbx starts by showing a list of task numbers that comprise the parallel job. The debugger obtains this information by reading a configuration file created by **poe** when it begins a job step. At this point you must choose a subset of that list to attach the debugger. Once you make a selection and the attach debug session starts, you cannot make additions or deletions to the set of tasks attached to. It is possible to attach a different set of tasks by detaching the debugger and attaching again, then selecting a different set of tasks.

The debugger attaches to the specified tasks. The selected executables are stopped wherever their program counters happen to be, and are then under the

control of the debugger. The other tasks in the original **poe** application continue to run. **pdbx** displays information about the attached tasks using the task numbering of the original **poe** application partition.

Attach Screen

Figure 1 shows a sample **pdbx** Attach screen.

```
pdbx Version 2, Release 3 -- Mar 1 1997 15:33:03

ATTENTION: 0029-9049 The following environment variables have been
ignored since they are not valid when starting the debugger
in attach mode -
  'MP_PROCS'.

To begin debugging in attach mode, select a task or tasks to attach.

Task       IP Addr                Node                 PID     Program
0        9.117.8.62          pe02.kgn.ibm.com        23870      ftoc
1        9.117.8.63          pe03.kgn.ibm.com        14908      ftoc
2        9.117.8.64          pe04.kgn.ibm.com        14400      ftoc
3        9.117.8.65          pe05.kgn.ibm.com        13114      ftoc
4        9.117.8.66          pe06.kgn.ibm.com        11330      ftoc
5        9.117.8.67          pe07.kgn.ibm.com        19784      ftoc
6        9.117.8.68          pe08.kgn.ibm.com        19524      ftoc
7        9.117.8.69          pe09.kgn.ibm.com        22086      ftoc

At the pdbx prompt enter the "attach" command followed by a
list of tasks or "all".  (ex. "attach 2 4 5-7" or "attach all")
You may also type "help" for more information or "quit" to exit
the debugger without attaching.

pdbx(none)
```

Figure 1. pdbx Attach screen

The **pdbx** Attach screen contains a list of tasks and, for each task, the following information:

- Task - the task number
- IP - the ip address of the node on which the task/application is running
- Node - the name of the node on which the task/application is running, if available
- PID - the process identifier of the task/application
- Program - the name of the application and arguments, if any.

After initiating attach mode, you can select a set of tasks to attach to. At the command prompt:

ENTER **attach all**

OR

ENTER **attach** followed by the *task_list* (see “Syntax for task_list” on page 10 for the correct syntax).

It is also possible to use the **quit** or **help** command at this prompt. Any other command will produce an error message. The **quit** command will not kill the application at this point, since the debugger has not been attached as of yet.

Note: When debugging in attach mode, the **load** subcommand is not available. An error message is displayed if an attempt is made to use it.

Other Compiling Options

pdbx provides substantial information when debugging an executable compiled with the **-g** option. However, you may find it useful to attach to an application not compiled with **-g**. **pdbx** allows you to attach to an application not compiled with **-g**, however, the information provided is limited to a stack trace.

You can also attach **pdbx** to an application compiled with both the **-g** and optimization flags. However, the optimized code can cause some confusion when debugging. For example, when stepping through code, you may notice the line marker points to different source lines than you would expect. The optimization causes this re-mapping of instructions to line numbers.

Command Line Arguments

You should not use certain command line arguments when debugging in attach mode. If you do, the debugger will not start, and you will receive a message saying the debugger will not start. In general, do not use any arguments that control how the debugger partition is set up or that specify application names and arguments. You do not need information about the application, since it is already running and the debugger uses the PID of the **poe** process to attach. Other information the debugger needs to set up its own partition, such as node names and PIDs, comes from the configuration file and the set of tasks you select. See Appendix B, “Command Line Flags for Normal or Attach Mode” on page 283 for a list of command line flags showing which ones are valid in normal and in attach debugging mode.

The information in the appendix is also true for the corresponding environment variables, however **pdbx** ignores the invalid setting. The debugger displays a message containing a list of the variables it ignores, and continues.

For example, if you had **MP_PROCS** set, when the debugger starts in attach mode it ignores the setting. It displays a message saying it ignored **MP_PROCS**, and continues initializing the debug session.

Loading the Partition with the Load Subcommand

Before you can debug a parallel program with the **pdbx** debugger, you need to load your partition. If you specified a program name on the **pdbx** command, it is already loaded on each task of your partition. If not, you need to load your partition using the **load** subcommand. **pdbx** will look for the program in the current directory unless a relative or absolute pathname is specified. The Partition Manager allocates the tasks of your partition when you enter the **pdbx** command. It does this either by connecting to the Resource Manager or by looking to your host list file. The number of tasks in the partition depends on the value of the **MP_PROCS** environment variable (or the value specified on the **-procs** flag) when you enter the **pdbx** command.

The following **pdbx** commands are available before the program is loaded on all tasks:

- alias
- group
- help
- load
- on
- quit
- source
- tasks
- unalias

To load the same executable on all tasks of the partition:	**To load separate executables on the partition:**
CHECK the **pdbx** command prompt to make sure the command context is on all tasks. The context *all* is the default when you start the **pdbx** debugger, so the prompt should read: `pdbx(all)` If the command context is not set on *all* tasks, reset it. To do this: **ENTER** **on all** Once the command context is on all tasks: **ENTER** **load** *program* [program_options] • The specified program is loaded onto all tasks in the partition, and the message "Partition loaded..." displays. The parallel program runs up to the first executable statement and stops. **Note:** The example above has the same effect as putting the program name and options on the command line.	**SET** the command context before loading each program. For example, say your partition consists of five tasks numbered 0 through 4. To load a program named *program1* on task 0 and a program named *program2* on tasks 1 through 4, you would: **ENTER** **on** *0* • The debugger sets the command context on task 0 **ENTER** **load** *program1* [program_options] The debugger loads *program1* on task 0. **ENTER** **group add** *groupa 1-4* • The debugger creates a task group named *groupa* consisting of tasks 1 through 4. **ENTER** **on** *groupa* The debugger sets the command context on tasks 1 through 4. **ENTER** **load** *program2* [program_options] • The debugger loads *program2* onto tasks 1 through 4, and the message "Partition loaded..." displays. The parallel program runs up to the first executable statement and stops.

Displaying Tasks and their States

With the **tasks** subcommand, you display information about all the tasks in the partition. Task state information is always displayed (see Table 3 on page 13 for information on task states). If you specify "long" after the command, it also displays the name, ip address, and job manager number associated with the task.

Following is an example of output produced by the **tasks** and **tasks long** command.

```
pdbx(others) tasks
  0:D     1:D     2:U     3:U     4:R     5:D     6:D     7:R

pdbx(others) tasks long
  0:Debug ready   pe04.kgn.ibm.com                9.117.8.68       -1
  1:Debug ready   pe03.kgn.ibm.com                9.117.8.39       -1
  2:Unhooked      pe02.kgn.ibm.com                9.117.11.56      -1
  3:Unhooked      augustus.kgn.ibm.com            9.117.7.77       -1
  4:Running       pe04.kgn.ibm.com                9.117.8.68       -1
  5:Debug ready   pe03.kgn.ibm.com                9.117.8.39       -1
  6:Debug ready   pe02.kgn.ibm.com                9.117.11.56      -1
  7:Running       augustus.kgn.ibm.com            9.117.7.77       -1
```

Grouping Tasks

You can set the context on a group of tasks by first using the context insensitive **group** subcommand to collect a number of tasks under a group name you choose. None of these tasks need to have been loaded for you to include them in a group. Later, you can set the context on all the tasks in the group by specifying its group name with the **on** subcommand.

For example, you could use the **group** subcommand to collect a number of tasks (tasks 0, 1, and 2) as a group named *groupa*. Then, to set the context on tasks 0, 1, and 2, you would:

ENTER **on** *groupa*

- The debugger sets the command context on tasks 0, 1, and 2.

Another use of the **group** subcommand is to give a name to a task. For example, assume you have a typical master/worker program. Task 0 is the master task, controlling a number of worker tasks. You could create a group named *master* consisting of just task 0. Then, to set the context on the master task you would:

ENTER **on** *master*

- The debugger sets the command context on task 0. Entering **on** *master*, therefore, is the same as entering **on** *0*, but would be more meaningful and easier to remember.

The **group** subcommand has a number of actions. You can use it to:

- Create a task group, or add tasks to an existing task group
- Delete a task group, or delete tasks from an existing task group
- Change the name of an existing task group
- List the existing task groups, or list the members of a particular task group.

Syntax for group_name: Provide a group name that is no longer than 32 characters which starts with an alphabetic character, and is followed by any alphanumeric character combination.

Syntax for task_list: To indicate a range of tasks, enter the first and last task numbers, separated by a colon or dash. To indicate individual tasks, enter the numbers, separated by a space or comma.

Note: Group names “all,” “none,” and “attached” are reserved group names. They are used by the debugger and cannot be used in the **group add** or **group delete** commands. However, the group “all” or “attached” can be

renamed using the **group change** command, if it currently exists in the debugging session.

Adding a Task to a Task Group

To add a task to a new or already existing task group, use the **add** action of the **group** subcommand. The syntax is:

group add *group_name task_list*

If the specified *group_name* already exists, then the debugger adds the tasks in *task_list* to that group. If the specified *group_name* does not yet exist, the debugger creates it.

The variable *task_list* **can be:**	**For example, to add the following task(s) to** *groupa***:**	**You would ENTER:**	• **The following message displays:**
a single task	task 6	**group add** *groupa 6*	`1 task was added to group "groupa".`
a collection of tasks	tasks 6, 8, and 10	**group add** *groupa 6 8 10*	`3 tasks were added to group "groupa".`
a range of tasks	tasks 6 through 10	**group add** *groupa 6:10*	`5 tasks were added to group "groupa".`
a range of tasks	tasks 6 through 10	**group add** *groupa 6-10*	`5 tasks were added to group "groupa".`

Deleting Tasks from a Task Group

To delete tasks from a task group, use the **delete** action of the **group** subcommand. The syntax is:

group delete *group_name* **[***task_list***]**

The variable *task_list* can be:	For example, to delete the following from *groupa*:	You would ENTER:	• The following message displays:
a single task	task 6	**group delete** *groupa 6*	`Task: 6 was successfully deleted from group "groupa".`
a collection of tasks	task 6, 8, and 10	**group delete** *groupa 6 8 10*	`Task: 6 was successfully deleted from group "groupa". Task: 8 was successfully deleted from group "groupa". Task: 10 was successfully deleted from group "groupa".`
a range of tasks	tasks 6 through 10	**group delete** *groupa 6:10*	`Task: 6 was successfully deleted from group "groupa". Task: 7 was successfully deleted from group "groupa". Task: 8 was successfully deleted from group "groupa". Task: 9 was successfully deleted from group "groupa". Task: 10 was successfully deleted from group "groupa".`
a range of tasks	tasks 6 through 8	**group delete** *groupa 6-8*	`Task: 6 was successfully deleted from group "groupa". Task: 7 was successfully deleted from group "groupa". Task: 8 was successfully deleted from group "groupa".`

You can also use the **delete** action of the **group** subcommand to delete an entire task group. For example, to delete the task group *groupa*, you would:

ENTER **group delete** *groupa*

- The debugger deletes the task group.

Note: The pre-defined task group *all* cannot be deleted.

Changing the Name of a Task Group

To change the name of an existing task group, use the **change** action of the **group** subcommand. The syntax is:

group change *old_group_name new_group_name*

For example, say you want to change the name of task group *group1* to *groupa*. At the **pdbx** command prompt, you would:

ENTER **group change** *group1 groupa*

- The following message displays:

```
Group "group1" has been renamed to "groupa".
```

Listing Task Groups

To list task groups, their members, and task states use the **list** action of the **group** subcommand. The syntax is:

group list [*group_name*]

You can use the list action to:	For example, if you ENTER:	•

list all the task groups.	**group list**	The debugger displays a list of all existing task groups and their members. An example of such a list is shown below.
		`pdbx(0) group list`
		`allTasks 0:R 1:D 2:D 3:U 4:U 5:D 6:D`
		`7:D 8:D 9:D 10:D 11:D`
		`evenTasks 0:R 2:D 4:U 6:D 8:D 10:R`
		`oddTasks 1:D 3:U 5:D 7:D 9:D 11:R`
		`master 0:R`
		`workers 1:D 2:D 3:U 4:U 5:D 6:D 7:D`
		`8:D 9:D 10:R 11:R`

list all the members of a single task group	**group list** *oddTasks*	The debugger displays a list of all the members of task group *oddTasks*. `1:D 3:U 5:D 7:D 9:D 11:R`

When you list tasks, a single letter will follow each task number. The following table represents the state of affairs on the remote tasks. Common states are "debug ready," where **pdbx** commands can be issued, and running, where the application has control and is executing.

Table 3. Task States

This letter displayed after a task number:	Represents:	And indicates that:
N	Not loaded	the remote task has not yet been loaded with an executable.
S	Starting	the remote task is being loaded with an executable.
D	Debug ready	the remote task is stopped and debug commands can be issued.
R	Running	the remote task is in control and executing the program.
X	Exited	the remote task has completed execution.
U	Unhooked	the remote task is executing without debugger intervention.
E	Error	the remote task is in an unknown state.

Figure 2 on page 14 illustrates the relationship between the **pdbx** debugger, which runs on the home task, and the various **dbx** processes running on the remote tasks. When thinking about "task states," consider the perspective of the remote tasks which are each running a copy of **dbx**. **pdbx** attempts to coordinate activities in multiple **dbx** sessions. There are times when this is not possible, typically when one or more tasks have not yet stopped. In this case, from a remote task's **dbx** perspective, a **dbx** prompt has not yet been displayed, and your application is still running. Similarly, **pdbx** will not display a **pdbx** prompt until all the remote **dbx** sessions are "debug ready."

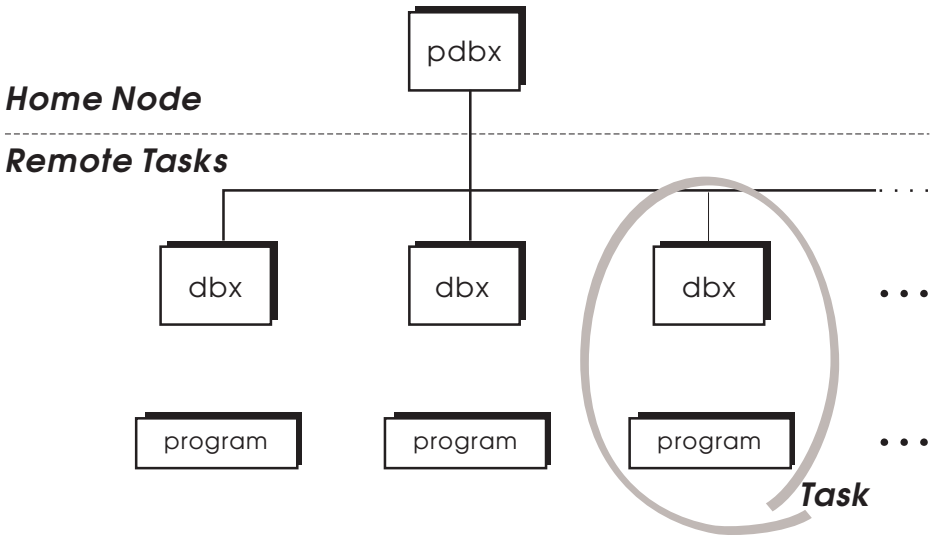


Figure 2. Relationship between home node (pdbx) and remote tasks (dbx processes)

Setting Command Context

You can set the current command context on a specific task or group of tasks so that the debugger directs subsequent context sensitive subcommands to just that task or group. This section also shows how you can temporarily deviate from the current command context you set.

Setting the Current Command Context

When you begin a **pdbx** session, the default command context is set on all tasks. The **pdbx** command prompt always indicates the current command context setting, so it initially reads:

```
pdbx(all)
```

You can use the **on** subcommand to set the current command context on one task or a group of tasks. The debugger then directs context sensitive subcommands to just the task(s) specified by group or task name.

You can use the **on** subcommand to set the current command context *before* you load your partition. The debugger will only direct context sensitive subcommands to the tasks in the current context. The syntax of the **on** subcommand is:

on {*group_name* | *task_id*}

For example, assume you have a parallel program divided into tasks numbered 0 through 4. To set the current command context on just task 1:

ENTER **on** *1*

- The **pdbx** command prompt indicates the new setting of the current command context.

```
pdbx(1)
```

You can also use the **on** subcommand to set the current command context on all the tasks in a specified task group. The task group *all* – consisting of all tasks – is

automatically defined for you and cannot be deleted. To set the command context back on all tasks, you would:

ENTER **on** *all*

- The **pdbx** command prompt shows that the current command context has changed, and that the debugger will now direct context sensitive subcommands to all tasks in the partition.

```
pdbx(all)
```

When you switch context using **on** *context_name*, and the new context has at least one task in the "running" state, a message is displayed stating that at least one task is in the "running" state. No **pdbx** prompt is displayed until all tasks in this context are in the "debug ready" state.

When you switch to a context where all tasks are in the "debug ready" state, the **pdbx** prompt is displayed immediately, indicating **pdbx** is ready for a command.

At the **pdbx** subset prompt, **on** *context_name* causes one of the following to happen: either a **pdbx** prompt is displayed; or a message is displayed indicating the reason why the **pdbx** prompt will be displayed at a later time. This is generally because one of the tasks is in "running" state. See "Context Switch when Blocked" on page 16 for more information.

Temporarily Deviating from the Current Command Context

There are times when it is convenient to deviate from the current command context for a single command. You can temporarily deviate from the command context by entering the **on** subcommand with, on the same line, a context sensitive subcommand. The **pdbx** prompt will be presented after all of the tasks in the temporary context have completed the command specified. It is possible, using <**Ctrl-c**> followed by the **back** or the **on** command, to issue further **pdbx** commands in the original context. The syntax is:

on {*group_name* | *task_id*} [*subcommand*]

For example, assume a task group named *groupa* contains tasks 3 through 5. The current command context is on this group. You want to set a breakpoint at line 99 of task 3 only, and then continue directing commands to all three members of *groupa*. One way to do this is to enter the three subcommands shown in the following example. This example shows the **pdbx** command prompt for additional illustration.

```
pdbx(groupa) on 3
pdbx(3) stop at 99
pdbx(3) on groupa
pdbx(groupa)
```

It is easier, however, to temporarily deviate from the current command context.

```
pdbx(groupa) on 3 stop at 99
pdbx(groupa)
```

The context sensitive **stop** subcommand is directed to task 3 only, but the current command context is unchanged. The next command entered at the **pdbx** command prompt is directed to all the tasks in the *groupa* task group.

At a **pdbx** prompt, you cannot use **on** *context_name pdbx_command* if any of the tasks in the specified context are running.

Context Switch when Blocked

When a task is blocked (there is no **pdbx** prompt), you can press <**Ctrl-c**> to acquire control. This displays the **pdbx** subset prompt `pdbx-subset([group | task])`, and provides a subset of **pdbx** functionality including:

- Changing the current context
- Displaying information about groups/tasks
- Interrupting the application
- Showing breakpoint/tracepoint status
- Getting help
- Exiting the debugger.

You can change the subset of tasks to which context sensitive commands are directed. Also, you can understand more about the current state of the application, and gain control of your application at any time, not just at user-defined breakpoints.

When a **pdbx** subset prompt is encountered, all input you type at the command line is intercepted by **pdbx**. All commands are interpreted and operated on by the home node. No data is passed to the remote nodes and standard input (STDIN) is not given to the application. Most commands in the **pdbx** subset produce information about the application and display the **pdbx** subset prompt. The exceptions are the **halt**, **back**, **on**, and **quit** commands. The **halt**, **back**, and **on** commands cause the **pdbx** prompt to be displayed when all of the tasks in the current context are in “debug ready” state.

The following example shows how the function works. A user is trying to understand the behavior of a program when tasks in the current context hang. This is a four task job with two groups defined called `low` and `high`. `Low` has tasks 0 and 1 while `high` has tasks 2 and 3. A breakpoint is set after a blocking read in task 2, and somewhere else in task 3. Group `high` is allowed to continue, and task 2 has a blocking read that will be satisfied by a write from task 0. Since task 0 is not executing, the job is effectively deadlocked and the **pdbx** prompt will not be displayed. The “effective deadlock” happens because the debugger controls some of the tasks that would otherwise be running. This could be called a debugger induced deadlock.

Using <**Ctrl-c**> allows the debugger to switch to task 0, then step past the write that satisfies the blocking read in task 2. A subsequent switch to group `high` shows task 2.

pdbx Subset Commands

The following table shows some commands that are uniquely available at the **pdbx** subset prompt, plus other **pdbx** commands that can be used. Certain commands are not allowed. The available commands keep the same command syntax as the **pdbx** subcommands (see “pdbx Subcommands” on page 1).

This subset command:	Is used to:	For more information see:
alias [alias_name string]	Set or display aliases.	"Creating, Removing, and Listing Command Aliases" on page 30
back	Return to a **pdbx** prompt.	"Returning to a pdbx Prompt" on page 17
group <action> [group_name] [task_list]	Manipulate groups. The actions are **add**, **change**, **delete**, and **list**. To indicate a range of tasks, enter the first and last task numbers, separated by a colon or dash. To indicate individual tasks, enter the numbers, separated by a space or comma.	"Grouping Tasks" on page 10
halt [all]	Interrupt all tasks in the current context that are running. If "all" is specified, all tasks, regardless of state, are interrupted. This command always returns to a **pdbx** prompt.	"Interrupting Tasks" on page 20
help [subject]	Display a list of **pdbx** commands and topics or help information about them.	"Accessing Help for pdbx Subcommands" on page 29
on <[group | task]>	Set the current context for later subcommands. This command always returns to a **pdbx** prompt.	"Setting Command Context" on page 14
source <cmd_file>	Execute subcommands stored in a file. **Note:** The file may contain context sensitive commands.	"Reading Subcommands From a Command File" on page 31
status [all]	Display the trace and stop events within the current context. If "all" is specified, all events, regardless of context, are displayed.	"Checking Event Status" on page 23
tasks [long]	Display processes (tasks) and their states.	"Displaying Tasks and their States" on page 9
quit	Exit the **pdbx** program and kill the application.	"Exiting pdbx" on page 34
unalias alias_name	Remove a previously defined alias.	"Creating, Removing, and Listing Command Aliases" on page 30
<**Ctrl-c**>	Has no effect, except to display the following message: `Typing Ctrl-c from the pdbx subset prompt has no effect. Use the halt command to interrupt the application. Use the quit command to quit pdbx. Type help then enter to view brief help of the commands available.`	"Context Switch when Blocked"

Returning to a pdbx Prompt

The **back** command causes the pdbx prompt to be displayed, when all the tasks in the current context are in "debug ready" state. You can use the **back** command if you want the application to continue as it was before <**Ctrl-c**> was issued. Also, you can use it if during subset mode all of the nodes are checked into debug ready state, and you want to do normal **pdbx** processing. The **back** command is only valid in **pdbx** subset mode.

It is also possible to return to the **pdbx** prompt using the **on** and the **halt** commands.

Controlling Program Execution

Like the **dbx** debugger, **pdbx** lets you set breakpoints and tracepoints to control and monitor program execution. *Breakpoints* are stopping places in your program. They halt execution, enabling you to then examine the state of the program. *Tracepoints* are places in the program that, when reached during execution, cause the debugger to print information about the state of the program. An occurrence of either a breakpoint or a tracepoint is called an *event*.

If you are already familiar with breakpoints and tracepoints as they are used in **dbx**, be aware that they work somewhat differently in **pdbx**. The subcommands for setting, checking, and deleting them are similar to their counterparts in **dbx**, but have been modified for use on parallel programs. These differences stem from the fact that they can now be directed to any number of parallel tasks.

This section describes how to:

- Set a breakpoint for tasks in the current context using the **stop** subcommand.
- Use the **halt** subcommand to interrupt tasks in the current context.
- Set a tracepoint for tasks in the current context using the **trace** subcommand.
- Use the **delete** subcommand to remove events for tasks in the current context.
- Use the **status** subcommand to display events set for tasks in the current context.

If you are already familiar with the **dbx** subcommands **stop**, **trace**, **status**, and **delete**, read the following as a discussion of how these subcommands are changed for **pdbx**.

The next few pages should act as an introduction to breakpoints and tracepoints if you are unfamiliar with **dbx**.

Refer to *IBM AIX Version 4 Commands Reference* and *IBM AIX Version 4 General Programming Concepts: Writing and Debugging Programs* for more information on subcommands.

Setting Breakpoints

The **stop** subcommand sets breakpoints for all tasks in the current context. When all tasks reach some breakpoint, execution stops and you can then examine the state of the program using other **pdbx** or **dbx** subcommands. These breakpoints can be different on each task.

The syntax of this context sensitive subcommand is:

stop if <*condition*>

stop at <*source_line_number*> [**if** <condition>]

stop in <*procedure*> [**if** <condition>]

stop <*variable*> [**if** <condition>]

stop <*variable*> **at** <*source_line_number*> [**if** <condition>]

stop <*variable*> **in** <*procedure*> [**if** <condition>]

Specifying **stop at** <*source_line_number*> causes the breakpoint to be triggered each time that source line is reached.

Specifying **stop in** <*procedure*> causes the breakpoint to be triggered each time the program counter reaches the first executable source line in the procedure (function, subroutine).

Using the <*variable*> argument to stop causes the breakpoint to be triggered when the contents of the variable changes. This form of breakpoint can be very time consuming. For better results, when possible, further qualify these breakpoints with a *source_line* or *procedure* argument.

Specify the <*condition*> argument using the syntax described by "Specifying Expressions" on page 31.

For example, to set a breakpoint **at** line 19 for all tasks in the current context, you would:

ENTER **stop at** *19*

- The debugger displays a message reporting the event it has built. The message includes the current context, the event ID associated with your breakpoint, and an interpretation of your command. For example:

  ```
  all:[0] stop at "ftoc.c":19
  ```

  The message reports that a breakpoint was set for the tasks in the task group *all*, and that the event ID associated with the breakpoint is *0*. Notice that the syntax of the interpretation is not exactly the same as the command entered.

Notes:

1. The **pdbx** debugger will not set a breakpoint at a line number in a group context if the group members have different current source files. Instead, the following error message will be displayed.

   ```
   ERROR: 0029-2081 Cannot set breakpoint or tracepoint event in
                    different source files.
   ```

   If this happens, you can either:

   - change the current context so that the **stop** subcommand will be directed to tasks with identical source files.
   - set the same source file for all members of the group using the **file** subcommand.

2. When specifying a variable name on the **stop** subcommand in **pdbx**, it is important to use fully-qualified names as arguments. See "Specifying Variables On the Trace and Stop subcommands" on page 21 for more information.

3. For further details on the **stop** subcommand, refer to its use on the **dbx** command as described in *IBM AIX Version 4 Commands Reference* and *IBM AIX Version 4 General Programming Concepts: Writing and Debugging Programs*.

Interrupting Tasks

By using the **halt** command, you interrupt all tasks in the current context that are running. This allows the debugger to gain control of the application at whatever point the running tasks happen to be in the application. To a **dbx** user, this is the same as using <**Ctrl-c**>. This command works at the **pdbx** prompt and at the **pdbx** subset prompt. If you specify "all" with the **halt** command, all running tasks, regardless of context, are interrupted.

Note: At a **pdbx** prompt, the **halt** command never has any effect without "all" specified. This is because by definition, at a **pdbx** prompt, none of the tasks in the current context are in "running" state.

The **halt all** command at the **pdbx** prompt affects tasks outside of the current context. Messages at the prompt show the task numbers that are and are not interrupted, but the **pdbx** prompt returns immediately because the state of the tasks in the current context is unchanged.

When using **halt** at the **pdbx** subset prompt, the **pdbx** prompt occurs when all tasks in the current context have returned to "debug ready" state. If some of the tasks in the current context are running, a message is presented.

Setting Tracepoints

The **trace** subcommand sets tracepoints for all tasks in the current context. When any task reaches a tracepoint, it causes the debugger to print information about the state of the program for that task.

The syntax of this context sensitive subcommand is:

trace [**in** <*procedure*>] [**if** <condition>]

trace <*source_line_number*> [**if** <condition>]

trace <*procedure*> [**in** <*procedure*>] [**if** <condition>]

trace <*variable*> [**in** <*procedure*>] [**if** <condition>]

trace <*expression*> **at** <*source_line_number*> [**if** <condition>]

Specifying **trace** with no arguments causes trace information to be displayed for every source line in your program.

Specifying **trace** <*source_line_number*> causes the tracepoint to be triggered each time that source line is reached.

Specifying **trace** [**in** <*procedure*>] causes the tracepoint to be triggered each time your program executes a source line within the procedure (function, subroutine).

Using the <*variable*> argument to trace causes the tracepoint to be triggered when the contents of the variable changes. This form of tracepoint can be very time consuming. For better results, when possible, further qualify these tracepoints with a *source_line_number* or *procedure* argument.

Specify the <*condition*> argument using the syntax described by "Specifying Expressions" on page 31.

The **trace** subcommand prints tracing information for a specified *procedure*, *function*, *sourceline*, *expression*, *variable*, or *condition*. For example, to set a tracepoint for the variable *foo* at line 21 for all tasks in the current context, you would:

ENTER **trace** *foo* **at** *21*

- The debugger displays a message reporting the event it has built. The message includes the current context, the event ID associated with your tracepoint, and an interpretation of your command. For example:

  ```
  all:[1] trace foo at "bar.c":21
  ```

  This message reports that the tracepoint was set for the tasks in the task group *all*, and that the event ID associated with the tracepoint is *1*. Notice that the syntax of the interpretation is not exactly the same as the command entered.

Notes:

1. The **pdbx** debugger will not set a tracepoint at a line number in a group context if the group members have different current source files. Instead, the following error message will be displayed.

   ```
   ERROR: 0029-2081 Cannot set breakpoint or tracepoint event in
                    different source files.
   ```

   If this happens, you can either:

   - change the current context so that the **trace** subcommand will be directed to tasks with identical source files.
   - set the same source file for all members of the group using the **file** subcommand.

2. When specifying a variable name on the **trace** subcommand in **pdbx**, it is important to use fully-qualified names as arguments. See “Specifying Variables On the Trace and Stop subcommands” for more information.
3. For further detail on the **trace** subcommand, refer to its use on the **dbx** command as described in *IBM AIX Version 4 Commands Reference*.

Specifying Variables On the Trace and Stop subcommands

When specifying a variable name as an argument on either the **stop** or **trace** subcommand, you should use fully-qualified names. This is because, when the **stop** or **trace** subcommand is issued, the tasks of your program could be in different functions, and the variable name may resolve differently depending on a task's position.

For example, consider the following SPMD code segment in *myfile.c*. It is running as two parallel tasks – task 0 and task 1. Task 0 is in *func1* at line 4, while task 1 is in *func2* at line 9.

```
1  int i;
2  func1()
3  {
4      i++;
5  }
6  func2()
7  {
8      int i;
9      i++;
10  }
```

To display the full qualification of a given variable, you use the **which** subcommand. For example, to display the full qualification of the variable *i* if the current context is *all*:

ENTER **which** *i*

- The **pdbx** debugger displays the full qualification of the variable specified.

```
0:@myfile.i          (from line 1 of previous example)
1:@myfile.func2.i      (from line 8 of previous example)
```

Because the tasks are at different lines, issuing the following **stop** command would set a different breakpoint for each task:

stop if (*i* == *5*)

The debugger would display a message reporting the event it has built.

```
all:[0] stop if (i == 5)
```

The *i* for task 0, however, would represent the global variable (*@myfile.i*) while the *i* for task 1 would represent the local variable *i* declared within *func2* (*@myfile.func2.i*). To specify the global variable *i* without ambiguity on the **stop** subcommand, you would:

ENTER **stop if** (*@myfile.i* **==** *5*)

- The debugger reports the event it has built.

```
all:[0] stop if (@myfile.i == 5)
```

Deleting pdbx Events

The **delete** subcommand removes events (breakpoints and tracepoints) of the specified **pdbx** event numbers. To indicate a range of events, enter the first and last event numbers, separated by a colon or dash. To indicate individual events, enter the numbers, separated by a space or comma. You can specify " * ", which deletes all events that were created in the current context. You can also specify "all", which deletes all events regardless of context. The syntax of this context sensitive subcommand is:

delete [*event_list* | * | **all**]

The event number is the one associated with the breakpoint or tracepoint. This number is displayed by the **stop** and **trace** subcommands when an event is built. Event numbers can also be displayed using the **status** subcommand. The output of the status command shows the creating context as the first token on the left before the colon.

Event numbers are unique to the context in which they were set, but not globally unique. Keep in mind that, in order to remove an event, the context must be on the appropriate task or task group, except when using the “all” keyword. For example, say the current context is on task 1 and the output of the **status** subcommand is:

```
1:[0] stop in celsius
all:[0] stop at "foo.c":19
all:[1] trace "foo.c":21
```

To delete all these events, you would do one of the following:

ENTER **on** *1*

delete *0*

on *all*

delete *0*,*1*

OR

ENTER **on** *1*

delete *0*

on *all*

delete ***

OR

ENTER **delete** *all*

Checking Event Status

A list of **pdbx** events can be displayed using the **status** subcommand. You can specify “all” after this command to list all events (breakpoints and tracepoints) that have been set in all groups and tasks. This is valid at the **pdbx** prompt and the **pdbx** subset prompt.

The following shows examples of **status**, **status all**, and incorrect syntax with different breakpoints set on three different groups and two tasks.

```
pdbx(all) status
all:[0] stop at "test/vtsample.c":60

pdbx(all) status all
1:[0] stop in main
2:[0] stop in mpl_ring
all:[0] stop at "test/vtsample.c":60
evenTasks:[0] stop at "test/vtsample.c":58
oddTasks:[0] stop at "test/vtsample.c":56

pdbx(all) status woops
0029-2062 The correct syntax is either 'status' or 'status all'.
```

Because the **status** command (without “all” specified) is context sensitive, it will not display status for events outside the context.

Unhooking and Hooking Tasks

The **unhook** subcommand lets you unhook a task so that it executes without intervention from the debugger. This subcommand is context sensitive and similar to the **detach** subcommand in **dbx**. The important difference is that you can regain control over a task that has been unhooked, while you cannot regain control over one that has been detached. To regain control over an unhooked task, use the **hook** subcommand. **Detach** is not supported in **pdbx**.

To better understand the **hook** and **unhook** subcommands, consider the following example. You are debugging a typical master/worker program containing many blocking sends and receives. You have created two task groups. One – named *workers* – contains all the worker tasks, and the other – named *master* – contains the master task. You would like to manipulate the master task and let the worker tasks process without debugger interaction. This would save you the bother of switching the command context back and forth between the two task groups.

Since the **unhook** subcommand is context sensitive, you must first set the context on the *workers* task group using the **on** subcommand. At the **pdbx** command prompt:

ENTER **on** *workers*

- The debugger sets the command context on the task group *workers*.

ENTER **unhook**

- The debugger unhooks the tasks in the task group *workers*.

The worker tasks are still indirectly affected by the debugger since they might, for example, have to wait on a blocking receive for a message from the master task. However, they do execute without any direct interaction from the debugger. If you later wish to reestablish control over the tasks in the *workers* task group, you would, assuming the context is on the *workers* task group:

ENTER **hook**

- The debugger hooks any unhooked task in the current command context.

Note: The **hook** subcommand is actually an interrupt. When you interrupt a blocking receive, you cause the request to fail. If the program does not deal with an interrupted receive, then data loss may occur.

Examining Program Data

The following section explains the **where**, **print**, and **list** subcommands for displaying and verifying data.

Viewing Program Call Stacks

The **where** subcommand displays a list of active procedures and functions.

The syntax of this context sensitive subcommand is:

where

To view the stack trace, issue the **where** command. The following stack trace was encountered after halting task 1. You can see that the main routine at line 144 has issued an **mpi_recv()** call.

```
pdbx(1) where
read(??, ??, ??) at 0xd07b5ce0
readsocket() at 0xd07542f4
kickpipes() at 0xd0750e14
mpci_recv() at 0xd076032c
_mpi_recv() at 0xd0700e2c
MPI__Recv() at 0xd06ffab8
mpi__recv() at 0xd03c4474
main(), line 144 in "send1.f"
```

Viewing Program Variables

The **print** subcommand does either of the following:

- Prints the value of a list of expressions, specified by the *expression* parameters.
- Executes a procedure, specified by the *procedure* parameter, and prints the return value of that procedure. Parameters that are included are passed to the procedure.

The syntax of this context sensitive subcommand is:

print *expression* ...

print *procedure* ([*parameters*])

See “Specifying Expressions” on page 31 for a description of valid expressions.

Following are some examples of printing portions of a two dimensional array of *floats* in a c program which is running on two nodes.

To display the type of array `ff`, enter:

```
pdbx(all) whatis ff
  0:float ff[10][10];
  1:float ff[10][10];
```

We can see the differences in the array values across the two nodes.

To show elements 4 through 7 of rows 2 and 3, enter:

```
pdbx(all) print ff[2..3][4..7]
  0:[2][4] = 30.0000076
  0:[2][5] = 42.0
  0:[2][6] = 0.0
  0:[2][7] = -3.52516241e+30
  0:[3][4] = -3.54361545e+30
  0:[3][5] = -3.60971468e+30
  0:[3][6] = 2.68063283e-09
  0:[3][7] = 4.65661287e-10
  0:
  1:[2][4] = -1.60068157e+10
  1:[2][5] = 0.0
  1:[2][6] = 0.0
  1:[2][7] = -3.52516241e+30
  1:[3][4] = -3.54361545e+30
  1:[3][5] = -3.60971468e+30
  1:[3][6] = 2.63675126e-09
  1:[3][7] = 1.1920929e-07
  1:
```

The same results as above could be achieved by entering:

```
print ff(2..3,4..7)
```

The array `ff` is being processed within a loop with loop counters `i` and `j`. The following demonstrates printing multiple variables and using program variables to specify the array elements.

```
pdbx(all) print "i is:", i, "\tj is:", j, "\n", ff[i][j..j+1]
  1:i is: 0     j is: 1
  1: [0][1] = -3.54331806e+30
  1:[0][2] = 4.40487202e-10
  1:
  0:i is: 2     j is: 6
  0: [2][6] = 0.0
  0:[2][7] = -3.52516241e+30
  0:
```

Following are some examples which display the elements of an array of *structs*:

The command **whatis** here is used to show that the type of the variable `tree` is an array size `4` of `wood_attr_t`'s.

```
pdbx(0) whatis tree
  0:wood_attr_t tree[4];
```

Here the **whatis** command shows that `wood_attr_t` is a `typedef` for the listed structure.

```
pdbx(0) whatis wood_attr_t
  0:typedef struct {
  0:    int max_age;
  0:    int max_size;
  0:    int is_hard_wood;
  0:} wood_attr_t;
```

This **whatis** command shows that `this_tree` is a `wood_attr_t` ptr.

```
pdbx(0) whatis this_tree

  0:wood_attr_t *this_tree;
```

To display the elements of the first three entries in the tree array, enter:

```
pdbx(0) print tree[0..2]
  0:[0] = (max_age = 150, max_size = 120, is_hard_wood = 0)
  0:[1] = (max_age = 250, max_size = 150, is_hard_wood = 1)
  0:[2] = (max_age = 200, max_size = 125, is_hard_wood = 0)
  0:
```

To display the element `max_size` of entry 1 of the `tree` array, enter:

```
pdbx(0) p tree[1].max_size
  0:150
```

To display the entry that `this_tree` is pointing to, enter:

```
pdbx(0) p *this_tree
  0:(max_age = 200, max_size = 125, is_hard_wood = 0)
```

To display just the `max_size` of the entry that `this_tree` is pointing to, enter:

```
pdbx(0) p this_tree->max_size
  0:125
```

Following are some examples of displaying elements of a two dimensional array of *reals* in a Fortran program:

To take a look at the type of `var43`:

```
pdbx(all) whatis var43
 real*4  var43(4,3)
```

To display the entire array `var43`, enter:

```
pdbx(all) print var43
(1,1)   11.0
(2,1)   21.0
(3,1)   31.0
(4,1)   41.0
(1,2)   12.0
(2,2)   22.0
(3,2)   32.0
(4,2)   42.0
(1,3)   13.0
(2,3)   23.0
(3,3)   33.0
(4,3)   43.0
```

To display a portion of the array `var43`, enter:

```
pdbx(all)  print var43(1..2, 2..3)
(1,2) = 12.0
(2,2) = 22.0
(1,3) = 13.0
(2,3) = 23.0
```

Refer to *IBM AIX Version 4 General Programming Concepts: Writing and Debugging Programs* for more information on expression handling.

Displaying Source

The **list** subcommand displays a specified number of lines of the source file. The number of lines displayed is specified in one of two ways:

Tip: Use **on** <*task*> **list**, or specify the ordered standard output option.

- By specifying a procedure using the *procedure* parameter.

  In this case, the **list** subcommand displays lines starting a few lines before the beginning of the specified procedure and until the list window is filled.

- By specifying a starting and ending source line number using the *sourceline-expression* parameter.

  The *sourceline-expression* parameter should consist of a valid line number followed by an optional + (plus sign), or – (minus sign), and an integer. In addition, a *sourceline* of $ (dollar sign) can be used to denote the current line number. A *sourceline* of @ (at sign) can be used to denote the next line number to be listed.

  All lines from the first line number specified to the second line number specified, inclusive, are then displayed, provided these lines fit in the list window.

  If the second source line is omitted, 10 lines are printed, beginning with the line number specified in the *sourceline* parameter.

  If the **list** subcommand is used without parameters, the default number of lines is printed, beginning with the current source line. The default is 10.

To change the number of lines to list by default, set the special debug program variable, *$listwindow*, to the number of lines you want. Initially, *$listwindow* is set to 10.

The syntax of this context sensitive subcommand is:

list [*procedure* | *sourceline-expression*[,*sourceline-expression*]]

Other Key Features

Some other features offered by **pdbx** include the following subcommands:

- **help**
- **dhelp**
- **alias**
- **source**

Also, this section includes information about how to specify expressions for the **print**, **stop**, and **trace** commands.

Accessing Help for pdbx Subcommands

The **help** command with no arguments displays a list of **pdbx** commands and topics about which detailed information is available.

If you type “help” with one of the **help** commands or topics as the argument, information will be displayed about that subject.

The syntax of this context insensitive command is:

help [*subject*]

Accessing Help for dbx Subcommands

The **dhelp** command with no arguments displays a list of **dbx** commands about which detailed information is available.

If you type “dhelp” with an argument, information will be displayed about that command.

Note: The partition must be loaded before you can use this command, because it invokes the **dbx help** command. It is also required that a task be in “debug ready” state to process this command. After the program has finished execution, the **dhelp** command is no longer available.

The syntax of this context insensitive command is:

dhelp [**dbx_command**]

Creating, Removing, and Listing Command Aliases

The **alias** subcommand specifies a command alias. You could use it to reduce the amount of typing needed, or to create a name more easily remembered. The syntax of this context insensitive subcommand is:

alias [*alias_name* **[***alias_string***]]**

For example, assume that you have organized all tasks into two convenient groups – *master* and *workers*. During the execution of a program, you need to switch the command context back and forth between these two groups. You could save yourself some typing by creating one alias for *on workers* and one for *on master*. At the **pdbx** command prompt, you would:

ENTER **alias** *mas on master*
alias *wor on workers*

Now to set the command context on the task group *master*, all you have to do is:

ENTER **mas**

Likewise, you can now enter **wor** instead of **on** *workers*.

In addition to any aliases you create, there are a number of aliases supplied by **pdbx** when the partition is loaded. To display the list of all existing aliases, use the **alias** subcommand with no parameters. At the **pdbx** command prompt:

ENTER **alias**

- The debugger displays a list of existing aliases. The example listing below shows all the default aliases provided by **pdbx**, as well as the two aliases – *mas* and *wor* – created in the previous example.

```
t                 where
j                 status
st                stop
s                 step
x                 registers
q                 quit
p                 print
n                 next
m                 map
l                 list
h                 help
d                 delete
c                 cont
mas               on master
wor               on workers
th                thread
mu                mutex
cv                condition
attr              attribute
active            tasks
threads           thread
```

Any aliases you create are not saved between **pdbx** sessions. You can also remove command aliases using the **unalias** subcommand. The syntax of this context insensitive subcommand is:

unalias *alias_name*

For example, to remove the alias *mas* defined above, you would:

ENTER **unalias** *mas*

Note: You can create, remove, and list command aliases as soon as you start the debugger. The partition does not need to be loaded.

Reading Subcommands From a Command File

The **source** subcommand enables you to read a series of subcommands from a specified command file. The syntax of this context-insensitive subcommand is:

source *command_file*

The *command_file* should reside on the home node, and can contain any of the subcommands that are valid on the **pdbx** command line. For example, say you have a commands file named *myalias* which contains a number of command alias settings. To read its commands:

ENTER **source** *myalias*

- The debugger reads the commands listed in *myalias* as if they had each been entered at the command line.

Notes:

1. You can also read commands from a file when starting the debugger. This is done using the **-c** flag on the **pdbx** command, or via a *.pdbxinit* file, as described in Table 2 on page 5. The *.pdbxinit* file would be a great way to automatically create your common aliases. When using a *.pdbxinit* file or the **-c** flag, you need to keep in mind that only a limited set of commands are supported until the partition is loaded.
2. STDIN cannot be included in a command file.

Specifying Expressions

Expressions are commonly used in the **print** command, and when specifying conditions for the **stop** or **trace** command.

You can specify conditions with a subset of C syntax, with some Fortran extensions. The following operators are valid:

Arithmetic Operators

+	Addition
-	Subtraction
-	Negation
*****	Multiplication
/	Floating point division
div	Integer division
mod	Modulo
exp	Exponentiation

Relational and Logical Operators

<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equal to
=	Equal to
!=	Not equal to
< >	Not equal to
||	Logical OR
or	Logical OR
&&	Logical AND
and	Logical AND

Bitwise Operators

bitand	Bitwise AND
|	Bitwise OR
xor	Bitwise exclusive OR
~	Bitwise complement
<<	Left shift
>>	Right shift

Data Access and Size Operators

[]	Array element
()	Array element
*	Indirection or pointer dereferencing
&	Address of a variable
.	Member selection for structures and unions
.	Member selection for pointers to structures and unions
->	Member selection for pointers to structures and unions
sizeof	Size in bytes of a variable

Miscellaneous Operators

()	Operator grouping
(Type)Expression	Type cast
Type(Expression)	Type cast
Expression\Type	Type cast

Other Important Notes on pdbx

Initial Breakpoint

The initial automatic breakpoint, which is set by default at function main, for **pdbx** can be redefined by the environment variable **MP_DEBUG_INITIAL_STOP**. See the manual page for the **pdbx** command in Appendix A, “Parallel Environment Tools Commands” on page 235 for more information.

Overloaded Symbols

While **pdbx** recognizes function names, it is the combination of a function's name and its parameters, or the function name and the shared object it resides in, that uniquely identify it to **pdbx**. When encountering ambiguous functions, **pdbx** issues the Select menu, which lets the user choose the desired instance of the function.

The Select menu looks like this:

```
pdbx(all) stop in f1
1.ambig.f1(double)
2.ambig.f1(float)
3.ambig.f1(char)
4.ambig.f1(int)
Select one or more of [1 - 4]:
```

The **whatis** subcommand can be used to determine whether or not a function is ambiguous. If **whatis** returns more than one function definition for a given symbol, **pdbx** will consider it ambiguous.

There are a few restrictions for the **pdbx** select menu:

- All tasks in the context must have an identical view of the ambiguous function because **pdbx** will only present one menu to the user that covers all tasks. As a result, you my need to create additional groups. The view of the ambiguous function is determined by the result of the **whatis** subcommand. In the example above, *whatis f1* should have returned the same result on all tasks, in order to proceed.
- The **hook** subcommand will not restore the set of events generated by the Select menu.
- The **trace** and **print** subcommands do not support ambiguous functions within complex expressions. For example, simple expressions are always allowed:

  ```
  trace myfunc

  print myfunc(parm1, parm2)
  ```

  but complex expressions are not allowed when a function (myfunc) is ambiguous:

  ```
  trace myvar-myfunc(parm1, parm2)

  print myvar*myfunc(parm1)
  ```

Exiting pdbx

It is possible to end the debug session at any time using either the **quit** subcommand, or the **detach** subcommand if debugging in attach mode.

To end a debug session in normal mode:

ENTER **quit**

- This returns you to the shell prompt.

To end a debug session in attach mode, you can choose either **quit** or **detach**. Quitting causes the debugger and all the members of the original **poe** application partition to exit. Detaching causes only the debugger to exit and leaves all the tasks running.

ENTER **quit**

- The debugger session ends, along with the **poe** application partition tasks.

OR

ENTER **detach**

- The debugger session ends. All tasks have been detached, but stay running.

Note: You can enter the **quit** and **detach** subcommands from either the **pdbx** prompt or **pdbx** subset prompt.

Choosing **detach** causes **pdbx** to exit, and allows the program to which you had attached to continue execution if it hasn't already finished. If this program has finished execution, and is part of a series of job steps, then detaching allows the next job step to be executed.

If instead you want to exit the debugger and end the program, choose **quit** as described above.

Chapter 2. Using the pedb Debugger

This chapter describes the **pedb** debugger. The **pedb** debugger provides a simplified, Motif graphical point-and-click interface. **pedb** is designed to debug parallel C or Fortran applications. The **pedb** debugger is a **poe** application with some modifications on the *home node* to provide a user interface. This means that most of the setup for the debugger is identical to the setup for **poe**.

pedb can be used to debug an application either by starting the application under the control of the debugger, or by attaching to an already running **poe** application.

If starting the application under the control of the debugger, it is first necessary to compile the program and set the execution environment. See *IBM Parallel Environment for AIX: Operation and Use, Volume 1, Using the Parallel Operating Environment* for more information on the following:

- Compiling the program. Be sure to specify the **-g** flag when compiling the program. This produces an object file with symbol table references needed for symbolic debugging. It is also advisable to not use the optimization option, **-O**. Using the debugger on optimized code may produce inconsistent and erroneous results. For more information on the **-g** and **-O** compiler options, refer to their use on other compiler commands such as **cc** and **xlf**. These compiler commands are described in *IBM AIX Version 4 Commands Reference*, or your online manual pages.
- Copying files to individual nodes. Like **poe**, **pedb** requires that your application program be available to run on each node in your partition. To support source level debugging, **pedb** requires the source files to be available too, but they are only required on the home node.
- Setting up the execution environment.

If using **pedb** to attach to an application, much of the setup described above is not necessary since the application is already running. However, it is still highly desirable, but not absolutely necessary, to have the application compiled with the **-g** option. When **pedb** attaches to an application that is not compiled with **-g**, the debug information is limited to a stack trace.

As you read these steps, keep in mind that **pedb** accepts almost all the option flags that **poe** accepts, and responds to almost all of the same environment variables.

This release of **pedb** does not support the debugging of applications that were compiled with previous releases of **poe**.

Starting the pedb Debugger

You can start the **pedb** debugger in either *normal* mode or *attach* mode. In normal mode your program runs under the control of the debugger. In attach mode you attach to a program that is already running. Certain options and functions are only available in one of the two modes. Since **pedb** is a source code debugger, some files need to be compiled with the **-g** option so that the compiler provides debug symbols, source line numbers, and data type information. When the application is started using **pedb**, debugger control of the application is given to the user by default at the first executable source line within the main routine. This is function

© Copyright IBM Corp. 1996, 1997

main in C code or the the routine defined by the *program* statement in Fortran. In Fortran, if there is no *program* statement, the program name defaults to *main*. If the file containing the main routine is not compiled with **-g** the debugger will exit. The environment variable **MP_DEBUG_INITIAL_STOP** can be set before starting the debugger to manually set an alternate file name and source line where the user initially receives debugger control of the application. Refer to the appendix on POE environment variables and command-line flags in *IBM Parallel Environment for AIX: Operation and Use, Volume 1, Using the Parallel Operating Environment*.

Normal Mode

The way you start the debugger in normal mode depends on whether the program(s) you are debugging follow the SPMD or MPMD model of parallel programming. In the SPMD model, the same program runs on each of the nodes in your partition. In the MPMD model, different programs can run on the nodes of your partition.

If you are debugging an SPMD program, you can enter its name on the **pedb** command line. It will be loaded on all the nodes of your partition automatically. If you are debugging an MPMD program, you will load the tasks of your partition after the debugger is started.

ENTER **pedb** [[*program*] *program options*] [*poe options*] [*X options*] [[**-I** *source directory*]...] [**-d** *nesting depth*] [**-x**]

- This starts **pedb**. You will see the **pedb** main window open. If you specified a *program*, it is loaded on each node of your partition and you see the message:

```
0030-0101 Partition loaded.
```

ENTER **pedb -a** *poe process id* [*limited poe options*] [*X options*] [[**-I** *source directory*]...] [**-d** *nesting depth*] [**-x**]

- This starts **pedb** in attach mode. See “Attach Mode” on page 37 for more information.

ENTER **pedb -h**

- This writes the **pedb** usage to STDERR. It includes **pedb** command line syntax and a description of **pedb** options.

The options you specify with the **pedb** command can be program options, POE options, or **pedb** options listed in Table 4 on page 37. Program options are those that your application program will understand. You can also specify certain X-Windows options with the **pedb** command.

For the most part, you can use the same command-line flags on the **pedb** command as you use when invoking a parallel program using the **poe** command. For example, you can override the **MP_PROCS** variable by specifying the number of processes with the **-procs** flag. Or you could use the **-hostfile** flag to specify the name of a host list file. For more information on the POE command-line flags, see *IBM Parallel Environment for AIX: Operation and Use, Volume 1, Using the Parallel Operating Environment*.

Table 4. Debugger Option Flags (pedb)

Use this flag:	To:	For example:
-a	Attach to a running **poe** job by specifying its process id. This must be executed from the node where the **poe** job was initiated. When using the debugger in attach mode there are some debugger command line arguments that should not be used. In general, any arguments that control how the partition is set up or specify application names and arguments should not be used.	To start **pedb** in attach mode: **ENTER** **pedb -a** *<poe PID>*
-d	Set the limit for the nesting of program blocks. The default nesting depth limit is 25.	To specify a nesting depth limit: **ENTER** **pedb -d** *nesting.depth*
-h	Write the **pedb** usage to STDERR then exit. This includes **pedb** command line syntax and a description of **pedb** flags.	To write the **pedb** usage to STDERR: **ENTER** **pedb -h**
-I (upper case i)	Specify a directory to be searched for an executable's source files. This flag must be specified multiple times to set multiple paths. (Once **pedb** is running, this list can also be updated using the Update Source Path window.)	To add *directory1* to the list of directories to be searched when starting the **pedb** debugger: **ENTER** **pedb -I** *dir1* You can add as many directories as you like to the directory list in this way. For example, to add two directories: **ENTER** **pedb -I** *dir1* **-I** *dir2*
-x	Prevents stripping _ (trailing underscore) characters from symbols originating in Fortran source code. This enables distinguishing between symbols which are identical except for an underscore character, such as xxx and xxx_.	To prevent trailing underscores from being stripped from symbols in Fortran source code: **ENTER** **pedb -x**

Attach Mode

The **pedb** debugger provides an attach feature, which allows you to attach the debugger to a parallel application that is currently executing. This feature is typically used to debug large, long running, or apparently “hung” applications. The debugger attaches to any subset of tasks without restarting the executing parallel program.

Parallel applications run on the partition managed by **poe**. For attach mode, you must specify the appropriate process identifier (PID) of the **poe** job, so the debugger can attach to the correct application processes contained in that particular job. To get the PID of the **poe** job, enter the following command on the node where **poe** was started:

```
$ ps -ef | grep poe
```

You initiate attach mode by invoking **pedb** with the **-a** flag and the PID of the appropriate **poe** process:

```
$ pedb -a <poe PID>
```

For example, if the process id of the **poe** is `12345` then the command would be:

```
$ pedb -a 12345
```

pedb starts by showing a list of task numbers that comprise the parallel job. The debugger obtains this information by reading a configuration file created by **poe** when it begins a job step. At this point you must choose a subset of that list to attach the debugger. Once you make a selection and the attach debug session starts, you cannot make additions or deletions to the set of tasks attached to. It is

possible to attach a different set of tasks by detaching the debugger and attaching again, then selecting a different set of tasks.

Note: The debugger supports up to 24 nodes. When attaching to jobs larger than 24 nodes, it is suggested you select a subset of tasks less than or equal to 24.

The debugger attaches to the specified tasks. The executable is stopped wherever its program counter happens to be, and is then under the control of the debugger. The other tasks in the original **poe** application continue to run. **pedb** displays information about the attached tasks using the task numbering of the original **poe** application partition.

Attach Window

Figure 3 shows the **pedb** Attach window.

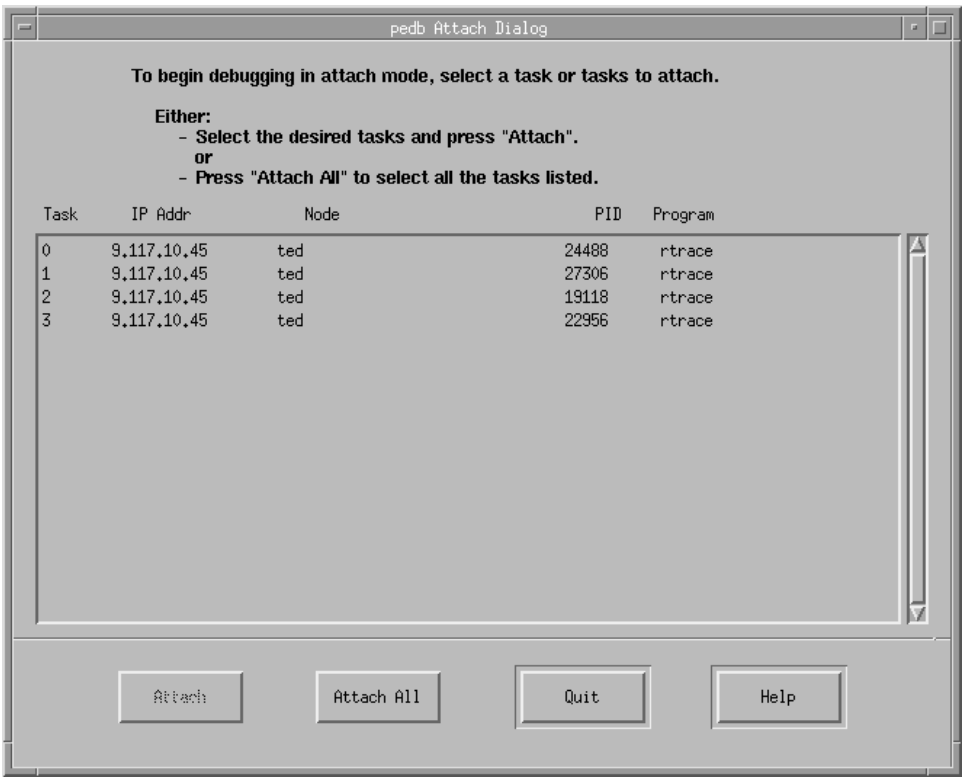


Figure 3. pedb Attach window

The **pedb** Attach window contains a list of tasks and, for each task, the following information:

- Task - the task number
- IP - the ip address of the node on which the task/application is running
- Node - the name of the node on which the task/application is running, if available
- PID - the process identifier of the task/application
- Program - the name of the application and arguments, if any.

At the bottom of the window there are four buttons:

- Attach - causes the debugger to attach to the tasks you select. It remains grayed out until you make a selection.
- Attach All - causes the debugger to attach to all tasks listed in the window. You do not need to make any specific selections.
- Quit - closes the Attach window and exits the debugger, leaving the **poe** job running.
- Help - accesses help information about the Attach window.

At this point you can select a set of tasks to which the debugger attaches:

PRESS **Attach All** to select all tasks

OR

SELECT individual tasks by holding down the **Ctrl** key and clicking with the left mouse button.

PRESS **Attach**

- The window closes and the **pedb** Main Window appears.

Task buttons appear for each task selected for debugging. Once the debugger attaches to the selected tasks, the buttons change from a label of “UA” (unattached) to “D” (debug state), and from the default color of “wheat” to “green.”

The default group button is labelled “Attached” and consists of all the tasks chosen for attach.

When starting the debugger in attach mode, the default context is “Attached,” as indicated at the top of the main window:

```
pedb: View - 1, Context - Group Attached
```

Other Compiling Options

pedb provides substantial information when debugging an executable compiled with the **-g** option. However, you may find it useful to attach to an application not compiled with **-g**. **pedb** allows you to attach to an application not compiled with **-g**, however, the information provided is limited to a stack trace.

You can also attach **pedb** to an application compiled with both the **-g** and optimization flags. However, the optimized code can cause some confusion when debugging. For example, when stepping through code, you may notice the line marker points to different source lines than you would expect. The optimization causes this re-mapping of instructions to line numbers.

Command Line Arguments

You should not use certain command line arguments when debugging in attach mode. If you do the debugger will not start, and you will receive a message saying the debugger will not start. In general, do not use any arguments that control how the debugger partition is set up or that specify application names and arguments. You do not need information about the application, since it is already running and the debugger uses the PID of the **poe** process to attach. Other information the debugger needs to set up its own partition, such as node names and PIDs, comes from the configuration file and the set of tasks you select. See Appendix B, “Command Line Flags for Normal or Attach Mode” on page 283 for a list of

command line flags showing which ones are valid in normal and in attach debugging mode.

The information in the appendix is also true for the corresponding environment variables, however **pedb** ignores the invalid setting. The debugger displays a message containing a list of the variables it ignores, and continues.

For example, if you had **MP_PROCS** set, when the debugger starts in attach mode it ignores the setting. It displays a message saying it ignored **MP_PROCS**, and continues initializing the debug session.

The pedb Main Window

As mentioned previously, you have the option of specifying the name of your application when you invoke **pedb** which causes it to be loaded on all the tasks automatically. This method is generally used to debug SPMD codes. If you need to load an MPMD code, or prefer to use the file selection window to load your partition you should not specify your application name on the **pedb** command line.

The initial **pedb** window you see will have blank areas as illustrated in Figure 4 on page 41. If you specified an application name on the command line, the debugger will continue, by loading your application for each task which will fill in the main window as illustrated Figure 6 on page 45.

Following is a brief overview of the **pedb** main window layout.

- Across the top is a menu bar, which contains the functions you will need for debugging.

  SELECT **File → Load Executables ...** to choose a program to debug.

  SELECT **Find → Find in Source Window** to position source by search strings.

- The left half of the screen contains the source code window and the **pedb** control buttons.
  - Double click on a source line to set a breakpoint.
  - Control execution of your application with the buttons below the source code.
- Application data by task is shown in the windows on the right side of the display.
  - Global variable on request.
  - Variables local to the current block (or stack frame).
  - Calling stack listing.
  - Threads listing.
  - Event data (Break and Trace points).
- Context selection buttons at the lower right
- At the bottom, a message window.

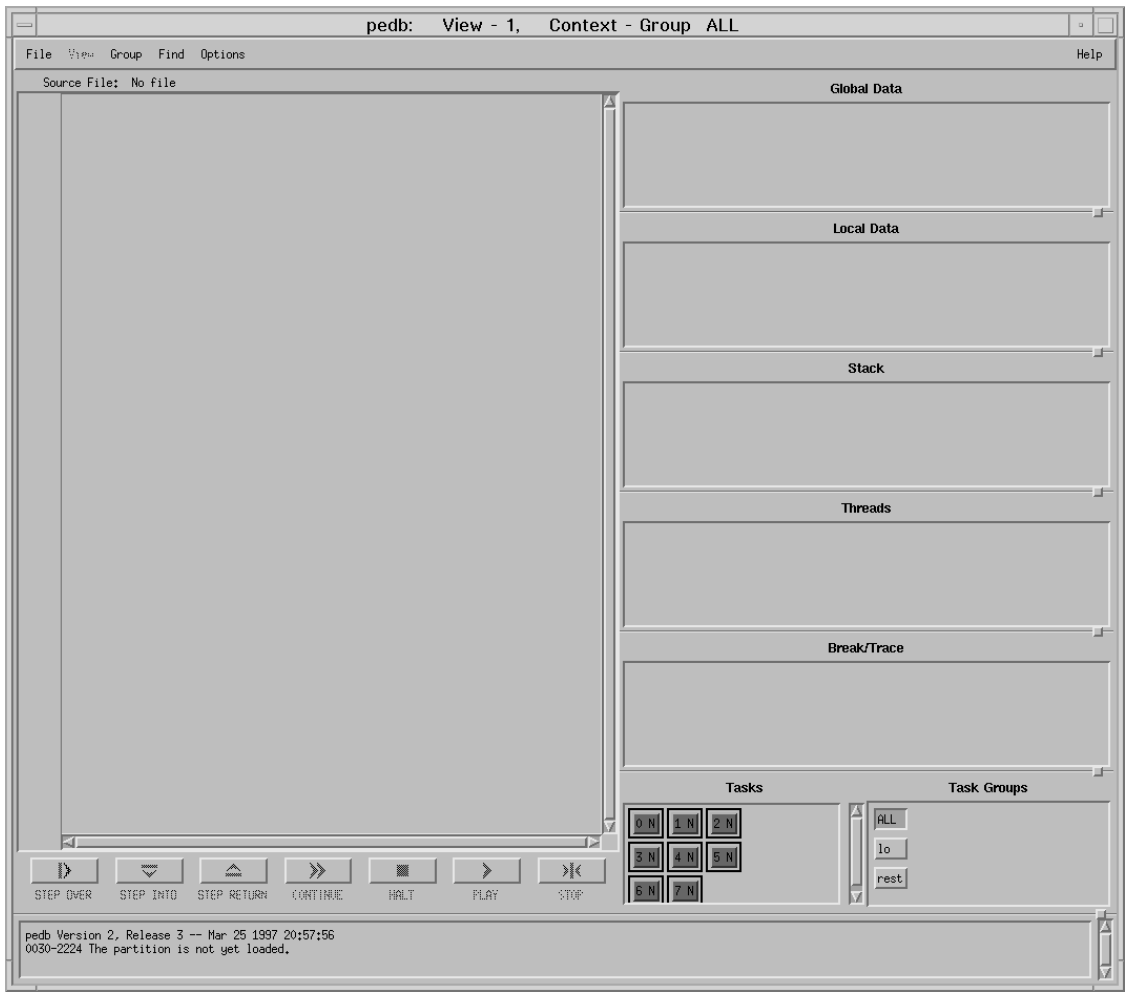


Figure 4. pedb main window before the partition is loaded

If you didn't do an SPMD load from the **pedb** command line, the initial screen opens with many options unavailable. For example, the View option and control buttons are inactive. These options will become available after all the tasks have been loaded.

During this initial loading phase, you can:

- create or delete groups
- load programs, tasks, or groups
- set a different context
- get help
- select hide/show options
- update the source path
- change context to group or task
- quit **pedb**

Loading the Partition from the Load Executables Window

If you did not specify a program to load on the **pedb** command line, you will use the Load Executables window. In this case, a partition has been created to support the number of tasks that were defined for the application. In general, the term *task* refers to an individual program that is part of a parallel application. The number of tasks was determined by the value of the **MP_PROCS** environment variable, or the value specified by the **-procs** flag, if entered on the command line when invoking **pedb**.

A partition may be thought of as a system of one or more physical processor nodes, along with the infrastructure necessary to execute a parallel application. When you load a partition, you provide programs for the infrastructure to run.

When you specify a program to run by invoking **pedb** with a program name on the command line, it assumes an SPMD model and automatically loads all tasks with this program. With the Load Executables window however, you also have the ability to load different executables on different tasks or groups of tasks (as in the case of an MPMD application), or to load the same executable on all tasks in the instance when the file is not located in a shared file system or in the same directory on all tasks. You can load programs one task at a time by selecting a different button in the Tasks area before opening the Load Executables window. You can also load a subset of all the tasks at one time by first creating the desired task group(s), and then selecting the corresponding group button in the Task Groups area before using the Load Executables window.

Program Search Path: Like **POE**, **pedb** uses the normal shell search path that is established by the environment variable **PATH** if you don't explicitly give a path. **pedb** checks this path and loads the first occurrence of the program you specify (scanning from left to right) for each task. The mechanism for finding source files is different from this. See “Source Code Search Path” on page 85 for information on the source code search path.

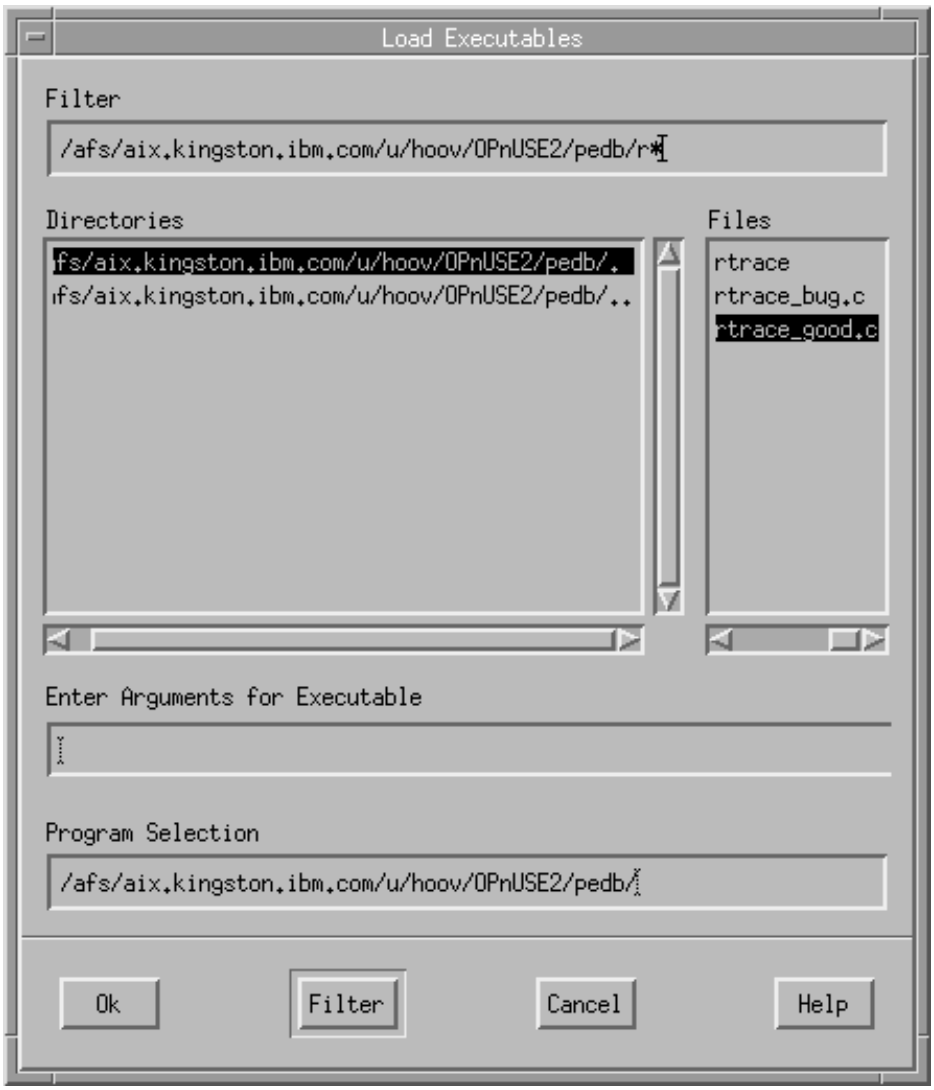
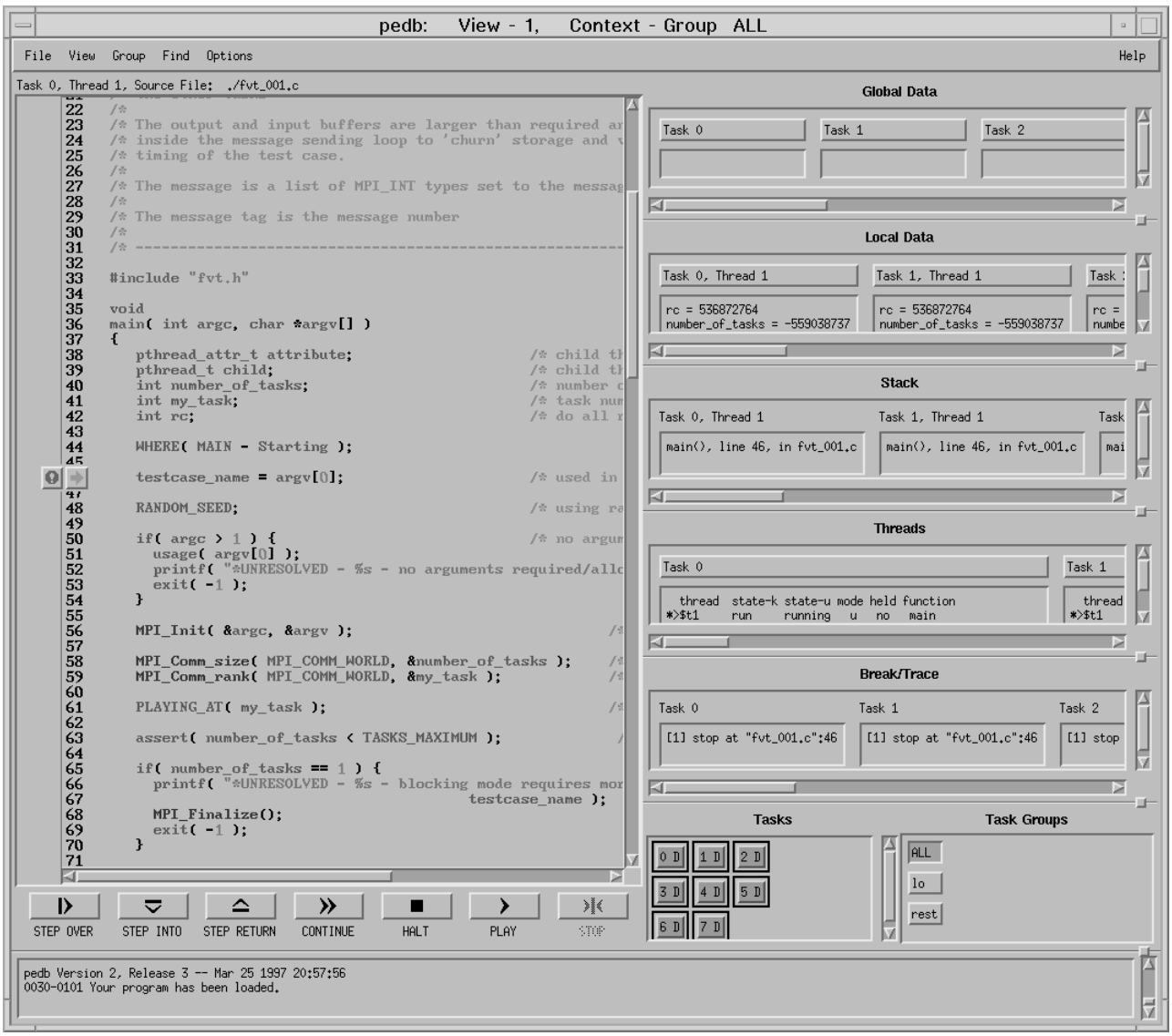


Figure 5. Load Executables window

To load the same executable for all tasks (SPMD):	To load different executables (MPMD):
CHECK the title bar of the **pedb** window to make sure the context is set to all tasks. This is the default when you start the **pedb** debugger. If the context is not on the task group *ALL*, reset it. To set the context on all tasks: **PRESS** the task group button labeled *ALL* in the Task Groups Area. Once the context is set on all tasks: **SELECT** **File → Load Executables ...** • The Load Executables window opens. This window allows you to choose the appropriate directory and select the corresponding executable file. **TYPE IN** any command line arguments to the executable selected for loading. **SELECT** the directory and executable file you want to load by clicking on each name. Double clicking on the file name automatically loads the program. **PRESS** **OK** • The Load Executables window closes, and the specified program is loaded for all tasks. Each task stops at the first executable source line. **MP_DEBUG_INITIAL_STOP** can be set to override the default of the first executable source line in *main()*. Set **MP_DEBUG_INITIAL_STOP** to the *file: linenumber*. **DISPLAYS MESSAGE** 0030-0101 Partition Loaded	Set the context before loading each program. For example, suppose there will be five tasks numbered 0 through 4. To load a program for task 0 and a program for tasks 1 through 4, you would: **PRESS** the task button labeled *0* in the Task Area. **SELECT** **File → Load Executables ...** • The Load Executables window opens. This window allows you to choose the appropriate directory and select the corresponding executable file. **SELECT** the directory and executable file you want to load by clicking on each name. Double clicking on the file name automatically loads the program. **TYPE IN** the command line arguments to the executable selected for loading. **PRESS** **OK** • The Load Executables window closes and the debugger loads the program for task 0. Create a group, say *group1*, containing tasks 1 through 4. **PRESS** the task group button labeled *group1* to set the context. **SELECT** **File → Load Executables ...** • The Load Executables window opens. **SELECT** the directory and executable file you want to load by clicking on each name. Double clicking on the file name automatically loads the program. **TYPE IN** any command line arguments to the executable selected for loading. **PRESS** **OK** • The Load Executables window closes and the debugger loads the program for tasks 1 through 4. Each tasks stops at the first executable statement. **MP_DEBUG_INITIAL_STOP** can be set to override the default of the first executable source line in *main()*. Set **MP_DEBUG_INITIAL_STOP** to the *file: linenumber*. **DISPLAYS MESSAGE** 0030-0101 Partition Loaded

The pedb Window with a Partition Loaded

Once the partition is loaded, the **pedb** window will make all of its options available.



| *Figure 6. pedb main window after partition is loaded*

This window consists of:

- The *Title Bar*. The Title Bar is located at the top most part of the window. It identifies the view and context of the program.
- A menu bar with the following options:
  - File
  - View
  - Group
  - Find
  - Options
  - Help
- The *Source File Label*. This label displays the name of the source file you are currently debugging and the task number with which the source file is associated.

- The *Source Area*. This area displays the source code of the parallel program you are debugging. This area has both horizontal and vertical scroll bars for reading text displayed outside it.
- The *Message Area*. This area displays informational and status messages about events and actions that occur. Messages about errors, warnings, and other severe conditions may *not* appear here; instead, they may appear in a message pop-up window. The contents of this message area window is controlled by a fixed-size buffer. When the buffer fills, earlier messages may no longer be accessible from the message area window. However, all error messages are duplicated in the window from which **pedb** was started.
- The *Global Data Area*. The global variable viewer displays the variables that are defined globally within the executing task(s). Global variables are only relevant when debugging C programs. For more information on global data, see "Examining Program Data" on page 62. The Global Data Area has both horizontal and vertical scroll bars for reading text displayed outside it.
- The *Local Data Area*. This area displays the values of the current routine's local variables. The Data Area has both horizontal and vertical scroll bars for reading text displayed outside it.
- The *Stack Area*. This area displays the call stack for the current procedure or function. The Stack Area has both horizontal and vertical scroll bars for reading text displayed outside it. See "Displaying Local Variables Within the Program Stack" on page 63 for more information.
- The *Threads Area*. This area displays the threads contained in the task. The Threads Area has both horizontal and vertical scroll bars for reading text displayed outside it. See "Displaying Threads Information" on page 67 for more information.
- The *Break/Trace Area*. This area displays the active Break/Trace points for the tasks in the current context. The Break/Trace Area has both horizontal and vertical scroll bars for reading text displayed outside it. See "Locating Breakpoint in Source" on page 63 for more information.
- The *pedb Execution Controls*. These controls are directly below the Source Area and allow you to control the execution of the application you are debugging. These controls are similar to those you might find on a VCR or CD player, and are described in "Controlling Program Execution" on page 51.
- A *Task Area*. This area contains a number of *task* and *task group* push buttons that you can use to select tasks, or task groups, when you are defining current context. See "Setting the Context" and "Creating Task Groups" on page 48 for more information.

Setting the Context

In **pedb**, context is defined as a task or group of tasks to which the debugger directs certain actions or requests. The context sensitive controls, directly below the Source Area (lower-left), only affect those tasks in the current context. The context also determines which task's variables and stack traces will be displayed.

When you start a **pedb** session, the context is initially set to all tasks. As illustrated in Figure 4 on page 41, the title bar of the pedb window reads

pedb: View - 1, Context - Group ALL.

If you want the current context to be something other than all tasks, you can use the push buttons in the Task and Task Groups areas to change it. Press the button that corresponds to the task or group of tasks you wish to include in the current context. Note that you can select only one task or task group at a time.

For example, assume you have a parallel program that is divided into five tasks. The tasks are numbered 0 through 4, and each has a task push button in the Task area. To set the context to just task 1:

PRESS the task push button labeled 1 in the Task Area.

- The **pedb** debugger sets the context to task 1. To illustrate this, the debugger highlights the task's push button and updates the title bar of the **pedb** window to read *pedb: View - 1, Context - 1*.

You can also define the current context by specifying groups of tasks. When you start a **pedb** session, a task group is automatically defined that consists of all tasks. This task group is named ALL. See “Creating Task Groups” on page 48 for information on how to create task groups.

To set the command context back to the task group ALL:

PRESS the task group push button labeled ALL in the Task Area.

- The **pedb** debugger sets the context to all tasks. To illustrate this, the debugger highlights the ALL task group push button, as well as the other task push buttons, and updates the title bar of the **pedb** window to read *pedb: View - 1, Context - ALL*.

You can change the context at any time during the debugging session.

Creating and Deleting Task Groups

In general, the term *task* refers to an individual program that is part of a parallel application.

You can collect a number of tasks under a common group name. When you do this, the debugger creates a push button for the task group in the Task Groups area. You can then set the context to include the tasks in the group by pressing its push button.

To understand why you would want to define your own task groups, consider the following example. You are debugging a master/worker program containing many blocking sends and receives. The program has ten tasks. Task 0 is the master task, and tasks 1 through 9 are the workers. During debugging you might start off by running the master until a blocking receive operation cannot complete. Then you could set the context on all the workers and run them past the matching send. This will allow the master task to proceed. Then set the context back on the master and run it some more.

Since you plan to keep switching the context back and forth between the master and workers, you might find it helpful to group tasks 1 through 9 into a task group named *workers*. Then you would be able to press a task group button to set the context on the workers only.

You could also create a group named *master* containing just task 0. Although the "group" in this case has only one task, the name *master* is more meaningful than a task number and is therefore easier to remember.

Provide a group name that is no longer than 32 characters which starts with an alphabetic character, and is followed by any alphanumeric character combination.

Creating Task Groups

You can create a group at any time during the debugging session using the Add Group window.

To create a task group:

SELECT Group → Add Group

- The Add Group window opens.

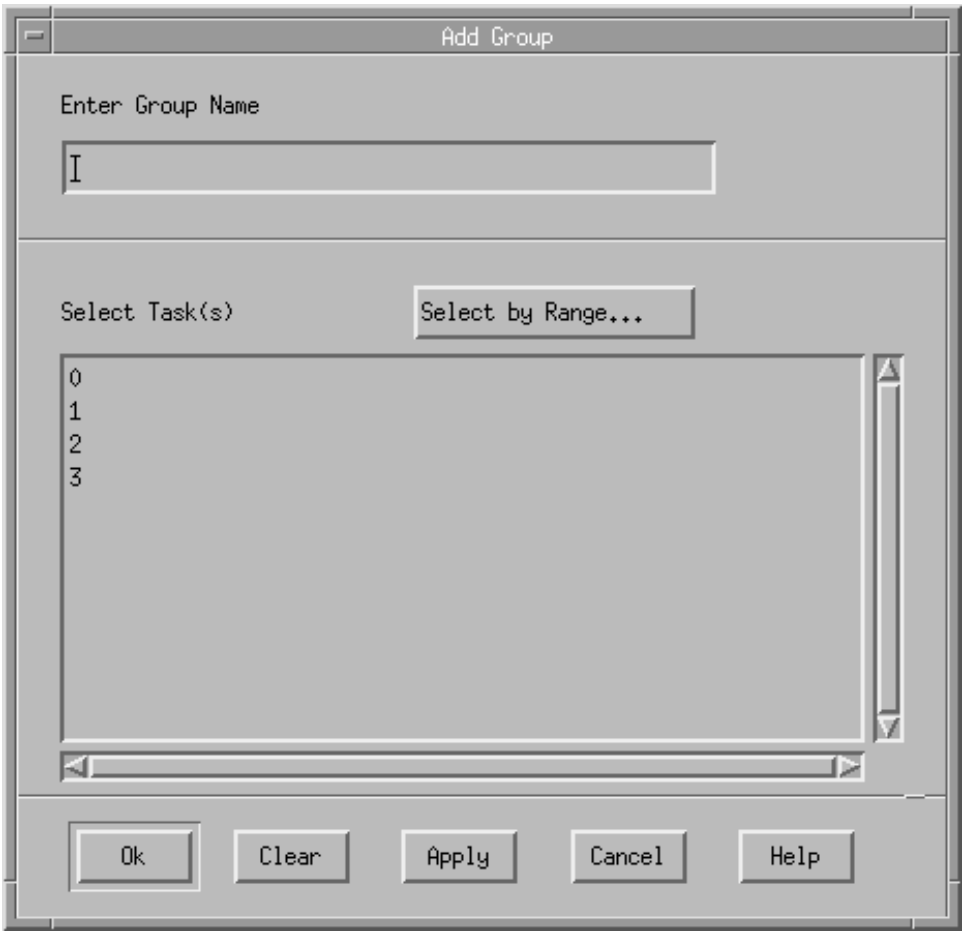


Figure 7. Add Group window

FOCUS on the Enter Group Name entry field.

TYPE IN the name of the group to be added.

FOCUS on the Select Task(s) area.

SELECT

a task by placing the cursor over a task number and depressing the left mouse button.

or

a range of tasks by depressing the left mouse button over the first task, and dragging it over the range of tasks to be included in the group.

or

a set of nonconsecutive tasks by selecting the first task, and while holding down the control key, selecting the next task(s). Note that selecting a previously selected variable will de-select it.

PRESS **Apply**

or

OK

- **Apply** creates the task group. A button containing the name of the group appears in the Task Groups area of the main window. The Add Group window remains open for further selections. **OK** creates the group, as above, and closes the Add Group window.

Clear removes all task selections and clears the text in the group name field.

Cancel closes the Add Group window.

The Select By Range feature is useful when you need to select a large range of tasks. To indicate the tasks using the Select By Range button:

SELECT the Select By Range button.

- The Select By Range window opens.

FOCUS on the **Enter range of tasks to select:** field.

TYPE IN the task list. To indicate a range of tasks, enter the first and last task numbers, separated by a colon or dash. To indicate individual tasks, enter the numbers, separated by a space or a comma.

For example:

To add:	Type in:
task 6 and 8	6,8
tasks 6 and 8	6 8
6 through 8, and 75	6:8 75
6 through 8, and 75	6-8 75

- **Apply** adds the selected tasks to the Add Group window and leaves the Select By Range window open for other selections.

  OK adds the tasks to the Add Group window and closes the Select By Range window.

  Cancel closes the Select By Range window.

Deleting Task Groups

If a particular task group no longer seems necessary, you can delete it using the Delete Group window. You may delete a group (except “all” or “attached”) at any time during the debugging session.

To delete a task group:

SELECT **Group → Delete Group**

- The Delete Group window opens.

PLACE the cursor over the name of the group.

PRESS the left mouse button.

- The group name highlights to show that you have selected it.

PRESS **Apply**

or

OK

- **Apply** deletes the group and removes the group button from the Task Groups area. The Delete Group window remains open. **OK** deletes the group, as above, and closes the Delete Group window.

  Cancel closes the Delete Group window.

Task Status Information

The task buttons used to change **pedb** context also display information about the status of each task. During your debug session, the color and the code letter change during different activities. For example, during the load process, the color will change from red to yellow to green and the status code letter will change from *N* to *L* to *D*.

Note: These are the default colors, but you can configure them by updating your *.Xdefaults* file.

Since each task runs independently of **pedb**, the debugger maintains status information for each task in the partition. The following table shows the status codes that are displayed in each task button and describes their meanings.

Table 5. Status Codes

Code	Default Color	Status	Description
N	Red	Not loaded	The task has not yet been loaded with an executable.
L	Yellow	Loaded	The task has been loaded with an executable.
D	Green	Debug Ready	The task is stopped and can be debugged using **pedb**.
R	White	Running	The task is in control and running.
P	MediumSeaGreen	Playing	The Play button has been depressed. The task is switching between Playing and Running, with some limited function.
x	Khaki	Exit Requested	The task in the parallel application has issued exit or returned from main, and is thus waiting in the POE specific exit code for its peer tasks to indicate that they too are waiting to exit.
X	Goldenrod	Exited	The task has reached the **hidden_exit** breakpoint that was set by **libdbx**.
E	Orange	Error	The task is in an unknown state.
U	LightSteelBlue	Unhooked	The task has been unhooked.
UA	Wheat	Unattached	The task has not yet been attached.

Controlling Program Execution

The **pedb** debugger lets you control execution by setting breakpoints in, or else by stepping through, the source code. This section describes how to perform these familiar debugging tasks using the **pedb** debugger. It also describes some additional functions **pedb** provides such as unhooking tasks so they can run without intervention from the debugger.

The simplest method of controlling program execution with **pedb** is by manipulating the control buttons located directly below the Source Area on the **pedb** window. From left to right, the control buttons are Step Over, Step Into, Step Return, Continue, Halt, Play, and Stop.

Table 6. Control Buttons	
In order to:	**Press this Control Button:**
Manually step the execution of tasks in the current context, by a line of source code, stepping *over* subroutines and functions.	Step Over
Manually step the execution of tasks in the current context, by a line of source code, stepping *into* subroutines and functions.	Step Into
Return from the current function to the function which called it. This typically reduces the call stack by one function.	Step Return
Continue executing the tasks in the current context up to the next breakpoint or to the program's completion.	Continue
Interrupt execution of *running* tasks. The Halt button is used for situations in which a process is in a running state, such as blocked, and must be interrupted.	Halt
Have the debugger repeatedly execute the tasks. Available Play choices are Step Over, Step Into, and Continue.	Play
Break out of Play mode (as tasks finish, they will stop)	Stop
Note: To modify the function of the Play button, refer to "Customizing the Play Control Button" on page 59.	

Every time execution of a task in the current context stops, the debugger updates the **pedb** window to display the current information. In the Source Area, the debugger uses a line marker to identify the line of code at which execution has stopped. The debugger draws a line marker as an arrow pointing at the line of code. For example, when you first load a parallel program onto the partition, it runs up to the first executable statement and stops. In Fortran and C programs, this is the first executable source line defined by the user, so the debugger draws a line marker there. Since you can set the context on a subset of tasks and run them independently of the others, you can have a number of line markers displayed. When more than one task is at the same line of code, the line marker appears as a button.

If you are unsure of the task(s) associated with a particular line marker:

PLACE the mouse cursor over the line marker.

PRESS the left mouse button.

- A label appears identifying the tasks and threads at that line of code. For threaded programs, each task number is followed by the displayed thread number, which is the thread whose source file, local variables, and stack are displayed. The label is visible only while you hold the mouse button down.

Line markers may be thought of as ad hoc groups which can be used to manipulate the parallel application. This manipulation is independent of the current context. For example, say you are debugging a parallel program with three tasks numbered 0, 1, and 2. Tasks 0 and 1 are at line 22, while task two is at line 24. You want to step tasks 0 and 1 to the same line as task two. To step tasks 0 and 1 as if they were in a group:

PLACE the mouse cursor over the line marker at line 22.

PRESS the right mouse button.

- A menu appears containing four items – Step Over, Step Into, Step Return, and Continue. These menu items correspond to the Step Over, Step Into, Step Return, and Continue **pedb** control buttons.

SELECT **Step Over**

The debugger runs tasks 0 and 1 one line and stops them at line 23.

REPEAT these steps so that tasks 0, 1, and 2 are all at line 24.

The line marker, as used above, allows you to perform simple operations on single or multiple tasks that are not in a predefined task group. If tasks 0 and 1 comprised a group, you could change the context to that group, and then use the control buttons as above, then restore the original context.

Note: For more information on stepping, see "Stepping Execution" on page 57.

Threaded Programs

Each task can contain multiple threads. The threads for the tasks are listed in the threads pane on the right hand side of the **pedb** main window just below the stack pane. The list of threads for a single task can also be displayed in a separate window known as the Threads Viewer window.

When a task is in "debug ready" state, the *interrupted* thread is defined as the thread that stopped due to encountering a breakpoint or a signal. When this thread stopped, it in turn stopped all of the other threads in the process. The interrupted thread is treated specially by single step execution control. Subsequent step execution control that is issued will have the effect of stepping the interrupted thread while letting all other threads continue. It is not possible to change the interrupted thread to another thread. The interrupted thread is denoted by an asterisk at the start of the threads row.

Initially, when a task reaches "debug ready" state, the *displayed* thread is the same as the interrupted thread. The displayed thread for a task is alterable by the user. When changed to another thread, the stack, local variables, source line arrow, and the source file will be updated to reflect those for the new displayed thread. The displayed thread is denoted by a "greater than" (>) operator at the start of the threads row.

Setting Breakpoints

The **pedb** debugger lets you set stopping places, called breakpoints, in your program. You mark which lines are to be breakpoints for the tasks in the current context and then run the program using the Continue button. When the tasks reach a breakpoint, execution stops and you can then examine the state of the program.

In threaded programs, setting a breakpoint on a task, sets the breakpoint for all threads in the task. When any thread in a task hits a breakpoint, all other threads in the task are also stopped.

Note: The **Play** button will not stop execution at breakpoints, so it is suggested that you use the Continue button.

To set a breakpoint:

PLACE the mouse cursor over an executable source line in the Source Area.

PRESS the left mouse button.

- The line highlights to show that you have selected it.

PRESS the right mouse button.

- A selection menu appears.

SELECT **Breakpoint**

- The debugger sets, for each task in the current context, a breakpoint at the marked line of code.

  Note: You can also set a breakpoint by double clicking the left mouse button after placing the cursor over an executable source line.

  In the Source Area, the debugger places a stop marker (drawn to look like a stop sign containing an exclamation point) next to the line with the breakpoint.

  In addition to the stop marker, the debugger displays a breakpoint event message (one for each of the tasks in the current context) in the Break/Trace area. The message includes an interpretation of the breakpoint. For example:

  ```
  [1] stop at "ptst4.f":22
  ```

Note: The debugger sets a separate breakpoint for each task in the context.

You can also specify a condition when setting a breakpoint. The task then stops executing at the breakpoint only if the condition evaluates to true.

Specifying Conditions for Breakpoints and Tracepoints

You can specify conditions with a subset of C syntax, with some Fortran extensions. The following operators are valid:

Arithmetic Operators

+	Addition
-	Subtraction
-	Negation
*	Multiplication
/	Floating point division
div	Integer division
mod	Modulo
exp	Exponentiation

Relational and Logical Operators

<	Less than
>	Greater than
<=	Less than or equal to

| **>=** Greater than or equal to
| **==** Equal to
| **=** Equal to
| **!=** Not equal to
| **< >** Not equal to
| || Logical OR
| **or** Logical OR
| **&&** Logical AND
| **and** Logical AND

| Bitwise Operators
| **bitand** Bitwise AND
| | Bitwise OR
| **xor** Bitwise exclusive OR
| ~ Bitwise complement
| **<<** Left shift
| **>>** Right shift

| Data Access and Size Operators
| **[]** Array element
| **()** Array element
| * Indirection or pointer dereferencing
| **&** Address of a variable
| **.** Member selection for structures and unions
| **.** Member selection for pointers to structures and unions
| **->** Member selection for pointers to structures and unions
| **sizeof** Size in bytes of a variable

| Miscellaneous Operators
| **()** Operator grouping
| **(Type)Expression** Type cast
| **Type(Expression)** Type cast
| **Expression\Type** Type cast

To set a conditional breakpoint:

PLACE the mouse cursor over an executable source line.

PRESS the left mouse button.

- The line highlights to show that you have selected it.

PRESS the right mouse button.

- A selection menu appears.

SELECT **Conditional Breakpoint**

- The debugger displays the Conditional Breakpoint window.

FOCUS on the text entry field of the Conditional Breakpoint window.

TYPE IN the condition that must evaluate to true for the execution to stop at the breakpoint. For example, to stop execution at the breakpoint only if the variable x is greater than 19, you would type in `x > 19`.

PRESS **OK**

- The debugger closes the Conditional Breakpoint window and sets a breakpoint for the tasks in the current context.

  As with regular breakpoints, the debugger places a stop marker next to the line with the breakpoint in the Source Area. In the Break/Trace area, it adds a message reporting the conditional breakpoint the debugger has built for each of the tasks in the current context. For example:

```
[1] { stop } at "ptst4.f":22 if x > 19
```

Specifying Thread Specific Breakpoints and Tracepoints

Thread specific conditions can be added to breakpoints and tracepoints using the conditional breakpoint and conditional tracepoint windows. You set the conditions using the variable **$running_thread**. This variable represents the thread that encountered the breakpoint first. This thread (the interrupted thread) will cause all of the other threads to stop. For example, adding the condition `$running_thread == 1` after selecting line `234` in your source would result in the program stopping when the thread labeled `$t1` encountered line `234`. To state this another way, the breakpoint is triggered only when thread `1` encounters the breakpoint. Refer to *IBM AIX Version 4 General Programming Concepts: Writing and Debugging Programs* for more details.

Identifying the Tasks Associated with a Breakpoint: When you set a breakpoint by following the previous instructions, remember that a separate breakpoint is set for each of the tasks in the current context. If you wish to see a list of the task(s) associated with a particular stop marker in the Source Area:

PLACE the mouse cursor over the stop marker.

PRESS the left mouse button.

- A label appears identifying the tasks with a breakpoint at that line of code. The label is visible only while you hold the mouse button down.

  When multiple breakpoints are set for a given task, the breakpoint will appear multiple times in the Break/Trace area. The corresponding breakpoint events will also be highlighted in the Break/Trace Area. This graphically shows the breakpoints that would be deleted if you used the procedure for *removing all breakpoints at the same line of code*, described in Table 7 on page 57.

Removing Breakpoints: Any number of the active tasks may have one or more breakpoints at that same line of code. You can remove:

- a breakpoint for a single task

  or
- all the breakpoints at that line of code for all tasks in the current context.

The following table shows how to remove breakpoints.

Table 7. Removing Breakpoints

To remove the breakpoint for a single task:	**To remove all breakpoints at the same line of code:**
PLACE the mouse cursor over the breakpoint's event message for that task in the Break/Trace Area. **Note:** The task must be in the current context for the event message to be displayed. **PRESS** the left mouse button • The breakpoint's event message highlights to show that you have selected the breakpoint. **PRESS** the right mouse button. • A selection menu appears. **SELECT** **Delete** • The breakpoint's event message disappears to show that the debugger has removed the breakpoint for the task.	**PLACE** the mouse cursor over the stop marker at that line of code. **PRESS** the right mouse button. • A selection menu appears. **SELECT** **Delete** • The debugger removes all breakpoints set at the line of code for the tasks in the current context. The stop marker disappears as well as the event strings *highlighted* in the Break/Trace Area.

Stepping Execution

The **pedb** debugger lets you single–step execution of your program. In other words, you can run the tasks in the current context one source code line at a time. For threaded programs, single step execution has the effect of single stepping the interrupted thread, while letting all other non-held threads in the task continue freely without stopping at any breakpoints. All threads will again be stopped when the stepping thread reaches the appropriate source line. You can manually control each step, or have the debugger repeatedly step through the tasks in the current context at a selected interval. There are three methods of stepping the execution of your program. You can use the Step Over button to step *over* the subroutines and functions of your program. The Step Into button lets you step *into* the subroutines and functions of your program. Also, the Step Return button lets you return to the calling function.

To step execution, stepping *over* subroutines and functions:

PRESS the **Step Over** Control Button.

• The debugger runs one line of the source code for the tasks in the current context and stops.

The function of the Step Over Control Button is to:

- step one line of source code
- step over functions, keeping the scope within the current function.

To step execution, stepping *into* subroutines and functions:

PRESS the **Step Into** Control Button.

• The debugger runs one line of the source code for the tasks in the current context and stops.

The function of the Step Into Control Button is to:

- step one line of source code
- step into functions, following execution into called functions with debugging information.

The debugger changes the source code displayed in the Source Area to that of the called function, and adds the function call to the Stack Area.

To step execution, returning to the calling function:

PRESS the **Step Return** Control Button.

- The debugger returns to the calling function and stops.

The function of the Step Return Control Button is to:

- execute the remainder of the current function
- stop in the calling function.

To automatically repeat execution:

PRESS the **Play** Control Button.

- The debugger repeatedly steps execution of the tasks in the current context.

When using the **Play** Control Button, execution continues for the tasks in the current context until you press the **Stop** Control Button. The Play function allows you to execute multiple iterations of **Step Over**, **Step Into** or **Continue** (see "Customizing the Play Control Button" on page 59 for more information). It updates the **pedb** window for each play cycle executed.

The Continue function of the **Play** Control Button can be particularly valuable when looking at loop processing. A break point may be set within a loop and the application data will be updated for each loop iteration.

Since the debugger has to keep updating the **pedb** window when in play mode, this is a slower form of execution than using the Continue Control Button. The advantage is that it provides you with more intermediate information.

For example, say you are debugging a master/worker program containing many blocking sends and receives. The context is on the worker tasks. You set a breakpoint and then press the Continue Control Button to continue execution of the worker tasks. Before reaching the breakpoint, however, the tasks hit a blocking receive intended to synchronize execution between the workers and the master task. Because the master is not in the current context, the receive operation cannot complete and so the workers cannot reach the breakpoint. Since the debugger cannot refresh the **pedb** window until the pending Continue function completes, the problem is not immediately observable. However, if you were repeatedly stepping the tasks using the Play function, you would see the line marker and other application information in effect just prior to the pending Step. You would see the task buttons holding in the *running* state and have a clear indication of where the program is hung.

To stop execution (stop playing):

PRESS the **Stop** Control Button.

- The debugger stops executing the program's tasks.

To interrupt execution (interrupt a waiting process):

PRESS the **Halt** Control Button.

- The debugger interrupts execution and returns control to the user.

Customizing the Play Control Button: When you press the Play Control Button, by default it repeatedly executes **Step Into**(s), with one-second between each **Step Into**. However, you can customize the Play Control Button to:

- select which command to repeatedly execute
- specify the delay between the command iterations in tenths of a second.

You can set these options from the main menu, using the **Options** pulldown.

To specify the command:

SELECT **Options → Change Play Command**

- Another menu appears with the command choices.

SELECT the command you want the debugger to execute repeatedly when you press the Play Control Button.

- The menu closes, and the Play Control Button is set to execute the command you specified.

To specify the delay between command iterations:

SELECT **Options → Change Play Delay**

- The Change Play Delay window opens.

FOCUS on the text entry field in this window.

TYPE IN The new delay time in tenths of seconds.

PRESS **OK**

- The Change Play Delay window closes, and the Play Control Button is set to execute its command with the new delay you specified.

You can also set these options from the pop-up menu on the Play Control Button:

To specify the command:	**To specify the delay between command iterations:**
PLACE the cursor over the Play Control Button.	**PLACE** the cursor over the Play Control Button.
PRESS the right mouse button.	**PRESS** the right mouse button.
• The Play Menu appears.	• The Play Menu appears.
SELECT **Change Play Command**	**SELECT** **Change Play Delay**
• Another menu appears with the command choices.	• The Change Play Delay window opens.
SELECT the command you want the debugger to execute repeatedly when you press the Play Control Button.	**FOCUS** on the text entry field in this window.
	TYPE IN The new delay time in tenths of seconds.
	PRESS **OK**
• The menu closes, and the Play Control Button is set to execute the command you specified.	• The Change Play Delay window closes, and the Play Control Button is set to execute its command with the new delay you specified.

Tracing Program Execution

The **pedb** debugger lets you set tracepoints in your program. When tasks reach a tracepoint during execution, the debugger writes information regarding the state of the program to the window from which **pedb** was invoked.

For threaded programs, tracepoints are set for all threads in the task by default. Each time a thread in the task encounters the tracepoint, a trace record is written.

Tracepoints can be set at any executable line of code within the program.

To set a tracepoint:

PLACE the mouse cursor over an executable source line.

PRESS the left mouse button.

- The line highlights to show that it is selected.

PRESS the right mouse button.

- The Break/Trace menu appears.

SELECT **Tracepoint**

- The debugger sets a tracepoint at the selected line for the tasks in the current context.

  In the Source Area, the debugger places a blue trace marker next to the line with the tracepoint. The trace marker is drawn as two eyes looking at the line of code.

  In addition to the trace marker, the debugger displays a tracepoint event message (one for each of the tasks in the current context) in the Break/Trace area. The message includes an interpretation of the tracepoint preceded by the event ID associated with it. For example:

  ```
  [6] trace at "mikia.f":15
  ```

Note: The debugger sets a separate tracepoint for each task in the context.

You can also specify a condition when setting a tracepoint. The tasks then write trace information only if the condition evaluates to true. See “Specifying Conditions for Breakpoints and Tracepoints” on page 54 for more information.

Thread specific tracepoints can be set in a similar fashion to thread specific breakpoints. See “Specifying Thread Specific Breakpoints and Tracepoints” on page 56 for more information.

To set a conditional tracepoint:

PLACE the mouse cursor over a source line.

PRESS the left mouse button.

- The line highlights to show that you have selected it.

PRESS the right mouse button.

- A selection menu appears.

SELECT **Conditional Tracepoint**

- The debugger displays the Conditional Tracepoint window.

FOCUS on the text entry field of the Conditional Tracepoint window.

TYPE IN the condition that must evaluate to true for trace information to be written. For example, `x > 19`.

PRESS **OK**

- The debugger closes the Conditional Tracepoint window and sets a tracepoint for the tasks in the current context.

  As with regular tracepoints, the debugger places a trace marker next to the line with the tracepoint in the Source Area. In the Break/Trace area, it adds a message reporting the conditional tracepoint for each of the tasks in the current context. For example:

  ```
  [7] trace at "blist.f":23 if x > 19
  ```

Identifying the Tasks Associated with a Tracepoint: If you wish to see a list of the task(s) associated with a particular trace marker in the Source Area:

PLACE the mouse cursor over the trace marker.

PRESS the left mouse button.

- A label appears identifying the tasks with a tracepoint at that line of code. The label is visible only while you hold the mouse button down.

  When multiple tracepoints are set for a given task, the tracepoint will appear multiple times in the Break/Trace area. The corresponding breakpoint events will also be highlighted in the Break/Trace Area. This graphically shows the tracepoints that would be deleted if you used the procedure for removing all tracepoints at the same line of code, described in Table 8.

Removing Tracepoints: Any number of the active tasks may have a tracepoint at that same line of code. You can remove:

- a tracepoint for a single task

  or
- all tracepoints at that line of code for all tasks in the current context.

The following table shows how to remove tracepoints.

Table 8. Removing Tracepoints

To remove the tracepoint for a single task:		**To remove all tracepoints at the same line of code:**	
PLACE	the mouse cursor over the tracepoint's event message for that task in the Break/Trace Area.	**PLACE**	the mouse cursor over the trace marker at that line of code.
	Note: The task must be in the current context for the event message to be displayed.	**PRESS**	the right mouse button.
PRESS	the left mouse button		• The Break/Trace menu appears.
	• The tracepoint's event message highlights to show that you have selected the tracepoint.	**SELECT**	**Delete**
PRESS	the right mouse button.		• The debugger removes all tracepoints for the tasks in the current context. The trace marker disappears as well as the event strings *highlighted* in the Break/Trace Area.
	• A selection menu appears.		
SELECT	**Delete**		
	• The tracepoint's event message disappears to show that the debugger has removed the tracepoint for the task.		

Unhooking Tasks

A task or group of tasks may be *unhooked* so that they execute without intervention from the debugger.

To unhook a task or group of tasks:

PLACE the mouse over a task or group button in the task area of the **pedb** window.

PRESS the right mouse button.

SELECT the Unhook option.

- The debugger unhooks the selected task or group of tasks. The task buttons are set to the appropriate color (the default is blue) to indicate that they have been unhooked. Note that you can change the default colors used by **pedb** by updating the X defaults file.

Examining Program Data

This section describes how to use the Data Area of the **pedb** Window to examine your program's data. This area shows the names and values of variables in the current routine.

Each time execution of the program stops, the debugger automatically updates the information displayed.

Data, Stack, Threads, and Break/Trace Information

In the **pedb** window, the Local Data, Global Data, Stack, Threads, and Break/Trace areas present information for each task in the current context. There are times when you want to stop displaying information for a particular task or task group in one or more of these areas. This allows you to conserve space in the area and improve the readability of information still displayed there.

For example, say you are debugging a master/worker program. The program has 15 identical worker tasks and you are stepping execution through them. Since the information displayed for each task is essentially the same, you might want to hide all but one. The information will then be easier to read and the information refreshes will be faster.

To hide a task's data, stack, threads,and break/trace information:

PLACE the mouse cursor over a task button or a task group button in the task area of the **pedb** window.

PRESS the right mouse button.

- A selection menu appears.

SELECT either **Hide Local Data**, **Hide Global Data**, **Hide Stack**, **Hide Threads**, **Hide Break/Trace**, or **Hide All**

- The debugger no longer displays information for the task in the specified window. If you selected **Hide All**, the debugger hides the tasks information in all four areas. When you hide a window, the **Hide** option on the selection menu toggles to **Show**. You can then repeat these steps to again display the task's information in the specified window.

Locating Breakpoint in Source

You can select a Break/Trace event and show the source line associated with it.

DOUBLE CLICK on an item in one of the Break/Trace window lists.

- The source window centers at the source line that is associated with the selected breakpoint and highlights that line.

OR

PLACE the mouse cursor on an item in one of the Break/Trace window lists.

PRESS the left mouse button to highlight your selection.

PRESS the right mouse button

- A menu pops up with two choices: **Delete** and **Goto Source**. Selecting **Goto Source** has the same effect as the double click described above.

Displaying Local Variables Within the Program Stack

pedb displays the variables that are in scope within the local program block. The Stack Area lets you display, for any of the functions or subroutines listed, the local variables that are outside the local execution block (not on the top of the stack). To display these variables:

PLACE the mouse button on a line in the Stack Area.

DOUBLE-CLICK the left mouse button.

- The line highlights to show that it is selected, and the local variables associated with the function, subroutine, or unnamed block are displayed within the Local Data. All the data variable menu options are available.

Note: Local variables for the associated tasks that are outside the local execution block (not on the top of the stack), are displayed only until you issue another Stack Area selection or execution function within **pedb**.

Displaying Local Variables and Program Stack Within a Thread

The **pedb** debugger will display program states about one thread of each task at a time. The source code, local variables, and stack trace for a task will be those of the displayed thread for the task. To select the displayed thread:

DOUBLE-CLICK on the thread.

OR

SELECT the thread from the pull-down available from each thread entry.

To select a stack entry:

PLACE the mouse button on a line in the Stack Area.

DOUBLE-CLICK the left mouse button.

- The line highlights to show that it is selected, and the local variables associated with the function, subroutine, or unnamed block are displayed within the Local Data. All the data variable menu options are available.

Understanding Data Types

In **pedb**, you can view program data through either the Global Variable Viewer or the Local Variable Viewer. These windows display a specific type of data (global or local), and the way you use them depends on the programming language.

Local Variables: The Local Variable Viewer displays the variables, in C and Fortran, that are currently visible within your local execution block.

Stepping in and out of functions and subroutines during the debugging session will alter the list of local variables that the Local Variable Viewer displays. The Local Variable Viewer displays the set of variables for the function or subroutine that is at the top of the execution stack for a particular task.

Global Variables: The Global Variable Viewer displays the variables that are defined globally within the executing task. Global variables are only relevant when debugging C programs. Unless you specifically modify it, the list of global variables displayed in the Global Variable Viewer remains constant throughout the debugging session. Initially no global variables are displayed.

Note: **xlf** and **xlc** optimize out variables that are not referenced within a program. As a result, these variables may not be available in the Global or Local Data areas.

Data Display Policies

The following table shows how variables are initially displayed in the Data Area.

In the Data Area, this type of variable:	Will initially be displayed:	For example:
scalar	with its value formatted according to its default type.	x = 300 a = -.0001 b = 331.789978 char_val = 'W'
structure	with its type indicated.	struc_1 = <struct MyStruct_t>
array	with its type indicated.	a = <array 8192 of int> x = <array 5 struct foo> z = <array 10 * of int>
pointer	with its type indicated	Character Pointer examples: x = (nil); (unreferenced) x = → 0x4a567 (ptr to address) x = →→4; (dereferenced) Structure Pointer Examples: structx = (nil); (unreferenced) structx = → <struct foo> (dereferenced)
union	with its type indicated	MyOwnUnion_T = <union MyOwnUnion_T>
enum	with its value formatted according to its default type.	x = foo
logical	with its value formatted according to its default type.	do_this = .false

Viewing Variables with the Variable Viewer: Both the local and global variable windows are physically limited by the number and size of the variable information to be shown in the display. Therefore, conditions can exist where the user is unable to view all the data contained within the variable viewer due to geometry restriction on the **pedb** main window; or that the amount of data to be displayed exceeds the

limitations of the window. In either of these cases, all of the variable list can be viewed using the Variable Viewer, which is an expanded form of the data window.

Initially, the variable list (local and global) is displayed in a list form, or in an iconic form. Note that when the condition of overflow occurs the variable list is replaced by the overflow icon.

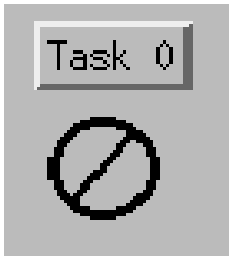


Figure 8. Overflow icon

To view the variable list in its own window,

PLACE the cursor over the task number label of the task, in the data area, of the variable you want to view.

PRESS the right mouse button.

- A pop-up menu appears with an option **Variable Viewer...**.

Note: The pop-up menu that appears for the local and global variable viewer will present a different set of options.

SELECT the **Variable Viewer** Option

- A separate window appears displaying the list of variables that was previously displayed in the local or global data area. The list (or icon) in the data areas on the main window of **pedb** should be replaced with the Variable Viewer icon.

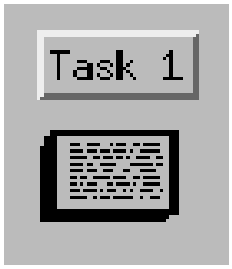


Figure 9. Variable Viewer icon

Using the Variable Viewer Window: The Variable Viewer window displays a list of global or local variables for a specified task. The task and type of data being displayed is identified by the title of the window. Figure 10 on page 66 shows the Variable Viewer window displaying local variables for the specified task. Note that all of the variable pop-up menus options that are described in "Policies for Global Variables" on page 66 are also available in this window.

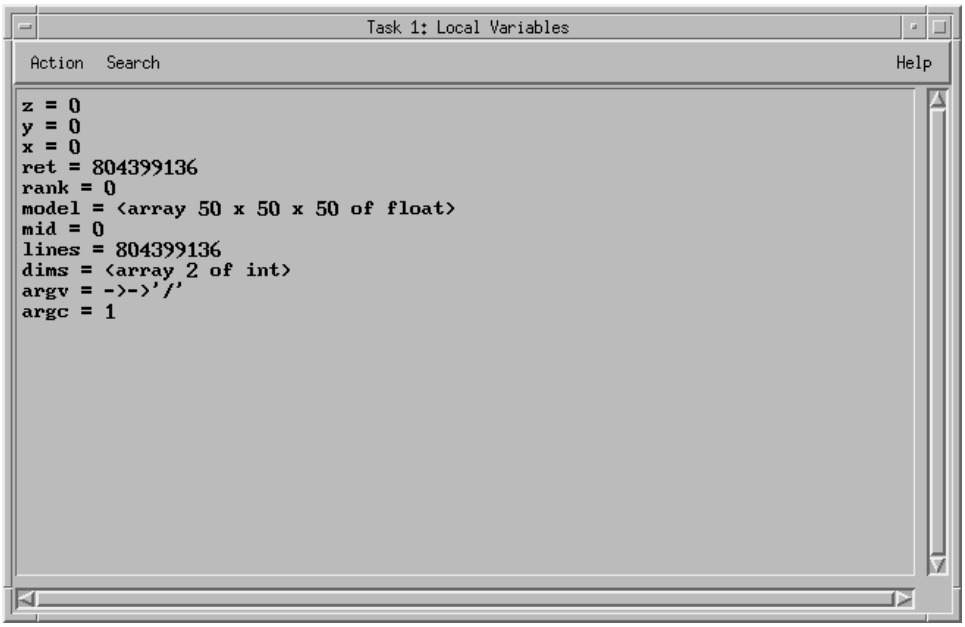


Figure 10. Variable Viewer window

To close the variable window and return the variable list back to the main window:

PLACE the mouse cursor over the **Action** pulldown on the menubar.

PRESS the left mouse button.

- A pop-up menu should appear with a **Close** option.

SELECT the close option with the left mouse button.

- The window will close and return the contents back to the main window data area.

Note: Variables displayed in a task that is not in the current context will not be refreshed.

Policies for Local Variables: The Local Variable subwindow for each task displays all the variables within the local execution block.

Policies for Global Variables: The Global Variable subwindow requires you to explicitly select the global variables you want to view. You can do this with the Variable Selection window.

To display a global variable:

PLACE the cursor over the task number label of the task, in the Global Data area, for which you wish to choose global variables.

PRESS the right mouse button.

- a pop-up menu appears with the following options:
  - Variable Selection, which provides a list from which you can select variables
  - Show All, which shows all global variables for the task
  - Hide All, which hides all global variables for the task
  - Variable Viewer, which moves the display for the variables of this task to a separate window.

SELECT the **Variable Selection...** option.

- The Variable Selection window appears.

Note: The Variable Selection window shows only the global variables in your program that have accessible source files. If the source code of a particular program is not accessible, you may wish to use the **-l** flag, or the Source Path Window.

SELECT

a single variable by placing the cursor over it and depressing the left mouse button.

or

multiple, contiguous variables by depressing the left mouse button and dragging it over the range of variables.

or

multiple, non-contiguous variables by selecting the first variable and, while holding down the control key, selecting the next variable(s). Note that selecting a previously selected variable will de-select it.

PRESS **Apply**

or

OK

- **Apply** selects the variable(s) and leaves the Variable Selection window open. The variable(s) appear under the corresponding task label in the Global Data area.

OK selects the variable(s), as above, and closes the window.

Cancel discards your selection and closes the window.

Displaying Threads Information

In **pedb**, you can display threads data for tasks in the current context. You can view a list of threads and some detailed information about the condition variables, attributes, and mutual exclusion locks pertaining to each task.

The Threads area of the **pedb** main window displays a list of threads for each task. Any one of the threads is available to select. Initially, the interrupted thread is the displayed thread. You are free to change the displayed thread for any task.

Like the local and global variable viewers, when a window representing a task in the threads area reaches a threshold, the overflow icon is displayed. See “Understanding Data Types” on page 64 for information on the local and global data viewer. At this time, you can open a Threads Viewer window, which contains the same information in the same format as in the Threads area for that task.

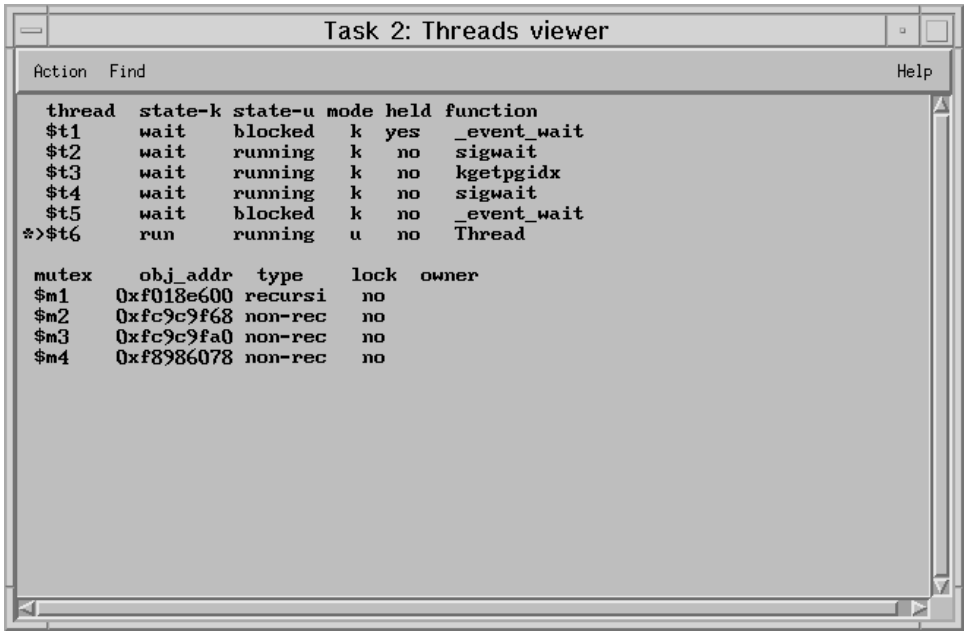


Figure 11. Threads Viewer window

The interrupted and displayed thread are denoted by "*" and ">" respectively. The ">" may move as you select different threads to display. The "*" does not change unless the program is executed and then stops again. The interrupted thread is important because it has an effect on further single stepping execution control. The displayed thread identifier is important because it denotes the thread that is represented in the stack, local variables, and source code windows.

Selecting a Thread to Display

To display a thread, go to the Threads area of the **pedb** main window:

SELECT a thread by pressing the left mouse button to highlight it.

PRESS the right button.

- The Thread menu appears.

SELECT the **Show thread** option.

- The currently highlighted thread becomes the displayed thread. The stack, local variables, and source windows will update accordingly. You can also double click on a thread with the left mouse button to display it. In this way you bypass the Thread menu.

Holding a Thread from Further Execution

You can hold a thread from execution, and then release it again for execution. To do this, enable the Thread menu as in "Selecting a Thread to Display" above, then:

SELECT the **Hold thread** option.

- This option controls whether the thread is dispatchable or not, and will affect subsequent execution control of the program. Held threads will not execute when single stepping or allowing the program to continue. Note that it is possible to induce hangs on other threads by holding a thread. When you select this option, the Thread menu disappears. The next time you enable the menu, the option will read "Release thread."

The held field in the Threads area (of the **pedb** main window) for this thread will update appropriately when you select either of these options.

Displaying Thread Details

You can display the details of threads by opening the Threads Display menu. From the **pedb** main window:

PRESS any of the task buttons in the Threads area.

- The Threads Display menu appears.

CHOOSE **Select Display Details...**

or

Open Threads Viewer...

- The **Select Display Details...** option opens the Threads Details Selections window. From here you can choose what thread details to view. This option is available only when the task is in "debug ready" state; otherwise it is disabled. The **Open Threads Viewer...** option opens a separate window to display threads information for the task.

The Threads Details Selections Window: This window contains three toggle buttons that specify the additional thread information to be displayed in the Threads area or the Threads Viewer. You can choose any or all of:

- Display Attributes
- Display Conditions
- Display Mutexes.

After using this window to change the level of thread detail displayed, these changes you made for this task will persist across further execution control.

The Threads Viewer: The Threads Viewer is another way to view threads data. It is similar to the local and global data viewer concept. One Threads Viewer is available for each task. It exists for two reasons. First, it overcomes the limitation in the display areas on the right side of the **pedb** main window when displaying large amounts of data. Second, it provides you with a separate and larger area for displaying threads data that interests you. You can iconify the Threads Viewer window separately.

The same actions are available in the Threads Viewer as in the Thread menu. Refer to "Selecting a Thread to Display" on page 68 and "Holding a Thread from Further Execution" on page 68 for this information. There is also a find selection in the Threads Viewer window main menu bar to allow finding strings within the displayed threads data.

The Threads Viewer window menu bar has three options available: **Action**, **Find**, and **Help**.

SELECT **Action**

CHOOSE **Select Display Details...**

or

Close Viewer

- The **Select Display Details...** option opens the Threads Details Selections window, as shown in "Displaying Thread Details" on page 69. This option is only available when the task is in **debug** state, otherwise it is disabled. The **Close Viewer** option closes the Threads Viewer window, and either redisplays the contents of the window in the threads area (if there is enough room), or displays the overflow icon.

For a description of the **Find** option, see "Source File, Variable Viewer, and Threads Viewer Find" on page 86.

Threaded MPI Library

If your application uses a threaded implementation of MPI, the debugger will display the existence and status of these threads even though you may not have explicitly coded threads. Refer to *IBM Parallel Environment for AIX: Operation and Use, Volume 1, Using the Parallel Operating Environment*, and *IBM Parallel Environment for AIX: MPI Programming and Subroutine Reference* for details.

Data Display Techniques

This section describes how, with appropriate mouse clicks in the Data Area, you can bring up menus and windows to:

- Select and display a variable
- Display a variable in more or less detail
- Change a variable's value
- Change a variable's format
- Display the variable's declaration
- Select the subrange of an array.

Displaying More or Less Detail for a Variable

The **More Detail** and **Less Detail** variable options on the variable options menu operate differently depending on the data type of the selected variable. The following describes these differences.

- Simple types

  Scalers, logicals, and enumerated types have one level of detail. The name of the variable and its value are displayed by default. The **More Detail** and **Less Detail** options are not available on the Variable Options menu for these variables.

- Complex types

  Structures and unions have two levels of detail. They have their type displayed by default. To show more detail:

  PLACE the mouse cursor over the variable name, equal sign, or variable type.

  PRESS the left mouse button.

  - The selection is highlighted.

  PRESS the right mouse button.

  - The Variable Options menu appears.

SELECT the **More Detail** option.

- This option shows the structure or union expanded with all members having their respective default levels of more or less.

After selecting **More Detail**, the **Less Detail** option then becomes available. When selected, **Less Detail** also shows the type of the structure or union. To show less detail:

PLACE the mouse cursor over the variable name or equal sign, and follow the same procedure as above for showing more detail.

- Arrays

An array has three levels of detail. The first level (the default) is its type, the second displays the array elements horizontally, and the third displays the elements vertically. When displaying the second level (horizontal), rows of more than 1000 characters are broken into multiple lines.

At the first level of detail, the **More Detail** option is available and when selected, shows the array elements horizontally. At the second level of detail, both the **More Detail** and **Less Detail** options are available. The **Less Detail** option will revert to the default of displaying the array type. The **More Detail** option, when selected, will then display the array elements vertically. At the third level of detail, the **Less Detail** option is available, and when selected, again shows the array elements horizontally.

Note that the default is to display one element of an array. To display more array elements, the **Select Subrange** option of the Variable Options menu must be selected to bring up the Array Subrange window (see “Viewing the Contents of an Array” on page 73 for more information). This window allows selecting ranges, slices, and strides within the selected array. There is a limitation of displaying 1000 elements per array at one time.

- Pointers

Pointers to any other type have two levels of detail. By default, the second level of detail is displayed, any dereferencing is done, and the value of the native type is displayed. To show less detail:

PLACE the mouse cursor over anywhere from the “-” portion of the arrow and to its left.

PRESS the left mouse button.

- The Variable Options menu appears.

SELECT the **Less Detail** option.

- This option shows the value of the pointer in ‘hex’ format, or the string at the pointer location in the case where the native type is ‘char’.

Pointers with multiple levels of indirection, which point to other than ‘char’ types, have a level of detail for the native type and another level of detail for each level of indirection. By default, all pointer dereferencing is done and the value of the native type is shown. To show less detail on any level of indirection:

PLACE the mouse cursor over anywhere from the “-” portion of the arrow and to its left, and follow the same procedure as above for showing less detail.

After any **Less Detail** option has been selected on any of the arrow pointers, subsequent selections anywhere on this variable to the right of any remaining arrows will result in bringing up a menu with the **More Detail** option available. If there are no remaining arrows, then selections anywhere on this variable will result in bringing up a menu with the **More Detail** option available. The **More Detail** option will always bring the variable back to the default of displaying the value of the native type.

Changing a Variable's Value

You can select a variable in the Data area and modify its value.

To select a variable and change its value:

PLACE the mouse cursor over a variable in the Data Area.

PRESS the left mouse button.

- The selection position is highlighted.

PRESS the right mouse button.

- A selection menu appears.

SELECT **Change Value**

- The debugger displays a Change Value window that corresponds to the type of variable you selected.

FOCUS on the text entry field of the Change Value window.

TYPE IN the value you want to set the selected variable to.

PRESS **OK**

- The debugger closes the Change Value window, and sets the variable to the value you specified.

Cancel closes the Change Value window and discards any changes.

Changing a Variable's Format

You can select a variable in the Data area and change its displayed format. You could modify the format of a variable to:

- default
- decimal
- hex
- octal
- scientific
- char
- string
- declaration

Note: Some options may be inactive, depending on the type of variable selected.

To select a variable and change its format:

PLACE the mouse cursor over a variable in the Data Area.

PRESS the left mouse button.

PRESS the right mouse button.

- A selection menu appears. This menu is type-sensitive, so the only options it makes available to you are those that correspond to the type of variable you have chosen.

SELECT **Change Format**

- A selection menu appears listing the possible display formats.

SELECT the format you want.

- The debugger formats the selected variable accordingly.

Display Declaration of a Variable

After selecting a variable in the Local Variable Viewer, Global Variable Viewer, or Variable Viewer for text, you can press the right mouse button to view the Variable Selection menu. From here you can select:

Change Format → Declaration

When selected, the declaration of the variable which was previously selected is displayed after the equal sign. This is only available for scalar types.

```
Default display:
   a = 5
Display after declaration format is chosen:
   a = int a;
      or
   a = integer*4 a
```

Viewing the Contents of an Array

The Array Subrange window allows you to view the elements of an array. By defining the range of elements, you can control the portion of the array that you see. To open the Array Subrange window:

PLACE the mouse cursor over the array you want to select.

PRESS the left mouse button to highlight your selection.

PRESS the right mouse button

- the Variable Options menu appears.

SELECT **Array Subrange**

- The Array Subrange window opens, shown in Figure 12.

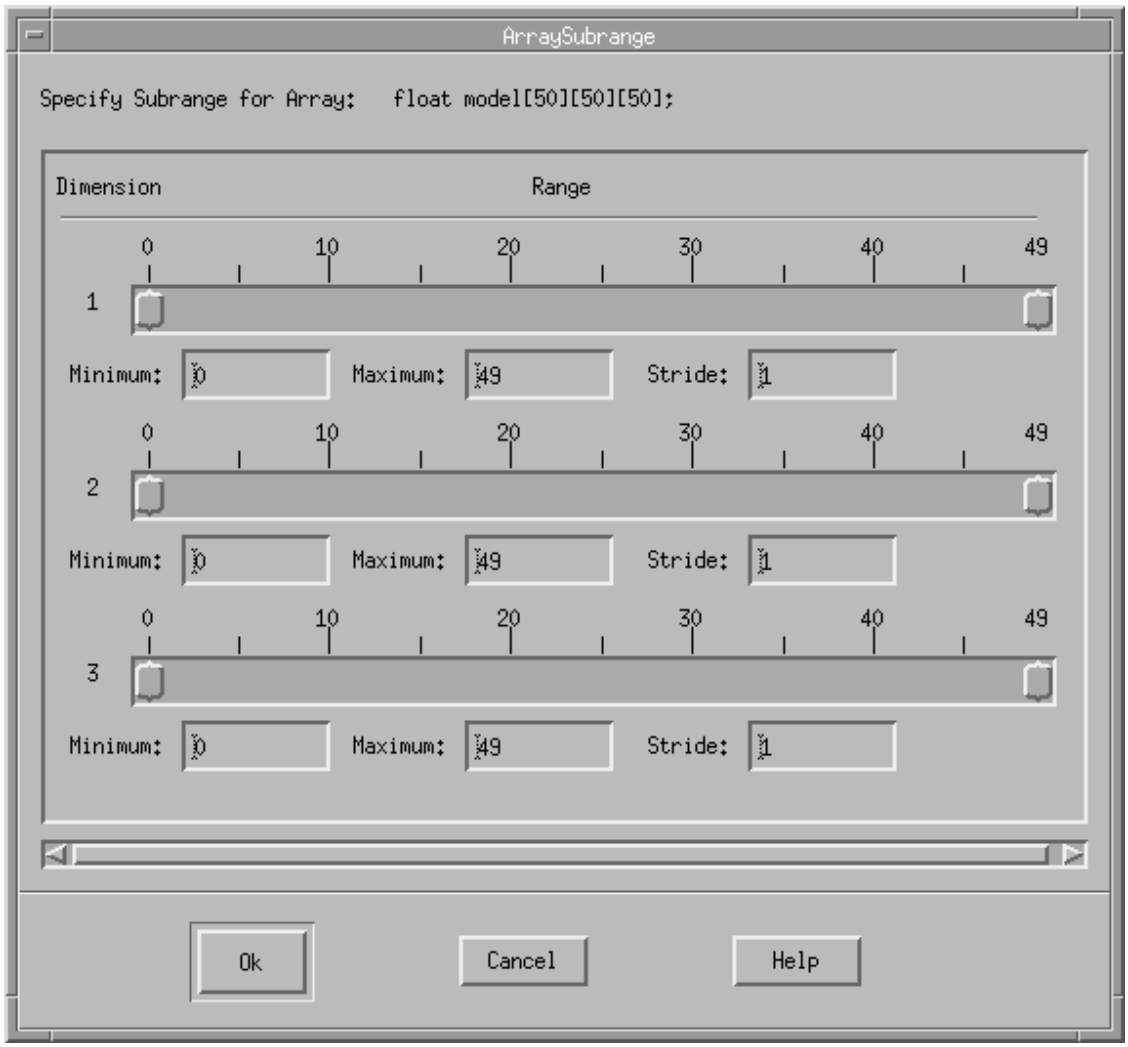


Figure 12. Array Subrange window

Specifying the Array Subrange

The **Specify Subrange for Array:** area of the Array Subrange window allows you to specify the set of array cells that you want to display (per dimension). Each array dimension is presented in the form of a slider, which is labelled with a scale that represents the actual range for that dimension of the array. The slider has two markers; one marks the minimum value of the subrange, the other marks the maximum value. These values are reflected in the Minimum and Maximum fields below the slider. To define a subrange for display, you choose the minimum and maximum values of the subrange individually for each dimension.

You can define a subrange in one of two ways:

PRESS the left mouse button to move the sliders horizontally left or right to mark the minimum and maximum values. These values appear in the Minimum and Maximum fields below the scale.

OR

CLICK ON the Minimum or Maximum field and clear the current value.

TYPE IN a value to define the new subrange.

The *Stride field* accepts non-zero integer values. This value allows you to skip elements for each range. The default is 1, which selects every element within the subrange for that dimension. A stride of 2 specifies every other element, and so forth. Specifying a negative stride value reverses the order of elements from which the subrange is selected. After doing this, the order goes from Maximum to Minimum, instead of Minimum to Maximum.

Note: The maximum number of array elements that can be displayed is 1000.

After you define the subrange with the appropriate values:

PRESS the **OK** button.

- The contents of the array elements that you specified are displayed, and the Array Subrange window closes. All subrange and stride specifications are retained for the next time the Array Subrange window is opened for this array on this task.

Cancel and Help Buttons

PRESS the **Cancel** button.

- The Array Subrange window closes, and all changes are discarded. All subrange and stride specifications when the window was opened are retained for the next time it is opened for this array on this task.

PRESS the **Help** button.

- Help information is displayed for the Array Subrange window.

Exporting Array Information to File

The Export window allows you to write elements of a C or Fortran array to a file in a specified data format. By defining the range of elements, you can control the portion of the array that is written to the file.

The format of the file is Hierarchical Data Format (HDF) Version 3.3. HDF is a standard format for scientific and visualization data. It was developed at the National Center for Supercomputing Applications (NCSA). See Appendix C, “Exporting Arrays to Hierarchical Data Format (HDF)” on page 285 for more information on HDF.

The Export function is only available for arrays of integer and floating point data types.

The function of the Export window is completely independent from the function of the Array Subrange window, and vice versa. This means that any specifications for a particular array in one of these windows will not affect the specifications for the same array in the other.

To open the Export window:

PLACE the mouse cursor over an array in the Local or Global variable list.

PRESS the left mouse button to highlight your selection.

PRESS the right mouse button

- The Variable Options menu appears.

SELECT Export to File...

- The Export window opens, shown in Figure 13.

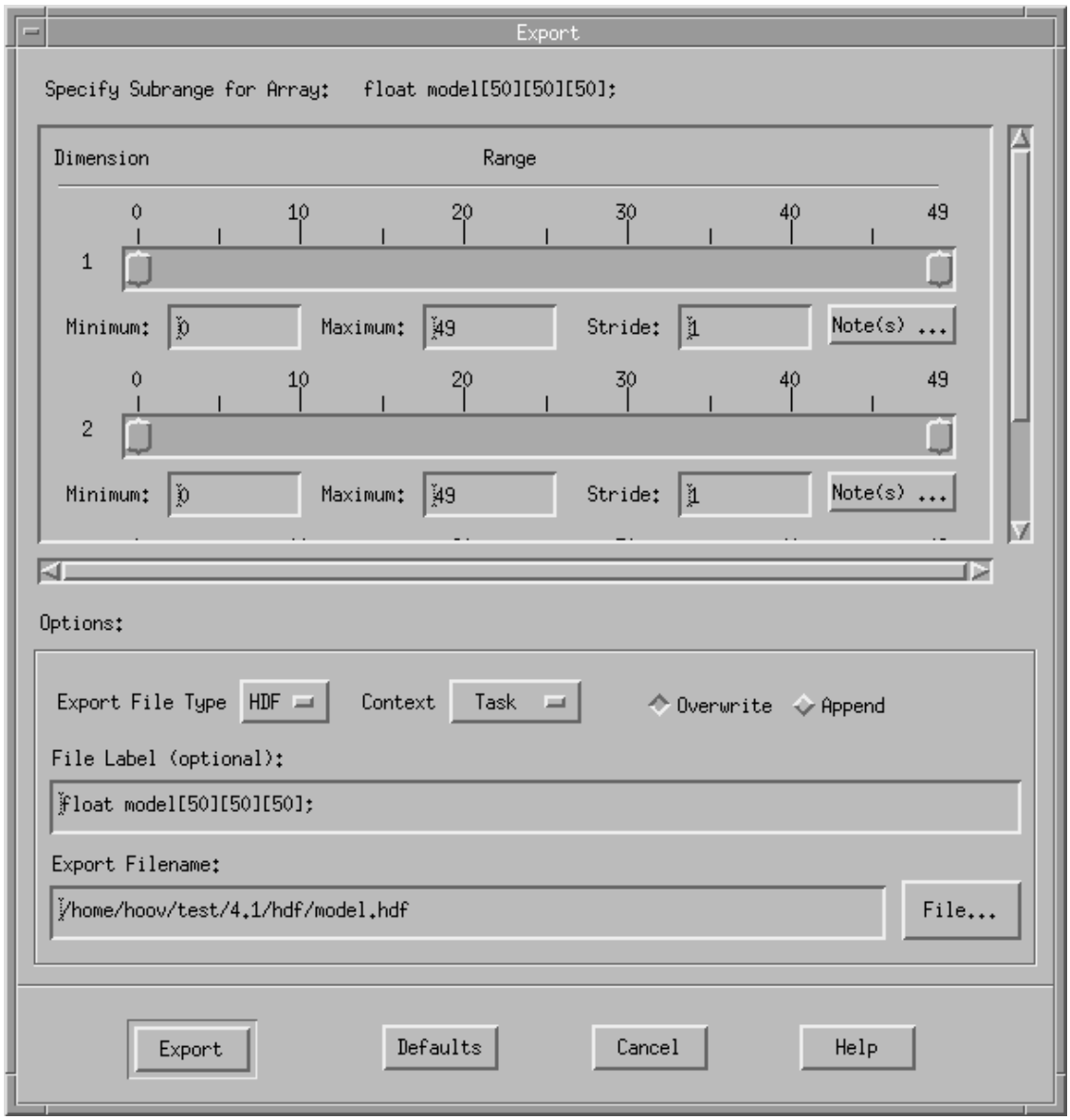


Figure 13. Export window

Specifying the Subrange for Export

As with the Array Subrange window, the Export window allows you to specify the subrange of an array. For details on this area, see “Specifying the Array Subrange” on page 74.

Dimension Attributes

The **Note(s) ...** pushbutton allows you to enter optional information pertaining to the associated array dimension. Clicking on this button displays the Optional Notes window, shown in Figure 14, with fields to enter attribute annotations that are specific to the output file type. To edit any of the text fields, click on the field and type in the desired text.

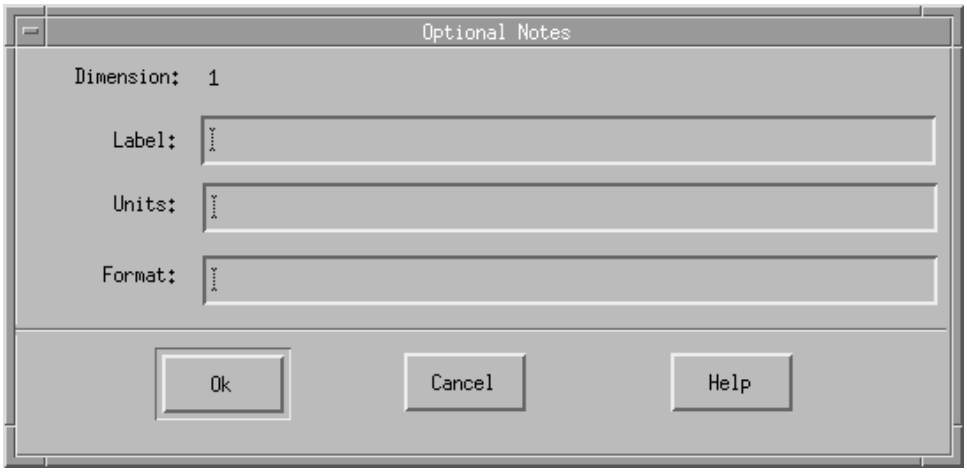


Figure 14. Optional Notes window

After you annotate the dimensions with the appropriate information:

PRESS the **OK** button.

- The notes you entered are saved and the Optional Notes window closes.

Cancel and Help Buttons

PRESS the **Cancel** button.

- The Optional Notes window closes, discarding any changes.

PRESS the **Help** button.

- Help information is displayed for the Optional Notes window.

Export Options

The following fields and buttons are available for you to specify aspects of the export:

Export File Type: Clicking on this with the left mouse button displays a menu that allows you to select the output format of the data to be written to a file. Currently, this menu has only one choice, which is HDF. This will result in the array contents being written in Hierarchical Data Format.

Context Setting: This option menu lets you select the tasks that will participate in the export, based on the criteria described below. This allows the selected array data to be written to one file in separate Scientific Data Sets, one set for each participating task.

The three choices for context are:

- "Task" - export the array data for the task from which this Export window was opened.
- "Current" - export the array data from each task within the current context. In this context the array must be displayed in all of the Global Data area task windows in the current context for all tasks to participate.
- "All" - export the array data from each of the tasks that are executing in this **pedb** debugging session. This is the same as the tasks which are included in

the task group “All” if you are running **pedb** in normal mode, or the task group “Attached” if you are running in attach mode. In this context, the array must be displayed in all of the Global Data area task windows for all tasks to participate.

If the context setting is either “Current” or “All,” the following criteria must be met for a task within the specified context to participate in the Export:

1. An array of the same name exists on each task (within the local or global block, depending on where the variable was selected).
2. The array on that task must have the same number of dimensions as the array on the task from which the Export window was opened.
3. The minimum element number for each dimension of the array must match those for the array on the task where the export is initiated. This is only a consideration with Fortran arrays, where a program can have arrays that are declared with any integer as the minimum element number. The maximum element numbers are not checked, in order to allow support for pointer variables and allocatable arrays.
4. If the array is a global array variable, then the array must be displayed in the associated task window of the Global Data area.
5. The task must be in “debug ready” state.

   If any of the tasks within the context do not meet all of the above criteria, they will be excluded from the export, and a message will be displayed to inform you of this.

   As stated above, the selected array data will be output to the HDF file as separate Scientific Data Sets. These data sets will be written to the file in order by task number.

Append and Overwrite Buttons: Clicking on the Append button results in the selected array contents to be appended to the end of an existing HDF file. This allows you to collect the contents of multiple subranges from the same array, or contents of multiple arrays, and group this information in any order within the file. Clicking on the Overwrite button causes the specified file to be created if it doesn't already exist, or be completely overwritten if it does exist. The default file writing method is Overwrite.

File Label: The File Label is a text string annotation that is written to the file just prior to writing the selected array element data for each task participating in the export. The File Label is an optional annotation which has the declaration of the array as a default text string. You can either change the label or eliminate it entirely by clicking in the text field and editing it.

Filename Specification: The **Export Filename:** field is a required field that specifies the name and path of the file that the array export data is written to. You can specify both the name of the file, and the directory where it will be located. There are two ways to do this:

CLICK ON the **Export Filename:** field with the left mouse button, and edit the current path/file text string.

OR

CLICK ON the **File...** button located to the right of the **Export Filename:** field. This opens the Export File Selection window, that allows you to select location and file name by choosing directories and files from selection lists.

Figure 15 shows the Export File Selection window. After making these selections, the **Export Filename:** field is updated with your choice.

The **Export Filename:** field contains a default path and file name every time the Export window is opened. The default path is the directory from which **pedb** was started, and the file name consists of the name of the array that you are exporting data for, along with a .hdf suffix.

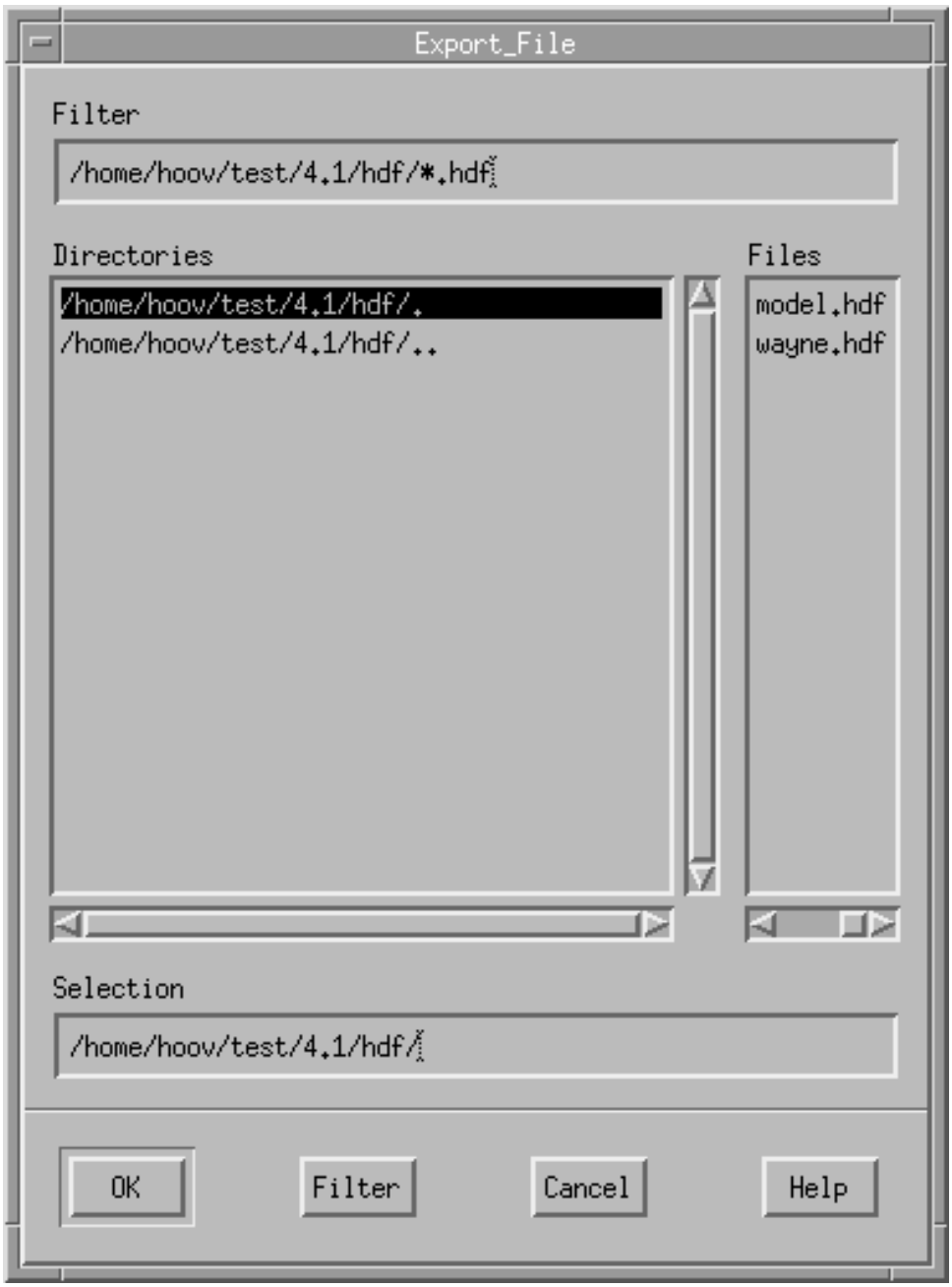


Figure 15. Export File Selection window

Export Button
Clicking on this button initiates the export of the selected array data to the specified file. The Export window remains open to allow additional exports on this array.

Stop Sign Icon: The export process can typically take more than a few seconds. When the export begins, an icon in the shape of a stop sign appears in the upper right hand corner of the Export window, and remains there until the export has completed. If you wish to stop an export that is in progress, click on this icon with the left mouse button.

Defaults Button
Clicking on this button will reset all fields and states in the Export window back to the default settings that were used the first time this window was opened for this array on this task. The subranges for all array dimensions will be set back to their full ranges.

Cancel Button
Pressing this button closes the window without exporting any data. The settings and specifications from the last export are retained for the next time the Export window is opened for this array on this task. If an export was not performed while the window was open, the settings and specifications when the window was opened for this array on this task are retained.

Help Button
This button displays help information for the Export window.

Visualizing Program Arrays
The Visualization window allows you to select elements of a C or Fortran array, and display the data in those elements using data visualization tools of your choice. By defining the range of elements, you can control the portion of the array that is visualized. Two types of visualizations are available with this feature: *pre-packaged* and *user-defined*.

The Visualization function is only available for arrays of integer and floating point data types.

To open the Visualization window:

PLACE the mouse cursor over an array from a local or global variable list.

PRESS the left mouse button to highlight your selection.

PRESS the right mouse button

- A pop-up menu appears.

SELECT **Visualize...**

- The Visualization window opens, shown in Figure 16.

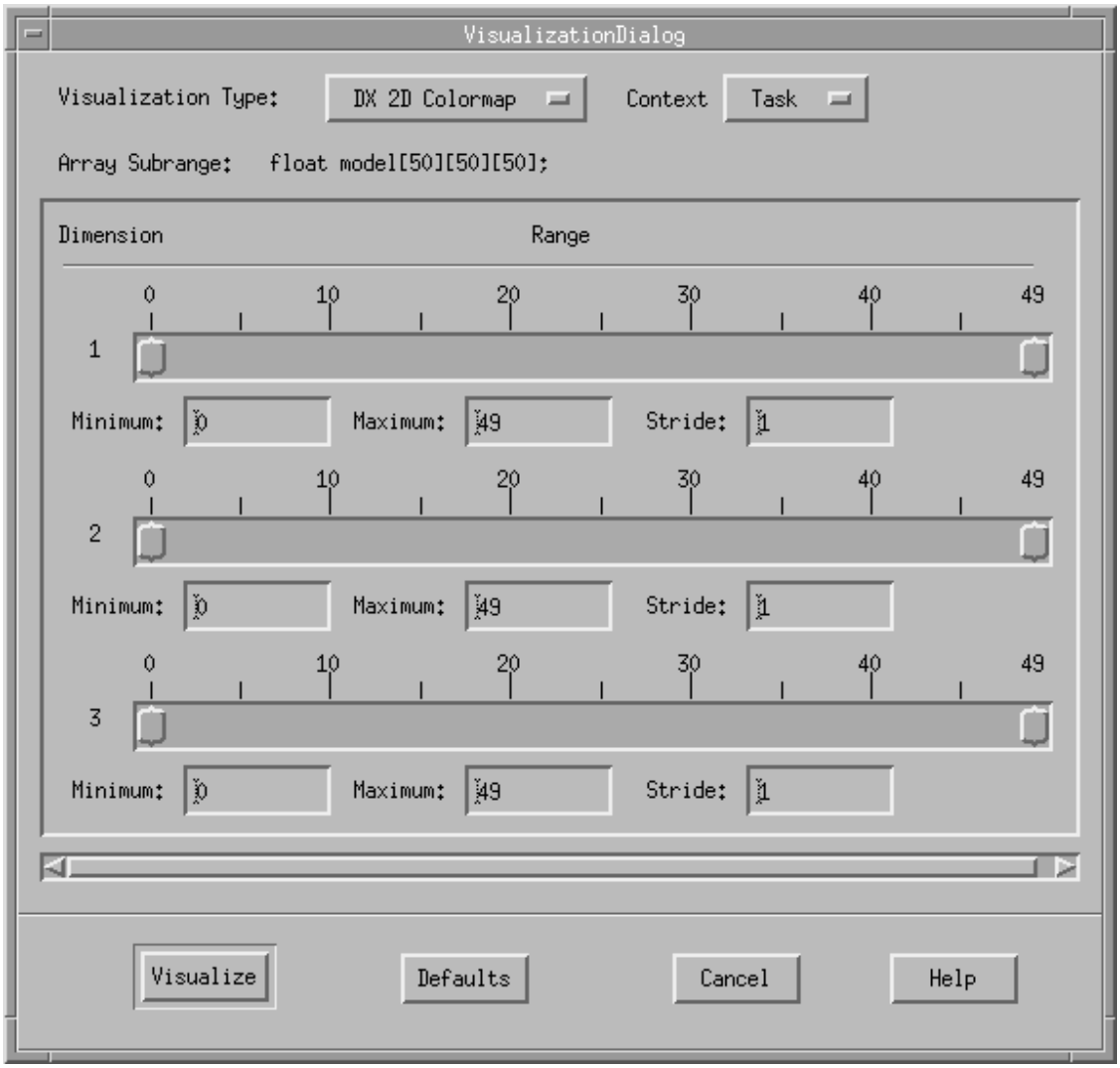


Figure 16. Visualization window

Visualization Type

Clicking on the raised button in this field results in a menu being displayed to allow you to choose the method for visualizing the selected array data. The first five menu options are pre-defined visualization methods for IBM's Visualization Data Explorer product. To execute these visualizations, the Data Explorer runtime fileset is required (see Appendix D, “Visualization Customization and Data Explorer Samples” on page 287 for more information). The last five options are available for you to call user-defined visualization tools. The default value is set to “DX 2D Colormap.” This menu, and the tools that can be invoked from it, are able to be customized.

The interface to the pre-defined samples is the same as that used with user-defined visualizations. Therefore, you can modify or replace them, if desired. Additional Data Explorer visualizations are included in a samples directory, but not pre-defined on the **pedb** user interface. The user-defined options are set up through the X defaults file and through a separate configuration script.

The following Data Explorer visualizations are pre-defined:

- 2D Colormap
- 2D Contour map
- Statistical analysis (text)
- 2D Height field (with a colormap)
- 3D Volume rendering.

The following are included in a samples directory:

- 3D Isosurface (3D contours)
- 3D Volume slicer.

The samples directory holds all of the visualization scripts (*.net* and *.cfg* files in Data Explorer terminology). It also includes the source for the program with which **pedb** interfaces to Data Explorer, and a makefile for the sample source.

User-defined Visualizations

You can insert your own visual tools for the debugger to invoke. When you define visualizations, you need a common method for integrating your applications.

A two step process is required to enable the tools you define and integrate with **pedb**:

1. Each menu option label has a corresponding label string entry in *Pedb.ad* (the **pedb** X defaults file). You change this string entry to your user tool name:

   ```
   Pedb*VisualTypeOption_1.labelString:  DX Colormap

   changes to

   Pedb*VisualTypeOption_1.labelString:  User Tool 1
   ```

   Note: You can edit your *.Xdefaults* file from your **$HOME** directory using the *Pedb.ad* file as a reference to the things you can change.

2. Each menu option has a corresponding Korn shell script in */usr/lpp/ppe.pedb/bin* that is executed to start the visualization:

   ```
   /usr/lpp/ppe.pedb/bin/VisualTypeOption_1.ksh
   ```

   By modifying this shell script, you can direct **pedb** to call your own tools.

Context Setting

This option menu lets you select the tasks for which array data will be visualized, based on the criteria described below. In the visualization process, the selected array data is written in HDF format to a temporary file located in /tmp. Certain context settings allow the selected array data to be written to the temporary file in separate Scientific Data Sets, one set for each participating task.

The three choices for context are:

- “Task” - visualize the array data for the task from which this Visualization window was opened.
- “Current” - visualize the array data from each task within the current context. In this context the array must be displayed in all of the Global Data area task windows in the current context for all tasks to participate.

- "All" - visualize the array data from each of the tasks that are executing in this **pedb** debugging session. This is the same as the tasks which are included in the task group "All" if you are running **pedb** in normal mode, or the task group "Attached" if you are running in attach mode. In this context, the array must be displayed in all of the Global Data area task windows for all tasks to participate.

Note: Context settings of "Current" or "All" result in multiple data sets being written to a temporary file for visualization. Verify that the tool you have chosen to perform the visualization allows you to visualize more that one data set at a time.

If the context setting is either "Current" or "All," the following criteria must be met for a task within the specified context to participate in the data visualization:

1. An array of the same name exists on each task (within the local or global block, depending on where the variable was selected).
2. The array on that task must have the same number of dimensions as the array on the task from which the Visualization window was opened.
3. The minimum element number for each dimension of the array must match those for the array on the task where the visualization is initiated. This is only a consideration with Fortran arrays, where a program can have arrays that are declared with any integer as the minimum element number. The maximum element numbers are not checked.
4. If the array is a global array variable, then the array must be displayed in the associated task window of the Global Data area.
5. The task must be in "debug ready" state.

   If any of the tasks within the context do not meet all of the above criteria, they will be excluded from the visualization, and a message will be displayed to inform you of this.

   As stated above, the selected array data will be output to the HDF file as separate Scientific Data Sets. These data sets will be written to the file in order by task number. The visualization program is required to determine the order in which the data will be visualized.

Array Subrange Area

For details on this area, see "Specifying the Array Subrange" on page 74.

Visualize Button

Clicking on this button initiates the visualization of the selected array data. The Visualization window remains open to allow additional visualizations on this array. The visualization program will be re-initialized each time this button is pressed. This will create multiple instances of the visual, rather than refreshing the data from the previous visualization.

Stop Sign Icon: Transferring data for visualization can typically take more than a few seconds. When the data transfer begins, an icon in the shape of a stop sign appears in the upper right hand corner of the Visualization window, and remains there until the data transfer has completed. If you wish to stop a data transfer that is in progress along with the visualization, click on this icon with the left mouse button.

Defaults Button

Clicking on this button will reset all fields and states in the Visualization window back to the default settings that were used the first time this window was opened for this array on this task. The subranges for all array dimensions will be set back to their full ranges.

Cancel Button

Pressing this button closes the window without initiating the visualization. The settings and specifications from the last visualization are retained for the next time the Visualization window is opened for this array on this task. If a visualization was not performed while the window was open, the settings and specifications when the window was opened for this array on this task are retained.

Help Button

This button displays help information for the Visualization window.

Source Code Control

During a **pedb** session, the source code file displayed in the Source Area may change many times. Each time execution of the tasks in a context stops, the debugger updates the **pedb** window and checks that the source code displayed matches the program counter. For example, if execution has stopped in a procedure located in a different file, the debugger automatically updates the Source Area to display the current source.

When tasks have stopped in different source files, the lowest numbered task in debugged state in the group determines the source file displayed. This excludes tasks which are unhooked, exit requested, or exited.

To display the source from a different task, you can either change the context or open a view with a different context. You can also change the current source code displayed by opening a source code file using the Source File(s) window, selecting a line in the stack window, or double clicking on a thread in the Threads window.

Opening a Source Code File

You can open a source code file and display it in the Source Area using the Source File(s) window. To do this:

SELECT File → Get Source File ...

- The Source File(s) window opens.

This window contains a list of accessible source files associated with your program that have been compiled with the **-g** flag. The source path is used to find the files.

PRESS the left mouse button to select a file in the list.

PRESS **OK**

OR

DOUBLE-CLICK on the desired file in the list.

- The File Selection window closes, and the source code of the selected file appears in the Source Area.

Source Code Search Path

The first default path searched is " **.** " (the current directory). If you do not explicitly specify a path when choosing a file to load, **pedb** uses the AIX path established by the **PATH** environment variable to locate the file. The path in which the program is located, whether explicitly specified or not, is added to the end of the list of directories searched for source files.

You may explicitly set the source code search path on the command line when invoking **pedb** using **-I** flags. The effective search path is set to the **-I** paths specified, in the order they appear on the command line. If you do not explicitly set it, the source code search path is based on the program(s) you load for debugging in the partition, as described above.

Note: In addition to having access to your program on each remote node, **pedb** requires source files on the home node to do source level debugging. See *IBM Parallel Environment for AIX: Operation and Use, Volume 1, Using the Parallel Operating Environment* for more information.

Source Path Window

During your **pedb** session, the search path used to locate source files may be modified. You may edit it, adding new paths or deleting or changing existing ones. This is helpful when source is distributed in multiple directories and you step into a source file which is located in a directory you missed at startup.

SELECT **File → Update Source Path ...**

- The Update Source Path window opens.

FOCUS on the edit field.

TYPE IN the new source search path, or modify the existing one.

PRESS OK

- The Update Source Path window closes. Subsequent source files will be accessed using the new path.

  Cancel closes the Update Source Path window without changing the current source path.

Edit Current Source File

You can edit the source file which is shown in the source area by selecting the **Edit Source File** menu from the **File** pulldown menu on the main window.

PRESS **File → Edit Source File**

- You open an edit session in an aixterm window.

Notes:

1. The editor used is determined by the **$EDITOR** environment variable.
2. If the source file in the Source Area is modified using the **Edit Source File** option, the program counter icon (→) may then be out of synch. This is because the line number information is based on the compiled version of the source. If you wish to continue debugging after editing your source file, consider saving it under a different name or directory instead of overwriting the copy that the debugger is referring to.

Source File, Variable Viewer, and Threads Viewer Find

Use the **Find** option to locate text in the source code, Variable Viewer, or Threads Viewer. You first open the Find window and specify the text to find. Once you have entered text, the find options are enabled. The find options are available from the menu bar pulldown and from buttons in the Find window. Accelerators <**ctrl-f**>, <**ctrl-n**>, <**ctrl-p**>, and <**ctrl-l**> are available for **First**, **Next**, **Previous**, and **Last** respectively. Search results are displayed differently for the source code window and the Variable or Threads Viewer.

To find text in the source code window, Variable Viewer, or Threads Viewer, go to the menu bar.

PRESS **Find**

- You see a pulldown menu with the following options:
  - **Open Find Dialog ...** - This option opens the Find window where you will enter the text to find.
  - **First** - Finds the first occurrence of the text.
  - **Next** - From the current line, finds the next occurrence of the text.
  - **Previous** - From the current line, finds the previous occurrence of the text.
  - **Last** - Finds the last occurrence of the text.

Using the Find Window

To open the Find window, go to the menu bar in the Main window or Variable Viewer.

From the Main window:

SELECT **Find → Find Text in the Source Window ...**

From the Variable Viewer:

SELECT **Find → Find ...**

- The Find window opens as shown in Figure 17. The Find window title will indicate if you are using **Find** from the source code window or the Variable Viewer.

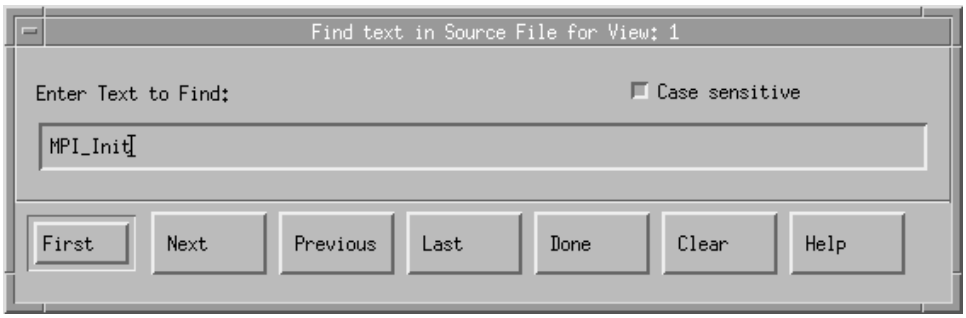


Figure 17. Find window

Enter the text to find in the text field. Entering text automatically enables the find option buttons in the window and the menu bar. Pressing **Enter** in the text field is the same as pressing the **Next** button. Use the **Case sensitive** toggle button to ignore the case of the text when searching. At the bottom of the Find window are the following buttons:

- Find options - These buttons (**First**, **Next**, **Previous**, and **Last**) and their corresponding buttons on the menu bar pulldown initiate a search for the text you typed.
- Done - closes the Find window.
- Clear - clears the text field. Note that the find options are desensitized (grayed out) when there is no text in the text field.
- Help - displays help text for using **Find**.

If the search fails, a message is displayed in the information area indicating the search direction and the actual string used in the search. You may want to broaden the search by specifying fewer characters or using the **Case sensitive** toggle button to ignore case.

When text is found in the source window or Threads Viewer, the entire line containing the text is highlighted. This becomes the current line, and the reference point for locating the next and previous occurrences.

When text is found in the Variable Viewer, the text that was matched is highlighted. The first character of the text is the reference point for the **Variable Options** pop-up menu as described in “Data Display Techniques” on page 70.

Source Emphasis

pedb provides source code emphasis when displaying code in the source area of the main window. For example, the language symbols, variable and function names, and comments are all displayed in different colors. See Table 9 below for details.

The debugger scans the current source file to identify elements of the language. Each element is then drawn with a different foreground and background color to emphasize that element. This may help you to quickly identify points of interest in your source code. It is particularly useful when you are not familiar with the code being debugged. Instead of scrutinizing the code to identify variables and comment blocks, their color will automatically alert you to their function.

To turn this feature off, set `Pedb*SourceEmphasis: False` in your *.Xdefaults* file, or use the toggle button on the menu bar:

File → Source Emphasis

The following table lists the default color scheme for each language element identified. Each resource can be given a unique color, but be aware that each unique color used will increase the total number of colors required for **pedb**. Resource values are any valid color specification for your system, for example, the values found in */usr/lpp/X11/lib/X11/rgb.txt*.

Table 9. Default Color Scheme

Resource	Language Element
Pedb*AlphaForeground: Pedb*AlphaBackground:	Alphanumerics - variable and functions names.
Pedb*CommentForeground: Pedb*CommentBackground:	Comments
Pedb*MessageForeground: Pedb*MessageBackground:	Message passing routines - MPL and MPI.

Pedb*KeywordForeground:
Pedb*KeywordBackground:
	Language specific keywords. For example
	`int`
	`class`
	`subroutine`

Pedb*LiteralForeground: Pedb*LiteralBackground:	Literal (quoted) strings.
Pedb*LinenumForeground: Pedb*LinenumBackground:	Line numbers. See note 2 below.
Pedb*NumberForeground: Pedb*NumberBackground:	Numbers.

Pedb*PreprocForeground:
Pedb*PreprocBackground:
	Preprocessor directives, for example
	`#ifdef`
	`#include`
	`#pragma`

Pedb*SymbolForeground: Pedb*SymbolBackground:	Language symbols (punctuation).
Pedb*LabelForeground: Pedb*LabelBackground:	Fortran labels.
Pedb*FortranProfile:	Fortran parser profile. See note 3 below.
Pedb*ProfilePath:	Path to the Fortran parser profile. See note 3 below.

Notes:

1. This feature may not properly identify all elements of your source code in all instances. **pedb** can only scan the current source file. It is not aware of other aspects of the compilation used to produce the executable. For example, a section of code may be bracketed by the directive

   ```
   #ifdef DEBUG
   #endif
   ```

   pedb will identify the directives and the elements within the block, but cannot determine if `DEBUG` was set at compile time.

2. This feature may also be used to turn off the display of line numbers in the source code area. By setting the *Pedb*LinenumForeground:* resource to the same value as *Pedb*LinenumBackground:*.
3. For expert users, **pedb** uses a profile driven parser for the Fortran source emphasis. There are 2 profiles provided in */usr/lpp/ppe.pedb/bin/FOR.PPR* and */usr/lpp/ppe.pedb/bin/FF.PPR*. *FF.PPR* supports free form Fortran, the default *FOR.PPR* supports fixed form. Experienced users may wish to change the default profile using the *Pedb*FortranProfile:* resource, or use a local copy by changing the *Pedb*ProfilePath:*.

Other Key Features

Some other features offered by **pedb** include the capabilities of displaying multiple views, and linking to online help.

Debugging Programs Using Multiple Views

You can think of **pedb** as a *window* into the *debug space*. The window you have is just one way of looking into the debug space, and depends on the current context, the source code displayed in the Source Area, the variables, and the stack trace. You can open multiple **pedb** windows and have multiple views into the same debug space.

For example, you have two tasks – tasks 0 and 1 – involved in message passing. You could open two **pedb** windows to follow send and receive pairs between the two tasks. In one window, you would set the context on task 0. In the other, you would set the context on task 1. You could then step execution past the send in one window, and then step execution past the receive in the other.

When dealing with multiple views into the same debug space, keep in mind that actions made on one **pedb** window may be reflected on the others. For example, say you have two views into the same debug space. The context of one is set on just task 0, while the context of the other is set on all the tasks including task 0. If you step all the tasks in the second window, the first window also reflects the step.

To open another **pedb** window to provide an additional view into the debug space:

SELECT View → Open new view

- Another **pedb** window opens.

To close a **pedb** window:

SELECT View → Close this view

Getting Help

There are help buttons on most windows and help options on many menus. Selecting these help buttons or options provide built-in online help for the particular window or menu from which the help was selected.

The **pedb** main window includes a Help button to access online help in a variety of ways by selecting one of the following options:

Help on Main Window: Displays built-in online help information about the **pedb** main window. There are main window left and right button presses that may not be obvious. Here you will find a list of the actions available using the buttons on the main window.

Help on Main Window Menu Bar: Displays built-in online help for the main window menu bar pop-up menus: **File**, **View**, **Group**, **Find**, and **Options**.

Index of Online Help Topics: Displays a list of the built-in online help items that are available on the various windows and menus throughout the **pedb** debugger. Any of the listed items can be selected, which results in a window displaying that help section.

Notes:

1. The main window menu bar pull downs (**File**, **View**, **Group**, **Find**) do not have help options. You can get help about options by pressing the Help button in the upper right corner of the main window, then selecting the **Help on Main Menu Bar** option.
2. The menu bar pull downs in the Local or Global Variable windows do not have help options. You can get help about these options by pressing the Help button in the upper right hand corner of the Local or Global Variable windows.

Customizing pedb Resources

Customizable resources for **pedb** are defined in */usr/lpp/ppe.pedb/defaults/Pedb.ad* (the **pedb** X defaults file). In this file is a set of X resources for defining graphical user interfaces based on the following criteria:

- Window geometry
- Push button and label text
- Pixmaps.

Leaving pedb

It is possible to end the debug session at any time using either the **Quit** option, or the **Detach** option if debugging in attach mode.

To end a debug session in normal mode:

SELECT **File** → **Quit** from the **pedb** Main Window.

- The Quit Confirmation window appears.

PLACE the mouse cursor over **OK**.

PRESS the left mouse button.

- The **pedb** window closes and you return to the window from which you started **pedb**.

To end a debug session in attach mode, you can choose either **Quit** or **Detach**. Quitting causes the debugger and all the members of the original **poe** application partition to exit. Detaching causes only the debugger to exit and leaves all the tasks running.

SELECT **File** → **Quit** from the **pedb** Main Window.

- The Quit Confirmation window appears.

PLACE the mouse cursor over **OK**.

PRESS the left mouse button.

- The debugger session ends, along with the **poe** application partition tasks.

OR

SELECT **File** → **Detach** from the **pedb** Main Window.

- The Detach Confirmation window appears.

PLACE the mouse cursor over **Detach**.

PRESS the left mouse button.

- The debugger session ends. All tasks have exited, but stay running.

Clicking on this button causes **pedb** to exit, and allows the program to which you had attached to continue execution if it hasn't already finished. If this program has finished execution, and is part of a series of job steps, then detaching allows the next job step to be executed.

If instead you want to exit the debugger and end the program, cancel the Detach Confirmation window and use the **Quit** option as described above.

Chapter 3. Profiling Parallel Programs with Xprofiler

This chapter describes how to profile your programs with the Xprofiler profiling tool of the IBM Parallel Environment for AIX. This chapter explains how to use the Xprofiler graphical user interface (GUI) to profile your application, so it is best to read it while you have the GUI up and running.

If you intend to use the AIX **gprof** command to profile your parallel application, see Appendix F, "Profiling Programs with the AIX prof and gprof Commands" on page 297 for information on how to do so. You may also find it helpful to consult the *IBM AIX Version 4 Commands Reference*.

You do not need to be familiar with the AIX **gprof** command to use Xprofiler.

Xprofiler is a tool that helps you analyze your parallel application's performance quickly and easily. It uses data collected by the **-pg** compiling option to construct a graphical display of the functions within your application. Xprofiler provides quick access to the profiled data, which lets you identify the functions that are the most CPU-intensive. The graphical user interface also lets you manipulate the display in order to focus on the application's critical areas.

Before You Begin

About Xprofiler

Xprofiler lets you profile both serial and parallel applications. The difference is that when you run a serial application, a single profile data file is generated, while a parallel application produce multiple profile data files. Either way, you can use Xprofiler to analyze the resulting profiling information.

Xprofiler provides a set of resource variables that let you customize some of the features of the Xprofiler window and reports. For information about customizing resources for Xprofiler, see *IBM Parallel Environment for AIX: Operation and Use, Volume 2, Tools Reference*.

The word *function* is used frequently throughout this chapter. Consider it to be synonymous with the terms *routine*, *subroutine*, and *procedure*.

Requirements and Limitations

To use Xprofiler, your application must be compiled with the **-pg** option. For more information about compiling, see "Compiling Applications to be Profiled" on page 94.

Like gprof, Xprofiler lets you analyze CPU (busy) usage only. It cannot give you other kinds of information such as CPU idle, I/O, or communication.

To run Xprofiler, AIX 4.2.1 (or later) must be installed on your machine.

If you compile your application on one machine, and then analyze it on another, you must first make sure that both machines have similar library configurations, at least for the system libraries used by the application. For instance, say you ran an HPF application on an SP, then tried to analyze the profiled data on a workstation.

© Copyright IBM Corp. 1996, 1997

The levels of HPF runtime libraries must match, and must be placed in a location that Xprofiler recognizes on the workstation. Otherwise, Xprofiler produces unpredictable results.

Since Xprofiler collects data by sampling, short-executing functions may show no CPU use.

Xprofiler does not give you information about the specific threads in a multi-threaded program. The data that Xprofiler presents is a summary of the activities of all the threads.

Xprofiler versus gprof

With Xprofiler, you can produce the same tabular reports that you may be accustomed to seeing with **gprof**. Just as with **gprof**, you can generate the Flat Profile, Call Graph Profile, and Function Index reports. Xprofiler is different from **gprof** in that it provides a graphical user interface (GUI) from which you can profile your application. It generates a graphical display of your application's performance, as opposed to just a text-based report. Unlike **gprof**, Xprofiler also lets you profile your application at the source statement level.

From the Xprofiler GUI, you can use all the same command line flags as gprof, plus a few more that are unique to Xprofiler.

Compiling Applications to be Profiled

In order to use Xprofiler, you must compile and link your application with the **-pg** option of the compiler command. This applies regardless of whether you are compiling a serial or parallel application. You can compile and link your application all at once, or perform the compile and link operations separately. Here's an example of how you would compile and link all at once:

```
cc -pg -o foo foo.c
```

And here's an example of how you would first compile your application and then link it. To compile:

```
cc -pg -c foo.c
```

To link:

```
cc -pg -o foo foo.o
```

Notice that when you compile and link separately, you must use the -pg option with *both* the compile and link commands.

The **-pg** option compiles and links the application so that when you run it, the CPU usage data gets written to one or more output files. For a serial application, this output consists of just one file called *gmon.out*, by default. For parallel applications, the output is written into multiple files, one for each task that is running in the application. To prevent each output file from overwriting the others, POE appends the task ID to each gmon.out file. For instance, *gmon.out.10*.

The **-pg** option is not a combination of the **-p** and the **-g** compiling options.

Note: You must set the LIBPATH environment variable to the profiled POE libraries in order to profile system libraries (like libc). For more information,

see *IBM Parallel Environment for AIX: Operation and Use, Volume 2, Tools Reference*.

In order to get a complete picture of your parallel application's performance, you must indicate all of its gmon.out files when you load the application into Xprofiler. When you specify more than one gmon.out file, Xprofiler shows you the sum of the profile information contained in each file.

The Xprofiler GUI provides the capability of viewing included functions. Note, however, that your application must also be compiled with the **-g** option in order for Xprofiler to display the included functions.

The **-g** option, in addition to the **-pg** option, is also required for source statement profiling.

Starting Xprofiler

You start Xprofiler from the AIX command line, using the **xprofiler** command. To use Xprofiler, you also need to specify the executable file (a.out), the profile data file (gmon.out), and any command line options.

Under some circumstances you may have multiple profile data files (gmon.out files). You will have more than one of these files if you are profiling a parallel application, because a gmon.out file is created for each task in the application when it is run. If you are running a serial application, there may be times when you want to summarize the profiling results from multiple runs of the application. In these cases, you will need to specify each of the profile data files you want to profile with Xprofiler.

You start Xprofiler by issuing the **xprofiler** command from the AIX command line. You also need to specify the executable, profile data file(s), and options, which you can do one of two ways. You can either specify them on the command line, with the **xprofiler** command, or you can issue the **xprofiler** command alone, then specify them from within the GUI. See “Loading Files from the Xprofiler GUI” on page 98.

To start Xprofiler and specify the executable, profile data file(s), and options:

ENTER **xprofiler** *a.out gmon.out*... [*options*]

where *a.out* is the binary executable file, *gmon.out* is the name of your profile data file(s), and *options* may be one or more of the options listed in “Xprofiler Command Line Options.”

To print basic Xprofiler command syntax to the screen, use the *-h* or *-?* option with the **xprofiler** command from the command line. For example, **xprofiler** *-h*.

Xprofiler Command Line Options

You can specify the same command line options with the **xprofiler** command that you do with **gprof**, plus one additional option (**-disp_max**), which is specific to Xprofiler. The command line options let you control the way Xprofiler displays the profiled output.

When you enter an option, there must be a space between the option and its corresponding value. For example,

```
-e main
```

You can specify the following options from either the Xprofiler GUI or the command line. See “Specifying Command Line Options (from the GUI)” on page 103 for more information.

The Xprofiler command line options are:

Table 10 (Page 1 of 3). Xprofiler Command Line Options

Use this flag:	To:	For example:
-b	Suppresses the printing of the field descriptions for the Flat Profile, Call Graph Profile, and Function Index reports when they are written to a file with the **Save As** option of the File menu.	To suppress printing of the field descriptions for the Flat Profile, Call Graph Profile, and Function Index reports in the saved file: file, **ENTER** **xprofiler -b** *a.out gmon.out*
-s	If multiple gmon.out files are specified when Xprofiler is started, produces the gmon.sum profile data file. The gmon.sum file represents the sum of the profile information in all the specified profile files. Note that if you specify a single gmon.out file, the gmon.sum file contains the same data as the gmon.out file.	To write the sum of the data from three profile data files, *gmon.out.1*, *gmon.out.2*, and *gmon.out.3*, into a file called *gmon.sum*: **ENTER** **xprofiler -s** *a.out gmon.out.1 gmon.out.2 gmon.out.3*
-z	Includes functions that have both zero CPU usage and no call counts in the Flat Profile, Call Graph Profile, and Function Index reports. A function will not have a call count if the file that contains its definition was not compiled with the **-pg** option, which is common with system library files.	To include all functions used by the application, in the Flat Profile, Call Graph Profile, and Function Index reports, that have zero CPU usage and no call counts: **ENTER** **xprofiler -z** *a.out gmon.out*
-disp_max	Sets the number of function boxes that Xprofiler initially displays in the function call tree. The value supplied with this flag can be any integer between 0 and 5,000. Xprofiler displays the function boxes for the most CPU-intensive functions through the number you specify. For instance, if you specify 50, Xprofiler displays the function boxes for the 50 functions in your program that consume the most CPU. After this, you can change the number of function boxes that are displayed via the Filter menu options. This flag has no effect on the content of any of the Xprofiler reports.	To display the function boxes for only 50 most CPU-intensive functions in the function call tree: **ENTER** **xprofiler -disp_max** *50 a.out gmon.out*
-e	De-emphasizes the general appearance of the function box(es) for the specified function(s) in the function call tree, and limits the number of entries for these function in the Call Graph Profile report. This also applies to the specified function's descendants, as long as they have not been called by non-specified functions. In the function call tree, the function box(es) for the specified function(s) appears greyed-out. Its size and the content of the label remain the same. This also applies to descendant functions, as long as they have not been called by non-specified functions. In the Call Graph Profile report, an entry for the specified function only appears where it is a child of another function, or as a parent of a function that also has at least one non-specified function as its parent. The information for this entry remains unchanged. Entries for descendants of the specified function do not appear unless they have been called by at least one non-specified function in the program.	To de-emphasize the appearance of the function boxes for *foo* and *bar*, as well as their qualifying descendants in the function call tree, and limit their entries in the Call Graph Profile report: **ENTER** **xprofiler -e** *foo* **-e** *bar a.out gmon.out*

Table 10 (Page 2 of 3). Xprofiler Command Line Options

Use this flag:	To:	For example:
-E	Changes the general appearance and label information of the function box(es) for the specified function(s) in the function call tree. Also limits the number of entries for these functions in the Call Graph Profile report, and changes the CPU data associated with them. These results also apply to the specified function's descendants, as long as they have not been called by non-specified functions in the program. In the function call tree, the function box for the specified function appears greyed-out, and its size and shape also changes so that it appears as a square of the smallest allowable size. In addition, the CPU time shown in the function box label, appears as 0 (zero). The same applies to function boxes for descendant functions, as long as they have not been called by non-specified functions. This option also causes the CPU time spent by the specified function to be deducted from the left side CPU total in the label of the function box for each of the specified function's ancestors. In the Call Graph Profile report, an entry for the specified function only appears where it is a child of another function, or as a parent of a function that also has at least one non-specified function as its parent. When this is the case, the time in the *self* and *descendants* columns for this entry is set to 0 (zero). In addition, the amount of time that was in the *descendants* column for the specified function is subtracted from the time listed under the *descendants* column for the profiled function. As a result, be aware that the value listed in the *% time* column for most profiled functions in this report will change.	To change the display and label information for *foo* and *bar*, as well as their qualifying descendants in the function call tree, and limit their entries and data in the Call Graph Profile report: **ENTER** **xprofiler -E** *foo* **-E** *bar a.out gmon.out*
-f	De-emphasizes the general appearance of all function boxes in the function call tree, *except* for that of the specified function(s) and its descendant(s). In addition, the number of entries in the Call Graph Profile report for the non-specified functions and non-descendant functions is limited. The **-f** flag overrides the **-e** flag. In the function call tree, all function boxes *except* for that of the specified function(s) and its descendant(s) appear greyed-out. The size of these boxes and the content of their labels remain the same. For the specified function(s), and its descendant(s), the appearance of the function boxes and labels remain the same. In the Call Graph Profile report, an entry for a non-specified or non-descendant function only appears where it is a parent or child of a specified function or one of its descendants. All information for this entry remains the same.	To de-emphasize the display of function boxes for all functions in the function call tree *except* for *foo*, *bar*, and their descendants, and limit their types if entries in the Call Graph Profile report: **ENTER** **xprofiler** -f *foo* **-f** *bar a.out gmon.out*

Table 10 (Page 3 of 3). Xprofiler Command Line Options

Use this flag:	To:	For example:
-F	Changes the general appearance and label information of all function boxes in the function call tree *except* for that of the specified function(s) and its descendants. In addition, the number of entries in the Call Graph Profile report for the non-specified and non-descendant functions is limited, and the CPU data associated with them is changed. The **-F** flag overrides the **-E** flag. In the function call tree, all function boxes *except* for that of the specified function(s) and its descendant(s) appear greyed-out. The size and shape of these boxes also changes so that they appear as squares of the smallest allowable size. In addition, the CPU time shown in the function box label, appears as 0 (zero). In the Call Graph Profile report, an entry for a non-specified or non-descendant function only appears where it is a parent or child of a specified function or one of its descendants. When this is the case, the time in the *self* and *descendants* columns for this entry is set to 0 (zero). As a result, be aware that the value listed in the *% time* column for most profiled functions in this report will change.	To change the display and label information of the function boxes for all functions *except* the functions *foo* and *bar* and their descendants, and limit their types of entries and data in the Call Graph Profile: **ENTER** **xprofiler** -F *foo* **-F** *bar a.out gmon.out*
-L	Sets the pathname for locating shared libraries. If you plan to specify multiple paths, use the *Set File Search Path* option of the File menu on the Xprofiler GUI. See "Setting the File Search Sequence" on page 105 for information.	To specify */lib/profiled/libc.a:shr.o* as an alternate pathname for your shared libraries: **ENTER** **xprofiler -L** */lib/profiled/libc.a:shr.o*

After you issue the **xprofiler** command, the Xprofiler main window appears, and displays your application's data.

Loading Files from the Xprofiler GUI

If you issue the **xprofiler** command without specifying an executable file, a profile data file, or options, you may do so from within the Xprofiler GUI. You use the *Load File* option of the File menu to do this.

When you issue the **xprofiler** command alone, the Xprofiler main window appears. Since you did not load an executable or specify a profile data file, the window will be empty.

If you issue the **xprofiler** command with the **-h** option only, Xprofiler displays the syntax for the command and then exits.

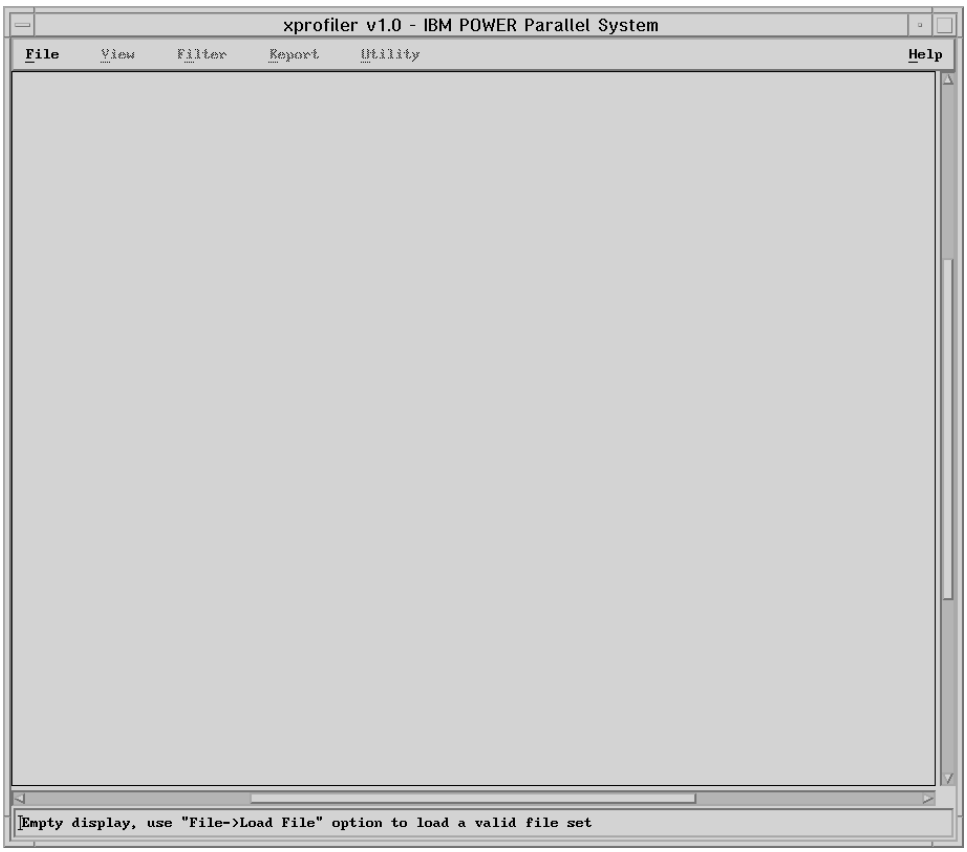


Figure 18. Xprofiler Main Window with No Executables Loaded

Select the File menu, and then the *Load File* option. The *Load Files Dialog* window appears.

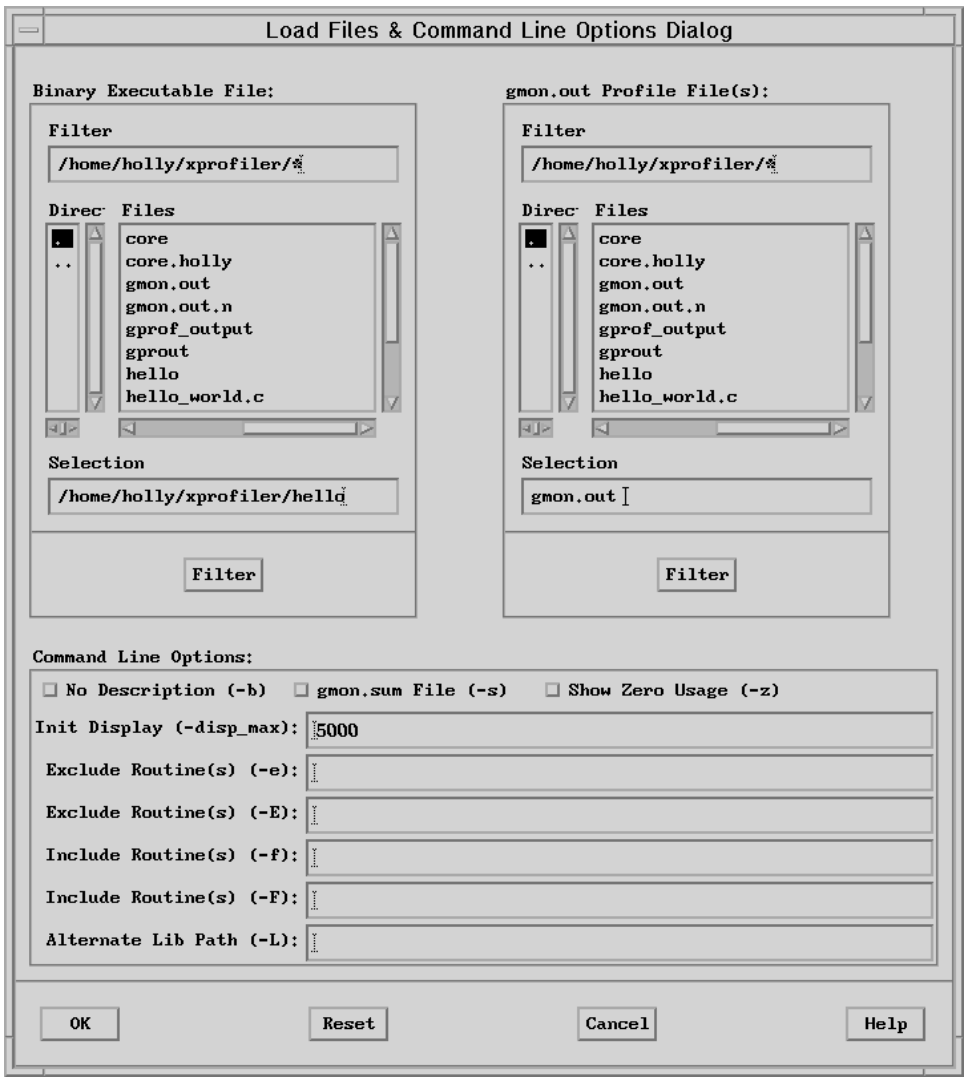


Figure 19. Load Files Dialog Window

The Load Files Dialog window lets you specify your application's executable, and its corresponding profile data (gmon.out) files. When you load a file, you can also specify the various command line options that let you control the way Xprofiler displays the profiled data.

To load the files for the application you wish to profile, you need to specify the:

- *Binary Executable* (required)
- *Profile Data File(s)* (required)
- *Command Line Options* (optional)

Specifying the Binary Executable

You specify the binary executable from the *Binary Executable File* of the *Load Files Dialog* window. The Binary Executable File area looks similar to this:

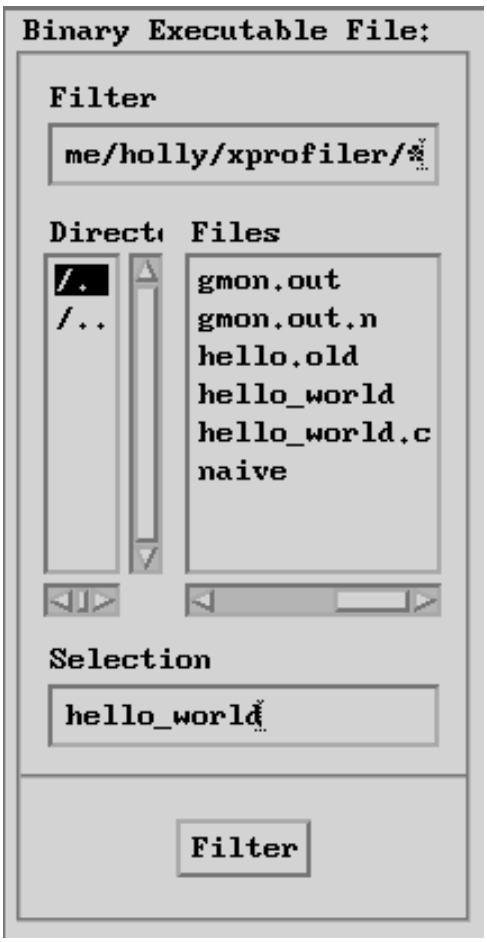


Figure 20. Binary Executable File Area

Use the scroll bars of the *Directories* and *Files* selection boxes to locate the executable file you wish to load. By default, all the files in the directory from which you invoked Xprofiler appear in the Files selection box. To select a file, click on it with the left mouse button.

To make locating your binary executable files easier, the Binary Executable File area includes a *Filter* button. Filtering lets you limit the files that are displayed in the *Files* selection box to those of a specific directory or of a specific type. For information on using the Filter, see "Using the Dialog Window Filters" on page 113.

Specifying the Profile Data File(s)

You specify the profile data file(s) from the *gmon.out Profile Data File(s)* area of the *Load Files Dialog* window. The gmon.out Profile Data File(s) area looks similar to this:

Figure 21. gmon.out Profile Data File(s) Area

When you started Xprofiler, with the **xprofiler** command, you were not required to indicate the name of the profile data file (which is probably why you are specifying it from the GUI). If you did not specify a profile data file, Xprofiler searches your directory for the presence of a file named *gmon.out* and, if found, places it in the *Selection* field of the gmon.out Profile File(s) area, as the default. Xprofiler then uses this file as input, even if it is not related to the binary executable file you specify. Since this will cause Xprofiler to display incorrect data, it is important that you enter the correct file into this field. So, if the profile data file you wish to use is named something other than what appears in the Selection field, you must replace it with the correct file name, as follows.

Use the scroll bars of the *Directories* and *Files* selection boxes to locate one or more of the profile data (gmon.out) files you wish to specify. The file you use does not have to be named *gmon.out*, and you may specify more than one profile data file. To select a file, click on it with the left mouse button. You can select multiple files by holding down the <**Ctrl**> key and clicking on each one with the left mouse button. To select multiple consecutive files, press and hold the left mouse button over the first file, and then drag the mouse over the other files. To de-select a file, press and hold the <**Ctrl**> key while clicking on the file.

To make locating your output files easier, the gmon.out Profile File(s) area includes a *Filter* button. Filtering lets you limit the files that get displayed in the Files selection box to those in a specific directory or of a specific type. For information on using the Filter, see “Using the Dialog Window Filters” on page 113.

Specifying Command Line Options (from the GUI)

You specify command line options from the *Command Line Options* area of the *Load Files Dialog* window. The Command Line Options area looks similar to this:

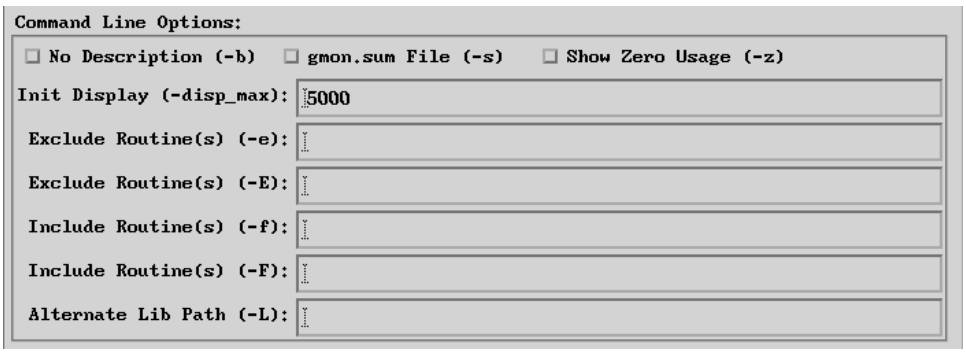


Figure 22. Command Line Options Area

You may specify one or more options as follows:

Table 11 (Page 1 of 3). Xprofiler GUI Command Line Options

Use this flag:	To:	For example:
-b (button)	Suppresses the printing of the field descriptions for the Flat Profile, Call Graph Profile, and Function Index reports when they are written to a file with the **Save As** option of the File menu.	To suppress printing of the field descriptions for the Flat Profile, Call Graph Profile, and Function Index reports in the saved file, set the *-b* button to the pressed-in position.
-s (button)	If multiple gmon.out files are specified in the *gmon.out Profile File(s)* area, produces the gmon.sum profile data file. The gmon.sum file represents the sum of the profile information in all the specified profile files. Note that if you specify a single gmon.out file, the gmon.sum file contains the same data as the gmon.out file.	To write the sum of the data from three profile data files, *gmon.out.1*, *gmon.out.2*, and *gmon.out.3*, into a file called *gmon.sum*, set the *-s* button to the pressed-in position to activate this option.
-z (button)	Includes functions that have both zero CPU usage and no call counts in the Flat Profile, Call Graph Profile, and Function Index reports. A function will not have a call count if the file that contains its definition was not compiled with the **-pg** option, which is common with system library files.	To include all functions used by the application, in the Flat Profile, Call Graph Profile, and Function Index reports, that have zero CPU usage and no call counts, set the *-z* button to the pressed-in position to activate this option.
-disp_max (field)	Sets the number of function boxes that Xprofiler initially displays in the function call tree. The value supplied with this flag can be any integer between 0 and 5,000. Xprofiler displays the function boxes for the most CPU-intensive functions through the number you specify. For instance, if you specify 50, Xprofiler displays the function boxes for the 50 functions in your program that consume the most CPU. After this, you can change the number of function boxes that are displayed via the Filter menu options. This flag has no effect on the content of any of the Xprofiler reports.	To display the function boxes for only 50 most CPU-intensive functions in the function call tree, type 50 in the *Init Display (-disp_max) field*

Table 11 (Page 2 of 3). Xprofiler GUI Command Line Options

Use this flag:	To:	For example:
-e (field)	De-emphasizes the general appearance of the function box(es) for the specified function(s) in the function call tree, and limits the number of entries for these function in the Call Graph Profile report. This also applies to the specified function's descendants, as long as they have not been called by non-specified functions. In the function call tree, the function box(es) for the specified function(s) appears greyed-out. Its size and the content of the label remain the same. This also applies to descendant functions, as long as they have not been called by non-specified functions. In the Call Graph Profile report, an entry for the specified function only appears where it is a child of another function, or as a parent of a function that also has at least one non-specified function as its parent. The information for this entry remains unchanged. Entries for descendants of the specified function do not appear unless they have been called by at least one non-specified function in the program.	To de-emphasize the appearance of the function boxes for *foo* and *bar*, as well as their qualifying descendants in the function call tree, and limit their entries in the Call Graph Profile report, type *foo* and *bar* in the *Exclude Routines (-e)* field. You specify multiple functions by separating each one with a space.
-E (field)	Changes the general appearance and label information of the function box(es) for the specified function(s) in the function call tree. Also limits the number of entries for these functions in the Call Graph Profile report, and changes the CPU data associated with them. These results also apply to the specified function's descendants, as long as they have not been called by non-specified functions in the program. In the function call tree, the function box for the specified function appears greyed-out, and its size and shape also changes so that it appears as a square of the smallest allowable size. In addition, the CPU time shown in the function box label, appears as 0 (zero). The same applies to function boxes for descendant functions, as long as they have not been called by non-specified functions. This option also causes the CPU time spent by the specified function to be deducted from the left side CPU total in the label of the function box for each of the specified function's ancestors. In the Call Graph Profile report, an entry for the specified function only appears where it is a child of another function, or as a parent of a function that also has at least one non-specified function as its parent. When this is the case, the time in the *self* and *descendants* columns for this entry is set to 0 (zero). In addition, the amount of time that was in the *descendants* column for the specified function is subtracted from the time listed under the *descendants* column for the profiled function. As a result, be aware that the value listed in the *% time* column for most profiled functions in this report will change.	To change the display and label information for *foo* and *bar*, as well as their qualifying descendants in the function call tree, and limit their entries and data in the Call Graph Profile report, type *foo* and *bar* in the *Exclude Routines (-E)* field. You specify multiple functions by separating each one with a space.
-f (field)	De-emphasizes the general appearance of all function boxes in the function call tree, *except* for that of the specified function(s) and its descendant(s). In addition, the number of entries in the Call Graph Profile report for the non-specified functions and non-descendant functions is limited. The **-f** flag overrides the **-e** flag. In the function call tree, all function boxes *except* for that of the specified function(s) and its descendant(s) appear greyed-out. The size of these boxes and the content of their labels remain the same. For the specified function(s), and its descendants, the appearance of the function boxes and labels remain the same. In the Call Graph Profile report, an entry for a non-specified or non-descendant function only appears where it is a parent or child of a specified function or one of its descendants. All information for this entry remains the same.	To de-emphasize the display of function boxes for all functions in the function call tree *except* for *foo*, *bar*, and their descendants, and limit their types if entries in the Call Graph Profile report, type *foo* and *bar* in the *Include Routines (-f)* field. You specify multiple functions by separating each one with a space.

Table 11 (Page 3 of 3). Xprofiler GUI Command Line Options

Use this flag:	To:	For example:
-F (field)	Changes the general appearance and label information of all function boxes in the function call tree *except* for that of the specified function(s) and its descendants. In addition, the number of entries in the Call Graph Profile report for the non-specified and non-descendant functions is limited, and the CPU data associated with them is changed. The **-F** flag overrides the **-E** flag. In the function call tree, all function boxes *except* for that of the specified function(s) and its descendant(s) appear greyed-out. The size and shape of these boxes changes so that they appear as squares of the smallest allowable size. In addition, the CPU time shown in the function box label, appears as 0 (zero). In the Call Graph Profile report, an entry for a non-specified or non-descendant function only appears where it is a parent or child of a specified function or one of its descendants. When this is the case, the time in the *self* and *descendants* columns for this entry is set to 0 (zero). As a result, be aware that the value listed in the *% time* column for most profiled functions in this report will change.	To change the display and label information of the function boxes for all functions *except* the functions *foo* and *bar* and their descendants, and limit their types of entries and data in the Call Graph Profile, type *foo* and *bar* in the *Include Routines (-F)* field. You specify multiple functions by separating each one with a space.
-L (field)	Sets the alternate pathname for locating shared objects. If you plan to specify multiple paths, use the *Set File Search Path* option of the File menu on the Xprofiler GUI. See "Setting the File Search Sequence" on page 105 for information.	Type the alternate library pathname in this field.

Once you have specified the binary executable, the profile data file, and any command line options you wish to use, press the **OK** button to save the changes and close the dialog window. Xprofiler loads your application and displays its performance data.

Setting the File Search Sequence

You can specify where you want Xprofiler to look for your library files and source code files by using the *Set File Search Paths* option of the File menu. By default, Xprofiler searches the default paths first and then any alternative paths you specify.

Default Paths

For library files, Xprofiler uses the paths recorded in the specified gmon.out file(s). If you use the -L command line option, the path you specify with this option will be used instead of those in the gmon.out file. (Note: -L allows only one path to be specified and you can use this option only once).

For source code files, the paths recorded in the specified a.out file are used.

Alternative Paths

These are the paths you specify with the *Set File Search Paths* option of the File menu.

For library files, if everything else failed, the search will be extended to the path(s) specified in the LIBPATH environment variable.

To specify alternative path(s), do the following:

- Select the *File* menu, and then the *Set File Search Paths* option. The *Alt File Search Path Dialog* window appears.

- Enter the name of the path in the *Enter Alt File Search Path(s)* text field. You can specify more than one path by separating each with colon (:) or a space.
- Click on the OK button. The paths you specified in the text field become the alternative paths.

Changing the Search Sequence: You can change the order of the search sequence for library files and source code files via the *Set File Search Paths* option of the File menu. To change the search sequence, do the following:

1. Select the *File* menu, and then the *Set File Search Paths* option. The *Alt File Search Path Dialog* window appears.
2. To indicate the file search should use alternative paths first, click on the *Check alternative path(s)* first button.
3. Click on the OK button. This changes the search sequence to the:
   a. Alternative paths
   b. Default paths
   c. Path(s) specified in LIBPATH (library files only)

To return the search sequence back to its default order, repeat steps 1 through 3, but in step 2 above, click on the *Check default path(s) first* button. When the action is confirmed (by clicking on the OK button), the search sequence will start with the default paths again.

Keep in mind that if a file is found in one of the alternative paths or a path in LIBPATH, this path now becomes the default path for this file throughout the current Xprofiler session (until you exit this Xprofiler session or load a new set of data).

Understanding the Xprofiler Display

The primary difference between Xprofiler and the UNIX **gprof** command is that Xprofiler gives you a graphical picture of your application's CPU consumption in addition to textual data. This allows you to focus quickly on the areas of your application that consume a disproportionate amount of CPU.

Xprofiler displays your profiled program in a single main window. It uses several types of graphic images to represent the relevant parts of your program. Functions appear as solid green boxes (called *function boxes*), and the calls between them appear as blue arrows (called *call arcs*). The function boxes and call arcs that belong to each library within your application appear within a fenced-in area called a *cluster box*. The way that functions, calls, and library clusters are depicted is discussed later.

The Xprofiler Main Window

The Xprofiler main window contains a graphical representation of the functions and calls within your application as well as their inter-relationships. It provides six menus, including one for online help.

The Xprofiler main window looks similar to this when an application has been loaded:

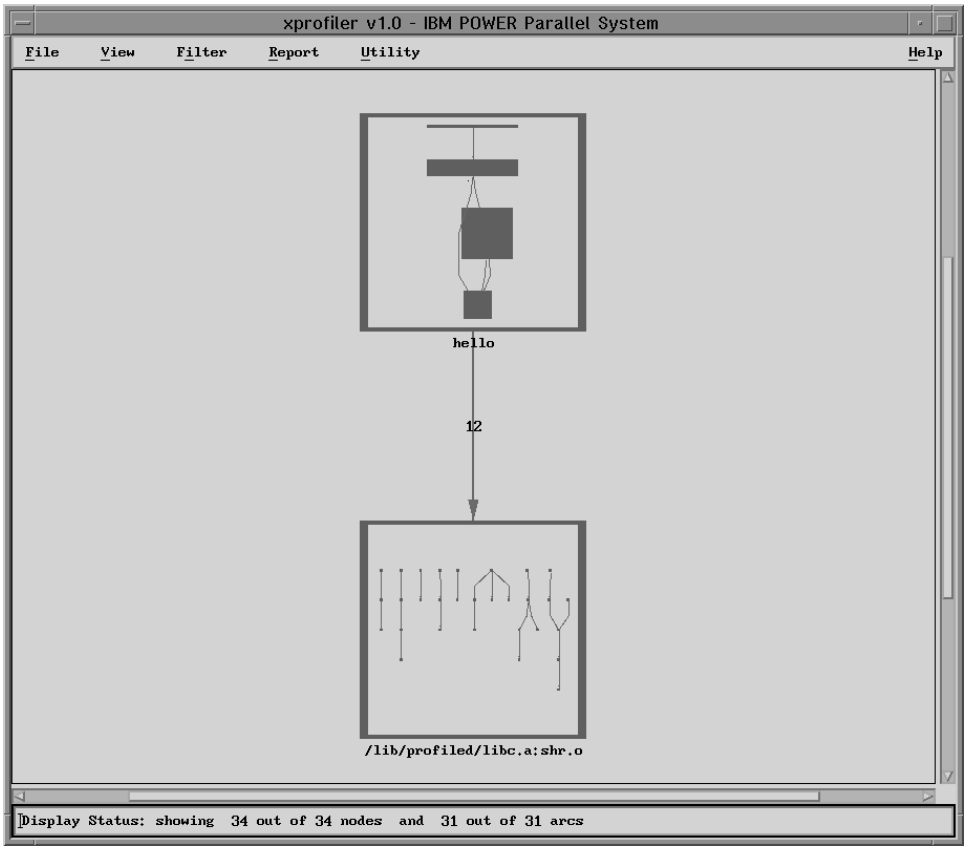


Figure 23. Sample Xprofiler Main Window

In the main window, Xprofiler displays the *function call tree*. The function call tree displays the function boxes, arcs, and cluster boxes that represent the functions within your application.

Note: When Xprofiler first opens, by default, the function boxes for your application will be *clustered* by library, as in the example above. This means that a cluster box appears around each library, and the function boxes and arcs within the cluster box are reduced in size. If you wish to see more detail, you need to uncluster the functions. To do this, select the *File* menu and then the *Uncluster Functions* option.

Xprofiler Main Menus

Along the upper portion of the main window is the *menu bar*. The left side of the menu bar contains the Xprofiler menus that let you work with your profiled data. On the right side of the menu bar, there is a Help menu for accessing online help.

The Xprofiler menus are described below:

File Menu: The File menu lets you specify the executable (a.out) files and profile data (gmon.out) files that Xprofiler will use. It also lets you control how your files are accessed and saved.

View Menu: The View menu lets you focus on specific portions of the function call tree in order to get a better view of the application's critical areas.

Filter Menu: The Filter menu lets you add, remove, and change specific parts of the function call tree. By controlling what Xprofiler displays, you can focus on the objects that are most important to you.

Report Menu: The Report menu provides several types of profiled data in a textual and tabular format. In addition to presenting the profiled data, the options of the Report menu let you:

- Display textual data
- Save it to a file
- View the corresponding source code
- Locate the corresponding function box or call arc in the function call tree.

Utility Menu: The Utility menu contains one option; *Locate Function By Name*, which lets you highlight a particular function in the function call tree.

Xprofiler Hidden Menus

Function Menu: The Function menu lets you perform a number of operations for any of the functions shown in the function call tree. You can access statistical data, look at source code, and control which functions get displayed.

The Function menu is not visible from the Xprofiler window. You access it by clicking on the function box of the function in which you are interested with your right mouse button. By doing this, you not only bring up the Function menu, but you select this function as well. Then, when you select actions from the Function menu, they are applied to this function.

Arc Menu: The Arc menu lets you locate the caller and callee functions for a particular *call arc*. A call arc is the representation of a call between two functions within the function call tree.

The Arc menu is not visible from the Xprofiler window. You access it by clicking on the call arc in which you are interested with your right mouse button. By doing this, you not only bring up the Arc menu, but you select that call arc as well. Then, when you perform actions with the Arc menu, they are applied to that call arc.

Cluster Node Menu: The Cluster Node menu lets you control the way your libraries are displayed by Xprofiler. In order to access the Cluster Node Menu, the function boxes, in the function call tree, must first be clustered by library. See "Clustering Libraries Together" on page 125 for information about clustering and unclustering the function boxes of your application. When the function call tree is clustered, all the function boxes within each library appear within a *cluster box*.

The Cluster Node menu is not visible from the Xprofiler window. You access it by clicking on the edge of the cluster box in which you are interested with your right mouse button. By doing this, you not only bring up the Cluster Node menu, but you select that cluster as well. Then, when you perform actions with the Cluster Node menu, they are applied to the functions within that library cluster.

Display Status Field

At the bottom of the Xprofiler window is a single field that tells you:

- The name of your application
- The number of gmon.out files used in this session.
- The total amount of CPU used by the application.
- The number of functions and calls in your application, and how many of these are currently displayed

How Functions are Depicted

Functions are represented by green, solid-filled boxes in the function call tree. The size and shape of each function box indicates its CPU usage. The height of each function box represents the amount of CPU time it spent on executing itself. The width of each function box represents the amount of CPU time it spent executing itself, plus its descendant functions. As a result, a function box that is wide and flat represents a function that uses a relatively small amount of CPU on itself (it spends most of its time on its descendants). On the other hand, the function box for a function that spends most of its time executing only itself will be roughly square-shaped.

Under each function box in the function call tree is a label that contains the name of the function and related CPU usage data. For information on the function box labels, see “Getting Basic Data” on page 131.

The example below shows the function boxes for two functions, *sub1* and *printf*, as they would appear in the Xprofiler display.

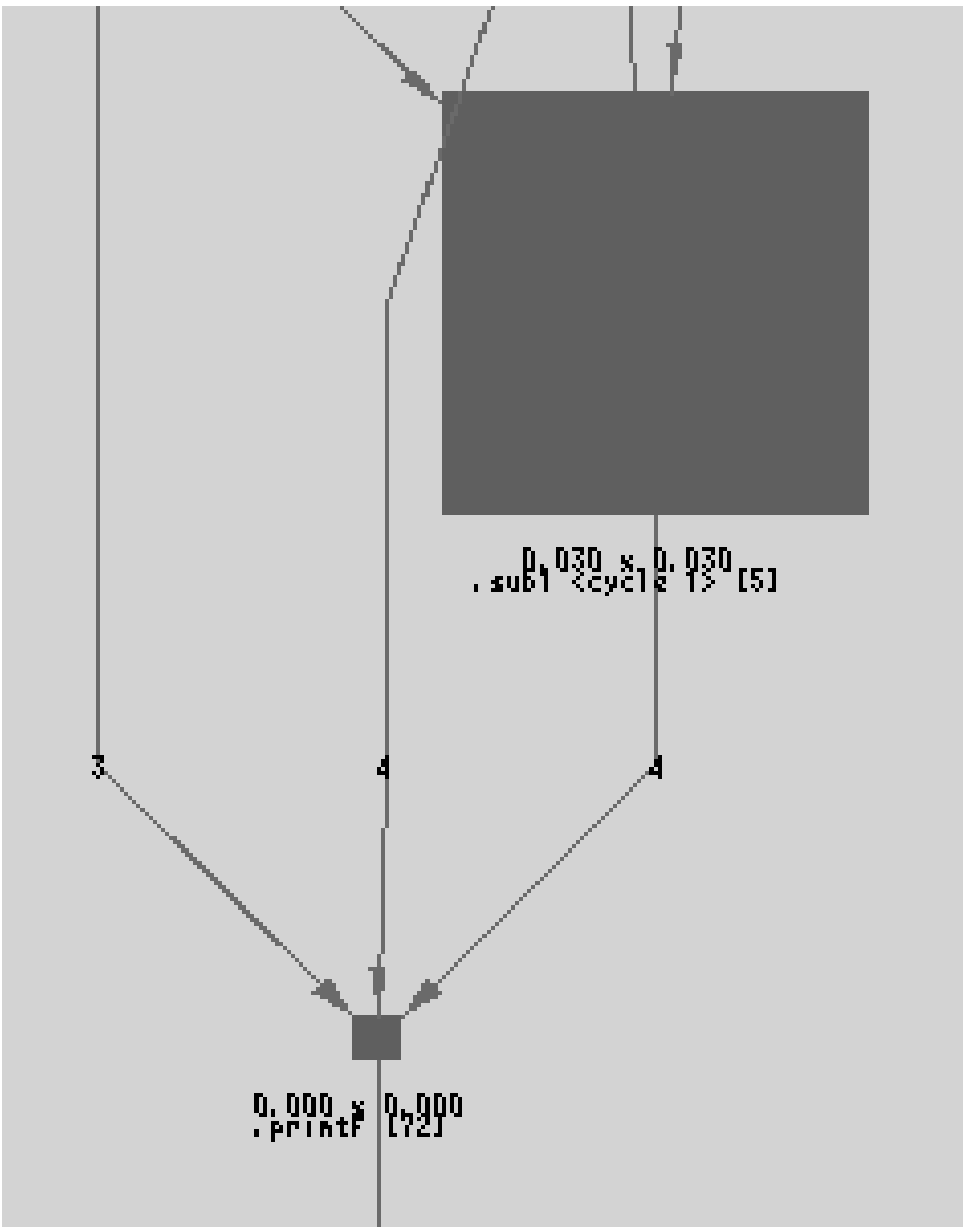


Figure 24. Example of Function Boxes and Arcs in Xprofiler Display

Each function box has its own menu. To access it, place your mouse cursor over the function box of the function in which you are interested, and press the right mouse button. Each function also has an information box that lets you get basic performance numbers quickly. To access the information box, place your mouse cursor over the function box of the function in which you are interested, and press the left mouse button.

How Calls Between Functions are Depicted

The calls made between each of the functions in the function call tree are represented by blue arrows extending between their corresponding function boxes. These lines are called *call arcs*. Each call arc appears as a solid blue line between two functions. The arrowhead indicates the direction of the call; the function represented by the function box it points to is the one that receives the call. The

function making the call is known as the *caller*, while the function receiving the call is known as the *callee*.

Each call arc includes a numerical label that tells you how many calls were exchanged between the two corresponding functions.

Figure 24 on page 110, above, shows several call arcs. For the call arc that connects *sub1* and *printf*, *sub1* is the caller and *printf* is the callee. The label tells you that *sub1* called *printf* four times.

Note that each call arc has its own menu that lets you locate the function boxes for its caller and callee functions. To access it, place your mouse cursor over the call arc for the call in which you are interested, and press the right mouse button. Each call arc also has an information box that shows you the number of times the caller function called the callee function. To access the information box, place your mouse cursor over the call arc for the call in which you are interested, and press the left mouse button.

How Library Clusters are Depicted

Xprofiler lets you collect the function boxes and call arcs that belong to each of your shared libraries into *cluster boxes*. Figure 23 on page 107 shows an example of an Xprofiler display in which the libraries are clustered.

Since there will be a box around each library, the individual function boxes and call arcs will be difficult to see. If you want to see more detail, you will need to uncluster the function boxes. To do this, select the Filter menu and then the *Uncluster Functions* option.

When viewing function boxes within a cluster box, note that the size of each function box is relative to those of the other functions within the same library cluster. On the other hand, when all the libraries are unclustered, the size of each function box is relative to all the functions in the application (as shown in the function call tree).

Each library cluster has its own menu that lets you manipulate the cluster box. To access it, place your mouse cursor over the edge of the cluster box you are interested in, and press the right mouse button. Each cluster also has an information box that shows you the name of the library and the total CPU usage (in seconds) consumed by the functions within it. To access the information box, place your mouse cursor over the edge of the cluster box you are interested in and press the left mouse button.

Using the Xprofiler Graphical User Interface

The Xprofiler graphical user interface (GUI) contains features and buttons that are common throughout the interface. This section explains how to use some of these common elements.

Using the Dialog Window Buttons

The buttons that appear on the Xprofiler dialog windows are explained below:

OK

Saves the changes, executes the action, and closes the dialog window.

Apply

Saves the changes, executes the action, but leaves the dialog window open.

Reset

Restores the fields of the dialog window to their original values (at the time you opened it), and keeps the dialog window open.

Cancel

Ignores changes and closes the dialog window.

Help

Brings up the Xprofiler online help.

Filter

Executes filtering criteria provided by you in the dialog window.

Using the Search Engine

Some of the Xprofiler windows that are accessible via the Report and Function menus provide a *Search Engine* field that lets you search for a specific string. In the Search Engine field, which is located at the bottom of these windows, you type the string in which you are interested. The first row that contains the string you specified is highlighted.

To use the Search Engine to search for a string:

1. Click on the *Search Engine* field with the left mouse button. The Search Engine field highlights to show that it is selected.
2. Type the string you are looking for in the Search Engine field.

   Extended regular expressions are allowed. For more information, see the explanation of the **regcmp** and **regcomp** commands in *AIX Technical Reference, Volume 2: Base Operating System and Extensions* (SC23-2615).
3. Press the <**Enter**> key. The first row, in the Report or Source Code window, that contains the string you specified is highlighted. Each time you press the <**Enter**> key, a subsequent occurrence of the string in highlighted. The Search wraps back to the first occurrence after all other occurrences have been highlighted.

Using the Save Dialog Windows

A *Save* dialog window appears when you choose the *Save As* option from any of the Xprofiler reports windows or from the File Menu. It allows you to save the data you see, in the report window that is currently open, to a file.

Note: If you choose the *Save As* option from one of the reports windows, the title of the dialog window included the name of the report (for example *Save Flat Profile*).

To save the current report data to a file using the Save dialog window:

1. Specify the file into which the data should be placed. You can specify either an existing file or a new one. If you specify and existing file, be aware that Xprofiler replaces the file altogether (instead of appending to the existing data). To replace an existing file, use the scroll bars of the *Directories* and the *Files* selection boxes to locate the file you want. To make locating your file easier, you can also use the *Filter* button (see "Using the Dialog Window Filters" for more information). To specify a new file, type its name in the *Selection* field.
2. Click on the *OK* button. A file containing the profiled window data appears in the directory you specified, under the name you gave it.

Using the Dialog Window Filters

Many of the Xprofiler dialog windows include a *Filter* button. The use of the Xprofiler Filter function follows the Motif standard. To use the Filter:

1. In the *Filter* field, specify the directory that contains the files that you wish to see in the *Files* selection box. You may specify an asterisk (*) as a wildcard.
2. Click on the *Filter* button with the left mouse button. The list of files in the Files selection box is updated to reflect your selection.

Using the Radio/Toggle Buttons and Sliders

Many of the dialog windows include buttons and sliders that allow you to select options and specify values.

Using the Buttons

Aside from push-buttons, the Xprofiler dialog windows also use radio buttons and toggle buttons. Radio buttons let you select one item from a set of items, while toggle buttons let you activate or de-activate a single item.

In the example below, the *Screen Dump Options Dialog* window uses both radio buttons and toggle buttons. For instance, under *Output To*, there are two radio buttons; **File** and **Printer**. You must select one or the other, but you cannot select both. Just above the *Default Directory* field, notice two toggle buttons; **Enable Landscape** and **Annotate Output**. By using the toggle buttons, you can activate either one or both of these options.

To select (or activate) an option with a radio or toggle button, set the button to the pressed-in position by by clicking on it. When a button is pressed-in, it appears shaded. Under *Output To*, in the example below, **File** is selected and **Printer** is not.

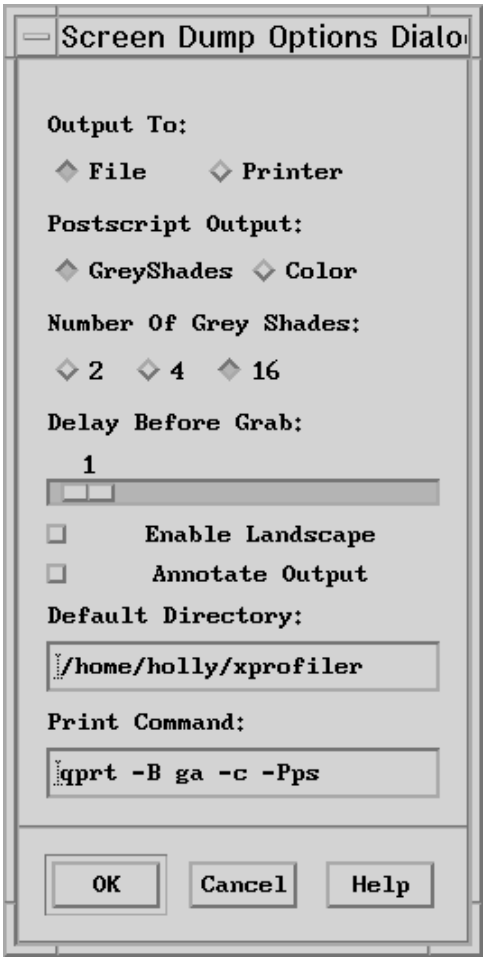


Figure 25. Example Showing Radio Buttons, Toggle Buttons, and Slider

Using the Sliders

Several of the Xprofiler dialog windows include *sliders* that let you specify a numerical value. In the example above, the *Delay Before Grab* slider lets you specify the number of seconds you want to pass before the screen image is actually captured.

Place your mouse cursor over the slider. Press and hold the left mouse button while moving the slider horizontally in either direction. The number above the slider changes as you move it, and indicates the number selected. Once the slider is at the setting you want, release the mouse button.

If the number of selectable values on the slider is high, you may want to have finer control over the placement of the slider. If so, click on the slider and then use the arrow keys on your keyboard to place it.

Manipulating the Function Call Tree

Xprofiler lets you look at your profiled data a number of ways, depending on what you want to see. It provides:

- Navigation that lets you move around the display and zoom in on specific areas
- Display options, based on your personal viewing preferences.
- Filtering capability, to let you include and exclude certain objects from the display

Zooming In on the Function Call Tree

Xprofiler lets you magnify specific areas of the window to get a better view of your profiled data. The View menu includes three options that let you do this:

- Overview
- Zoom In
- Zoom Out

To resize a specific area of the display, you can use either the *Overview* or *Zoom In* options of the View menu. To magnify an area with the *Overview* option:

1. Select the *View* menu, and then the *Overview* option. The Overview Window appears, as in the example below.

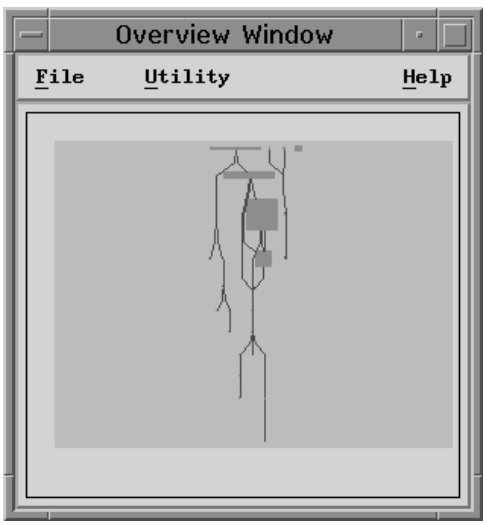


Figure 26. The Overview Window

The Overview Window contains a miniature view of the function call tree, just as it appears in the Xprofiler main display. When you open the Overview Window, the light blue highlight area represents the current view of the main window.

You control the size and placement of the highlight area with your mouse. Depending on where you place your mouse over the highlight area, your cursor changes to indicate the operation you can perform. Here is an explanation of the cursor images, and what they indicate to you:

When your cursor appears as two crossed arrows, it means that by pressing and holding your mouse button, you can control where the box is placed.

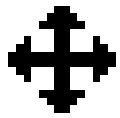


Figure 27. Cursor when movement of highlight box is under mouse control

When your cursor appears as a line with an arrow perpendicular to it, it means that your mouse button has grabbed the edge of the highlight area, and you now have the ability to resize it. By pressing and holding your mouse button, and dragging it in or out, you can increase or decrease the size of the box. Notice that as you move the edge in or out, the size of the entire highlight area changes.

Figure 28. Cursor when edge of highlight box is under mouse control

When your cursor appears as a right angle with an arrow pointing into it, it means that your mouse button has grabbed the corner of the highlight area and you now have the ability to resize it. By pressing and holding your mouse button, and dragging it diagonally up or down, you can increase or decrease the size of the box. Notice that as you move the corner up or down, the size of the entire highlight area changes.

Figure 29. Cursor when corner of highlight box is under mouse control

3. Place your mouse cursor within the light blue highlight area. Notice that the cursor changes to four crossed arrows. This indicates that your cursor has control over the placement of the box.

4. Move your cursor over one of the four corners of the highlight area. Notice that the cursor changes to a right angle with an arrow pointing into it. This indicates that you now have control over the corner of the highlight area.

5. Press and hold the left mouse button, and drag the corner of the box diagonally inward. The box shrinks as you move it. The example below shows the highlight area reduced in size, with only a few function boxes visible within it.

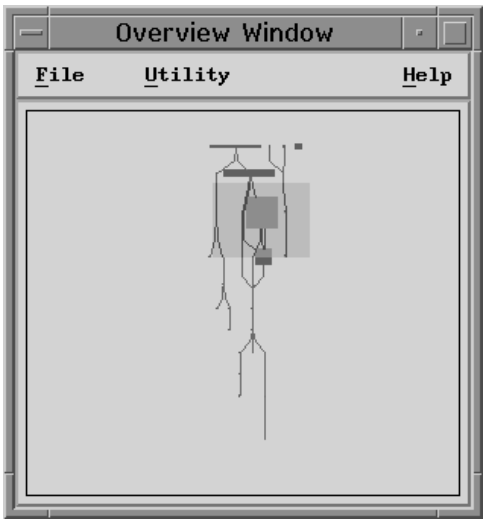


Figure 30. Highlight Area Reduced in Size

6. When the highlight area is as small as you would like it (or the smallest allowable size), release the mouse button. The Xprofiler main display redraws itself to contain only the functions within the highlight area, and in the same proportions. This has the effect of magnifying the items within the highlight area.

7. Place your mouse cursor over the highlight area. Your cursor again changes to four crossed arrows to indicate that you have control over the placement of the highlight area. Press and hold the left mouse button and drag the highlight area to the area of the Xprofiler display you want to magnify.

8. Release the mouse button. The Xprofiler main display now contains the items in which you are interested.

The example below shows the Xprofiler main display with the area, indicated by the highlight area in Figure 30, magnified. Note that the performance data, on the label of each function box, is now visible.

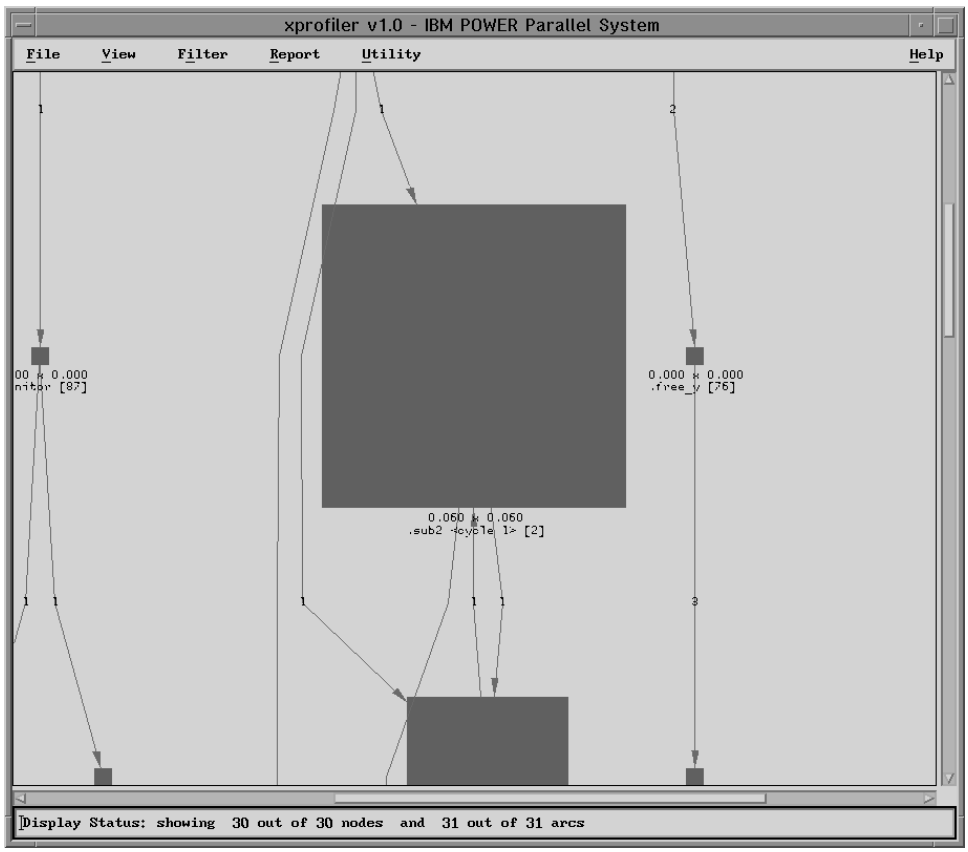


Figure 31. Magnified View of Xprofiler Display

To use the *Zoom In* option of the View menu to magnify a specific area of the function call tree:

1. Select the *View* menu, and then the *Zoom In* option. Once you select the *Zoom In* option, your cursor changes to a hand to indicate that your selection is active.
2. Place the mouse cursor in the upper left hand corner of the area you would like to view more closely. Press and hold the left mouse button while dragging it diagonally downward, until the rubber band box surrounds the area you want to view.
3. Release the mouse button. Xprofiler redraws the display so that the area of the function call tree you selected is centered and sized proportionately, according to the size of the rubber band box you drew.

To get an even closer view of the area you selected, choose the *Zoom In* option again and follow the steps above.

There may be times when you are looking at the function call tree too closely. The *Zoom Out* option lets you widen the view of the function call tree, as if you were taking a few steps back from a painting on a wall.

The *Zoom Out* option is most useful after using the *Zoom In* or *Overview* options to magnify an area of the function call tree. By default, the Xprofiler main window is completely zoomed out (it shows you the entire function call tree). The *Zoom Out* option helps you return the main window to this state.

To zoom out:

1. Select the *View* menu, and then the *Zoom Out* option. Once you select the *Zoom Out* option, your cursor changes to a hand to indicate that your selection is active.
2. Place the mouse cursor in the upper left hand corner of the area you want to view. Press and hold the left mouse button while dragging it diagonally downward, until the rubberband box surrounds the area you want to widen.
3. Release the mouse button. Xprofiler redraws the display so that the area of the function call tree you selected is centered and sized proportionately according to the size of the rubber band box that you drew.

To further step back from the area you selected, choose the *Zoom Out* option again, and follow the steps above.

Controlling How the Display is Updated: The Utility menu of the Overview window lets you choose the mode in which the display is updated. The default is the *Immediate Update* option, which causes the display to show you the items in the highlight area as you are moving it around. The *Delayed Update* option, on the other hand, causes the display to be updated only when you have moved the highlight area over the area in which you are interested, and released the mouse button. The *Immediate Update* option only applies to what you see when you move the highlight area; it has no effect on the resizing of items in highlight area, which is always delayed.

Other Viewing Options

Xprofiler lets you change the way it displays the function call tree, based on your own personal preferences.

Controlling the Graphic Style of the Function Call Tree: You can choose between 2-D and 3-D function boxes in the function call tree. The default style is 2-D, but you can change this to 3-D. To do this:

1. Select the View menu, and then the *3-D Image* option. The function boxes in the function call tree now appear in 3-D format.

Controlling the Orientation of the Function Call Tree: You can choose to have Xprofiler display the function call tree in either *Top-to-Bottom* or *Left-to-Right* format. The default is Top-to-Bottom. If you would rather see the function call tree displayed in Left-to-Right format:

1. Select the View menu, and then the *Layout: Left→Right* option. The function call tree appears in Left-to-Right format.

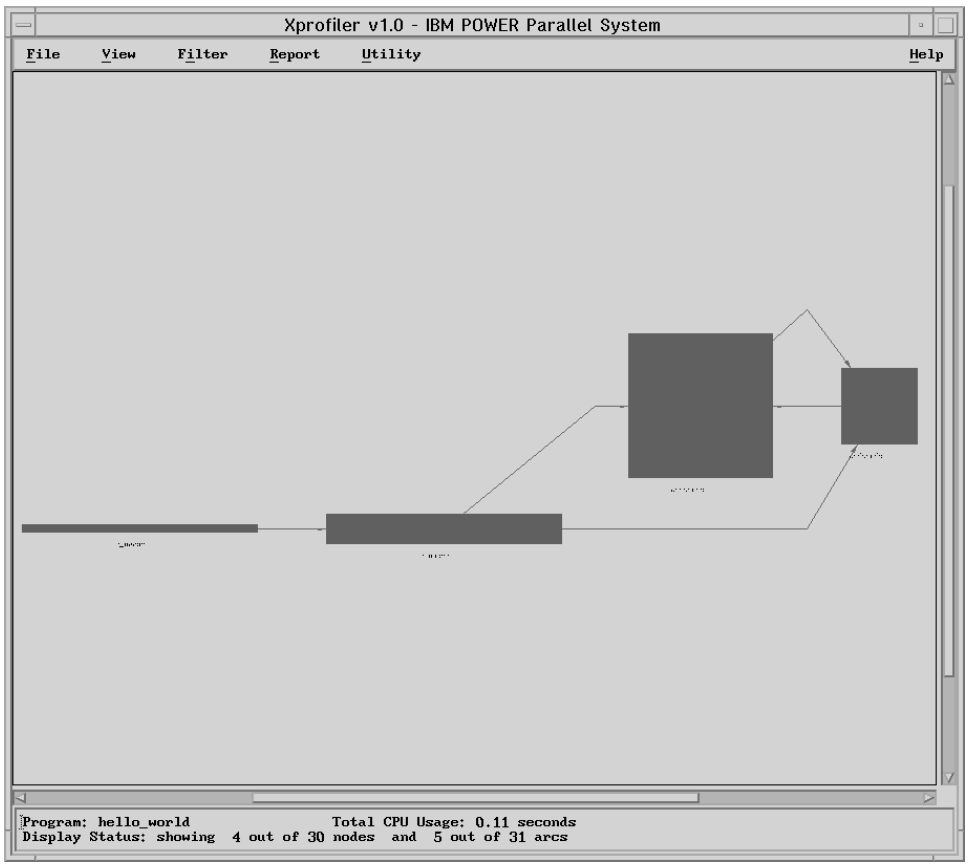


Figure 32. Left-to-Right Format

Filtering What You See

When Xprofiler first opens, the entire function call tree appears in the main window. This includes the function boxes and call arcs that belong to your executable as well as the shared libraries that it utilizes. At times, you may want to simplify what you see in the main window, and there are a number of ways to do this.

Note: Filtering options of the Filter menu let you change the appearance of the function call tree only. The performance data contained in the reports (via the Reports menu) is not affected.

Displaying the Entire Function Call Tree

When you first open Xprofiler, by default, all the function boxes and call arcs of your executable and its shared libraries appear in the main window. After that, you may choose to filter out specific items from the window. However, there may be times when you want to see the entire function call tree again, without having to reload your application. To do this:

1. Select the *Filter* menu, and then the *Show Entire Call Tree* option. Xprofiler erases whatever is currently displayed in the main window and replaces it with the entire function call tree.

Excluding and Including Specific Objects

There are a number of ways that Xprofiler lets you control the items that get displayed in the main window. For the most part, you will want to include or exclude certain objects so that you can more easily focus on the things that are of most interest to you.

Filtering Shared Library Functions: In most cases, your application will call functions that are within shared libraries. By default, these shared libraries will appear in the Xprofiler window along with your executable. As a result, the window may get crowded and obscure the items that you really want to see. If this is the case, you may want to filter the shared libraries from the display. To do this:

1. Select the *Filter* menu, and then the *Remove All Library Calls* option.

The shared library function boxes disappear from the function call tree, leaving only the function boxes of your executable file visible.

If you removed the library calls from the display, you may want to add all or some of them back. To do this:

1. Select the *File* menu, and then the *Add Library Calls* option

The function boxes once again appear with the function call tree. Note, however, that all of the shared library calls that were in the initial function call tree may not be added back. This is because the *Add Library Calls* option only adds back in the function boxes for the library functions that were called by functions that are currently displayed in the Xprofiler window.

There may be times when you want to add only specific function boxes back into the display. To do this:

1. Select the *Filter* menu, and then the *Filter by Function Names* option. The *Filter By Function Names Dialog* window appears.
2. From the Filter By Function Names Dialog window, select the *add these functions to graph* button, and then type the name of the function you want to add in the *Enter function name* field. If you enter more than one function name, you must separate them by putting a blank space between each function name string.

   If there are multiple functions in your program that include the string you enter in their names, the filter applies to each one. For example, say you specified *sub*, and *print*, and your program also included functions named *sub1*, *psub1* and *printf*. The *sub*, *sub1*, *psub1*, *print*, and *printf* functions would all be added to the graph.
3. Click on the *OK* button. The function box(es) appears in the Xprofiler display with the function call tree.

Filtering by Function Characteristics: The Filter menu of Xprofiler offers you three options that allow you to add or subtract function boxes from the main window, based on specific characteristics. The options are:

- Filter by Function Names
- Filter by CPU Time
- Filter by Call Counts

Each one of these options uses a different dialog window to let you specify the criteria by which you want to include or exclude function boxes from the window.

To filter by function names:

1. Select the *Filter* menu.

2. Select the *Filter by Function Names* option. The *Filter By Function Names Dialog* window appears.

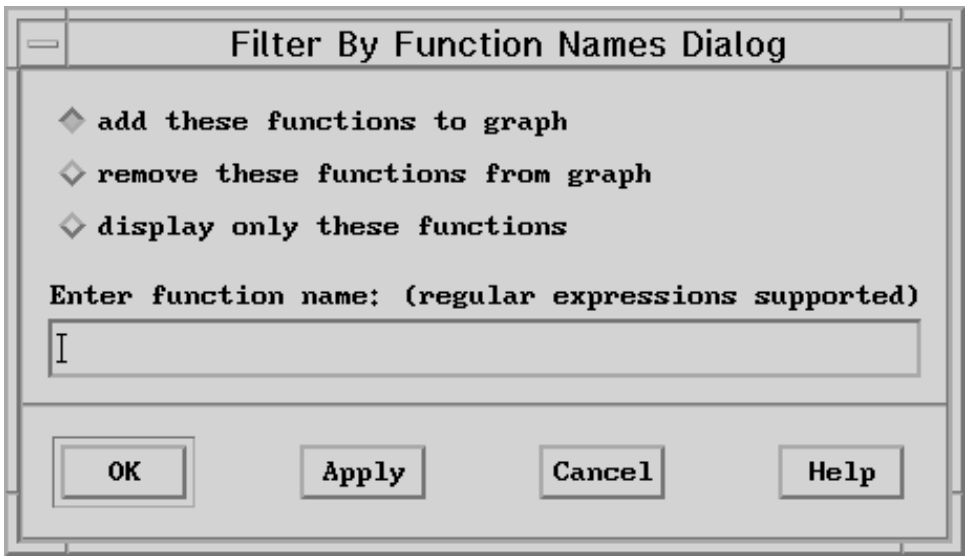


Figure 33. Filter By Function Names Dialog window

3. The Filter By Function Names Dialog window includes three options:

   - *add these functions to graph*
   - *remove these functions from the graph*
   - *display only these functions*

   From the Filter By Function Names Dialog window, select the option you want, and then type the name of the function(s) to which you want it applied in the *Enter function name* field. For example, say you wanted to remove function box for a function called *printf*, from the main window. You would click on the *remove this function from the graph* button and type *printf* in the *Enter function name* field.

   You can enter more than one function name in this field. If there are multiple functions in your program that include the string you enter in their names, the filter will apply to each one. For example, say you specified *sub* and *print*, and your program also included functions named *sub1*, *psub1*, and *printf*. The option you chose would be applied to the *sub*, *sub1*, *psub1*, *print*, and *printf* functions.

   5. Click on the *OK* button. The contents of the function call tree now reflect the filtering options you specified.

To filter by CPU time:

1. Select the *Filter* menu.

2. Select the *Filter by CPU Time* option. The *Filter By CPU Time Dialog* window appears.

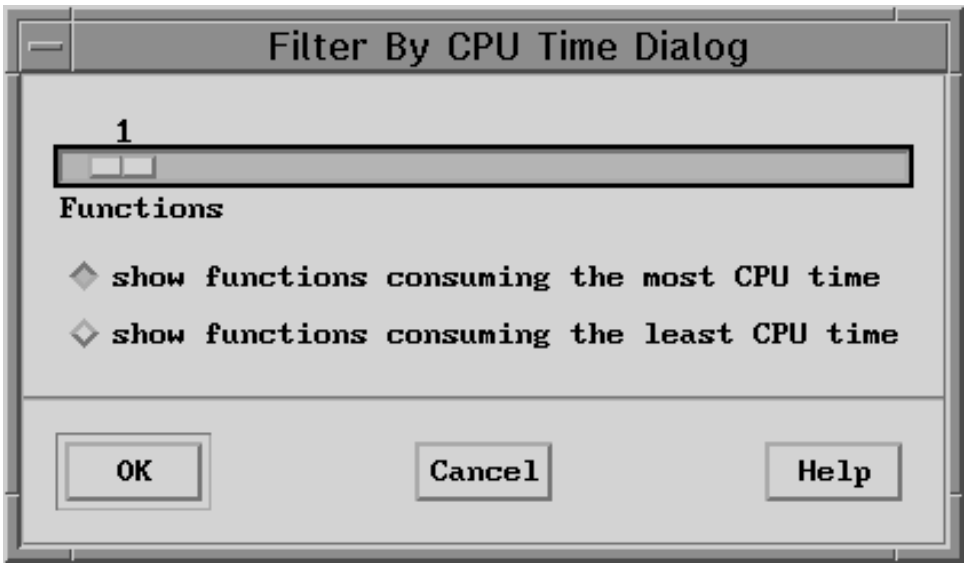


Figure 34. Filter By CPU Time Dialog window

3. The Filter By CPU Time Dialog window includes two options:

- *show functions consuming the most CPU time*
- *show functions consuming the least CPU time*

4. Click on the button for the option you want (*show functions consuming the most CPU time* is the default).

5. Use the slider to specify the number of function boxes you want displayed (1 is the default).

For example, if you wanted to display the the function boxes for the 10 functions in your application that consumed the most CPU, you would select the *show functions consuming the most CPU* button, and specify 10 with the slider.

6. Click on the *OK* button. The contents of the function call tree now reflect the filtering options you specified.

To filter by call counts:

1. Select the *Filter* menu.

2. Select the *Filter by Call Counts* option. The *Filter By Call Counts Dialog* window appears.

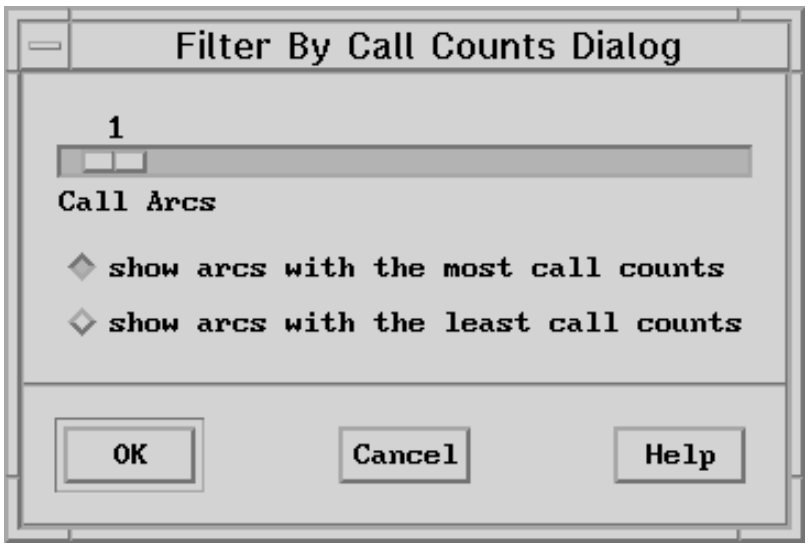


Figure 35. Filter By Call Counts Dialog window

3. The Filter By Call Counts Dialog window includes two options:
   - *show arcs with the most call counts*
   - *show arcs with the least call counts*

4. Click on the button for the option you want (*show arcs with the most call counts* is the default).

5. Use the slider to specify the number of arcs you want displayed (1 is the default).

   For example, if you wanted to display the 10 call arcs in your application that represented the least number of calls, you would select the *show arcs with the least call counts* button, and specify 10 with the slider.

6. Click on the *OK* button. The contents of the function call tree now reflect the filtering options you specified.

Including and Excluding Parent and Child Functions: When tuning the performance of your application, you will want to know which functions consumed the most CPU time, and then you will need to ask several questions in order to understand their behavior:

- Where did each function spend most of the CPU time?
- What other functions called this function? Were the calls made directly or indirectly?
- What other functions did this function call? Were the calls made directly or indirectly?

Once you understand how these functions behave, and are able to improve their performance, you can move on to analyzing the functions that consume less CPU.

When your application is large, the function call tree will also be large. As a result, the functions that are the most CPU-intensive may be difficult to see in the function call tree. To get around this, use the *Filter by CPU* option of the Filter menu, which lets you display only the function boxes for the functions that consume the most CPU time. Once you've done this, the Function menu for each function lets you

add the parent and descendant function boxes to the function call tree. By doing this, you create a smaller, simpler function call tree that displays the function boxes associated with most CPU-intensive area of the application.

A *child* function is one that is directly called by the function of interest. To see only the function boxes for the function of interest and its child functions:

1. Place your mouse cursor over the function box in which you are interested, and press the right mouse button. The Function menu appears.
2. From the Function menu, select the *Immediate Children* option, and then the *Show Child Functions Only* option.

   Xprofiler erases the current display and replaces it with only the function boxes for the function you chose, plus its child functions.

A *parent* function is one that directly calls the function of interest. To see only the function box for the function of interest and its parent functions:

1. Place your mouse cursor over the function box in which you are interested, and press the right mouse button. The Function menu appears.
2. From the Function menu, select the *Immediate Parents* option, and then the *Show Parent Functions Only* option.

   Xprofiler erases the current display and replaces it with only the function boxes for the function you chose, plus its parent functions.

There may be times when you may want to see the function boxes for both the parent and child functions of the function in which you are interested, without erasing the rest of the function call tree. This is especially true if you chose to display the function boxes for two or more of the most CPU-intensive functions with the *Filter by CPU* option of the Filter menu (you suspect that more than one function is consuming too much CPU). To do this:

1. Place your mouse cursor over the function box in which you are interested, and press the right mouse button. The Function menu appears.
2. From the Function menu, select the *Immediate Parents* option, and then the *Add Parent Functions to Tree* option.

   Xprofiler leaves the current display as it is, but adds the parent function boxes.
3. Place your mouse cursor over the same function box and press the right mouse button. The Function menu appears.
4. From the Function menu, select the *Immediate Children* option, and then the *Add Child Functions to Tree* option.

   Xprofiler leaves the current display as it is, but now adds the child function boxes in addition to the parents.

Clustering Libraries Together

When you first bring up the Xprofiler window, by default, the function boxes of your executable, and the libraries associated with it, are clustered. Since Xprofiler shrinks the call tree of each library when it places it in a cluster, you will need to uncluster the function boxes if you want to look closely at a specific function box label.

It is important to understand that you can see significantly more detail per function, when your display is in the unclustered or *expanded* state, than when it is in the clustered or *collapsed* state. So, depending on what you want to do, you will need to cluster or uncluster (collapse or expand) the display.

There may be times when the Xprofiler window is visually crowded. This is especially true if your application calls functions that are within shared libraries; function boxes representing your executable functions as well as the functions of the shared libraries get displayed. As a result, you may want to organize what you see in the Xprofiler window so you can focus on the areas that are most important to you. One way you can do this is to collect all the function boxes of each library into a single area, known as a library *cluster*.

When you choose to cluster your libraries, Xprofiler gathers all the functions for each one into a single area, and draws a green box around them. This is known as a *cluster box*. The name of the library also appears below the box.

The function boxes in the application shown in Figure 23 on page 107 have been clustered. Note that one cluster box holds the function boxes associated with the executable *hello_world*, while the other cluster box holds the function boxes of the shared library */lib/profiled/libc.a:shr.o*.

The example below shows the same application, *hello_world*, with its function boxes unclustered. Now that the function boxes of your executable and its shared libraries are displayed together, which means their relationships are more apparent.

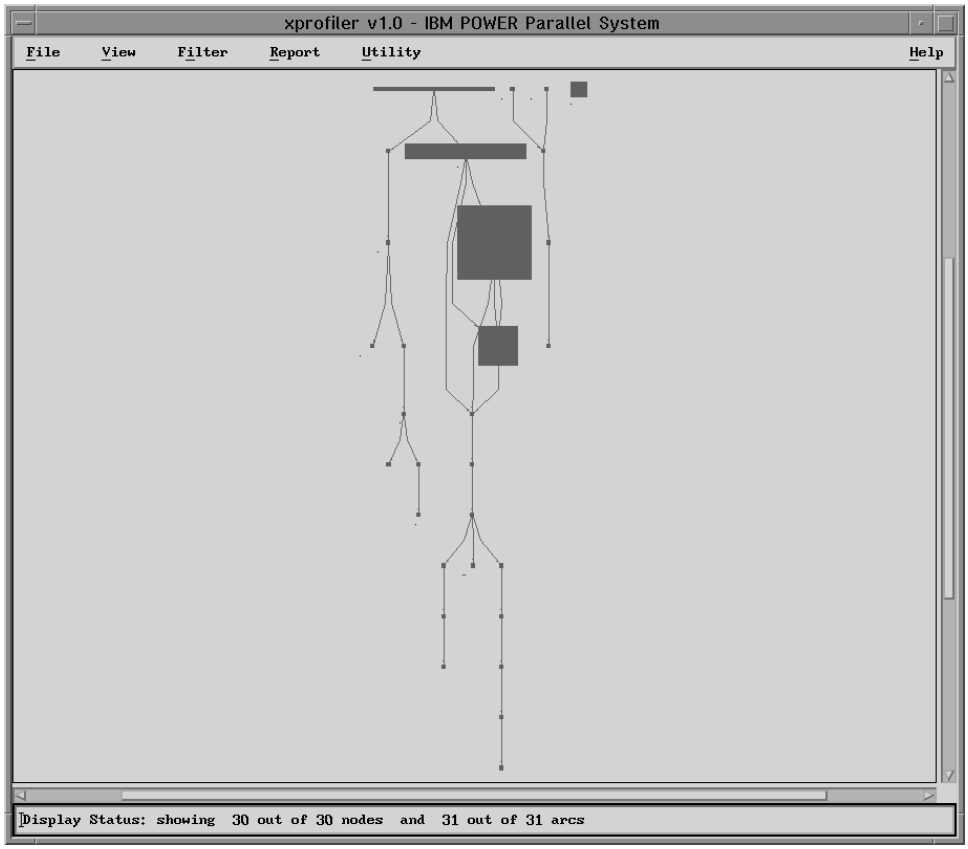


Figure 36. Xprofiler Window with Function Boxes Unclustered

Clustering Functions

If the functions within your application are unclustered, you can use an option of the Filter menu to cluster them.

1. Select the Filter menu, and then the *Cluster Functions by Library* option. The libraries within your application appear within their respective cluster boxes.

Once you cluster the functions in your application, as shown in Figure 23 on page 107, you can further reduce the size (also referred to as *collapse*) of each cluster box. To do this:

1. Place your mouse cursor over the edge of the cluster box and press the right mouse button. The Cluster Node Menu appears.
2. Select the *Collapse Cluster Node* option. The cluster box, and its contents, now appear as a small solid green box. In the example below, the library */lib/profiled/libc.a:shr.o* is collapsed.

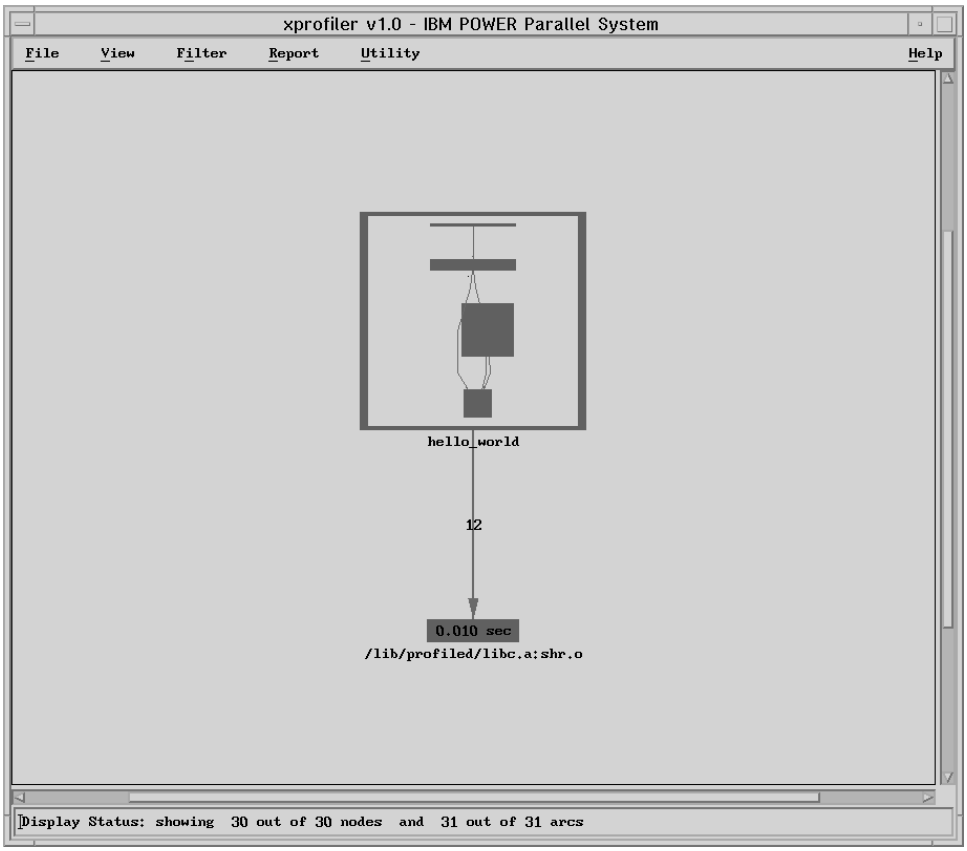


Figure 37. Xprofiler Window with One Library Cluster Box Collapsed

To return the cluster box to its original condition (*expand* it):

1. Place your mouse cursor over the collapsed cluster box and press the right mouse button. The Cluster Node Menu appears.
2. Select the *Expand Cluster Node* option. The cluster box, and its contents, appear once again.

Unclustering Functions

If the functions within your application are clustered, you can use an option of the Filter menu to uncluster them.

1. Select the Filter menu, and then the *Uncluster Functions* option. The cluster boxes disappear and the functions boxes of each library expand to fill the Xprofiler window.

If your functions have been clustered, you may want to remove one or more (but not all) cluster boxes. For example, say you wanted to uncluster only the functions of your executable, but keep its shared libraries within their cluster boxes. You would:

1. Place your mouse cursor over the edge of the cluster box that contains the executable and press the right mouse button. The Cluster Node Menu appears.
2. Select the *Remove Cluster Box* option. The cluster box disappears and the function boxes and call arcs, that represent the executable functions,now appear in full detail. The function boxes and call arcs of the shared libraries remain within their cluster boxes, which now appear smaller to make room for the unclustered executable function boxes.

The example below shows the executable, *hello_world* with its cluster box removed. Its shared library, */lib/profiled/libc.a:shr.o*, remains within its cluster box.

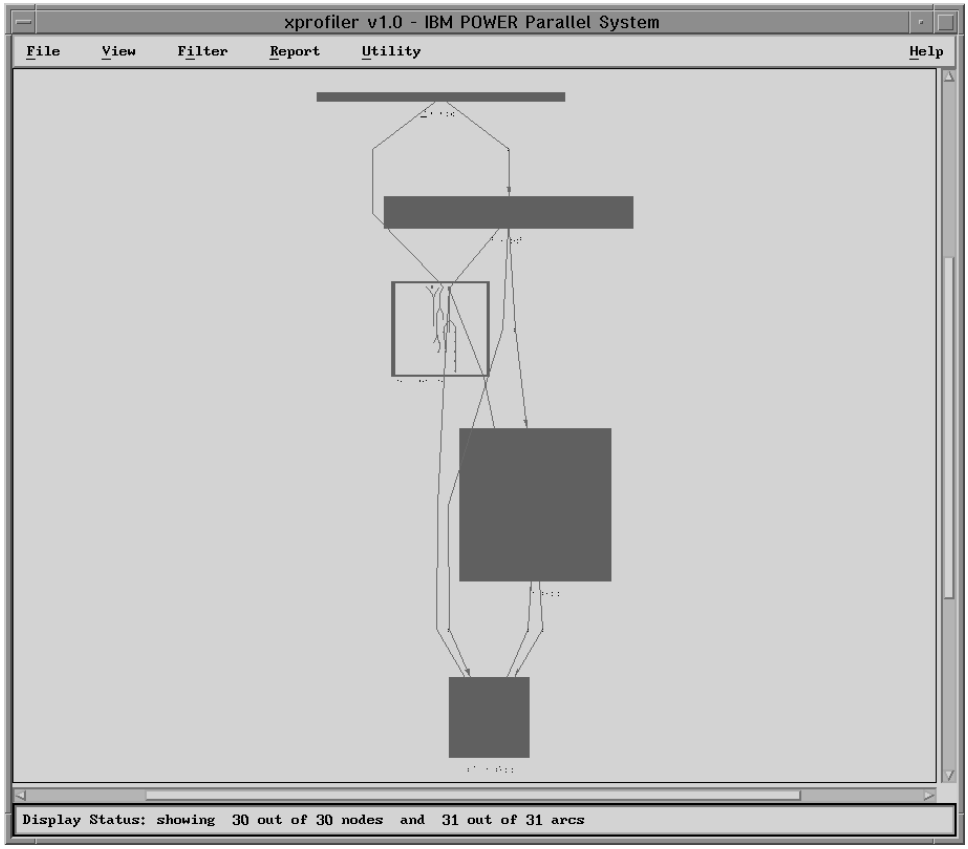


Figure 38. Xprofiler Window with One Library Cluster Box Removed

Locating Specific Objects in the Function Call Tree

If you are interested in one or more specific functions in a complex program, you may need help locating their corresponding function boxes in the function call tree.

If you would like to locate a single function, and you know its name, you can use the *Locate Function By Name* option of the Utility menu. To locate a function by name:

1. Select the Utility menu, and then the *Locate Function By Name* option. The *Search By Function Name Dialog* window appears.
2. Type the name of the function you wish to locate in the *Enter Function Name* field. The function name you type here must be a continuous string (it cannot include blanks).
3. Press the *OK* or *Apply* button. The corresponding function box is highlighted (its color changes to red) in the function call tree and Xprofiler zooms in on its location.

   To display the function call tree in full detail again, go to the View menu and use the *Overview* option.

There may also be times when you want to see only the function boxes for the functions you are concerned with, plus other specific functions that are related to it. For instance, suppose you want to see all the functions that directly called the function in which you are interested. It might not be easy to pick out these function boxes when you view the entire call tree, so you would want to display them, plus the function of interest, alone.

Each function has its own menu, called a *Function menu*. Via the Function menu, you can choose to see the following for the function in which you are interested:

- Parent functions (functions that directly call the function of interest)
- Child functions (functions that are directly called by the function of interest)
- Ancestor functions (functions that can call, directly or indirectly, the function of interest)
- Descendant functions (functions that can be called, directly or indirectly, by the function of interest)
- Functions that belong to the same cycle

When you use these options, Xprofiler erases the current display and replaces it with only the function boxes for the function of interest and all the functions of the type you specified.

Locating and Displaying Parent Functions

A *parent* is any function that directly calls the function in which you are interested. To locate the parent function boxes of the function in which you are interested:

1. Click on the function box of interest with the right mouse button. The Function menu appears.
2. From the Function menu, select *Immediate Parents→Show Parent Functions Only*. Xprofiler redraws the display to show you only the function boxes for the function of interest and its parent functions.

Locating and Displaying Child Functions

A *child* is any function that is directly called by the function in which you are interested. To locate the child functions boxes for the function in which you are interested:

1. Click on the function box of interest with the right mouse button. The Function menu appears.
2. From the Function menu, select *Immediate Children→Show Child Functions Only*. Xprofiler redraws the display to show you only the function boxes for the function of interest and its child functions.

Locating and Displaying Ancestor Functions

An *ancestor* is any function that can call, directly or indirectly, the function in which you are interested. To locate the ancestor functions:

1. Click on the function box of interest with the right mouse button. The Function menu appears.
2. From the Function menu, select *All Paths To→Show Ancestor Functions Only*. Xprofiler redraws the display to show you only the function boxes for the function of interest and its ancestor functions.

Locating and Displaying Descendant Functions

A *descendant* is any function that can be called, directly or indirectly, by the function in which you are interested. To locate the descendant functions (all the functions that the function of interest can reach, directly or indirectly):

1. Click on the function box of interest with the right mouse button. The Function menu appears.
2. From the Function menu, select *All Paths From→Show Descendant Functions Only*. Xprofiler redraws the display to show you only the function boxes for the function of interest and its descendant functions.

Locating and Displaying Functions on a Cycle

To locate the functions that are on the same cycle as the function you are interested in:

1. Click on the function of interest with the right mouse button. The Function menu appears.
2. From the Function menu, select *All Functions on the Cycle→Show Cycle Functions Only*. Xprofiler redraws the display to show you only the function of interest and all the other functions on its cycle.

Getting Performance Data for Your Application

With Xprofiler, you can get performance data for your application on a number of levels, and in a number of ways. You can easily view data pertaining to a single function, or you can use the reports provided to get information on your application as a whole.

Getting Basic Data

Xprofiler makes it easy to get data on specific items in the function call tree. Once you've located the item you are interested in, you can get data a number of ways. If you are having trouble locating a function in the function call tree, see “Locating Specific Objects in the Function Call Tree” on page 129.

Basic Function Data: Below each function box in the function call tree is a label that contains basic performance data. The example below shows the function box for the function *main*, and it's label.

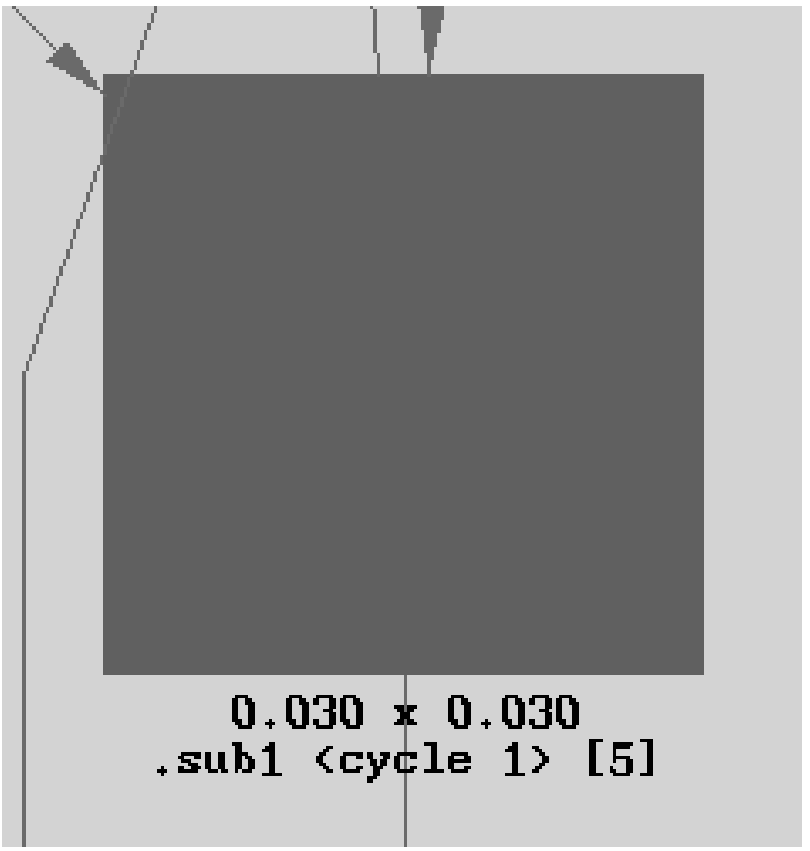


Figure 39. Example of a Function Box Label

The label contains:

- The name of the function, its associated cycle, if any, and its index. In the example above, the name of the function is *sub1*. It is associated with cycle 1, and its index is 5.
- The total amount of CPU time (in seconds) this function spent on itself plus the amount of CPU time it spent on its descendants (the number on the left of the *x*). In the example above, the function *sub1* spent .030 seconds on itself, plus its descendants.
- The amount of CPU time (in seconds) this function spent only on itself (the number on the right of the *x*). In the example above, the function *sub1* spent .030 seconds on itself.

Since labels are not always visible in the Xprofiler window when it is fully zoomed out, you may need to zoom in on it in order to see the labels. See “Zooming In on the Function Call Tree” on page 115 for information on how to do this.

Basic Call Data: Call arc labels appear over each call arc. The label shows you the number of calls that were made between the two functions (from caller to callee). For example:

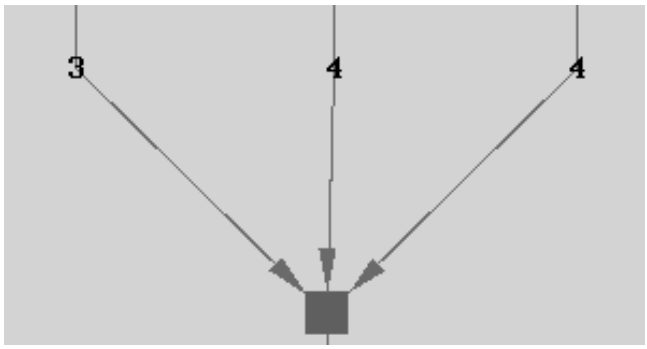


Figure 40. Example of a call arc label

In order to see a call arc label, you will probably need to zoom in on it. See “Zooming In on the Function Call Tree” on page 115 for information on how to do this.

Basic Cluster Data: Cluster box labels tell you the name of the library that is represented by that cluster. If it is a shared library, the label shows its full pathname.

Information Boxes: For each function box, call arc, and cluster box, there is a corresponding information box that you can access with your mouse. It gives you the same basic data that appears on the label. This is useful when the Xprofiler display is fully zoomed out and the labels are not visible. To access the information box, click on the function box, call arc, or cluster box (place it over the edge of the box) with the left mouse button. The information box appears.

For a function, the information box contains:

- The name of the function, its associated cycle, if any, and its index.
- The amount of CPU used by this function. There are two values supplied in this field. The first is the amount of CPU time spent on this function plus the time spent on its descendants. The second value represents the amount of CPU time this function spent only on itself.
- The number of times this function was called (by itself or any other function in the application).

  For a call, the information box contains:

  – The caller and callee functions (their names) and their corresponding indexes.

  – The number of times the caller function called the callee.

  For cluster, the information box contains:

- The name of the library

- The total CPU usage (in seconds) consumed by the functions within it.

Function Menu Statistics Report Option: You can get performance statistics for a single function via the *Statistics Report* option of the Function menu. It lets you see data on the CPU usage and call counts of the selected function. If you are using more than one gmon.out file, this option breaks down the statistics per each gmon.out file you use.

When you select the *Statistics Report* menu option, the *Function Level Statistics Report* window appears, as in the example below.

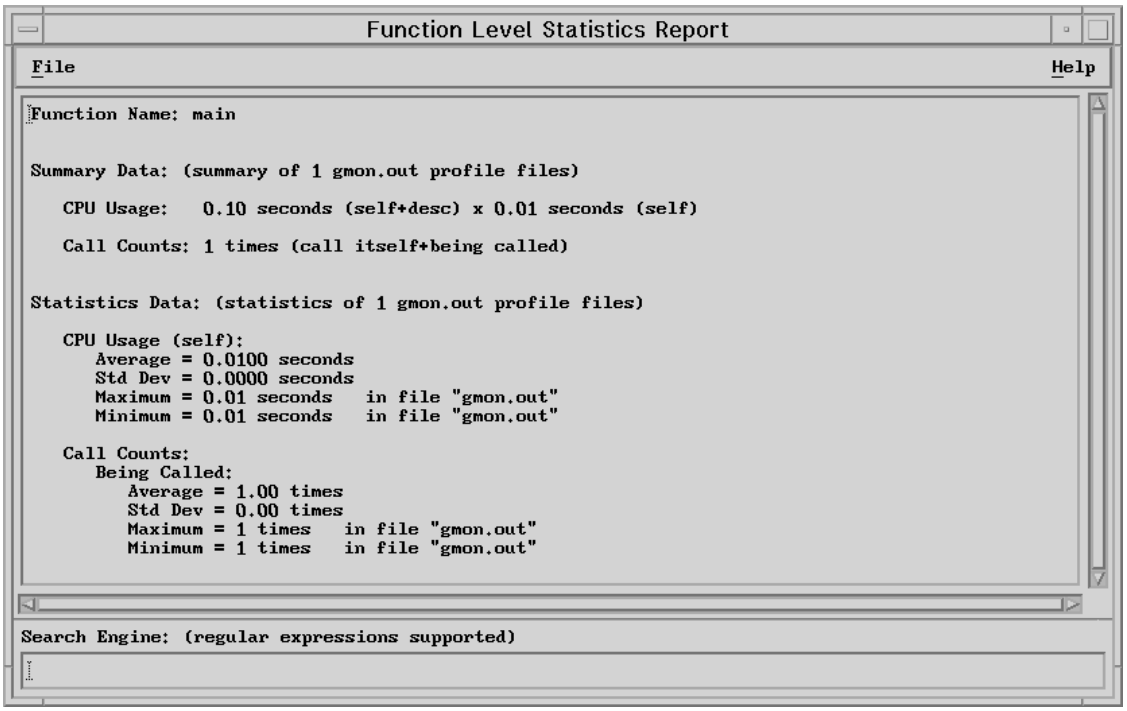


Figure 41. Function Level Statistics Report window

The Function Level Statistics Report window provides the following information:

Function Name: The name of the function you selected. In Figure 41, the function name is *main*.

Summary Data: The total amount of CPU used by this function. If you used multiple gmon.out files, the value shown here represents their sum.

The *CPU Usage* field shows you:

- The amount of CPU used by this function. There are two values supplied in this field. The first is the amount of CPU time spent on this function plus the time spent on its descendants. The second value represents the amount of CPU time this function spent only on itself.

  In Figure 41, CPU usage is listed as *0.10 seconds (self+desc) x 0.10 seconds (self)*

The *Call Counts* field shows you:

- The number of times this function called itself, plus the number of times it was called by other functions.

  In Figure 41 on page 133, the value in the Call Counts field is *1 times*.

Statistics Data: The CPU usage and calls made to or by this function, broken down by gmon.out file.

The *CPU Usage* field shows you:

- Average

  The average CPU time used by the data in each gmon.out file. In Figure 41 on page 133, the Average is listed as *0.0100 seconds*.

The average is calculated as:

$$\bar{x} = \frac{\Sigma x_i}{\mathrm{N}}$$

Where:

$\bar{x}$ = average value

N = number of gmon.out files

x_i = corresponding value of each gmon.out file

- Std Dev

  Standard deviation. A value that represents the difference in CPU usage samplings, per function, from one gmon.out file to another. The smaller the standard deviation, the more balanced the workload. In Figure 41 on page 133, the Std Dev is listed as *0.0000 seconds*

The standard deviation is calculated as:

$$\sqrt{\frac{\Sigma(x_i)^2}{\mathrm{N}} - (\bar{x})^2}$$

- Maximum

  Of all the gmon.out files, the maximum amount of CPU time used. The corresponding gmon.out file appears to the right. In Figure 41 on page 133, the Maximum is listed as *0.01 seconds*.

- Minimum

  Of all the gmon.out files, the minimum amount of CPU time used. The corresponding gmon.out file appears to the right. In Figure 41 on page 133, the Minimum is listed as *0.01 seconds*.

The *Call Counts* field shows you:

- Average

  The average number of calls made to this function or by this function, per gmon.out file. In Figure 41 on page 133, the Average is *1.00 times*.

- Std Dev

Standard deviation. A value that represents the difference in call count sampling, per function, from one gmon.out file to another. A small standard deviation value in this field means that the function was almost always called the same number of times in each gmon.out file. In Figure 41 on page 133, the Std Dev is *0.00 times*.

- Maximum

  The maximum number of calls made to this function or by this function in a single gmon.out file. The corresponding gmon.out file appears to the right. In Figure 41 on page 133, the Maximum is *1 times*.

- Minimum

  The minimum number of calls made to this function or by this function in a single gmon.out file. The corresponding gmon.out file appears to the right. In Figure 41 on page 133, the Minimum is *1 times*.

Getting Detailed Data via Reports

Xprofiler provides performance data in textual and tabular format. This data is provided in various tables called *reports*. If you are a **gprof** user, you are familiar with the *Flat Profile*, *Call Graph Profile*, and *Function Index* reports. Xprofiler generates these same reports, in the same format, plus two others.

You can access the Xprofiler reports from the Report menu. The Report menu lets you see the following reports:

- Flat Profile
- Call Graph Profile
- Function Index
- Function Call Summary
- Library Statistics

Each report window includes a File menu. Under the File menu is the *Save As* option which allows you to save the report to a file. For information on using the *Save File Dialog* window to save a report to a file, see “Using the Save Dialog Windows” on page 112.

Each report window also includes a *Search Engine* field, which is located at the bottom of the window. The Search Engine lets you search for a specific string in the report. For information on using the Search Engine field, see “Using the Search Engine” on page 112.

Note: If you select the *Save As* option from the Flat Profile, Function Index, or Function Call Summary report windows, you must either complete the save operation or cancel it before you can select any other option from the menus of these reports. You can, however, use the other menus of Xprofiler before completing the save operation or canceling it, with the exception of the Load Files option, of the File menu, which remains greyed out.

Each of the Xprofiler reports are explained below.

Flat Profile Report

When you select the *Flat Profile* menu option, the *Flat Profile* window appears. The Flat Profile report shows you the total execution times and call counts for each function (including shared library calls) within your application. The entries for the functions that use the greatest percentage of the total CPU usage appear at the top of the list, while the remaining functions appear in descending order, based on the amount of time used.

Unless you specified the **-z** command line option, the Flat Profile report does not include functions whose CPU usage is 0 (zero) and have no call counts.

Note that the data presented in the Flat Profile window is the same data that is generated with the UNIX **gprof** command.

The Flat Profile report looks similar to this:

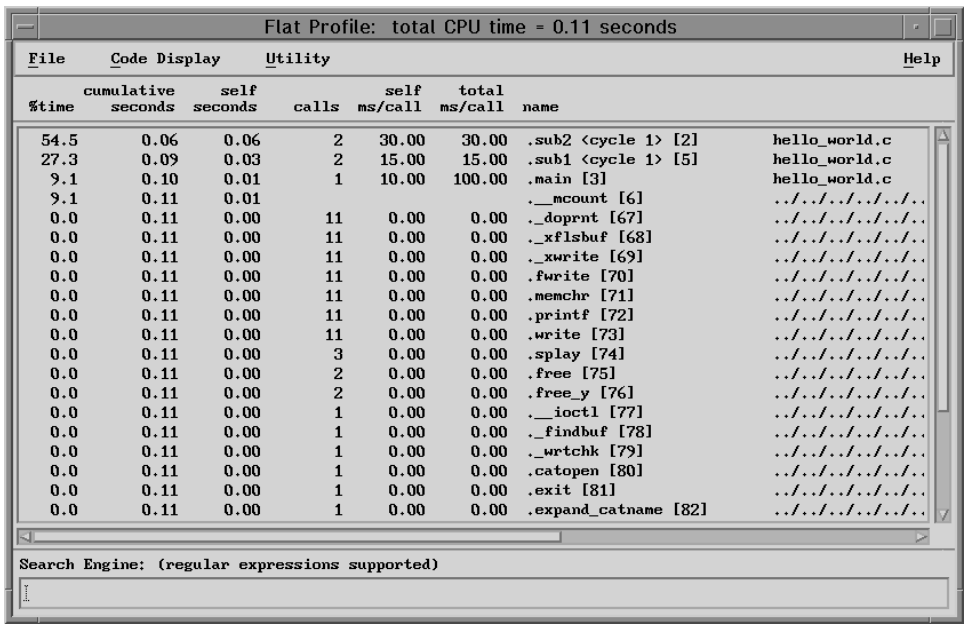


Figure 42. Flat Profile Report

Flat Profile Window Fields: The Flat Profile window fields are explained below.

- %time

  The percentage of the program's total CPU usage that is consumed by this function.

- cumulative seconds

  A running sum of the number of seconds used by this function and those listed above it.

- self seconds

  The number of seconds used by this function alone. The *self seconds* values are what Xprofiler uses to sort the functions of the Flat Profile report.

- calls

  The number of times this function was called (if this function is profiled). Otherwise, it is blank.

- self ms/call

  The average number of milliseconds spent in this function per call (if this function is profiled). Otherwise, it is blank.

- total ms/call

  The average number of milliseconds spent in this function and its descendants per call (if this function is profiled). Otherwise, it is blank.

- name

  The name of the function. The *index* appears in brackets [] to the right of the function name. The index serves as the function's identifier within Xprofiler. It also appears below the corresponding function in the function call tree.

Call Graph Profile Report

The *Call Graph Profile* menu option lets you view the functions of your application, sorted by the percentage of total CPU usage that each function, and its descendants, consumed. When you select this option, the *Call Graph Profile* window appears.

Unless you specified the **-z** command line option, the Call Graph Profile report does not include functions whose CPU usage is 0 (zero) and have no call counts.

Note that the data presented in the Call Graph Profile window is the same data that is generated with the UNIX **gprof** command.

The Call Graph Profile report looks similar to this:

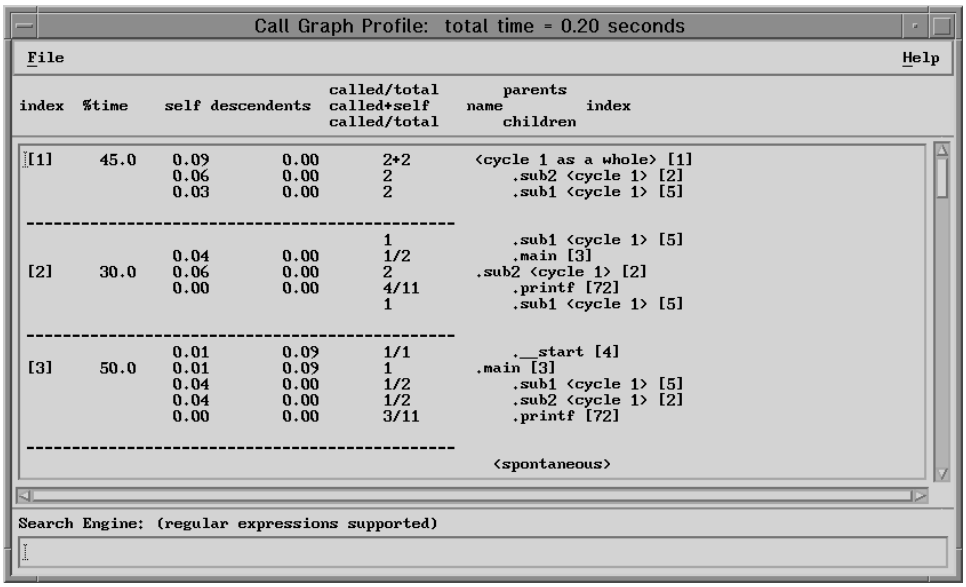


Figure 43. Call Graph Profile Report

Call Graph Profile Window Fields: The fields of the Call Graph Profile are explained below.

- index

  The index of the function in the Call Graph Profile. Each function in the Call Graph Profile has an associated index number which serves as the function's

identifier. The same index also appears with each function box label in the function call tree, as well as other Xprofiler reports.

- %time

  The percentage of the program's total CPU usage that was consumed by this function and its descendants.

- self

  The number of seconds this function spends within itself.

- descendants

  The number of seconds spent in the descendants of this function, on behalf of this function.

- called/total, called+self, called/total

  The heading of this column refers to the three different kinds of calls that take place within your program. The values in this field correspond to the functions listed in the *name, index, parents, children* field to its right. Depending on whether the function is a parent, child, or the function of interest (the function who's index is listed in the *index* field of this row), this value can stand for one of the following:

  – Number of times a parent called the function of interest

  – Number of times the function of interest called itself, recursively

  – Number of times the function of interest called a child

In the example below, *sub2* is the function of interest, *sub1* and *main* are its parents, and *printf* and *sub1* are its children.

```
called/total       parents
called+self    name           index
called/total       children

---------------
      1              .sub1 <cycle 1> [5]
      1/2            .main [3]
      2          .sub2 <cycle 1> [2]
      4/11           .printf [72]
      1              .sub1 <cycle 1> [5]
```

Figure 44. called/total, call/self, called/total field

- called/total

  For a parent function, this refers to the number of calls made to the function of interest, as well as the total number of calls it made to all functions. In the example above, one of the parent functions, *main*, made two calls; one to the function of interest, *sub2*, and one to another function.

- called+self

This refers to the number of times the function of interest called itself, recursively. In the example above, the function of interest, *sub2*, called itself two times. For a child function, this refers to the number of times the function of interest called this child. In the example above, one of the child functions, *printf*, was called eleven times; four times by the function of interest, *sub2*, and seven times by other functions.

- name, index, parents, children

  The layout of the heading of this column is indicative of the information that is provided. To the left is the name of the function, and to its right is the function's index number. Appearing above the function are its parents, and below are its children.

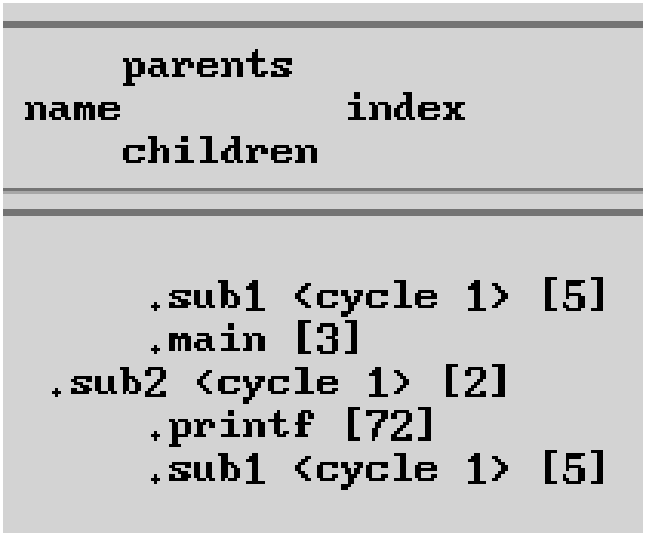


Figure 45. name/index/parents/children field

- name

  The name of the function, with an indication of its membership in a cycle, if any. Note that the function of interest appears to the left, while its parent and child functions are indented above and below it. In the example above, the name of the function is *sub2*.

- index

  The index of the function in the Call Graph Profile. This number corresponds to the index that appears in the *index* column of the Call Graph Profile and the on the function box labels in the function call tree. In the example above, the index of *sub2* is [2].

- parents

  The parents of the function. A *parent* is any function that directly calls the function in which you are interested. In the example above, the parents are *sub1* and *main*.

  If any portion of your application was not compiled with the **-pg** option, Xprofiler will not be able to identify the parents for the functions within those portions. As a result, these parents will be listed as *spontaneous* in the Call Graph Profile report.

- children

  The children of the function. A *child* is any function that is directly called by the function in which you are interested. In the example above, the children are *printf* and *sub1*.

Function Index Report

The *Function Index* menu option lets you view a list of the function names included in the function call tree. When you select this option, the *Function Index* window appears, and displays the function names in alphabetical order. To the left of each function name is its *index*, enclosed in brackets []. The index is the function's identifier, which is assigned by Xprofiler. An index also appears on the label of each corresponding function box in the function call tree as well as other reports.

Unless you specified the **-z** command line option, the Function Index report does not include functions whose CPU usage is 0 (zero) and have no call counts.

The Function Index report looks similar to this:

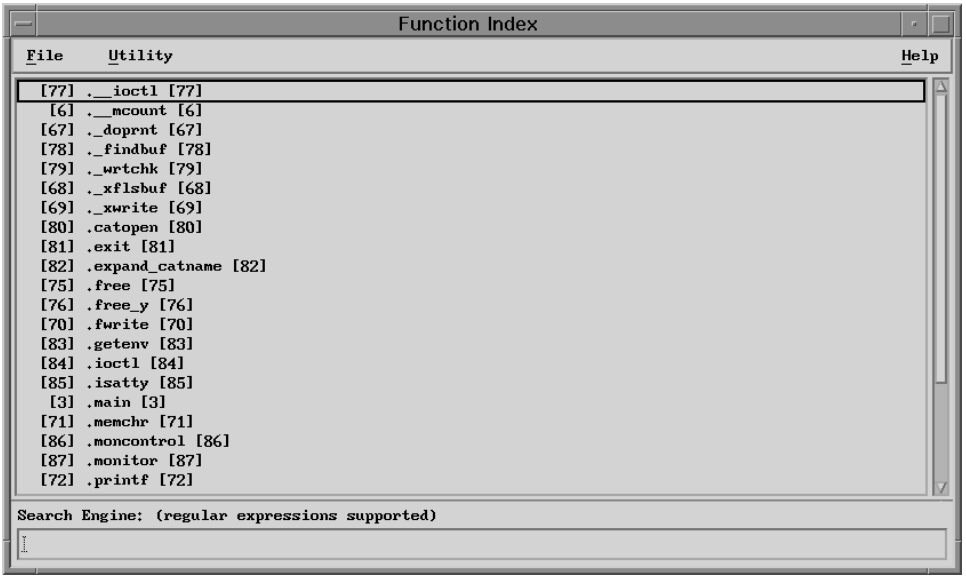


Figure 46. Sample Function Index Report

Function Call Summary Report

The *Function Call Summary* menu option lets you display all the functions in your application that call other functions. They appear as caller-callee pairs (call arcs, in the function call tree), and are sorted by the number of calls in descending order. When you select this option, the *Function Call Summary* window appears.

The Function Call Summary report looks similar to this: