

z/OS



# UNIX System Services Parallel Environment: Operation and Use



z/OS



# UNIX System Services Parallel Environment: Operation and Use

**Note**

Before using this information and the product it supports, be sure to read the general information under "Appendix G. Notices" on page 303

**First Edition, March 2001**

This edition applies to Version 1 Release 1 of z/OS (5694-A01), 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 below.

IBM® welcomes your comments. A form for readers' comments may be provided at the back of this publication, or you may address your comments to the following address:

IBM Deutschland Entwicklung GmbH  
Information Development, Dept. 3248  
Schönaicher Str. 220  
71032 Böblingen  
Germany

FAX (Germany): 07031-16-3456

FAX (Other Countries):

Your International Access Code +49-7031-16-3456

Internet e-mail: [s390id@de.ibm.com](mailto:s390id@de.ibm.com)

World Wide Web: <http://www.ibm.com/servers/eserver/zseries/zos>

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.

Permission to copy without fee all or part of *MPI: A Message Passing Interface Standard, Version 1.1* by Message Passing Interface Forum is granted, provided the University of Tennessee copyright notice and the title of the document appear, and notice is given that copying is by permission of the University of Tennessee. © 1993, 1995 University of Tennessee, Knoxville, Tennessee.

© Copyright International Business Machines Corporation 1997, 2001. All rights reserved.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

# Contents

**Tables . . . . . vii**

**Figures . . . . . ix**

**About this book . . . . . xi**

Who should use this book . . . . . xi  
How this book is organized . . . . . xi  
    Overview of contents . . . . . xi  
    Document conventions . . . . . xii  
National Language Support. . . . . xiv  
z/OS Migration Information. . . . . xv

**Chapter 1. Introduction . . . . . 1**

**Chapter 2. Preparing to use POE . . . . . 3**

Customizing INETD and setting up the Partition  
Daemon . . . . . 3  
    Configuring the inetd.conf file . . . . . 3  
    Configuring the services file . . . . . 3  
    Customizing Your Code Page for pmd. . . . . 3  
    WLM Support setup . . . . . 3  
Access . . . . . 4  
Host list file . . . . . 4  
Scheduling environments . . . . . 5  
WLM Multi-system Enclaves . . . . . 5  
Restrictions in security environments . . . . . 6

**Chapter 3. Getting started with POE . . . . . 7**

The Parallel Operating Environment . . . . . 7  
Starting the POE . . . . . 7  
Running simple commands . . . . . 7  
Compiling and running a simple parallel application 10  
Getting a little more information . . . . . 11  
    Most likely needed POE options . . . . . 13

**Chapter 4. Executing parallel programs 15**

Executing parallel programs using POE . . . . . 15  
    Step 1: Compile the program . . . . . 15  
    Step 2: Copy files to individual nodes . . . . . 16  
    Step 3: Set up the execution environment . . . . . 16  
    Step 4: Start the X-Windows analysis tool . . . . . 24  
    Step 5: Invoke the executable . . . . . 25  
Controlling program execution . . . . . 30  
    Specifying develop mode. . . . . 31  
    Making POE ignore arguments. . . . . 31  
    Managing standard input, output, and error . . . . . 32

**Chapter 5. Managing POE jobs . . . . . 41**

Stopping a POE job. . . . . 41  
Cancelling and killing a POE job . . . . . 41  
Detecting remote node failures . . . . . 41  
Asynchronous Interrupts support . . . . . 41  
    Using MP\_CSS\_INTERRUPT . . . . . 41  
    Support for Performance Improvements. . . . . 42

Parallel file copy utilities . . . . . 44

**Chapter 6. Monitoring program execution . . . . . 45**

Step 1: Call PM Array parallel utility functions . . . . . 46  
Step 2: Compile the program . . . . . 46  
Step 3: Set up your X-Windows environment . . . . . 46  
Step 4: Set the number of lights. . . . . 47  
Step 5: Open the PM Array window . . . . . 47  
Step 6: Invoke the program and monitor its  
execution . . . . . 48  
    Displaying details of a light . . . . . 48  
    Displaying task output . . . . . 48  
Step 7: Close the PM Array window . . . . . 49

**Chapter 7. Techniques for creating parallel programs . . . . . 51**

Message passing. . . . . 51  
Data decomposition . . . . . 52  
    Functional decomposition . . . . . 58  
    Duplication versus redundancy. . . . . 61

**Chapter 8. Programming considerations for user applications in POE . . . . . 63**

Environment overview . . . . . 63  
User authentication. . . . . 63  
Exit status . . . . . 64  
POE job-step function . . . . . 65  
POE additions to the user executable. . . . . 65  
    Initialization user exits CEEBXITA and CEEBINT 65  
    Signal Handlers . . . . . 65  
Limitations in setting the thread stacksize . . . . . 67  
Do not hard code file descriptor numbers . . . . . 67  
POE gets control first and handles process  
initialization . . . . . 67  
Termination of a parallel job. . . . . 67  
Your program cannot run as root . . . . . 67  
Forks are limited . . . . . 68  
Shell execution . . . . . 68  
Do not rewind stdin, stdout or stderr. . . . . 68  
Ensuring that string arguments are passed to your  
program correctly . . . . . 68  
Network tuning considerations. . . . . 68  
Standard I/O requires special attention . . . . . 69  
    STDIN/STDOUT piping example . . . . . 69  
Program and thread termination . . . . . 70  
Other thread-specific considerations . . . . . 70  
    Order requirement for system includes . . . . . 70  
    MPI\_Init . . . . . 70  
    Collective communications . . . . . 71  
Reserved environment variables . . . . . 71  
Message catalog considerations. . . . . 71  
MPI-IO Requires Shared HFS To Be Used Effectively 71

Using Shared Memory. . . . . 71

**Chapter 9. Debugging . . . . . 73**

Messages . . . . . 73  
    Message catalog errors. . . . . 73  
    Finding messages . . . . . 73  
    Logging POE errors to a file. . . . . 74  
    Message format . . . . . 74  
Cannot compile a parallel program . . . . . 74  
Cannot start a parallel job . . . . . 75  
Cannot execute a parallel program . . . . . 76  
The program runs but... . . . . 77  
    Debugging your parallel program . . . . . 77  
    No output at all . . . . . 78  
    It hangs . . . . . 78  
    Let us attach the debugger . . . . . 81  
    Other hang-ups . . . . . 87  
    Bad output . . . . . 88  
Debugging and Threads . . . . . 88  
Keeping an eye on progress . . . . . 89

**Chapter 10. Using the pdbx debugger 93**

pdbx subcommands . . . . . 93  
Starting the pdbx debugger . . . . . 96  
    Normal mode . . . . . 96  
    Attach mode . . . . . 99  
    Attach screen. . . . . 99  
Loading the partition with the load subcommand 101  
    Displaying tasks and their states . . . . . 102  
    Grouping tasks . . . . . 103  
    Controlling program execution . . . . . 110  
    Examining program data . . . . . 117  
    Other key features. . . . . 120  
    Other important notes on pdbx . . . . . 124  
    Exiting pdbx . . . . . 124

**Chapter 11. Using the pedb debugger 127**

Setting up the debugger environment . . . . . 127  
    Setting up your X-Window environment . . . . . 127  
    Be aware of storage requirements. . . . . 128  
    Setting up the pedb.ad file . . . . . 128  
Starting the pedb debugger. . . . . 128  
    Normal mode . . . . . 128  
    Attach mode . . . . . 130  
    Attach window. . . . . 131  
The pedb main window . . . . . 134  
Loading the partition from the Load Executables window . . . . . 135  
    Program search path . . . . . 136  
    The pedb window with a partition loaded. . . . . 138

**Appendix A. Parallel Environment commands . . . . . 185**

mcp . . . . . 186  
mcpqath . . . . . 188  
mcpscat . . . . . 192  
mpcc/mpCC . . . . . 195  
pdbx . . . . . 197  
pdbx alias subcommand. . . . . 202  
pdbx assign subcommand . . . . . 204

pdbx attach subcommand . . . . . 205  
pdbx back subcommand. . . . . 206  
pdbx case subcommand . . . . . 207  
pdbx catch subcommand . . . . . 208  
pdbx condition subcommand . . . . . 209  
pdbx cont subcommand . . . . . 210  
pdbx dbx subcommand . . . . . 211  
pdbx delete subcommand . . . . . 212  
pdbx detach subcommand . . . . . 213  
pdbx dhelp subcommand . . . . . 214  
pdbx display memory subcommand. . . . . 215  
pdbx down subcommand . . . . . 216  
pdbx dump subcommand . . . . . 217  
pdbx file subcommand . . . . . 218  
pdbx func subcommand. . . . . 219  
pdbx goto subcommand. . . . . 220  
pdbx gotoi subcommand . . . . . 221  
pdbx group subcommand . . . . . 222  
pdbx halt subcommand . . . . . 224  
pdbx help subcommand . . . . . 225  
pdbx hook subcommand . . . . . 226  
pdbx ignore subcommand . . . . . 227  
pdbx list subcommand . . . . . 228  
pdbx listi subcommand . . . . . 230  
pdbx load subcommand. . . . . 231  
pdbx map subcommand. . . . . 232  
pdbx mutex subcommand . . . . . 233  
pdbx next subcommand . . . . . 234  
pdbx nexti subcommand . . . . . 235  
pdbx on subcommand . . . . . 236  
pdbx print subcommand . . . . . 238  
pdbx quit subcommand . . . . . 239  
pdbx registers subcommand . . . . . 240  
pdbx return subcommand . . . . . 241  
pdbx search subcommand . . . . . 242  
pdbx set subcommand . . . . . 243  
pdbx sh subcommand . . . . . 244  
pdbx skip subcommand . . . . . 245  
pdbx source subcommand . . . . . 246  
pdbx status subcommand . . . . . 247  
pdbx step subcommand . . . . . 248  
pdbx stepi subcommand . . . . . 249  
pdbx stop subcommand . . . . . 250  
pdbx tasks subcommand . . . . . 252  
pdbx thread subcommand . . . . . 253  
pdbx trace subcommand . . . . . 254  
pdbx unalias subcommand. . . . . 256  
pdbx unhook subcommand. . . . . 257  
pdbx unset subcommand . . . . . 258  
pdbx up subcommand . . . . . 259  
pdbx use subcommand . . . . . 260  
pdbx whatis subcommand . . . . . 261  
pdbx where subcommand . . . . . 262  
pdbx whereis subcommand . . . . . 263  
pdbx which subcommand . . . . . 264  
pedb . . . . . 265  
pmarray . . . . . 268  
poe. . . . . 270  
poekill. . . . . 281

<b>Appendix B. POE environment variables and command-line flags . . .</b>	<b>283</b>
---	------------

<b>Appendix C. Command-line flags for Normal or Attach Mode . . . . .</b>	<b>291</b>
---	------------

<b>Appendix D. MPI safety . . . . .</b>	<b>293</b>
---	------------

Safe MPI coding practices . . . . .	293
What is a safe program? . . . . .	293
Some general hints and tips . . . . .	293
Order . . . . .	293
Progress . . . . .	294
Fairness . . . . .	294
Resource limitations . . . . .	295

<b>Appendix E. Copying Parallel Environment Executables in a Security Environment . . . . .</b>	<b>297</b>
---	------------

<b>Appendix F. Migration and Multi-release Compatibility . . . . .</b>	<b>299</b>
--	------------

Parallel Environment Compatibility . . . . .	299
Parallel Environment Releases . . . . .	299
Principles of Multi-release Compatibility . . . . .	299

Installing a Back-level Release . . . . .	300
Installing a Back-level poe . . . . .	300
Installing a Back-level pmd . . . . .	300
Installing a Back-level ppe.dll . . . . .	301
Running a Back-level Release of Parallel Environment . . . . .	301
Multi-release Compatibility for Statically Linked Programs . . . . .	302
Servicing Back-level Parallel Environment Releases	302

<b>Appendix G. Notices . . . . .</b>	<b>303</b>
--------------------------------------	------------

Trademarks . . . . .	304
Accessing Licensed Books on the Web . . . . .	305
LookAt System for Online Message Lookup . . . . .	305

<b>Bibliography . . . . .</b>	<b>307</b>
-------------------------------	------------

Related publications . . . . .	307
Parallel Environment publications . . . . .	307
Related z/OS publications . . . . .	307
Related non-IBM publications . . . . .	307

<b>Glossary of terms and abbreviations</b>	<b>309</b>
--	------------

<b>Index . . . . .</b>	<b>315</b>
------------------------	------------



---

## Tables

1. Context insensitive pdbx subcommands	94	13. POE environment variables and command-line flags for I/O control	286
2. Context sensitive pdbx subcommands.	94	14. POE environment variables and command-line flags for diagnostic information	287
3. Debugger option flags (pdbx)	97	15. POE environment variables and command-line flags for Message Passing Interface (MPI)	289
4. Task states	106	16. Other POE environment variables and command-line flags	290
5. Debugger option flags (pedb)	130	17. Command-line flags for Normal or Attach Mode	291
6. Status codes	143	18. Relation between OS releases, Parallel Environment releases and PTF's	299
7. Control buttons	144		
8. Removing breakpoints	150		
9. Removing tracepoints.	155		
10. Default color scheme	181		
11. POE environment variables and command-line flags for Partition Manager control.	284		
12. POE environment variables and command-line flags for job specification	285		



---

## Figures

1. Output from C-program compiled with mpcc	11	16. Add Group window	141
2. The Program Marker Array	45	17. Overflow icon	159
3. Formula for sine function	59	18. Variable Viewer icon	159
4. Formula for partial term	61	19. Variable Viewer window	160
5. Attach Dialog window	82	20. Threads Viewer window	162
6. pedb main window	83	21. Array Subrange window	168
7. Getting additional information about a task	84	22. Application Message Queues window	170
8. Global Data window	85	23. Select Filters window	172
9. Program Marker Array	90	24. Task Message Queue window	173
10. pdbx Attach screen	100	25. Point to Point Message Details window	175
11. Relationship between home node (pdbx) and remote tasks (dbx processes)	106	26. Send/Receive Message Details window	175
12. pedb Attach window	132	27. Collective Communications Details window	176
13. pedb main window before the partition is loaded	135	28. Early Arrival Message Details window	176
14. Load Executables window	136	29. Message Group Information window	177
15. pedb main window after the partition is loaded	138	30. Find window	180



---

## About this book

This book provides guidance for using the z/OS UNIX System Services Parallel Environment (PE).

The PE introduces new capabilities to enable developers to write and run parallel UNIX® applications on z/OS®, enabling new applications such as parallel data mining. Applications running simultaneously now can communicate with one another. The z/OS UNIX System Services Parallel Environment supports Workload Manager and takes advantage of the Parallel Sysplex® workload balancing of resource-intensive applications such as data mining.

The PE consists of two components:

### **Message Passing Interface (MPI)**

The Message Passing Interface is a standard for application communication in a parallel environment. In this book, you will find information on basic parallel programming concepts and on the (MPI) standard. You will also find information about the application development tools that are provided by PE such as the Parallel Operating Environment.

### **Parallel Operating Environment (POE)**

The POE provides the user with a single-job view for a set of parallel application processes including two parallel debuggers **pdbx** and **pedb**.

This book shows how to use POE's facilities to compile, run, and analyze parallel programs. For information on how to write parallel programs, refer to *z/OS UNIX System Services Parallel Environment: MPI Programming and Subroutine Reference*.

---

## 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 that is covered should interest system administrators.

---

## How this book is organized

### Overview of contents

This book contains the following information:

- “Chapter 1. Introduction” on page 1 is a quick overview of the PE. It describes the various PE components, and how you might use each in developing a parallel application program.
- “Chapter 2. Preparing to use POE” on page 3 contains information on what you must do before you can use the PE.
- “Chapter 3. Getting started with POE” on page 7 shows you how to start the PE, how to use simple commands, and how to compile and run a sample parallel application.
- “Chapter 4. Executing parallel programs” on page 15 describes how to compile and run parallel programs using the Parallel Operating Environment (POE) in more detail.
- “Chapter 5. Managing POE jobs” on page 41 describes the tasks involved with managing POE jobs.

- “Chapter 6. Monitoring program execution” on page 45 describes the Program Marker Array which allows you to monitor program execution online.
- “Chapter 7. Techniques for creating parallel programs” on page 51 covers parallelization techniques and discusses their advantages and disadvantages.
- “Chapter 8. Programming considerations for user applications in POE” on page 63 contains various information for user applications that are written to run under the PE.
- “Chapter 9. Debugging” on page 73 tells you about common problems you might run into, and what to do about them.
- “Chapter 10. Using the pdbx debugger” on page 93 describes the **pdbx** debugger. This debugger extends the **dbx** debugger’s line-oriented interface and subcommands.
- “Chapter 11. Using the pedb debugger” on page 127 describes the **pedb** debugger, which is designed to debug parallel C applications.
- “Appendix A. Parallel Environment commands” on page 185 contains detailed descriptions of the PE commands that are discussed throughout this book.
- “Appendix B. POE environment variables and command-line flags” on page 283 describes the environment variables you can set to influence the running of parallel programs. This appendix also describes the command-line flags that are associated with each of the environment variables. When starting a parallel program, you can use these flags to override the value of an environment variable.
- “Appendix C. Command-line flags for Normal or Attach Mode” on page 291 lists the **poe** command-line flags that **pdbx** and **pedb** use.
- “Appendix D. MPI safety” on page 293 provides you with some general guidelines for creating safe parallel MPI programs.
- “Appendix E. Copying Parallel Environment Executables in a Security Environment” on page 297 describes which actions should be taken after the PE executables were copied. These actions are only necessary when a security environment exists.
- “Appendix F. Migration and Multi-release Compatibility” on page 299 provides information how to simultaneously run older PE releases and the standard release installed on each system.

### Note on terminology

z/OS UNIX System Services Parallel Environment is the official name of the function. The abbreviation used in this book for the function is PE.

The UNIX System Services were formerly called OpenEdition<sup>®</sup>.

## Document conventions

This manual uses visual cues to help you locate and identify information quickly and easily. The highlighting conventions used in this book are outlined in the following table.

Typographic	Usage
<b>Bold</b>	<ul style="list-style-type: none"> <li>• <b>Bold</b> words or characters represent system elements that you must use literally, such as commands, flags, and path names.</li> <li>• <b>Bold</b> words also indicate the first use of a term included in the glossary.</li> </ul>
<i>Italic</i>	<ul style="list-style-type: none"> <li>• <i>Italic</i> words or characters represent variable values that you must supply.</li> <li>• <i>Italics</i> are also used for book titles and for general emphasis in text.</li> </ul>
Constant width	Examples and information that the system displays appear in constant width typeface.
[ ]	Brackets enclose optional items in format and syntax descriptions.
{ }	Braces enclose a list from which you must choose an item in format and syntax descriptions.
	A vertical bar separates items in a list of choices. (In other words, it means “or”.)
< >	Angle brackets (‘less than’ and ‘greater than’) enclose the name of a key on the keyboard. For example, <Enter> refers to the key on your terminal or workstation that is labeled with the word Enter.
...	An ellipsis indicates that you can repeat the preceding item one or more times.
<EscChar-x>	The notation <EscChar-x> indicates a control character sequence. For example, <EscChar-c> means that you enter the escape character followed by <c>.

## Syntax diagrams

This book uses railroad syntax diagrams to illustrate how to use commands. This is how you read a syntax diagram:

A command or keyword that you must enter (a required command) is displayed like this:



An optional keyword is shown below the line, like this:



A default is shown over the line, like this:



An item that can be repeated is shown like this:



---

## National Language Support

For National Language Support (NLS), PE components display messages that are located in externalized message catalogs. English versions, as well as Japanese versions, of the message catalogs are shipped as part of z/OS UNIX System Services, but users can use their own translated message catalogs. The UNIX System Services 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**.

The shipped message catalogs in English, for example, are activated through:

```
ENTER export NLSPATH=/usr/lib/nls/msg/%L/%N
      export LANG=C
```

The default message catalogs are located in the following directories:

```
/usr/lib/nls/msg/C (English)
/usr/lib/nls/msg/ja_JP (Japanese)
```

The default message catalogs in */usr/lib/nls/msg/C* must not be deleted, even if the user provides customized translated message catalogs. They contain default messages which are used in case a message cannot be found in the customized translated message catalogs.

For further information on National Language Support, see “Customizing Your Code Page for pmd” on page 3.

---

## z/OS Migration Information

This section is intended for customers migrating from PE releases of OS/390 V2R4-V2R8 to z/OS V1R1. It contains specific information on some differences between earlier releases that you need to consider prior to using z/OS Parallel Environment.

**Existing applications:** Applications from previous versions of Parallel Environment are binary compatible with z/OS Parallel Environment with the following exceptions:

- In order to run under z/OS Parallel Environment, you must re-link any statically bound applications that were created with releases of OS/390 V2R4-V2R8.
- User profiling applications which have been compiled with V2R4-V2R6 Parallel Environment must be recompiled with z/OS PE using the option `-p` of `mpcc/mpCC`.

**Multi-release Compatibility:** For information on Multi-release Compatibility, see “Appendix F. Migration and Multi-release Compatibility” on page 299.

**Hex floating point:** z/OS Parallel Environment supports only programs compiled for z/OS Hex floating point. IEEE floating point is not supported at the moment.

**Further information:** Refer to the *z/OS UNIX System Services Planning* manual for additional migration information.



---

## Chapter 1. Introduction

PE is a distributed memory message passing system which runs in the UNIX System Services (also known as OpenEdition) environment of OS/390 V2R9.

Specifically, PE can be used to develop and run parallel programs on:

- A single multi processor system
- or
- a parallel sysplex
- or
- on network (TCP/IP) connected systems.

The S/390<sup>®</sup> 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 processor nodes. The group of parallel tasks is called a *partition*.

PE supports the two basic parallel programming models – SPMD and MPMD. In the *SPMD (Single Program Multiple Data) model*, the programs running the parallel tasks of your partition are identical. The tasks, however, work on different sets of data. In the *MPMD (Multiple Program Multiple Data) model*, each node may be running a different program. A typical example of this is the master/worker MPMD program. In a master/worker program, one task – the master – coordinates the execution of all the others – the workers.

The application developer begins by creating a parallel program's source code. The application developer might create this program from scratch or could modify an existing serial program. In either case, the developer places calls to **Message Passing Interface (MPI)** routines so that it can run as a number of parallel tasks. This is known as *parallelizing* the application.

The message passing calls enable the parallel tasks of your partition to communicate data and coordinate their execution. The message passing routines in turn call communication subsystem library routines which handle communication among the processor nodes. The communication subsystem library is implemented using the User Datagram Protocol (UDP) or possibly shared memory, if processes run on the same node.

After writing the parallel program, the application developer then begins a cycle of modification and testing. The application developer now compiles and runs the program from the **home node** using the **Parallel Operating Environment (POE)**. The home node can be any z/OS system in a parallel sysplex that has the UNIX System Services configured. POE is an execution environment designed to hide, or at least smooth, the differences between serial and parallel execution.

In general, with POE, you invoke a parallel program from your home node and run its parallel tasks on a number of **remote nodes**. When you invoke a program on your home node, POE starts your **Partition Manager** which allocates the nodes of your partition and initializes the local environment. Depending on your hardware and configuration, the used nodes can be defined either explicitly with a **host list file** or implicitly which means that all available nodes in a Sysplex may be used. There exist two possible allocation schemes: one uses Round-Robin scheme and the other is based on actual WLM capacity data.

For Single Program Multiple Data (SPMD) applications the Partition Manager executes the same program on all nodes. For Multiple Program Multiple Data (MPMD) applications, the Partition Manager prompts you for the name of the program to load on each node. The Partition Manager connects the standard I/O streams to each remote node so the parallel tasks can communicate with the home node. Although you are running tasks on remote nodes, POE allows you to continue using the standard System Services execution techniques. For example, you can redirect input and output, pipe the output of programs, or use shell tools. The POE includes:

- A number of **parallel compiler scripts**. These are shell scripts that call the C or C++ compilers while also linking in an interface library to enable communication between your home node and the parallel tasks running on the remote nodes. You dynamically link in a communication subsystem implementation when you invoke the executable.
- A number of **POE environment variables** you can use to set up your execution environment. These are System Services environment variables you can set to influence the operation of POE. These environment variables control such things as how processor nodes are allocated, and how standard I/O between the home node and the parallel tasks should be handled. Most of the POE environment variables also have associated command-line flags that enable you to temporarily override the environment variable value when invoking POE and your parallel program.
- Two **parallel debugging facilities**. The first – **pdbx** – is a line oriented debugger based on the **dbx** debugger. The other – **pedb** – is a Motif based debugger.
- The **Program Marker Array**. This is a programmable array of small boxes, or *lights*, which are associated with parallel tasks. Under program control, these lights can change color to provide you with immediate visual feedback as your program executes. See page “Step 5: Open the PM Array window” on page 47 for a description how to use this tool.

---

## Chapter 2. Preparing to use POE

This chapter describes what must be done before you can use POE.

---

### Customizing INETD and setting up the Partition Daemon

The system administrator must configure the `inetd` daemon so it can start the Partition Manager Daemon (`pmd`) when requests from a Parallel Environment home node arrive on a system.

#### Configuring the `inetd.conf` file

You must add the following line to the `/etc/inetd.conf` file on all remote nodes:

```
pmv2 stream tcp nowait OMVSKERN /bin/pmd pmd
```

#### Configuring the `services` file

You must add the following line to the `/etc/services` file on the home node, and on all remote nodes:

```
pmv2 6125/tcp # POE Partition Manager Daemon
```

You can select a different port number, but the number must be the same on all nodes.

#### Customizing Your Code Page for `pmd`

**National Language support:** If your site is using its own translation of message catalogs, you need to configure `inetd` with your settings of the `LANG` and `NLSPATH` environment variables. This setting enables `pmd` to write error messages and warnings to its log-files in the defined language. If you keep the default setting, the messages will be English.

For example to set up Japanese error messages for `pmd`, do the following (you need super user authority):

- Create a shell script, e.g. in `/bin/pmd_Ja_JP`:

```
#!/bin/sh
LANG=Ja_JP NLSPATH=/usr/lib/nls/msg/%L/%N /bin/pmd
```
- Ensure that the shell script is executable by `inetd`
- Modify the following entry in `/etc/inetd.conf`

```
pmv2 stream tcp nowait OMVSKERN /bin/pmd_Ja_JP pmd_Ja_JP
```
- Force `inetd` to read the new setting:

```
> kill -HUP <inetd pid>
```

For further information on NLS, see “National Language Support” on page xiv.

#### WLM Support setup

Refer to the manual *z/OS UNIX System Services Planning*, GA22-7800.

---

## Access

Before you can run your job, you must first have access to the compute resources in your system. Here are some things to think about:

- You must have the *same* user ID and password on the home node and each remote node on which you will be running your parallel application.
- POE will not allow you to run your application as root.

Each user must have an account on all nodes where a job runs. The user name, the user ID, and the password must be the same on all nodes. Also, the user must be a member of the same named group on the home and the remote nodes. For more details about how user authentication is handled, see “User authentication” on page 63.

---

## Host list file

One way to tell POE where to run your program is by using a *host list* file. The host list file is generally in your current working directory, but you can move it anywhere you like by specifying certain parameters. This file can be given any name, but the default name is *host.list*. Many people use *host.list* as the name to avoid having to specify another parameter (as we will discuss later).

This file contains either Transmission Control Protocol (TCP) host names, Internet Protocol (IP) addresses, or MVS system names. The specified identifier is used for communication with the inetd server, that is, it must be resolvable to an IP address through the TCP/IP Domain Name Servers or the local TCP/IP HOSTS file.

A host list file consists of pin entries and pool entries. Each line in a host list file is to be considered as one entry. Pin entries are used for specific allocation and consist of a host name followed by a list of task numbers. It is possible to directly map task numbers to nodes with specific allocation.

Pool entries are used for automatic allocation. A pool entry consists of a single node name in the host list file. After specific allocation is done, all pool entries are used for automatic allocation to allocate nodes for the remaining tasks. There are two modes for automatic allocation - the Round-Robin mode (RR mode) and the Workload manager mode (WLM mode). While using the WLM mode in a Sysplex the automatic allocation is based on actual capacity data.

Here is an example of a host list file that specifies both pin and pool entries:

```
! Pin entries:
9.164.156.194           0 1
boesys1.sw.boeblingen.ibm.com 4
! Pool entries:
boesys1.sw.boeblingen.ibm.com
boesys2.sw.boeblingen.ibm.com
```

Using this host list file would execute tasks 0 and 1 on “9.164.156.194” and task 4 on “boesys1.sw.boeblingen.ibm.com”. The remaining tasks would be distributed in automatic allocation either in WLM mode or RR mode (depending on the setting of MP\_RESD).

When using WLM mode the pool entries in the host list file must serve two purposes: they must be valid MVS system names, and they must also be known to TCP/IP as TCP names. You can resolve the name restriction problem by defining

your MVS system names as aliases in the Domain Name Servers or the HOSTS file, respectively. That way, you can then always use the MVS system names.

Here's an example of a host list file that specifies the MVS names of four systems in a sysplex:

```
!pool list
SYS1
SYS2
SYS3
SYS4
```

The corresponding entries in the local TCP/IP HOSTS file look like this:

```
9.164.156.191 boesys1.sw.boeblingen.ibm.com SYS1 BOESYS1
9.164.156.192 boesys2.sw.boeblingen.ibm.com SYS2 BOESYS2
9.164.156.193 boesys3.sw.boeblingen.ibm.com SYS3 BOESYS3
9.164.156.194 boesys4.sw.boeblingen.ibm.com SYS4 BOESYS4
```

If you use automatic allocation without a host list file, all systems in the sysplex participate in automatic allocation. The host list file allows you to prevent parallel jobs from running on certain systems, for example, systems running key production applications.

---

## Scheduling environments

Another method of distributing your work across the nodes is to use WLM Resource Affinity Scheduling. Resource Affinity Scheduling allows you to define a set of resources, known as a *scheduling environment*, that a job needs in order to execute and to control the state of each resource for each individual system.

For example, a batch job submitted to JES can have a scheduling environment specified in the JCL. The job is scheduled to run on a system where the scheduling environment resources are in the correct state.

If you want to use Resource Affinity Scheduling, you must:

- Add new scheduling environment and resource names (done by a system programmer)
- Set the states of additional resources (done by an MVS operator)
- Give the POE user read access to the BPX.WLMSEVER facility class:  
PERMIT BPX.WLMSEVER ACCESS(READ) CLASS(FACILITY) ID(<user>)

---

## WLM Multi-system Enclaves

The WLM Multi-system Enclave support of PE allows a parallel PE job to run under a single WLM Multi-system Enclave. If all tasks of a POE job are in the same enclave, WLM recognizes that all tasks are part of the same work request and therefore make better decisions in making performance adjustments affecting the POE job. For detailed information on the concept of WLM Multi-system enclaves and for configuration details, please refer to *z/OS MVS Planning: Workload Management*.

If you want to use WLM Multi-system enclaves, you must give the userid, associated to the PMD (e.g. OMVSKERN) read access to the BPX.WLMSEVER facility class:

```
PERMIT BPX.WLMSEVER ACCESS(READ) CLASS(FACILITY) ID(<user>)
```

**Note:** The userid associated to the PMD is specified in the `/etc/inetd.conf` file.

The second prerequisite is that the Coupling Facility structure `SYSWLM_WORKUNIT` must be defined. Refer to the manual *z/OS MVS Planning: Workload Management* for a detailed description of the definition. The z/OS system command "DISPLAY WLM" can be used to verify that WLM is connected to this structure.

**Note:** Do not use UNIX System Services aliases for MVS user Ids when working with MVS Multi-System enclaves.

---

## Restrictions in security environments

In a security environment where the `BPX.DAEMON` facility class is defined, every parallel application that is built with the compile script `mpcc` or `mpCC`, for example

```
mpcc -o hello hello.c
```

must run in its own address space.

Therefore, if the application shall be started by its name, for example

```
hello
```

the environment variable `_BPX_SHAREAS` must be unset or set to *no*.

If `_BPX_SHAREAS` is set to *yes*, the owner of the application (or the super user) must unset the 'share address space' attribute of the application, for example

```
extattr -s hello
```

before the parallel application can be called.

Alternatively, the user can use the `poe` command to start parallel applications, for example

```
poe hello
```

without setting the 'share address space' attribute of the application.

If `_BPX_SHAREAS` is not set or set to *no*, there are no restrictions for calling the application program.

The reason for the restriction mentioned above is that each program compiled with `mpcc` or `mpCC` internally executes `poe`, which is program-controlled.

---

## Chapter 3. Getting started with POE

This chapter describes:

- The Parallel Operating Environment (POE)
- Starting the POE
- Running simple commands
- Compiling and running a simple parallel application
- Environment setup and debugging

---

### The Parallel Operating Environment

This chapter describes the Parallel Operating Environment (POE). POE is designed to ease the transition from serial to parallel application development and execution.

The purpose of the Parallel Operating Environment (POE) is to allow you to develop and execute your parallel applications across multiple processors, called nodes. When using POE, there is a single node (a UNIX System Services system) called the *home node* that manages interactions with users. POE transparently manages the allocation of remote nodes, where your parallel application actually runs. It also handles the various requests and communication between the home node and the remote nodes via the underlying network.

---

### Starting the POE

The **poe** command enables you to load and execute programs on remote nodes. The syntax is:

```
▶▶ poe [program] [options] ▶▶
```

When you call **poe** from the command line, you are prompted for your password.

When you call **poe**, it allocates processor nodes for each task and initializes the local environment. It then loads your program and reproduces your local shell environment on each processor node. POE also passes the user program arguments to each remote node.

The exact description of the command is in “Appendix A. Parallel Environment commands” on page 185.

---

### Running simple commands

Let us try some simple examples. When you try these examples on your system, use a host list file that contains the node names. These examples also assume at least a four-node parallel environment. If you have more than four nodes, feel free to use more. If you have less than four nodes, write the names of the available systems to the host list file. This example assumes that your file is called *host.list*, and is in the directory from which you are submitting the parallel job. If either of

these conditions are not true, POE will not find the host list file unless you use the **-hostfile** option (covered on page “-hostfile or -hfile” on page 14).

The **-procs 4** option tells POE to run this command on four nodes. If no other POE options are set, it will use the first four in the host list file.

```
$ poe id -procs 4
```

```
Enter password:
uid=2715(MOST) gid=3243(DE#03243)
uid=2715(MOST) gid=3243(DE#03243)
uid=2715(MOST) gid=3243(DE#03243)
uid=2715(MOST) gid=3243(DE#03243)
```

If less than four systems are in the host list file, the available systems will be used in Round-Robin order.

What you see is the output from the **id** command run on each of the remote nodes. POE has taken care of submitting the command to each node, collecting the standard output and standard error from each remote node, and sending it back to your workstation. One thing that you do not see is which task is responsible for each line of output. In a simple example like this, it is not that important but if you had many lines of output from each node, you’d want to know which task was responsible for each line of output. To do that, you use the **-labelio** option:

```
$ poe id -procs 4 -labelio yes
```

```
Enter password:
0:uid=2715(MOST) gid=3243(DE#03243)
1:uid=2715(MOST) gid=3243(DE#03243)
2:uid=2715(MOST) gid=3243(DE#03243)
3:uid=2715(MOST) gid=3243(DE#03243)
```

This time, notice how each line starts with a number and a colon? Notice also that the numbering started at 0 (zero). The number is the task id that the line of output came from (it is also the line number in the host list file that identifies the host which generated this output). Now we can use this parameter to identify lines from a command that generates more output.

Try this command:

```
$ poe cat /etc/inetd.conf -procs 2 -labelio yes
```

You should see something similar to this:

```
Enter password:
0:#
0:# /etc/inetd.conf
0:#
0:#
0:# INTERNET SERVER CONFIGURATION DATABASE
0:#
0:#=====
0:#      !      !      !      !      !      !
0:# service ! socket ! protocol ! wait/ ! user ! server ! server prog.
0:# name    ! type   !          ! nowait !      ! prog.  ! arguments
0:#      !      !          !      !      !      !
0:#=====
0:#
0:shell  stream tcp  nowait OMVSKERN /usr/sbin/rshd rshd
```

```

0:login    stream tcp  nowait  OMVSKERN  /usr/sbin/rlogind rlogind -m
0:exec     stream tcp  nowait  OMVSKERN  /usr/sbin/rexecd rexecd
0:otelnets stream tcp  nowait  OMVSKERN  /usr/sbin/otelnetsd otelnetsd -k -t
1:#
1:# /etc/inetd.conf
1:#
1:#
1:# INTERNET SERVER CONFIGURATION DATABASE
1:#
1:#=====
1:#      !      !      !      !      !      !
1:# service ! socket ! protocol ! wait/  ! user ! server ! server prog.
1:# name    ! type  !          ! nowait !      ! prog.  ! arguments
1:#      !      !          !      !      !      !
1:#=====
1:#
1:shell    stream tcp  nowait  OMVSKERN  /usr/sbin/rshd rshd
1:login    stream tcp  nowait  OMVSKERN  /usr/sbin/rlogind rlogind -m
1:exec     stream tcp  nowait  OMVSKERN  /usr/sbin/rexecd rexecd
1:otelnets stream tcp  nowait  OMVSKERN  /usr/sbin/otelnetsd otelnetsd -k -t
0:daytime stream tcp  nowait  OMVSKERN  internal
0:time     stream tcp  nowait  OMVSKERN  internal
0:# Entry for the POE Partition Manager Daemon
0:pmv2     stream tcp  nowait  OMVSKERN  /bin/pmd pmd
1:daytime  stream tcp  nowait  OMVSKERN  internal
1:time     stream tcp  nowait  OMVSKERN  internal
1:# Entry for the POE Partition Manager Daemon
1:pmv2     stream tcp  nowait  OMVSKERN  /bin/pmd pmd

```

The `cat` command is listing the contents of the file `/etc/inetd.conf` on each of the remote nodes. But notice how the output from each of the remote nodes is intermingled? This is because as soon as a buffer is full on the remote node, POE sends it back to your home node for display (in case you had any doubts that these commands were really being executed in parallel). The result is the jumbled mess that can be difficult to interpret. Fortunately, we can ask POE to clear things up with the `-stdoutmode` parameter.

Try this command:

```
$ poe cat /etc/inetd.conf -procs 2 -labelio yes -stdoutmode ordered
```

You should see something similar to this:

```

Enter password:
0:#
0:# /etc/inetd.conf
0:#
0:#
0:# INTERNET SERVER CONFIGURATION DATABASE
0:#
0:#=====
0:#      !      !      !      !      !      !
0:# service ! socket ! protocol ! wait/  ! user ! server ! server prog.
0:# name    ! type  !          ! nowait !      ! prog.  ! arguments
0:#      !      !          !      !      !      !
0:#=====
0:#
0:shell    stream tcp  nowait  OMVSKERN  /usr/sbin/rshd rshd
0:login    stream tcp  nowait  OMVSKERN  /usr/sbin/rlogind rlogind -m
0:exec     stream tcp  nowait  OMVSKERN  /usr/sbin/rexecd rexecd
0:otelnets stream tcp  nowait  OMVSKERN  /usr/sbin/otelnetsd otelnetsd -k -t
0:daytime stream tcp  nowait  OMVSKERN  internal
0:time     stream tcp  nowait  OMVSKERN  internal
0:# Entry for the POE Partition Manager Daemon

```

```

0:pmv2      stream tcp nowait OMVSKERN /bin/pmd pmd
1:#
1:# /etc/inetd.conf
1:#
1:#
1:# INTERNET SERVER CONFIGURATION DATABASE
1:#
1:#=====
1:#      !      !      !      !      !      !      !
1:# service ! socket ! protocol ! wait/ ! user ! server ! server prog.
1:# name    ! type   !         ! nowait !      ! prog.  ! arguments
1:#      !      !         !      !      !      !
1:#=====
1:#
1:shell     stream tcp nowait OMVSKERN /usr/sbin/rshd rshd
1:login     stream tcp nowait OMVSKERN /usr/sbin/rlogind rlogind -m
1:exec      stream tcp nowait OMVSKERN /usr/sbin/rexecd rexecd
1:otelnets  stream tcp nowait OMVSKERN /usr/sbin/otelnetsd otelnetsd -k -t
1:daytime   stream tcp nowait OMVSKERN internal
1:time      stream tcp nowait OMVSKERN internal
1:# Entry for the POE Partition Manager Daemon
1:pmv2      stream tcp nowait OMVSKERN /bin/pmd pmd

```

This time, POE keeps all the output until the jobs either finish or POE itself runs out of space. If the jobs finish, POE displays the output from each remote node together. If POE runs out of space, it prints everything, and then starts a new page of output. You get less of a sense of the parallel nature of your program, but it is easier to understand. Note that the **-stdoutmode** option consumes a significant amount of system resources, which may affect performance.

---

## Compiling and running a simple parallel application

To show you how compiling works, we have selected the *Hello World* program, which uses some MPI function calls:

```

/*****
 *
 * Hello World Example
 *
 * Basic program to demonstrate compilation
 * and execution techniques
 *
 * To compile:
 * mpcc -o hello hello.c
 *
 *****/

#include <mpi.h>
void main()
{
    int rank;

    MPI_Init(0,0);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    printf("Task %i says: Hello, World!\n",rank);
    MPI_Finalize();
    exit(0);
}

```

To compile this program, you just invoke the appropriate compiler script:

```

export MP_VERBOSE=YES
mpcc -o hello hello.c

```

```
c89 -W c,dll,'LANG(EXTENDED)' -D_THREAD_SAFE -o hello hello.c /usr/lib/pe_CEEBXITA.o \
/usr/lib/pe_CEEBINT.o /usr/lib/ppe.x
```

**mpcc** is a POE script that links the parallel libraries which allow your programs to run in parallel and use MPI calls. The compiler scripts accept all the options that the non-parallel compilers do, as well as some options specific to POE. For a complete list of all parallel-specific compilation options, see “Appendix A. Parallel Environment commands” on page 185.

Running the **mpcc** script, as we have shown you, creates an executable version of your source program that takes advantage of POE. However, before POE can run your program, you need to make sure that it is accessible on each remote node. You can do this by copying it there.

Here is the output of the C program:

```
poe hello -procs 4
Enter password:
Task 0 says: Hello, World!
Task 1 says: Hello, World!
Task 2 says: Hello, World!
Task 3 says: Hello, World!
```

Figure 1. Output from C-program compiled with mpcc

---

## Getting a little more information

You can control the level of messages you get from POE as your program executes by using the **-infolevel** option of POE. The default setting is 1 (normal), which says that warning and error messages from POE will be written to STDERR. However, you can use this option to get more information about how your program executes.

For a description of the various **-infolevel** settings, see “Appendix B. POE environment variables and command-line flags” on page 283.

Here is the *Hello World* program again:

```
$ poe hello -procs 2 -labelio yes -infolevel 2
```

You should see output similar to the following:

```
Enter password:
0:INFO: FOM00724 Executing program: <hello>
1:INFO: FOM00724 Executing program: <hello>
0:INFO: FOM00619 MPCl library was compiled at Thu Oct 7 16:48:11 1999
0:Task 0 says: Hello, World!
1:Task 1 says: Hello, World!
0:INFO: FOM00306 pm_atexit: pm_exit_value is 0.
1:INFO: FOM00306 pm_atexit: pm_exit_value is 0.
INFO: FOM00656 I/O file STDOUT closed by task 0
INFO: FOM00656 I/O file STDOUT closed by task 1
INFO: FOM00656 I/O file STDERR closed by task 0
INFO: FOM00656 I/O file STDERR closed by task 1
INFO: FOM00251 task 0 exited: rc=0
INFO: FOM00251 task 1 exited: rc=0
INFO: FOM00639 Exit status from pm_respond = 0
```

With **-infolevel** set to 2, you see messages from each node that indicate the executable they're running and what the return code from the executable is. In the example above, you can differentiate between the **-infolevel** messages that come from POE itself and the messages that come from the remote nodes, because the remote nodes are prefixed with their task ID. If we didn't set **-infolevel**, we would see only the output of the executable (Hello World!, in the example above), interspersed with POE output from remote nodes.

With **-infolevel** set to a higher value (maximum is 6), you get even more information. In the following example, we use a host list file that contains one pool entry (BOESYS3), when we invoke POE.

Look at the output, below. In this case, POE tells us that it's using Round-Robin allocation mode with a host list (PM\_ALLOC\_MODE\_RR\_LIST) and added BOESYS3 to the pool list. Then we see that POE allocated BOESYS3 (9.164.172.193) for tasks 0 and 1.

```
poe hello -procs 2 -labelio yes -infolevel 6
```

You should see output similar to the following:

```
Enter password:
INFO: DEBUG_LEVEL changed from 0 to 4
D4<L4>: ALLOC: allocation mode is PM_ALLOC_MODE_RR_LIST
D4<L4>: ALLOC: node BOESYS3 added to pool list
D1<L4>: Requesting service pmv2
D4<L4>: pm_contact: MPL_addr for task 0 is 9.164.172.193
D4<L4>: pm_contact: MPL_addr for task 1 is 9.164.172.193
D1<L4>: Socket file descriptor for task 0 (BOESYS3) is 3
D1<L4>: Socket file descriptor for task 1 (BOESYS3) is 5

.
.
.

D2<L4>: Sending PMD_EXIT to task 0
D2<L4>: Sending PMD_EXIT to task 1
D2<L4>: Elapsed time for pm_remote_shutdown: 5 seconds
D2<L4>: In pm_exit... Calling exit with status = 0 at Tue Oct 19 12:18:30 1999
```

The **-infolevel** messages give you more information about what is happening on the home node, but if you want to see what is happening on the remote nodes, you need to use the **-pmdlog** option. If you set **-pmdlog** to a value of *yes*, a log is written to each of the remote nodes that tells you what POE did while running each task.

If you issue the following command, a file is written in /tmp, of each remote node, called *mplog.pid.taskid*.

```
$ poe hello -procs 4 -pmdlog yes
```

**Note:** *pid* is the process id number of the corresponding pmd (Partition Manager Daemon). *taskid* corresponds to the number of the task.

If you don't know what the process numbers are, it's probably the most recent log file (or, if you're sharing the node with other POE users, *one* of the most recent log files).

Here is a part of a sample log file:

```

z/OS USS Parallel Environment pmd version @(#) 97/03/11 11:15:31
The ID of this process is 33554894
The version of this pmd for version checking is 3100
The hostname of this node is BOESYS3
The taskid of this task is 0
HOMENAME: BOESYS3.boeblingen.de.ibm.com
USERID: 2360
USERNAME: HOLZ
GROUPID: 3243
GROUPNAME: DE#03243
PWD: /u/PE390/PUBS_R9/HELLO
PRIORITY: 0
NPROCS: 1
PMDLOG: 1
NEWJOB: 0
PDBX: 0
LIBPATH: /lib:/usr/lib:./usr/lib
VERSION (of home node): 3100
WLM_ENCLAVE: 0
ENVC recv'd
envc: 37
envc is 37
env[0] = MAIL=/usr/mail/HOLZ
env[1] = PATH=/bin:/usr/bin:.
env[2] = XENVIRONMENT=/usr/adsm/DSMX
env[3] = EDITOR=emacs
env[4] = SHELL=/bin/sh
env[5] = _C89_SLIB_PREFIX=SYS1
env[6] = _C89_CLIB_PREFIX=SYS1
env[7] = _CXX_PLIB_PREFIX=SYS1
env[8] = PS1=SYS3 $PWD>
env[9] = _CC_PLIB_PREFIX=SYS1
env[10] = _BPX_SPAWN_SCRIPT=YES
env[11] = _=/bin/poe
env[12] = CLASSPATH=./local/java/
env[13] = LOGNAME=HOLZ
env[14] = STEPLIB=SYS1.SCLBDLL
env[15] = LANG=C

```

```

.
.
.

```

```

in pmd signal handler, signal 20
wait status is 00000000
exiting child pid = 33554895
err_data is 0
child exited and all pipes closed
err_data is 0
pmd_exit reached!, exit code is 0

```

## Most likely needed POE options

There are a number of options (command-line flags) that you may want to specify when invoking POE. These options are covered in full detail in “Appendix A. Parallel Environment commands” on page 185 but here are the ones you’ll most likely need to be familiar with at this stage.

### **-procs**

When you set **-procs**, you are telling POE how many tasks your program will run. You can also set the **MP\_PROCS** environment variable to do this (**-procs** can be used to temporarily override it).

### **-hostfile or -hfile**

The default host list file used by POE to allocate nodes is called `host.list`. You can specify a file other than `host.list` by setting the **-hostfile** or **-hfile** options when invoking POE. You can also set the `MP_HOSTFILE` environment variable to do this (**-hostfile** and **-hfile** can be used to temporarily override it).

### **-labelio**

You can set the **-labelio** option when invoking POE so that the output from the parallel tasks of your program is labeled by task id. This becomes especially useful when you are running a parallel program and your output is *unordered*. With labeled output, you can easily determine which task returns which message.

You can also set the `MP_LABELIO` environment variable to do this (**-labelio** can be used to temporarily override it).

### **-infolevel or -ilevel**

You can use the **-infolevel** or **-ilevel** options to specify the level of messages you want from POE. There are different levels of informational, warning, and error messages, plus several debugging levels. Note that the **-infolevel** option consumes a significant amount of system resources. Use it with care. You can also set the `MP_INFOLEVEL` environment variable to do this (**-infolevel** and **-ilevel** can be used to temporarily override it).

### **-pmdlog**

The **-pmdlog** option lets you specify that diagnostic messages should be logged to a file in `/tmp` on each of the remote nodes of your partition. These diagnostic logs are particularly useful for isolating the cause of abnormal termination. You can also set the `MP_PMDLOG` environment variable to do this (**-pmdlog** can be used to temporarily override it).

### **-stdoutmode**

The **-stdoutmode** option lets you specify how you want the output data from each task in your program to be displayed. When you set this option to *ordered*, the output data from each parallel task is written to its own buffer, and later, all buffers are flushed, in task order, to `STDOUT`. We showed you how this works in some of the examples in this section. You can also set the `MP_STDOUTMODE` environment variable to do this (**-stdoutmode** can be used to temporarily override it).

---

## Chapter 4. Executing parallel programs

This chapter describes the steps involved in compiling and executing your parallel C or C++ programs in more detail.

---

### Executing parallel programs using POE

This section discusses how to compile and execute your parallel C or C++ programs. It leaves out the first step in any application's life cycle which is actually writing the program. For information on writing parallel programs, refer to "Chapter 7. Techniques for creating parallel programs" on page 51, and *z/OS UNIX System Services Parallel Environment: MPI Programming and Subroutine Reference*.

In order to execute an MPI parallel program, you need to:

1. Compile and link the program using shell scripts or make files which call the C or C++ compilers while linking in the Partition Manager interface and message passing subroutines.
2. Copy your executable to the individual nodes in your partition.
3. Set up your execution environment. This includes setting the number of tasks, and determining the method of node allocation.
4. Start the X-Windows Analysis Tool if you wish to analyze your application.
5. Load and execute the parallel program on the processor nodes of your partition. You can
  - load a copy of the same executable on all nodes of your partition. This is the normal procedure for SPMD programs.
  - individually load the nodes of your partition with separate executables. This is the normal procedure for MPMD programs.
  - load and execute a series of SPMD or MPMD programs, in job-step fashion, on all nodes of your partition.

#### Step 1: Compile the program

As with a serial application, you must compile a parallel C or C++ program before you can run it. Instead of using the `c89` or `c++` commands, however, you use the commands `mpcc` or `mpCC`. The `mpcc` and `mpCC` commands not only compile your program, but also link in the Partition Manager and Message Passing Interface. When you later invoke the program, the subroutines in these libraries enable the home node Partition Manager to communicate with the parallel tasks, and the tasks with each other.

**Note:** To enable the MPI-2 C++ bindings `mpCC` has to be invoked with the `"-cpp"` flag.

These two compiler commands are actually shell scripts which call the appropriate compiler. You can use any of the `c89` or `c++` flags on the `mpcc` and `mpCC` commands.

#### Creating a static executable

**Note:** We discourage you from creating statically bound executables with POE. If service is ever applied that affects any of the Parallel Environment libraries,

you will need to recompile your application to create a new executable that will work with the new libraries. This could lead to a lot of work and may expose you to potential problems, which would be avoided if dynamic libraries are used.

In general, to create a static executable, do the following:

1. Create an object file of your program using **c89** or **c++** and the compile flags **-D\_THREAD\_SAFE** and **-W c,dll,'LANG(EXTENDED)'**.

For example:

```
c89 -D_THREAD_SAFE -W c,dll,'LANG(EXTENDED)' -c myprog.c
```

**Note:** Use these compile flags for all modules.

2. Create the executable by linking your program with the POE archive library, and the POE user initialization exit modules. For statically linking you should use **c++**, because the POE archive library includes some C++ modules.

For example:

```
c++ -o myprog -L/usr/lib myprog.o /usr/lib/pe_CEEBXITA.o /usr/lib/pe_CEEBINT.o -lpe
```

If you want use **c89** for linking you have to modify the environment for the **z/OS C** compiler. To do this you have to set the environment variable **'\_C89\_PSYSLIB'** in the following way:

```
export _C89_PSYSLIB="{_PLIB_PREFIX}.SCEE0BJ:{_PLIB_PREFIX}.SCEE0PP"
```

For more information on the **'\_C89\_PSYSLIB'** environment variable, refer to *z/OS UNIX System Services Command Reference*, description of **c89** command.

## Step 2: Copy files to individual nodes

Once you have generated the executable, you must copy it to all the nodes in the sysplex. You can do this with the **mcp** command.

For example, to make a copy of *my\_program* in the **/tmp** directory to the first six nodes listed in *host.list*:

ENTER

```
mcp my_program /tmp/my_program -procs 6
```

**Note:** If you are using a shared filesystem, it is not necessary to copy the executable.

## Step 3: Set up the execution environment

This step contains the following sections:

- “Defining the pin- and pool-list” on page 17
- “Defining the automatic allocation mode” on page 21
- “Defining the partition size” on page 22
- “Examples” on page 22
- “Creating an output host list file” on page 23
- “Setting the **MP\_WLM\_ENCLAVE** environment variable” on page 24

### Note

There are differences to previous releases using the old style node allocation.

In Parallel Environment R9 Process Pinning has been introduced. This changed the node allocation step in contrast to previous releases. The new style supersedes and augments the semantics of the old style with the exception of the following two marginal situations:

- If specifying in the host list file fewer nodes than processes in the parallel application, an error occurred in the previous releases when using specific allocation. Now, the Round-Robin scheduling is used, no error occurs. Therefore it is a valid host list file.
- If `MP_HOSTFILE` is not defined and no file `./host.list` exists, in the previous releases WLM selection without restrictions was used. Now an error message will be printed which says that `./host.list` could not be opened.

Before invoking your program, you need to set up your execution environment. There are a number of POE environment variables discussed throughout this book and summarized in "Appendix B. POE environment variables and command-line flags" on page 283. Any of these environment variables can be set at this time to later influence the execution of parallel programs. This step covers those environment variables which are most important for successful invocation of a parallel program. When you invoke a parallel program, your home node Partition Manager checks these environment variables to determine:

- The way to allocate processor nodes for the parallel tasks:
  - Pin- and pool-lists are specified by the `MP_HOSTLIST` and `MP_RMPOOL` environment variables
  - The automatic allocation mode is specified by the `MP_RESD` environment variable
- The number of tasks in your program as specified by the `MP_PROCS` environment variable.

## Defining the pin- and pool-list

In a given environment some z/OS systems are available - perhaps in a (parallel) Sysplex - where poe may run the parallel tasks. Node allocation deals with the question, on which systems (nodes) poe should run these tasks.

When a parallel job is started, poe numbers the parallel tasks beginning with 0. Normally it does not matter which task number is mapped to which system, you only want to parallelize the work and distribute the workload on your systems so that no system has too much load at a time. In this case you simply define which systems are available for the parallel program and leave it to poe to do the mapping of task numbers to systems in a meaningful way. We call this *automatic allocation*, because poe automatically performs the mapping between task numbers and systems. The list where you specify the systems which poe can use for automatic allocation is called "pool-list". In this release poe supports two modes for automatic allocation. One is the *Workload Manager mode* (WLM-mode) where poe allocates systems out of the pool-list according to the actual capacity information from WLM. The other is called *Round-Robin mode* (RR mode) where poe allocates systems from the pool-list in Round-Robin order.

In some cases it is important that special tasks run on specific systems. For example if the tasks need resources which are not available on all the systems. Think of local file systems, databases etc. ! So you have to tell poe that a specific task number has to be mapped to a specific system. This allocation step is called *specific allocation*, because you tell poe explicitly where to run some tasks. The list where the mapping between systems and task numbers is specified is called "pin-list".

It is possible to combine both pin- and pool-lists, if only some tasks must run on specific nodes and it does not matter where exactly the other tasks of the parallel job run.

**To summarize:**

POE uses two lists for task allocation:

- The pin-list is used for specific allocation
- The pool-list is used for automatic allocation

Now you know how poe does the mapping of task numbers to systems. In the following we describe how you define the contents of the pin- and the pool-list. There exist three possibilities to specify the two lists. These possibilities can be chosen by the user through specifying MP\_HOSTFILE and MP\_RMPOOL environment variables or the "-hostfile" and "-rmpool" poe commandline flags.

**Possibilities to specify pin- and pool-list:**

1. "Explicitly specify pin- and pool-list with a host list file"
2. "Use empty pin-list and for the pool-list all systems within a Parallel Sysplex" on page 20
3. "Use empty pin-list and for the pool-list all systems within a Parallel Sysplex, where a specific scheduling environment is available" on page 20

**Explicitly specify pin- and pool-list with a host list file:**

Use the following settings to specify a host list file:

- **MP\_HOSTFILE:** path of the host list file or unset if you use "./host.list"
- **MP\_RMPOOL:** unset

You have to define the pin- and pool-list in a host list file which contains pin- and pool entries. It is possible that one of the two lists is empty.

The host list file must contain valid TCP addresses, which must be either symbolic host names (e.g. "myhost") or TCP addresses in dotted decimal format (e.g. "9.167.5.8").

Create a host list file by using any editor. To make a comment, start the line with an exclamation mark or an asterisk. Blank lines and comments are ignored. You can specify pin- and pool entries in the host list file:

**Pin Entries:**

The tasks which are started by poe are numbered beginning with 0. A pin entry is a line in the host list file which consists of a node name and a non-empty list of task numbers. The tasks specified in the list of task numbers will be executed on the system defined by the pin entry. If the list of task numbers contains numbers which exceed the partition size, these task numbers are ignored.

A system which is specified in a pin entry is not used for automatic allocation unless there is an additional pool entry for this system.

It is possible to specify more than one pin entry for a system. For the final pin-list all pin entries in the host list file are accumulated.

It is erroneous to specify a task number more than once in the host list file.

**Pool Entries:**

A pool entry only consists of the node name of a system.

If using Round-Robin mode for automatic allocation, the host list file is processed top down. This means the first task which is not a member of a pin entry will be executed on the system specified by the first pool entry in the host list. The second task will be executed on the system of the second pool-list entry etc. In particular this means that systems for which more than one pool-list entry exist will be chosen more frequently. If the partition size exceeds the number of pool entries, the pool entries in the host list file are processed in a Round-Robin manner.

If using WLM mode for automatic allocation, neither the sequence nor the multiplicity of pool entries influences the allocation of tasks. POE allocates the tasks on nodes defined in the pool-list under consideration of performance data provided by WLM.

**Example for a simple host list file:**

```
! pin entries:
SYS3 0 2
! pool entries:
SYS1
SYS2
```

This host list file pins tasks 0 and 2 to System SYS3 (specific node allocation) and specifies the systems SYS1 and SYS2 for automatic allocation for the remaining tasks.

The default host list file used by the Partition Manager to allocate nodes is called *host.list* and is located in your current directory. You can specify a file other than *host.list* by setting the environment variable **MP\_HOSTFILE** to the name of a host list file, or by using either the **-hostfile** or **-hfile** flag when invoking the program. In either case, you can specify the file using its relative or full path name. For example, say you want to use the host list file *myhosts* located in the directory */u/hinkle*. You could:

Set the MP_HOSTFILE environment variable:	Use the -hostfile flag when invoking the program:
<p>ENTER  <b>export MP_HOSTFILE=/u/hinkle/myhosts</b></p>	<p>ENTER  <b>poe program -hostfile /u/hinkle/myhosts</b>  or  <b>poe program -hfile /u/hinkle/myhosts</b></p>

**Restrictions:**

When using automatic allocation in WLM mode (MP\_RESD=yes), pool entries must be valid TCP host names and MVS system names. For further information see "Host list file" on page 4.

**Use empty pin-list and for the pool-list all systems within a Parallel Sysplex:**

Use the following setting:

- **MP\_HOSTFILE:** "NULL" or "" (empty string)
- **MP\_RMPOOL:** unset

With this setting, poe will use WLM to find out which systems are within the Parallel Sysplex where poe is started. The pin-list will be empty and the pool-list consists of all systems within this Parallel Sysplex.

If you want to use all systems within the Sysplex for the pool-list, you must set **MP\_HOSTFILE** to an empty string or to the word "NULL". Otherwise the Partition Manager will look for a host list file. You can either:

Set the MP_HOSTFILE environment variable:	Use the -hostfile flag when invoking the program:
<p><b>ENTER</b></p> <pre>export MP_HOSTFILE=</pre> <p>or</p> <pre>export MP_HOSTFILE=""</pre> <p>or</p> <pre>export MP_HOSTFILE=NULL</pre>	<p><b>ENTER</b></p> <pre>poe program -hostfile ""</pre> <p>or</p> <pre>poe program -hostfile NULL</pre>

**Restrictions:**

For this setting it is necessary that you have a Sysplex which is running in goal mode and the MVS system names of all nodes in the Sysplex are valid TCP/IP host names of the respective systems or valid aliases. For further information on the necessary setup, see "Host list file" on page 4.

**Use empty pin-list and for the pool-list all systems within a Parallel Sysplex, where a specific scheduling environment is available:**

Use the following setting:

- **MP\_HOSTFILE:** "NULL" or "" (empty string)
- **MP\_RMPOOL:** environment\_name

MP\_RMPOOL specifies the scheduling environment to be used for automatic node allocation. With the above setting, the pin-list will be empty and the pool-list consists of all systems within the Parallel Sysplex which currently have the resources in the correct states for the specified scheduling environment.

You can either:

Set the MP_HOSTFILE and MP_RMPOOL environment variables:	Use the -hostfile and -rmpool flags when invoking the program:
<p>ENTER</p> <pre>export MP_HOSTFILE=NULL</pre> <p>and</p> <pre>export MP_RMPOOL=<i>environment_name</i></pre>	<p>ENTER</p> <pre><i>poe program -hostfile NULL -rmpool environment_name</i></pre>

### Restrictions:

For this setting it is necessary that you have a Sysplex which is running in goal mode and the MVS system names of all nodes in the Sysplex are valid TCP/IP host names of the respective systems or valid aliases. For information on the necessary setup, see "Scheduling environments" on page 5.

### Defining pin- and pool-list: Summary:

Value of MP_HOSTFILE	Value of MP_RMPOOL	Used pin- and pool-lists
Path of host list file or unset (default is <code>./host.list</code> )	Setting of MP_RMPOOL is ignored.	Pin-list and pool-list specified in the host list file.
"NULL" or ""	Unset (default)	Pool-list consists of all nodes within the Sysplex, the pin-list is empty. Works only if Sysplex is running in goal mode.
"NULL" or ""	Name of an existing scheduling environment	Pool-list consists of all nodes within the Sysplex, where the scheduling environment specified with MP_RMPOOL is defined. The pin-list is empty. Works only if Sysplex is running in goal mode.

## Defining the automatic allocation mode

### Use the following setting:

- **MP\_RESD:** "yes", "no" or unset

After you specified the pin- and pool-list to poe you have to tell poe what strategy it should use to choose the systems out of the pool-list for automatic allocation. There are two modes for automatic allocation - The Workload Manager mode (WLM mode) and a Round-Robin mode (RR mode). The mode can be controlled with the MP\_RESD option. If MP\_RESD is set to 'yes', the WLM is used to automatically select the nodes for task execution under consideration of the current load on the Sysplex. Otherwise if MP\_RESD is set to 'no', a Round-Robin scheduling for the nodes specified in the pool list is used. The default for MP\_RESD is 'no'.

### Example for RR mode:

- pool-list: SYS1,SYS2
- no pin-list defined

```

task 0 --> SYS1
task 1 --> SYS2
task 2 --> SYS1
task 3 --> SYS2
task 4 --> SYS1
.
.
.

```

You can either:

Set the MP_RESD environment variable:	Use the <code>-resd</code> flag when invoking the program:
<p>ENTER            <code>export MP_RESD=yes</code></p> <p>or</p> <p>ENTER            <code>export MP_RESD=no</code></p>	<p>ENTER            <code>poe program -resd yes</code></p> <p>or</p> <p>ENTER            <code>poe program -resd no</code></p>

### Restrictions:

Automatic allocation in WLM mode is only possible if WLM is running in goal mode and the MVS system names of all affected nodes in the Sysplex are valid TCP/IP host names of the respective systems or valid aliases. For further information on the necessary setup, see "Host list file" on page 4.

### Defining the partition size

Use the following setting:

- **MP\_PROCS**: number of tasks for the parallel poe job

After you decided which pin- and pool-lists and which automatic allocation mode poe should use, you need to set the partition size by defining the number of parallel processes to use. To do this, use the **MP\_PROCS** environment variable or its associated commandline flag **-procs**. The default value is 1. For example you want to specify the number of task processes to 6. You could:

Set the MP_PROCS environment variable:	Use the <code>-procs</code> flag when invoking the program:
<p>ENTER            <code>export MP_PROCS=6</code></p>	<p>ENTER            <code>poe program -procs 6</code></p>

### Examples

The following examples show how the settings of `MP_HOSTFILE`, `MP_RMPOOL`, `MP_RESD` and `MP_PROCS` can be used in combination.

#### Example 1:

Use a host list file with both pin- and pool-list: pin tasks 0 and 2 to system SYS3 (specific node allocation) and specify the systems SYS1 and SYS2 for automatic allocation for the remaining tasks.

./host.list:

SYS3 0 2  
SYS1  
SYS2

Start "myprog" with the following options:

```
> myprog -procs 1          ---> Task 0 is executed on SYS3
> myprog -procs 2 -resd no ---> Task 0 is executed on SYS3,
                               Task 1 is executed on SYS1
> myprog -procs 4 -resd no ---> Tasks 0 and 2 are executed on SYS3,
                               Task 1 is executed on SYS1,
                               Task 3 on SYS2
> myprog -procs 5 -resd no ---> Tasks 0 and 2 are executed on SYS3,
                               Task 1 and 4 are executed on SYS1,
                               Task 3 on SYS2
> myprog -procs 4 -resd yes ---> Tasks 0 and 2 are executed on SYS3,
                               Task 1 and 3 are executed on SYS1/SYS2
                               (nodes are selected by WLM)
```

### Example 2:

Use no host list (and therefore no pin-list). Use all systems in the Sysplex for automatic allocation. Assume that the Sysplex has the following systems:

- SYS1
- SYS2

Start "myprog" with the following options:

```
> myprog -procs 3 -resd no -hostfile NULL --> Task 0 is executed on SYS1
                                               Task 1 is executed on SYS2
                                               Task 2 is executed on SYS1
> myprog -procs 3 -resd yes -hostfile NULL --> Task 0 to 3 are executed on SYS1
                                               and SYS2 (nodes are selected by WLM)
```

### Example 3:

Use no host list (and therefore no pin-list): Use all systems in the Sysplex where the scheduling environment "database" is available for automatic allocation. Assume that the Sysplex has the following systems:

- SYS1 ("database" NOT available)
- SYS2 ("database" available)
- SYS3 ("database" available)

Start "myprog" with the following options:

```
> myprog -procs 3 -hostfile NULL -rmpool database --> Task 0 is executed on SYS2
                                               Task 1 is executed on SYS3
                                               Task 2 is executed on SYS2
```

## Creating an output host list file

Use the following setting:

- **MP\_SAVEHOSTFILE**: file name or unset

When running parallel programs in the PE environment, you can generate an output host list file of the nodes allocated by the Partition Manager. When you have the Partition Manager perform WLM automatic node allocation, this enables

you to learn which systems were allocated. This information is vital if you want to perform some postmortem analysis or file cleanup on those nodes. You can also use the generated host list for a rerun of the parallel job where each task runs on the same system as it did in the first run.

The generated host list will contain the MVS system name assigned to each task, not the TCP/IP host name. Assuming that the DNS is configured to accept MVS system names, the generated host list can be used afterwards to rerun the job with **MP\_RESD=no**.

To generate a host list file, set the **MP\_SAVEHOSTFILE** environment variable to a file name. You can specify this using a relative or full path name. As with most POE environment variables, you can temporarily override the value of **MP\_SAVEHOSTFILE** using its associated command-line flag **-savehostfile**.

Note that the name of your output host list file can be the same as your input host list file. If a file of the same name already exists, it is overwritten by the output host list file.

### Setting the **MP\_WLM\_ENCLAVE** environment variable

Use the following setting:

- **MP\_WLM\_ENCLAVE**: yes, no or unset

This option can be used to tell poe that it should run all tasks of the parallel job within one WLM multi system enclave. If the option is set, WLM considers the parallel job as a single work unit and can make better scheduling decisions. It is also possible to monitor the parallel tasks more easily when using e.g. performance monitor tools which support multi-system enclaves.

To request usage of WLM multi-system enclaves, you could:

Set the <b>MP_WLM_ENCLAVE</b> environment variable:	Use the <b>-wlm_enclave</b> flag when invoking the program:
ENTER <code>export MP_WLM_ENCLAVE=yes</code>	ENTER <code>poe program -wlm_enclave yes</code>

**MP\_WLM\_ENCLAVE** defaults to no.

#### Restrictions:

This option can only be used within a (parallel) Sysplex. If nodes outside the Sysplex are used, these nodes will not be part of the enclave. See chapter 2 "Preparing to use poe, WLM Multi-system Enclaves" for necessary setup!

## Step 4: Start the X-Windows analysis tool

If you wish to use the Program Marker Array to analyze your application, you should start it before invoking the executable. For more information, see "Chapter 6. Monitoring program execution" on page 45.

## Step 5: Invoke the executable

### Notes:

#### In order to perform this step

1. You need to have the same user ID and password on the home node, and each of the processor nodes that will be running your parallel application.
2. You cannot run your application as root.
3. The executable must be available on each node.

The **poe** command enables you to load and execute programs on remote nodes. You can use it to:

- load and execute a program onto all nodes of your partition. For more information, see “Invoking an SPMD program”.
- individually load the nodes of your partition. This capability is intended for MPMD programs. For more information, see “Invoking an MPMD program” on page 26.
- load and execute a series of SPMD or MPMD programs, in individual job steps, on the same partition. For more information, see “Loading a series of programs as job steps” on page 28.
- run non-parallel programs on remote nodes. For more information, see “Invoking a non-parallel program on remote nodes” on page 30.

When you invoke **poe**, the Partition Manager allocates processor nodes for each task and initializes the local environment. It then loads your program, and reproduces your local environment, on each processor node. The Partition Manager also passes the option list to each remote node.

Since the Partition Manager attempts to reproduce your local environment on each remote node, your current directory is important. When you invoke **poe**, the Partition Manager will, immediately before running your executable, change directory to your current working directory on each remote node. If you are in a local directory that does not exist on remote nodes, you will get an error as the Partition Manager attempts to change to that directory on remote nodes.

Before using the **poe** command, you can first specify which programming model you are using by setting the **MP\_PGMMODEL** environment variable to either *spmd* or *mpmd*. As with most POE environment variables, you can temporarily override the value of **MP\_PGMMODEL** using its associated command-line flag **-pgmmodel**. For example, if you want to run an MPMD program, you could:

Set the <b>MP_PGMMODEL</b> environment variable:	Use the <b>-pgmmodel</b> flag when invoking the program:
ENTER <code>export MP_PGMMODEL=mpmd</code>	ENTER <code>poe program -pgmmodel mpmd</code>

**Note:** If you do not set the **MP\_PGMMODEL** environment variable or **-pgmmodel** flag, the default programming model is SPMD.

### Invoking an SPMD program

If you have an SPMD program, you want to load it as a separate task on each node of your partition. To do this, follow the **poe** command with the program name and any options. The options can be program options or any of the POE

command-line flags shown in “Appendix B. POE environment variables and command-line flags” on page 283. You can also invoke a program by entering the program name and any options:

**ENTER**

`poe program [options]`

or

`program [options]`

You can also enter **poe** without a program name:

**ENTER**

`poe [options]`

- Once your partition is established, a prompt appears.

**ENTER**

the name of the program you want to load. You can follow the program name with any program options or a subset of the POE flags.

**Note:** For National Language Support, POE displays messages located in an externalized message catalog. POE checks the **LANG** and **NLSPATH** environment variables, and if either is not set, it will set up the following defaults:

- **LANG=C**
- **NLSPATH=/usr/lib/nls/msg/%L/%N**

For more information about the message catalog, see “National Language Support” on page xiv.

## Invoking an MPMD program

**Note:** You must set the **MP\_PGMMODEL** environment variable or **-pgmmodel** flag to `mpmd` in order to invoke an MPMD program .

With an SPMD application, the name of the same executable is sent to, and runs on, each of the processor nodes of your partition. If you are invoking an MPMD application, you are dealing with more than one program and need to individually load the nodes of your partition.

For example, say that you have two programs – *master* and *workers* – designed to run together and communicate via calls to message passing subroutines. The program *master* is designed to run on one processor node. The *workers* program is designed to run as separate tasks on any number of other nodes. The *master* program will coordinate and synchronize the execution of all the worker tasks. Neither program can run without the other, as *master* only does sends and the *workers* tasks only do receives.

You can establish a partition and load each node individually using:

- standard input (from the keyboard or redirected)
- a POE commands file

**Loading nodes individually from standard input:** To establish a partition and load each node individually using STDIN:

**ENTER**

`poe [options]`

- The Partition Manager allocates the processor nodes of your partition. Once your partition is established, a prompt containing both the logical node identifier 0 and the actual host name it maps to, appears.

**ENTER**

the name of the program you want to load on node 0. You can follow the program name with any program options or a subset of the POE flags.

- A prompt for the next node in the partition displays.

**ENTER**

the name of the program you want to load on each processor node as you are prompted.

- When you have specified the program to run on the last node of your partition, the message “Partition loaded...” displays and execution begins.

For additional illustration, the following shows the command prompts that would appear, as well as the program names you would enter, to load the example *master* and *workers* programs. This example assumes that the **MP\_PROCS** environment variable is set to 5.

```
$ poe -pgmmode1 mpmd
0:host1_name> master [options]
1:host2_name> workers [options]
2:host3_name> workers [options]
3:host4_name> workers [options]
4:host5_name> workers [options]
```

**Loading nodes individually using a POE commands file:** The **MP\_CMDFILE** environment variable, and its associated command-line flag **-cmdfile**, let you specify the name of a POE commands file. You can use such a file when individually loading a partition – thus freeing STDIN. The POE commands file simply lists the individual programs you want to load and run on the nodes of your partition. The programs are loaded in task order. For example, say you have a typical master/workers MPMD program that you want to run as 5 tasks. Your POE commands file would contain:

```
master [options]
workers [options]
workers [options]
workers [options]
workers [options]
```

Once you have created a POE commands file, you can specify it using a relative or full path name on the **MP\_CMDFILE** environment variable or **-cmdfile** flag. For example, if your POE commands file is */u/hinkle/mpmdprog*, you could:

Set the <b>MP_CMDFILE</b> environment variable:	Use the <b>-cmdfile</b> flag on the <b>poe</b> command:
<b>ENTER</b> <b>export MP_CMDFILE=/u/hinkle/mpmdprog</b>	<b>ENTER</b> <b>poe -cmdfile /u/hinkle/mpmdprog</b>

Once you have set the **MP\_CMDFILE** environment variable to the name of the POE commands file, you can individually load the nodes of your partition. To do this:

**ENTER**

**poe [options]**

- The Partition Manager allocates the processor nodes of your partition. The programs listed in your POE commands file are run on the nodes of your partition.

### Loading a series of programs as job steps

By default, the Partition Manager releases your partition when your program completes its run. However, you can set the environment variable **MP\_NEWJOB**, or its associated command-line flag **-newjob**, to specify that the Partition Manager should maintain your partition for multiple job steps.

For example, say you have three separate SPMD programs. The first one sets up a particular computation by adding some files to */tmp* on each of the processor nodes on the partition. The second program does the actual computation. The third program does some postmortem analysis and file cleanup. These three parallel programs must run as job steps on the same processor nodes in order to work correctly. While node allocation using a host list file might work, the requested nodes might not be available when you invoke each program. The better solution is to instruct the Partition Manager to maintain your partition after execution of each program completes. You can then read multiple job steps from:

- standard input
- a POE commands file using the **MP\_CMDFILE** environment variable.

In either case, you must first specify that you want the Partition Manager to maintain your partition for multiple job steps. To do this, you could:

Set the <b>MP_NEWJOB</b> environment variable:	Use the <b>-newjob</b> flag on the <b>poe</b> command:
<b>ENTER</b> <code>export MP_NEWJOB=yes</code>	<b>ENTER</b> <code>poe -newjob yes</code>

#### Note:

1. You can only load a series of programs as job steps using the **poe** command. You cannot do this with either one of the parallel debugger commands: **pdbx** and **pedb**.
2. **poe** is its own shell. If a step fails, any remaining steps continue.

**Reading job steps from standard input:** Say that you want to run three programs – *setup*, *computation*, and *cleanup* – as job steps on the same partition. Assuming STDIN is keyboard entry, and **MP\_NEWJOB** is set to *yes*, you would:

**ENTER**

`poe [poe-options]`

- The Partition Manager allocates the processor nodes of your partition, and prompts you for the program name.

**ENTER**

`setup [program-options]`

- The program *setup* executes on all nodes of your partition. When execution completes, POE prompts you for the next program name.

**ENTER**

`computation [program-options]`

- The program *computation* executes on all nodes of your partition. When execution completes, you are prompted again.

## ENTER

*cleanup* [*program-options*]

- The program *cleanup* executes on all nodes of your partition. When execution completes, you are prompted again.

## ENTER

**quit**

or

<EscChar-d><sup>1</sup>

- The Partition Manager releases the nodes of your partition.

### Notes:

1. You can also run a series of MPMD programs in job-step fashion from STDIN. If **MP\_PGMMODEL** is set to *mpmd*, the Partition Manager will, after each step completes, prompt you to individually reload the partition as described in “Loading nodes individually from standard input” on page 26.
2. When **MP\_NEWJOB** is *yes*, the Partition Manager, by default, looks to STDIN for job steps. However, if the environment variable **MP\_CMDFILE** is set to the name of a POE commands file as described in “Reading job steps from a POE commands file”, the Partition Manager will look to the commands file instead. To ensure that job steps are read from STDIN, check that the **MP\_CMDFILE** environment variable is unspecified.

**Multi-Step STDIN for newjob mode:** POE’s STDIN processing model allows redirected STDIN to be passed to all steps of a newjob sequence, when the redirection is from a file. If redirection is from a pipe, POE does not distribute the input to each step, only to the first step.

**Reading job steps from a POE commands file:** The **MP\_CMDFILE** environment variable and its associated command-line flag **-cmdfile** lets you specify the name of a POE commands file. If **MP\_NEWJOB** is *yes*, you can have the Partition Manager read job steps from a POE commands file. The commands file in this case simply lists the programs you want to run as job steps. For example, say you want to run the three SPMD programs *setup*, *computation*, and *cleanup* as job steps on the same partition. Your POE commands file would contain the following three lines:

```
setup [program-options]  
computation [program-options]  
cleanup [program-options]
```

Program-options represent the actual values you need to specify.

If you are loading a series of MPMD programs, the POE commands file is also responsible for individually loading the partition. For example, say you had three master/worker MPMD job steps that you wanted to run as 4 tasks on the same partition. The following is a representation of what your POE commands file would contain. Options represent the actual values you need to specify.

```
master1 [options]  
workers1 [options]  
workers1 [options]  
workers1 [options]  
master2 [options]  
workers2 [options]
```

---

1. The usage of the short keys depends on the used program. For 3270 always use <EscChar-d> to immediately release nodes and for **rlogin** use <Ctrl-d>.

```
workers2 [options]
workers2 [options]
master3 [options]
workers3 [options]
workers3 [options]
workers3 [options]
```

While you could also redirect STDIN to read job steps from a file, a POE commands file gives you more flexibility by not tying up STDIN. You can specify a POE commands file by using its relative or absolute path name. Say that your POE commands file is called /u/hinkle/jobsteps. To specify that the Partition Manager should read job steps from this file rather than STDIN, you could:

Set the MP_CMDFILE environment variable:	Use the -cmdfile flag on the poe command:
<p>ENTER</p> <pre>export MP_CMDFILE=/u/hinkle/jobsteps</pre>	<p>ENTER</p> <pre>poe -cmdfile /u/hinkle/jobsteps</pre>

Once **MP\_NEWJOB** is set to *yes*, and **MP\_CMDFILE** is set to the name of your POE commands file, you would:

ENTER

```
poe [poe-options]
```

- The Partition Manager allocates the processor nodes of your partition, and reads job steps from your POE commands file. The Partition Manager does not release your partition until it reaches the end of your commands file.

### Invoking a non-parallel program on remote nodes

You can also use POE to run non-parallel programs on the remote nodes of your partition. Any executable (binary file, shell script, UNIX utility) is suitable, and it does not need to have been compiled with **mpcc**, or **mpCC**. For example, if you wanted to check the process status (using the z/OS-command **ps**) for all remote nodes in your partition, you would:

ENTER

```
poe ps -a
```

The process status for each remote node is written to standard out (STDOUT) at your home node. How STDOUT from all the remote nodes is handled at your home node depends on the output mode. See “Managing standard output (STDOUT)” on page 35 for more information.

---

## Controlling program execution

This section describes a number of additional POE environment variables for monitoring and controlling program execution. It describes how to use the:

- **MP\_EUIDEVELOP** environment variable to specify that you want to run your program in message passing develop mode. In this mode, more detailed checking of your program is performed.
- **MP\_NOARGLIST** and **MP\_FENCE** environment variable to make POE ignore arguments.
- **MP\_STDINMODE** and **MP\_HOLD\_STDIN** environment variables to manage standard input.
- **MP\_STDOUTMODE** environment variable to manage standard output.

- **MP\_INFOLEVEL** environment variable to specify the level of messages you want reported to standard error.
- **MP\_PMDLOG** environment variable to generate a diagnostic log on remote nodes.
- **MP\_LABELIO** environment variable to label message output with task identifiers.
- **MP\_REMOTEDIR** specifies the name of the current directory to be used on the remote nodes. By default, the current directory is the current directory on the home node at the time POE is run.

For a complete listing of all POE environment variables, see “Appendix B. POE environment variables and command-line flags” on page 283.

## Specifying develop mode

You can run programs in one of two modes – *develop mode* or *run mode*. In develop mode, intended for developing applications, the Message Passing Interface performs more detailed checking during execution. Because of the additional checking it performs, develop mode can significantly slow program performance. In run mode, intended for completed applications, only minimal checking is done. While run mode is the default, you can use the **MP\_EUIDEVELOP** environment variable to specify message passing develop mode. As with most POE environment variables, **MP\_EUIDEVELOP** has an associated command-line flag **-euidesvelop**. To specify MPI develop mode, you could:

Set the <b>MP_EUIDEVELOP</b> environment variable:	Use the <b>-euidesvelop</b> flag when invoking the program:
ENTER <code>export MP_EUIDEVELOP=yes</code>	ENTER <code>poe program -euidesvelop yes</code>

To later go back to run mode, set **MP\_EUIDEVELOP** to *no*.

You can also use **MP\_EUIDEVELOP** for the **pedb** message queue facility by specifying the *DEB* value, for “debug”.

Set the <b>MP_EUIDEVELOP</b> environment variable:	Use the <b>-euidesvelop</b> flag when invoking the program:
ENTER <code>export MP_EUIDEVELOP=DEB</code>	ENTER <code>poe program -euidesvelop DEB</code>

To stop parameter checking, set **MP\_EUIDEVELOP** to *min*, for “minimum”.

## Making POE ignore arguments

When you invoke a parallel executable, you can specify an argument list that consists of a number of program options and POE command-line flags. The argument list is parsed by POE – the POE command-line flags are removed and the remainder of the list is passed on to the program. If any of your program arguments are identical to POE command-line flags, however, this can cause problems. For example, say you have a program that takes the argument **-pulse**. You invoke the program with the **-pulse** option, but it does not execute correctly. This is because there is also a POE command-line flag **-pulse**. POE parses the argument list and so the **-pulse** option is never passed on to your program. There are two ways to correct this sort of problem. You can:

- make POE ignore the entire argument list using the **MP\_NOARGLIST** environment variable.
- make POE ignore a portion of the argument list using the **MP\_FENCE** environment variable.

### Making POE ignore the entire argument list

When you invoke a parallel executable, POE, by default, parses the argument list and removes all POE command-line flags before passing the rest of the list on to the program. Using the environment variable **MP\_NOARGLIST**, you can prevent POE from parsing the argument list. To do this:

```
ENTER
    export MP_NOARGLIST=yes
```

When the **MP\_NOARGLIST** environment variable is set to *yes*, POE does not examine the argument list at all. It simply passes the entire list on to the program. For this reason, you can not use any POE command-line flags, but must use the POE environment variables exclusively. While most POE environment variables have associated command-line flags, **MP\_NOARGLIST**, for obvious reasons, does not. To specify that POE should again examine argument lists, either set **MP\_NOARGLIST** to *no*, or unset it.

```
ENTER
    export MP_NOARGLIST=no
    or
    unset MP_NOARGLIST
```

### Making POE ignore a portion of the argument list

When you invoke a parallel executable, POE, by default, parses the entire argument list and removes all POE command-line flags before passing the rest of the list on to the program. You can use a fence, however, to prevent POE from parsing the remainder of the argument list. A *fence* is simply a character string you define using the **MP\_FENCE** environment variable. Once defined, you can use the fence to separate those arguments you want parsed by POE from those you do not. For example, say that you have a program that takes the argument **-pulse**. Because there is also a POE command-line flag **-pulse**, you need to put this argument after a fence. To do this, you could:

```
ENTER
    export MP_FENCE=Q
    poe program -procs 26 -infolevel 2 Q -pulse RGB
```

While this example defines *Q* as the fence, keep in mind that the fence can be any character string. Any arguments placed after the fence are passed by POE, unexamined, to the program. While most POE environment variables have associated command-line flags, **MP\_FENCE** does not.

## Managing standard input, output, and error

POE lets you control standard input (STDIN), standard output (STDOUT), and standard error (STDERR) in several ways. You can continue using the traditional I/O manipulation techniques such as redirection and piping, and can also:

- determine whether a single task or all parallel tasks should receive data from STDIN.

- determine whether a single task or all parallel tasks should write to STDOUT. If all tasks are writing to STDOUT, you can further define whether or not the messages are ordered by task id.
- specify the level of messages that will be reported to STDERR during program execution.
- specify that messages to STDOUT and STDERR should be labeled by task id.

### Managing standard input (STDIN)

STDIN is the primary source of data that goes into a command. Usually, STDIN refers to keyboard input. If you use redirection or piping, however, STDIN could refer to a file or the output from another command (see “Using MP\_HOLD\_STDIN”). How you manage STDIN for a parallel application depends on whether or not its parallel tasks require the same input data. Using the environment variable **MP\_STDINMODE** or the command-line flag **-stdinmode**, you can specify that:

- all tasks should receive the same input data from STDIN. This is *multiple input mode*.
- STDIN should be sent to a single task of your partition. This is *single input mode*.
- no task should receive input data from STDIN.

**Multiple input mode:** Setting **MP\_STDINMODE** to *all* indicates that all tasks should receive the same input data from STDIN. The home node Partition Manager sends STDIN to each task as it is read.

To specify multiple input mode so all tasks receive the same input data from STDIN, you could:

Set the MP_STDINMODE environment variable:	Use the -stdinmode flag when invoking the program:
ENTER export MP_STDINMODE=all	ENTER poe program -stdinmode all

**Note:** If you do not set the **MP\_STDINMODE** environment variable or use the **-stdinmode** command-line flag, multiple input mode is the default.

**Single input mode:** There are times when you only want a single task to read from STDIN. To do this, you set **MP\_STDINMODE** to the appropriate task id. For example, say that you have an MPMD application consisting of two programs – *master* and *workers*. The program *master* is designed to run as a single task on one processor node. The *workers* program is designed to run as separate tasks on any number of other nodes. The *master* program handles all I/O, so only its task needs to read STDIN. If *master* is running as task 0, you need to specify that only task 0 should receive STDIN. To do this, you could:

Set the MP_STDINMODE environment variable:	Use the -stdinmode flag when invoking the program:
ENTER export MP_STDINMODE=0	ENTER poe program -stdinmode 0

### Using MP\_HOLD\_STDIN

The environment variable **MP\_HOLD\_STDIN** is used to defer sending of STDIN from the home node to the remote node(s) until the message passing partition has been established. The variable must be set to “yes” when using POE to invoke a

program which: (1) has been compiled with **mpcc** or **mpCC** and (2) will be reading STDIN from other than the keyboard (redirection or piping). Failing to export this environment variable when running these programs could likely result in the user program hanging.

In addition, if a program invoked using POE has not been compiled with **mpcc** or **mpCC**, the environment variable must not be set (or set to "no") to ensure that STDIN is delivered to the remote node(s).

To set **MP\_HOLD\_STDIN** correctly, you need to know the relative order of your program's use of stdin data and initialization of the message passing library.

If **MPI\_Init()** is called before any STDIN data is read, the use of redirected STDIN is explained below. If, however, all STDIN is read before **MPI\_Init()** is called, then **MP\_HOLD\_STDIN** should be set to "no", to allow STDIN data to be sent to the user's executable by POE.

## Using redirected STDIN

**Note:** Wherever the following description refers to a POE environment variable (starting with **MP\_**), the use of the associated command-line option produces the same effect, with the exception of **MP\_HOLD\_STDIN**, which has no associated command-line option.

A POE process can use its STDIN in two ways. First, if the program name is not supplied on the command line and no command file (**MP\_CMDFILE**) is specified, POE uses STDIN to resolve the names of the programs to be run as the remote tasks. Second, any "remaining" STDIN is then distributed to the remote tasks as indicated by the **MP\_STDINMODE** and **MP\_HOLD\_STDIN** settings. In this dual STDIN model, redirected STDIN can then pose two problems:

1. If using job steps (**MP\_NEWJOB=yes**), the "remaining" STDIN is always consumed by the remote tasks during the first job step.
2. If POE attempts program name resolution on the redirected STDIN, program behavior can vary when using job steps, depending on the type of redirection used and the size of the redirected STDIN.

The first problem is addressed in POE by performing a rewind of STDIN between job steps (only if STDIN is redirected from a file, for reasons beyond the scope of this document). The second problem is addressed by providing an additional setting for **MP\_STDINMODE** of "none", which tells POE to only use STDIN for program name resolution. As far as STDIN is concerned, "none" never gets delivered to the remote tasks. This provides an additional method of reliably specifying the program name to POE, by redirecting STDIN from a file or pipe, or by using the shell's here-document syntax in conjunction with the "none" setting. If **MP\_STDINMODE** is not set to "none" when POE attempts program name resolution on redirected STDIN, program behavior is undefined.

The following scenarios describe in more detail the effects of using (or not using) an **MP\_STDINMODE** of "none" when redirecting (or not redirecting) STDIN, as shown in the example:

	Is STDIN Redirected?	
	Yes	No
Is <b>MP_STDINMODE</b> set to "none"?	Yes    A	B
	No     C	D

### Scenario A

POE will use the redirected STDIN for program name resolution, only if no program name is supplied on the command line (**MP\_CMDFILE** is ignored when **MP\_STDINMODE=none**). No STDIN is distributed to the remote tasks. No rewind of STDIN is performed when **MP\_STDINMODE=none**. If **MP\_HOLD\_STDIN** is set to “yes”, this is ignored because no STDIN is being distributed.

### Scenario B

POE will use the keyboard STDIN for program name resolution, only if no program name is supplied on the command line (**MP\_CMDFILE** is ignored when **MP\_STDINMODE=none**). No STDIN is distributed to the remote tasks. No rewind of STDIN is performed when **MP\_STDINMODE=none** (also, STDIN is not from a file). If **MP\_HOLD\_STDIN** is set to “yes”, this is ignored because no STDIN is being distributed.

### Scenario C

POE will use the redirected STDIN for program name resolution, if required, and will distribute “remaining” STDIN to the remote tasks. If STDIN is intended to be used for program name resolution, program behavior is *undefined* in this case, since POE was not informed of this by setting **STDINMODE** to “none” (see Problem 2 above). If STDIN is redirected from a file, POE will rewind STDIN between each job step. If **MP\_HOLD\_STDIN** is set to “yes”, this feature will behave accordingly.

### Scenario D

POE will use the keyboard STDIN for program name resolution, if required. Any “remaining” STDIN is distributed to the remote tasks. No rewind of STDIN is performed since STDIN is not from a file. If **MP\_HOLD\_STDIN** is set to “yes”, it is ignored because STDIN is not redirected.

### Managing standard output (STDOUT)

STDOUT is where the data coming from the command will eventually go. Usually, STDOUT refers to the display. If you use redirection or piping, however, STDOUT could refer to a file or another command. How you manage STDOUT for a parallel application depends on whether you want output data from one task or all tasks. If all tasks are writing to STDOUT, you can also specify whether or not output is ordered by task id. Using the environment variable **MP\_STDOUTMODE**, you can specify that:

- all tasks should write output data to STDOUT asynchronously. This is *unordered output mode*.
- output data from each parallel task should be written to its own buffer, and later all buffers should be flushed, in task order, to STDOUT. This is *ordered output mode*.
- a single task of your partition should write to STDOUT. This is *single output mode*.

**Unordered output mode:** Setting **MP\_STDOUTMODE** to *unordered* specifies that all tasks should write output data to STDOUT asynchronously. To specify unordered output mode, you could:

Set the <b>MP_STDOUTMODE</b> environment variable:	Use the <b>-stdoutmode</b> flag when invoking the program:
ENTER: <code>export MP_STDOUTMODE=unordered</code>	ENTER: <code>poe program -stdoutmode unordered</code>

#### Notes:

1. If you do not set the **MP\_STDOUTMODE** environment variable or use the **-stdoutmode** command-line flag, unordered output mode is the default.

2. If you are using unordered output mode, you will probably want the messages labeled by task id. Otherwise it will be difficult to know which task sent which message. See "Labeling message output" on page 37 for more information.
3. You can also specify unordered output mode from your program by calling the `mpc_stdout_mode` Parallel Utility Function. Refer to *z/OS UNIX System Services Parallel Environment: MPI Programming and Subroutine Reference* for more information.
4. Although the above environment variable and Parallel Utility Function are both described as `MP_STDOUTMODE`, they are each used independently for their specific purposes.

**Ordered output mode:** Setting `MP_STDOUTMODE` to *ordered* specifies ordered output mode. In this mode, each task writes output data to its own buffer. Later, all the task buffers are flushed, in order of task id, to `STDOUT`. The buffers are flushed when:

- any one of the individual task buffers fills
- execution of the program completes.
- all tasks explicitly flush the buffers by calling `mpc_flush` Parallel Utility Function.
- tasks change output mode using calls to Parallel Utility Functions. For more information on Parallel Utility Functions, refer to *z/OS UNIX System Services Parallel Environment: MPI Programming and Subroutine Reference*.

**Note:** When running the parallel application under `pdbx` with `MP_STDOUTMODE` set to *ordered*, there will be a difference in the ordering from when the application is run directly under `poe`. The buffer size available for the application's `STDOUT` is smaller because `pdbx` uses some of the buffer, so the task buffers fill up more often.

To specify ordered output mode, you could:

Set the <code>MP_STDOUTMODE</code> environment variable:	Use the <code>-stdoutmode</code> flag when invoking the program:
ENTER <code>export MP_STDOUTMODE=ordered</code>	ENTER <code>poe program -stdoutmode ordered</code>

**Note:** You can also specify ordered output mode from your program by calling the `mpc_stdout_mode` Parallel Utility Function. Refer to *z/OS UNIX System Services Parallel Environment: MPI Programming and Subroutine Reference* for more information.

**Single output mode:** You can specify that only one task should write its output data to `STDOUT`. To do this, you set `MP_STDOUTMODE` to the appropriate task id. For example, say that you have an SPMD application in which all the parallel tasks are sending the exact same output messages. For easier readability, you would prefer output from only one task – task 0. To specify this, you could:

Set the <code>MP_STDOUTMODE</code> environment variable:	Use the <code>-stdoutmode</code> flag when invoking the program:
ENTER <code>export MP_STDOUTMODE=0</code>	ENTER <code>poe program -stdoutmode 0</code>

**Note:** You can also specify single output mode from your program by calling the `mpc_stdout_mode` Parallel Utility Function. Refer to *z/OS UNIX System Services Parallel Environment: MPI Programming and Subroutine Reference* for more information.

### Labeling message output

You can set the environment variable `MP_LABELIO`, or use the `-labelio` flag when invoking a program, so that output from the parallel tasks of your program are labeled by task id. While not necessary when output is being generated in *single* mode, this ability can be useful in *ordered* and *unordered* modes. For example, say that the output mode is *unordered*. You are executing a program and receiving asynchronous output messages from all the tasks. This output is not labeled, so you do not know which task has sent which message. It would be clearer if the unordered output was labeled. For example:

```
7: Hello World
0: Hello World
3: Hello World
23: Hello World
14: Hello World
9: Hello World
```

To have the messages labeled with the appropriate task id, you could:

Set the <code>MP_LABELIO</code> environment variable:	Use the <code>-labelio</code> flag when invoking the program:
ENTER <code>export MP_LABELIO=yes</code>	ENTER <code>poe program -labelio yes</code>

To no longer have message output labeled, set the `MP_LABELIO` environment variable to *no*.

### Setting the message reporting level for standard error (STDERR)

You can set the environment variable `MP_INFOLEVEL` to specify the level of messages you want from POE. You can set the value of `MP_INFOLEVEL` to one of the integers shown in the following table. The integers *0*, *1*, and *2* give you different levels of informational, warning, and error messages. The integers *3* through *6* indicate debug levels that provide additional debugging and diagnostic information.

Should you require help from the IBM Support Center in resolving a PE-related problem, you will probably be asked to run with one of the debug levels.

As with most POE environment variables, you can override `MP_INFOLEVEL` when you invoke a program. This is done using either the `-infolevel` or `-ilevel` flag followed by the appropriate integer.

This integer:	Indicates this level of message reporting:	In other words:
0	Error	Only error messages from POE are written to STDERR.
1	Normal	Warning and error messages from POE are written to STDERR. This level of message reporting is the default.
2	Verbose	Informational, warning, and error messages from POE are written to STDERR.

This integer:	Indicates this level of message reporting:	In other words:
3	Debug Level 1	Informational, warning, and error messages from POE are written to STDERR. Also written is some high-level debugging and diagnostic information.
4	Debug Level 2	Informational, warning, and error messages from POE are written to STDERR. Also written is some high- and low-level debugging and diagnostic information.
5	Debug Level 3	Debug level 2 messages plus some additional loop detail.
6	Debug Level 4	Debug level 3 messages plus other informational error messages for the greatest amount of diagnostic information.

Let us say you want the POE message level set to verbose. The following table shows the two ways to do this. You could:

Set the MP_INFOLEVEL environment variable:	Use the -infolevel flag when invoking the program:
<p>ENTER</p> <pre>export MP_INFOLEVEL=2</pre>	<p>ENTER</p> <pre>poe program -infolevel 2</pre> <p>or</p> <pre>poe program -ilevel 2</pre>

As with most POE command-line flags, the **-infolevel** or **-ilevel** flag temporarily override their associated environment variable.

### Generating a diagnostic log on remote nodes

Using the **MP\_PMDLOG** environment variable, you can also specify that diagnostic messages should be logged to a file in */tmp* on each of the remote nodes of your partition. The log file is named *mplog.pid.n*, where *pid* is the process id of the Partition Manager Daemon, and *n* is the task number. Should you require help from the IBM Support Center in resolving a PE-related problem, you will probably be asked to generate these diagnostic logs.

The ability to generate diagnostic logs on each node is particularly useful for isolating the cause of abnormal termination, especially when the connection between the remote node and the home node Partition Manager has been broken. As with most POE environment variables, you can temporarily override the value of **MP\_PMDLOG** using its associated command-line flag **-pmdlog**. For example, to generate a **pmd** log file, you could:

Set the MP_PMDLOG environment variable:	Use the -pmdlog flag when invoking the program:
<p>ENTER</p> <pre>export MP_PMDLOG=yes</pre>	<p>ENTER</p> <pre>poe program -pmdlog yes</pre>

**Note:** By default, **MP\_PMDLOG** is set to *no*. No diagnostic logs are generated. It is not recommended that you run **MP\_PMDLOG** routinely. Running this will greatly affect performance and fill up your file system space.

### **Changing current directories on remote nodes**

POE allows to change the user's current directory on the remote nodes. By default, the current directory of the home node is used by all tasks. Using the **MP\_REMOTEDIR** environment variable, you can specify the name of the current directory to be used on the remote nodes. For example, if you set **MP\_REMOTEDIR=/tmp**, the current directory on the remote nodes becomes `/tmp`, regardless of what it is on the home node.



---

## Chapter 5. Managing POE jobs

This chapter describes the tasks involved with managing POE jobs:

- Stopping a POE job
- Cancelling and killing a POE job
- Detecting remote node failures
- Using parallel file copy utilities

---

### Stopping a POE job

You can stop (suspend) a POE job by pressing `<EscChar-z>` or by sending POE a SIGTSTP signal. POE stops, and sends a SIGSTOP signal to all the remote tasks, which stops them. To resume the parallel job, issue the `fg` or `bg` command to POE. A SIGCONT signal will be sent to all the remote tasks to resume them.

---

### Cancelling and killing a POE job

You can cancel a POE job by pressing `<EscChar-c>`. This sends POE a SIGINT signal. POE terminates all the remote tasks and exits.

If POE is killed or terminated before the remote nodes are shut down, direct communication with the parallel job will be lost. In this situation, use the `poekill` script as a POE command to terminate the partition. `poekill` kills all instantiations of the program name on a remote node by sending it a SIGTERM signal. See the `poekill` script in `/bin`, and the description of the `poekill` command in “Appendix A. Parallel Environment commands” on page 185.

**Note:** *Do not* kill the `pmds` using the `poekill` command.

---

### Detecting remote node failures

POE and the Partition Manager use a *pulse* detection mechanism to periodically check each remote node to ensure that it is actively communicating with the home node. You specify the time interval (or *pulse* interval), of these checks with the `-pulse` flag or the `MP_PULSE` environment variable. When POE runs a job, POE and the Partition Manager Daemons check at the interval you specify that each node is running. When a node failure is detected, POE terminates the job on all remaining nodes and issues an error message.

The default pulse interval is 600 seconds (10 minutes). You can increase or decrease this value with the `-pulse` flag or the `MP_PULSE` environment variable. To completely disable the pulse function, specify an interval value of 0 (zero).

---

### Asynchronous Interrupts support

#### Using `MP_CSS_INTERRUPT`

The `MP_CSS_INTERRUPT` environment variable may take the value of either `yes` or `no`. By default it is set to `no`. In certain applications, setting this value to `yes` may provide improved performance.

Applications which have the following characteristics may see performance improvements from setting the POE environment variable `MP_CSS_INTERRUPT` to **yes**:

- Applications that use nonblocking send or receive operations for communication.
- Applications that have non-synchronized sets of send or receive pairs. In other words, the send from node0 is issued at a different point in time with respect to the matching receive in node1.
- Applications that do not issue waits for nonblocking send or receive operations immediately after the send or receive, but rather do some computation prior to issuing the waits.

In all of the above cases, the application is taking advantage of the asynchronous nature of the nonblocking communication subroutines. This essentially means that the calls to the nonblocking send or receive routines do not actually ensure the transmission of data from one node to the next, but only post the send or receive and then return immediately back to the user application for continued processing. However, since the communication subsystem executes within the user's process, it must regain control from the application to complete asynchronous requests for communication.

The communication subsystem can regain control from the application in any one of three different methods:

1. Any subsequent calls to the communication subsystem to post send or receive, or to wait on messages.
2. A timer signal is received periodically to allow the communication subsystem to do recovery from transmission errors.
3. If the value of `MP_CSS_INTERRUPT` is set to **yes**, the communication subsystem device driver will send a signal to the user application when data is received or buffer space is available to transmit data.

Method 1 and Method 2 are always enabled. Method 3 is controlled by the POE environment variable `MP_CSS_INTERRUPT`, and is enabled when this variable is set to **yes**.

For those applications that have the characteristics mentioned above, this implies that when using asynchronous communication the completion of the communication must occur through one of these three methods. In the case that `MP_CSS_INTERRUPT` is not enabled, only the first two methods are available to process communication. Depending upon the amount of time between the non-synchronized send or receive pairs, or between the nonblocking send or receive and the corresponding waits, the actual transmission of data may only complete at the matching wait call. If this is the case, it is possible that an application may see a performance degradation due to unnecessary processor stalling waiting for communication.

Finally, it should be noted that there is a cost associated with handling the signals when `MP_CSS_INTERRUPT` is set to **yes**. In some cases, this cost can degrade application performance. Therefore, `MP_CSS_INTERRUPT` should only be used for those applications that require it. `MP_CSS_INTERRUPT=yes` enables UDP to send a SIGIO signal when a message packet is received.

## Support for Performance Improvements

POE provides interfaces to improve interrupt mode latency.

## Interrupt Mode Control

When a node receives a packet and an interrupt is generated, the interrupt handler checks its tables for the process identifier (PID) of the user process and notifies the process. The process signal handler or service thread polls for at least two times the interrupt delay, checking to see if more packets will arrive. Waiting for more packets avoids the cost of incurring an interrupt each time a new packet arrives (interrupt processing is very expensive). However, the more packets that arrive, the more delay time is increased. Therefore, with these functions you can either tune the delay parameter based on your application, and/or dynamically turn interrupts on or off at selected nodes.

For an application with few nodes exchanging small messages, it will help latency if you keep the interrupt delay small. For an application with a large number of nodes, or one which exchanges large messages, keeping the delay parameter large will help the bandwidth. A large delay allows multiple read transmissions to occur in a single read cycle. You should experiment with different values and use the functions described below to achieve desired performance, depending on the communication pattern.

**MP\_INTRDELAY** is the environment variable which allows you to set the delay parameter for how long the signal handler or service thread waits for more data. The delay specified in the environment variable is set during initialization, before running the program. In this way, user programs can tune the delay parameter without having to recompile existing applications. If none is specified, the default value of 35 microseconds is used. The application can tune this parameter based on the communication pattern it has in different parts of the application.

Five application programming interfaces are provided to help you enable or disable interrupts on specific tasks, based on the communication patterns of the tasks. If a task is frequently in the communication library, then the application can turn interrupts off for that particular task for the duration of the program. The application can enable interrupts when the task is not going to be in the communication subsystem often. The enable or disable interfaces override the setting of the **MP\_CSS\_INTERRUPT** environment variable.

The first two functions allow you to query what the current delay parameter is and to set the delay parameter to a new value.

### **int mpc\_queryintrdelay()**

This function returns the current interrupt delay (in microseconds). If none was set by the user, the default is returned.

### **int mpc\_setintrdelay(int val)**

This function sets the delay parameter to the value, in microseconds, specified by "val". The function can be called at multiple places within the program to set the delay parameter to different values during execution.

The following three functions allow you to control dynamically masking interrupts on individual nodes, and query the state of interrupts. In the current system only "all" nodes or "none" can be selected to statically enable or disable running in interrupt mode.

### **int mpc\_queryintr()**

This function returns 0 if the node on which it is executed has interrupts turned off, and it returns 1 otherwise.

### **int mpc\_disableintr()**

This function disables interrupts on the node on which it is executed. Return code = 0, if successful, -1 otherwise.

### **int mpc\_enableintr()**

This function enables interrupts on the node on which it is executed. Return code = 0, if successful, -1 otherwise.

**Note:** The last two of the above functions override the setting of the environment variable **MP\_CSS\_INTERRUPT**. If they are not used properly they can deadlock the application. Please use these functions *only* if you are sure of what you are doing. These functions are useful in reducing latency if the application is doing blocking `recv/wait` and interrupts are otherwise enabled. Interrupts should be turned off before executing blocking communication calls and turned on immediately after those calls.

---

## Parallel file copy utilities

During the course of developing and running parallel applications on numerous nodes, the potential need exists to efficiently copy data and files to and from a number of places. POE provides three utilities for this reason:

1. **mcp** - to copy a single file from the home node to a number of remote nodes. This was discussed briefly in “Step 2: Copy files to individual nodes” on page 16.
2. **mcpscat** - to copy a number of files from task 0 and scatter them in sequence to all tasks, in a Round-Robin order.
3. **mcpgather** - to copy (or gather) a number of files from all tasks back to task 0.

**mcp** is for copying the same file to all tasks. The input file must reside on task 0. You can copy it to a new name on the other tasks, or to a directory. It accepts the source file name and a destination file name or directory, in addition to any POE command-line argument, as input parameters.

**mcpscat** is intended for distributing a number of files in sequence to a series of tasks, one at a time. It will use a Round-Robin ordering to send the files in a one to one correspondence to the tasks. If the number of files exceeds the number of tasks, the remaining files are sent in another round through the tasks.

**mcpgather** is for when you need to copy a number of files from each of the tasks back to a single location, task 0. The files must exist on each task. You can optionally specify to have the task number appended to the file name when it is copied.

Both **mcpscat** and **mcpgather** accept the source file names and a destination directory, in addition to any POE command-line argument, as input parameters. You can specify multiple file names, a directory name (where all files in that directory, not including subdirectories, are copied), or use wildcards to expand into a list of files as the source. Wildcards should be enclosed in double quotes, otherwise they will be expanded locally, which may not produce the intended file name resolution.

These utilities are actually message passing applications provided with POE. Their syntax is described in “Appendix A. Parallel Environment commands” on page 185.

---

## Chapter 6. Monitoring program execution

The Program Marker Array (shown in Figure 2) is an X-Windows run-time monitoring tool. This window consists of a number of small squares - called *lights* - that change color under program control. Each task in a parallel program has its own row of lights, and Parallel Utility Function calls from those tasks can change light colors. The calls can also send strings to the PM Array.

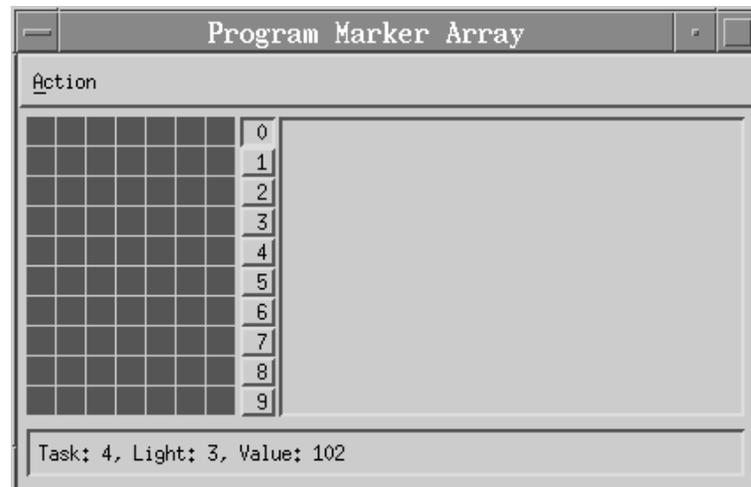


Figure 2. The Program Marker Array

The ability to color lights on, and send strings to, the PM Array window enables a parallel program to provide you with immediate visual feedback as it executes. A program could begin by coloring lights red and then slowly move through the spectrum towards blue as it executes. If a program takes a long time to run, this would give you an indication that it was indeed progressing. Should the program not be progressing, the PM Array would indicate that as well. For example, lights "stuck" on a particular color could indicate that the program is stuck as well. The strings displayed could provide additional information on the program's progress. In addition, the Program Marker Array is distributed as source code, so you can customize the program as you see fit. The source code is located in the directory `/samples/pe/marker`.

In order to use the PM Array to monitor program execution, you need to:

1. In your C, or C++ program, insert subroutine calls to color lights on, and send strings to, the PM Array. This is described in "Step 1: Call PM Array parallel utility functions" on page 46.
2. Compile and link the program using the `mpcc`, or `mpCC` command as described in "Step 1: Compile the program" on page 15. These commands call the C, or C++ compilers while linking in the Partition Manager interface and Parallel Utility subroutines.
3. Make sure that your X-Windows environment is set up properly. This is described in "Step 3: Set up your X-Windows environment" on page 46.
4. Set the environment variable `MP_PMLIGHTS` equal to the number of lights you would like displayed per program task. Alternatively, you can set the

number of lights using a command-line flag when invoking your parallel program. See “Step 4: Set the number of lights” on page 47.

5. Issue the **pmarray** command to start the PM Array program as described in “Step 5: Open the PM Array window” on page 47.
6. Invoke your parallel program and monitor its execution using the PM Array. This is described in “Step 6: Invoke the program and monitor its execution” on page 48.

---

## Step 1: Call PM Array parallel utility functions

In order for the PM Array to display meaningful information at run time, you need to place calls to Parallel Utility Functions within your program. At run time, your program can then:

- color lights on, or send output strings to, the PM Array Window. This is done by calling the **mpc\_marker** Parallel Utility Function.
- determine the number of lights that are displayed per task row. This is done by calling the **mpc\_nlights** Parallel Utility Function. Since the number of lights displayed for each task on the PM Array can vary from run to run, this capability is important. It enables your program to learn exactly how many lights are available to be set. It returns an integer value that can then be used by the program to resolve some conditional expression.

The syntax of these Parallel Utility Functions is shown in *z/OS UNIX System Services Parallel Environment: MPI Programming and Subroutine Reference*.

---

## Step 2: Compile the program

Once you have inserted calls to the **mpc\_marker** and **mpc\_nlights** functions into your program, you can compile it. Since this is the same procedure you follow when regularly compiling a parallel program with POE, see page 15 for more information.

---

## Step 3: Set up your X-Windows environment

If you are already running X-Windows applications, you can probably skip this step. However, you might check that the PM Array X defaults file has been changed by your System Administrator. For details, refer to *z/OS UNIX System Services Planning*.

First you must make sure that an X-server is running on the workstation on which you want to display the PM Array Window. On UNIX workstations the X-server is usually started by default at start time. On Personal Computers you may have to start it yourself (for example, for OS/2 there is an X-server called *PMX Server*). Next, you must give permission to the z/OS system (the X-client machine) to display something on the screen of the workstation. This is typically done on OS/2<sup>®</sup> with the **xhost** command on the workstation:

**ENTER**

```
xhost z/OS-machine-name
```

The *z/OS-machine-name* can be either a host name or an IP address.

To complete the X-Windows setup you must tell pmarray (the X-client) where to display its window. This is done using the DISPLAY environment variable:

```
ENTER
      export DISPLAY workstation-name:0
```

The *workstation-name* can be either a host name or an IP address.

---

## Step 4: Set the number of lights

When you open the PM Array window in the next step, the number of rows in the PM Array are set to the number of program tasks – the current setting of **MP\_PROCS**. You can also specify the number of lights you want displayed per task row. To do this, set the environment variable **MP\_PMLIGHTS** or specify the **-pmlights** command-line flag in “Step 6: Invoke the program and monitor its execution” on page 48.

For example, say that you want five lights displayed per task in the PM Array. You could:

Set the <b>MP_PMLIGHTS</b> environment variable:	Use the <b>-pmlights</b> flag when invoking your executable:
ENTER export MP_PMLIGHTS=5	ENTER poe program -pmlights 5

As with most POE command-line flags, the **-pmlights** flag temporarily overrides its associated environment variable.

### Notes:

1. Setting the **MP\_PMLIGHTS** environment variable, or the **-pmlights** flag, to *0* indicates that you do not want your program to communicate with the PM Array tool.
2. If you reset the **MP\_PMLIGHTS** environment variable, or the **-pmlights** flag, after the Program Marker Array tool is started, it will usually reset to the new number of lights. The only time it will not, however, is when the new value of **MP\_PMLIGHTS** is *0*.

---

## Step 5: Open the PM Array window

The **pmarray** command starts the PM Array program. You will probably want to use the **&** operator so the program runs in the background and does not tie up the shell.

```
ENTER
      pmarray &
```

- The PM Array window opens. The number of task rows displayed in the PM Array is equal to the current setting of **MP\_PROCS**. The number of lights per task row is determined by the current setting of **MP\_PMLIGHTS**. When you invoke your program in Step 6: Invoke the program and monitor its execution, you can override either of the environment variables using its associated command-line flag. Then the PM Array redisplay with the new number of rows or lights.

**Note:** The PM Array connects to the Partition Manager by using a socket that is assigned, by default, to port 9999. If you get an error message indicating

that the port is in use, specify a different port by setting the `MP_USRPORT` environment variable before entering the `pmarray` command. For example, to specify port 9998:

```
ENTER
      export MP_USRPORT=9998
```

---

## Step 6: Invoke the program and monitor its execution

Finally, you invoke your program. As the program runs, the Parallel Utility Function calls that are placed within it change the color of lights on the PM Array. With appropriate mouse clicks on this window, you can:

- display details of a light
- display output strings from a task
- close the window and discontinue monitoring

### Displaying details of a light

Each light on the PM Array is associated with a particular task, has a particular light number, and has a particular color value. You can display these details for each of the lights on the PM Array.

For example, say you have coded the PM Array subroutines into your program so that the lights slowly move during execution through the spectrum over color values 0 to 99. As the program runs, the lights start off black, and then turn brown, green, blue, and so on. By watching the lights as they change color, you get a general idea of the program's progress. For a more precise indication of the program's progress, you could display the actual color value number for a light. In this example, the closer this light's value is to 99, the closer execution is to being complete.

To display details of a light:

#### PLACE

the cursor over any light on the PM Array.

#### PRESS

the left mouse button.

The following information displays in the text area at the bottom of the PM Array window:

- the task identifier number
- the light number
- the color value number

This information is not updated until you select another light.

### Displaying task output

You can display output strings sent by the tasks of your program in the output display area of the PM Array window. This is the area to the right of the PM Array, and the strings displayed there are the ones you specified on the `mpc_marker` subroutine calls. Only one task's strings are displayed in this area at a time. By default, output from task 0 is displayed. You can select the task and display its output instead by pressing its task push button. Each task has a push button. It is just to the right of the task's row on the PM Array, and is labeled with the task identifier. To select, for example, task 3:

**PRESS**

the task push button labeled 3.

- Output strings from task 3 are displayed in the output display area. Only one string is displayed at a time.

**Note:** If a task not currently selected has sent new output to the PM Array window, its task push button will appear yellow.

---

**Step 7: Close the PM Array window**

The PM Array window remains open after your parallel program completes executing. You could then repeat Step 6: Invoke the program and monitor its execution to monitor the same, or a different program's execution. To close the PM Array window when you are done monitoring:

**SELECT**

**Action → Quit**



---

## Chapter 7. Techniques for creating parallel programs

This chapter discusses some of the techniques for creating a parallel program, using message passing, as well as the various advantages and pitfalls associated with each.

This chapter is not intended to be an in-depth tutorial on writing parallel programs. Instead, it's more of an introduction to basic message passing parallel concepts; it provides just enough information to help you understand the material covered in this book. If you want more information about parallel programming concepts, you may find some of the books listed in "Related non-IBM publications" on page 307 helpful.

You should *start with a working sequential program*. Complex sequential programs are difficult enough to get working correctly, without also having to worry about the additional complexity introduced by parallelism and message passing. The bottom line is that it's easier to convert a working serial program to parallel, than it is to create a parallel program from scratch. As you become proficient at creating parallel programs, you'll develop an awareness of which sequential techniques translate better into parallel implementations, and you can then make a point of using these techniques in your sequential programs. In this chapter, you'll find information on some of the fundamentals of creating parallel programs.

There are two common techniques for turning a sequential program into a parallel program; *data decomposition* and *functional decomposition*. Data decomposition has to do with distributing the data that the program is processing among the parallel tasks. Each task does roughly the same thing but on a different set of data. With functional decomposition, the function that the application is performing is distributed among the tasks. Each task operates on the same data, but does something different. Most parallel programs don't use data decomposition or functional decomposition exclusively, but rather a mixture of the two, weighted more toward one type or the other. One way to implement either form of decomposition is through the use of message passing.

---

### Message passing

The message passing model of communication is typically used in distributed memory systems, where each processor node owns private memory, and is linked by an interconnection network. With message passing, each task operates exclusively in a private environment, but must cooperate with other tasks in order to interact. In this situation, tasks must exchange messages in order to interact with one another.

The challenge of the message passing model is in reducing message traffic over the interconnection network while ensuring that the correct and updated values of the passed data are promptly available to the tasks when required. Optimizing message traffic is one way of boosting performance.

*Synchronization* is the act of forcing events to occur at the same time or in a certain order, while taking into account the logical dependence and the order of precedence among the tasks. The message passing model can be described as self-synchronizing because the mechanism of sending and receiving messages

involves implicit synchronization points. To put it another way, a message cannot be received if it has not already been sent.

---

## Data decomposition

A good technique for parallelizing a sequential application is to look for loops where each iteration does not depend on any prior iteration (this is also a prerequisite for either *unrolling* or eliminating loops). An example of a loop that has dependencies on prior iterations is the loop for computing the Factorial series. The value calculated by each iteration depends on the value that results from the previous pass. If each iteration of a loop does not depend on a previous iteration, the data being processed can be processed in parallel, with two or more iterations being performed simultaneously.

The C program example below includes a loop with independent iterations. This example does not include the routines for computing the coefficient and determinant because they are not part of the parallelization at this point.

```
/*
 *
 * Matrix Inversion Program - serial version
 *
 * To compile:
 * c89 -o inverse_serial inverse_serial.c
 *
 *****/

#include<stdlib.h>
#include<stdio.h>
#include<assert.h>
#include<errno.h>

float determinant(float **matrix, int size, int * used_rows, int * used_cols, int depth);
float coefficient(float **matrix,int size, int row, int col);
void print_matrix(FILE * fptr,float ** mat,int rows, int cols);
float test_data[8][8] = {
    {4.0, 2.0, 4.0, 5.0, 4.0, -2.0, 4.0, 5.0},
    {4.0, 2.0, 4.0, 5.0, 3.0, 9.0, 12.0, 1.0 },
    {3.0, 9.0, -13.0, 15.0, 3.0, 9.0, 12.0, 15.0},
    {3.0, 9.0, 12.0, 15.0, 4.0, 2.0, 7.0, 5.0 },
    {2.0, 4.0, -11.0, 10.0, 2.0, 4.0, 11.0, 10.0 },
    {2.0, 4.0, 11.0, 10.0, 3.0, -5.0, 12.0, 15.0 },
    {1.0, -2.0, 4.0, 10.0, 3.0, 9.0, -12.0, 15.0 },
    {1.0, 2.0, 4.0, 10.0, 2.0, -4.0, -11.0, 10.0 },
};
#define ROWS 8

int main(int argc, char **argv)
{
    float **matrix;
    float **inverse;
    int rows,i,j;
    float determ;
    int * used_rows, * used_cols;

    rows = ROWS;

    /* Allocate markers to record rows and columns to be skipped */
    /* during determinant calculation */
    used_rows = (int *) malloc(rows*sizeof(*used_rows));
    used_cols = (int *) malloc(rows*sizeof(*used_cols));

    /* Allocate working copy of matrix and initialize it from static copy */
    matrix = (float **) malloc(rows*sizeof(*matrix));
```

```

inverse = (float **) malloc(rows*sizeof(*inverse));
for(i=0;i<rows;i++)
{
    matrix[i] = (float *) malloc(rows*sizeof(**matrix));
    inverse[i] = (float *) malloc(rows*sizeof(**inverse));
    for(j=0;j<rows;j++)
        matrix[i][j] = test_data[i][j];
}
/* Compute and print determinant */
printf("The determinant of\n\n");
print_matrix(stdout,matrix,rows,rows);
determ=determinant(matrix,rows,used_rows,used_cols,0);
printf("\nis %f\n",determ);
fflush(stdout);
assert(determ!=0);

for(i=0;i<rows;i++)
{
    for(j=0;j<rows;j++)
        inverse[j][i] = coefficient(matrix,rows,i,j)/determ;
}

printf("The inverse is\n\n");
print_matrix(stdout,inverse,rows,rows);

return 0;
}

```

Before we talk about parallelizing the algorithm, let us look at what is necessary to create the program with the PE. The example below shows the same program, but it is now aware of PE. You do this by using three calls in the beginning of the routine, and one at the end.

The first of these calls (**MPI\_Init**) initializes the *MPI* environment and the last call (**MPI\_Finalize**) closes the environment. **MPI\_Comm\_size** sets the variable **tasks** to the total number of parallel tasks running this application, and **MPI\_Comm\_rank** sets **me** to the task ID of the particular instance of the parallel code that invoked it.

**MPI\_Comm\_size** actually gets the size of the communicator you pass in and **MPI\_COMM\_WORLD** is a pre-defined communicator that includes everybody. For more information about these calls, *z/OS UNIX System Services Parallel Environment: MPI Programming and Subroutine Reference* or other *MPI* publications may be of some help. See "Related non-IBM publications" on page 307.

```

/*****
*
* Matrix Inversion Program - serial version enabled for parallel environment
*
* To compile:
* mpcc -o inverse_parallel_enabled inverse_parallel_enabled.c
*
*****/

#include<stdlib.h>
#include<stdio.h>
#include<assert.h>
#include<errno.h>
#include<mpi.h>

float determinant(float **matrix, int size, int * used_rows, int * used_cols, int depth);
float coefficient(float **matrix,int size, int row, int col);
void print_matrix(FILE * fptr,float ** mat,int rows, int cols);
float test_data[8][8] = {
    {4.0, 2.0, 4.0, 5.0, 4.0, -2.0, 4.0, 5.0},

```

```

    {4.0, 2.0, 4.0, 5.0, 3.0, 9.0, 12.0, 1.0 },
    {3.0, 9.0, -13.0, 15.0, 3.0, 9.0, 12.0, 15.0},
    {3.0, 9.0, 12.0, 15.0, 4.0, 2.0, 7.0, 5.0 },
    {2.0, 4.0, -11.0, 10.0, 2.0, 4.0, 11.0, 10.0 },
    {2.0, 4.0, 11.0, 10.0, 3.0, -5.0, 12.0, 15.0 },
    {1.0, -2.0, 4.0, 10.0, 3.0, 9.0, -12.0, 15.0 } ,
    {1.0, 2.0, 4.0, 10.0, 2.0, -4.0, -11.0, 10.0 } ,
};
#define ROWS 8

int me, tasks, tag=0;

int main(int argc, char **argv)
{
    float **matrix;
    float **inverse;
    int rows,i,j;
    float determ;
    int * used_rows, * used_cols;

    MPI_Status status[ROWS]; /* Status of messages */
    MPI_Request req[ROWS]; /* Message IDs */

    MPI_Init(&argc,&argv); /* Initialize MPI */
    MPI_Comm_size(MPI_COMM_WORLD,&tasks); /* How many parallel tasks are there?*/
    MPI_Comm_rank(MPI_COMM_WORLD,&me); /* Who am I? */

    rows = ROWS;

    /* Allocate markers to record rows and columns to be skipped */
    /* during determinant calculation */
    used_rows = (int *) malloc(rows*sizeof(*used_rows));
    used_cols = (int *) malloc(rows*sizeof(*used_cols));

    /* Allocate working copy of matrix and initialize it from static copy */
    matrix = (float **) malloc(rows*sizeof(*matrix));
    inverse = (float **) malloc(rows*sizeof(*inverse));
    for(i=0;i<rows;i++)
    {
        matrix[i] = (float *) malloc(rows*sizeof(**matrix));
        inverse[i] = (float *) malloc(rows*sizeof(**inverse));
        for(j=0;j<rows;j++)
            matrix[i][j] = test_data[i][j];
    }
    /* Compute and print determinant */
    printf("The determinant of\n\n");
    print_matrix(stdout,matrix,rows,rows);
    determ=determinant(matrix,rows,used_rows,used_cols,0);
    printf("\nis %f\n",determ);
    fflush(stdout);
    assert(determ!=0);

    for(i=0;i<rows;i++)
    {
        for(j=0;j<rows;j++)
            inverse[j][i] = coefficient(matrix,rows,i,j)/determ;
    }

    printf("The inverse is\n\n");
    print_matrix(stdout,inverse,rows,rows);

    /* Wait for all parallel tasks to get here, then quit */
    MPI_Barrier(MPI_COMM_WORLD);
    MPI_Finalize();

    return 0;
}

```

```

float determinant(float **matrix,int size, int * used_rows, int * used_cols, int depth)
{
    int col1, col2, row1, row2;
    int j,k;
    float total=0;
    int sign = 1;

    /* Find the first unused row */
    for(row1=0;row1<size;row1++)
    {
        for(k=0;k<depth;k++)
        {
            if(row1==used_rows[k])
                break;
        }
        if(k>=depth) /* this row is not used */
            break;
    }
    assert(row1<size);

    if(depth==(size-2))
    {
        /* There are only 2 unused rows/columns left */
        /* Find the second unused row */
        for(row2=row1+1;row2<size;row2++)
        {
            for(k=0;k<depth;k++)
            {
                if(row2==used_rows[k])
                    break;
            }
            if(k>=depth) /* this row is not used */
                break;
        }
        assert(row2<size);

        /* Find the first unused column */
        for(col1=0;col1<size;col1++)
        {
            for(k=0;k<depth;k++)
            {
                if(col1==used_cols[k])
                    break;
            }
            if(k>=depth) /* this column is not used */
                break;
        }
        assert(col1<size);

        /* Find the second unused column */
        for(col2=col1+1;col2<size;col2++)
        {
            for(k=0;k<depth;k++)
            {
                if(col2==used_cols[k])
                    break;
            }
            if(k>=depth) /* this column is not used */
                break;
        }
        assert(col2<size);

        /* Determinant = m11*m22-m12*m21 */
        return matrix[row1][col1]*matrix[row2][col2]-matrix[row2][col1]*matrix[row1][col2];
    }
    /*There are more than 2 rows/columns in the matrix being processed */
}

```

```

/* Compute the determinant as the sum of the product of each element*/
/* in the first row and the determinant of the matrix with is */
/* and column removed */
total = 0;

used_rows[depth] = row1;
for(coll=0;coll<size;coll++)
{
    for(k=0;k<depth;k++)
    {
        if(coll==used_cols[k])
            break;
    }
    if(k<depth) /* This column is used */
        continue;
    used_cols[depth] = coll;
    total += sign*matrix[row1][coll]*determinant(matrix,size,used_rows,used_cols,depth+1);
    sign=(sign==1)?-1:1;
}
return total;
}

void print_matrix(FILE * fptr,float ** mat,int rows, int cols)
{
    int i,j;
    for(i=0;i<rows;i++)
    {
        for(j=0;j<cols;j++)
            fprintf(fptr,"%10.4f ",mat[i][j]);
        fprintf(fptr,"\n");
    }
    fflush(fptr);
}

float coefficient(float **matrix,int size, int row, int col)
{
    float coef;
    int * ur, *uc;

    ur = malloc(size*sizeof(matrix));
    uc = malloc(size*sizeof(matrix));
    ur[0]=row;
    uc[0]=col;
    coef = (((row+col)%2)?-1:1)*determinant(matrix,size,ur,uc,1);
    return coef;
}

```

This particular example is pretty ridiculous because each parallel task is going to determine the entire inverse matrix, and they're all going to print it out. As we saw in the previous section, the output of all the tasks will be intermixed, so it will be difficult to figure out what the answer really is.

A better approach is to figure out a way to distribute the work among several parallel tasks and collect the results when they are done. In this example, the loop that computes the elements of the inverse matrix simply goes through the elements of the inverse matrix, computes the coefficient, and divides it by the determinant of the matrix. Since there is no relationship between elements of the inverse matrix, they can all be computed in parallel. Keep in mind that every communication call has an associated cost, so you need to balance the benefit of parallelism with the cost of communication. If we were to totally parallelize the inverse matrix element computation, each element would be derived by a separate task. The cost of collecting those individual values back into the inverse matrix would be significant, and might outweigh the benefit of having reduced the computation

cost and time by running the job in parallel. So, instead, we are going to compute the elements of each row in parallel, and send the values back, one row at a time. This way we spread some of the communication overhead over several data values. In our case, we will execute loop 1 in parallel in this next example.

```

/*****
 *
 * Matrix Inversion Program - First parallel implementation
 *
 * To compile:
 * mpcc -o inverse_parallel inverse_parallel.c
 *
 *****/

#include<stdlib.h>
#include<stdio.h>
#include<assert.h>
#include<errno.h>
#include<mpi.h>

float determinant(float **matrix, int size, int * used_rows, int * used_cols, int depth);
float coefficient(float **matrix,int size, int row, int col);
void print_matrix(FILE * fptr,float ** mat,int rows, int cols);
float test_data[8][8] = {
    {4.0, 2.0, 4.0, 5.0, 4.0, -2.0, 4.0, 5.0},
    {4.0, 2.0, 4.0, 5.0, 3.0, 9.0, 12.0, 1.0 },
    {3.0, 9.0, -13.0, 15.0, 3.0, 9.0, 12.0, 15.0},
    {3.0, 9.0, 12.0, 15.0, 4.0, 2.0, 7.0, 5.0 },
    {2.0, 4.0, -11.0, 10.0, 2.0, 4.0, 11.0, 10.0 },
    {2.0, 4.0, 11.0, 10.0, 3.0, -5.0, 12.0, 15.0 },
    {1.0, -2.0, 4.0, 10.0, 3.0, 9.0, -12.0, 15.0 },
    {1.0, 2.0, 4.0, 10.0, 2.0, -4.0, -11.0, 10.0 },
};
#define ROWS 8

int me, tasks, tag=0;

int main(int argc, char **argv)
{
    float **matrix;
    float **inverse;
    int rows,i,j;
    float determ;
    int * used_rows, * used_cols;

    MPI_Status status[ROWS]; /* Status of messages */
    MPI_Request req[ROWS]; /* Message IDs */

    MPI_Init(&argc,&argv); /* Initialize MPI */
    MPI_Comm_size(MPI_COMM_WORLD,&tasks); /* How many parallel tasks are there?*/
    MPI_Comm_rank(MPI_COMM_WORLD,&me); /* Who am I? */

    rows = ROWS;

    /* We need exactly one task for each row of the matrix plus one task */
    /* to act as coordinator. If we don't have this, the last task */
    /* reports the error (so everybody doesn't put out the same message */
    if(tasks!=rows+1)
    {
        if(me==tasks-1)
            fprintf(stderr,"%d tasks required for this demo"
                "(one more than the number of rows in matrix)\n",rows+1);
        exit(-1);
    }

    /* Allocate markers to record rows and columns to be skipped */
    /* during determinant calculation */

```

```

used_rows = (int *) malloc(rows*sizeof(*used_rows));
used_cols = (int *) malloc(rows*sizeof(*used_cols));

/* Allocate working copy of matrix and initialize it from static copy */
matrix = (float **) malloc(rows*sizeof(*matrix));
for(i=0;i<rows;i++)
{
    matrix[i] = (float *) malloc(rows*sizeof(**matrix));
    for(j=0;j<rows;j++)
        matrix [i] [j] = test_data [i] [j] ;
}

/* Everyone computes the determinant (to avoid message transmission) */
determ=determinant(matrix,rows,used_rows,used_cols,0);

if(me==tasks-1)
{ /* The last task acts as coordinator */
    inverse = (float**) malloc(rows*sizeof(*inverse));
    for(i=0;i<rows;i++)
    {
        inverse[i] = (float *) malloc(rows*sizeof(**inverse));
    }
    /* Print the determinant */
    printf("The determinant of\n\n");
    print_matrix(stdout,matrix,rows,rows);
    printf("\nis %f\n",determ);
    /* Collect the rows of the inverse matrix from the other tasks */
    /* First, post a receive from each task into the appropriate row */
    for(i=0;i<rows;i++)
    {
        MPI_Irecv(inverse[i],rows,MPI_REAL,i,tag,MPI_COMM_WORLD,&(req[i]));
    }
    /* Then wait for all the receives to complete */
    MPI_Waitall(rows,req,status);
    printf("The inverse is\n\n");
    print_matrix(stdout,inverse,rows,rows);
}
else
{ /* All the other tasks compute a row of the inverse matrix */
    int dest = tasks-1;
    float *one_row;
    int size = rows*sizeof(*one_row);
    one_row = (float*) malloc(size);
    for(j=0;j<rows;j++)
    {
        one_row[j] = coefficient(matrix,rows,j,me)/determ;
    }
    /* Send the row back to the coordinator */
    MPI_Send(one_row,rows,MPI_REAL,dest,tag,MPI_COMM_WORLD);
}
/* Wait for all parallel tasks to get here, then quit */
MPI_Barrier(MPI_COMM_WORLD);
MPI_Finalize();
return(0);
}

```

## Functional decomposition

Parallel servers and data mining applications are examples of functional decomposition. With functional decomposition, the function that the application is performing is distributed among the tasks. Each task operates on the same data, but does something different. The sine series algorithm is also an example of functional decomposition. With this algorithm, the work being done by each task is trivial. The cost of distributing data to the parallel tasks could outweigh the value of running the program in parallel, and parallelism would increase total time. Another approach to parallelism is to invoke different functions, each of which

processes all of the data simultaneously. This is possible as long as the final or intermediate results of any function are not required by another function. For example, searching a matrix for the largest and smallest values as well as a specific value could be done in parallel.

This is a simple example, but suppose the elements of the matrix were arrays of polynomial coefficients, and the search involved actually evaluating different polynomial equations using the same coefficients. In this case, it would make sense to evaluate each equation separately.

On a simpler scale, let us look at the series for the sine function:

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots - \frac{x^{2n+1}}{(2n+1)!}$$

Figure 3. Formula for sine function

The serial approach to solving this problem is to loop through the number of terms desired, accumulating the factorial value and the sine value. When the appropriate number of terms has been computed, the loop exits. The following example does exactly this. In this example, we have an array of values for which we want the sine, and an outer loop would repeat this process for each element of the array. Since we don't want to recompute the factorial each time, we need to allocate an array to hold the factorial values and compute them outside the main loop.

```

/*****
*
* Series Evaluation - serial version
*
* To compile:
* c89 -o series_serial series_serial.c
*
*****/

#include<stdlib.h>
#include<stdio.h>
#include<math.h>

double angle[] = { 0.0, 0.1*M_PI, 0.2*M_PI, 0.3*M_PI, 0.4*M_PI,
                  0.5*M_PI, 0.6*M_PI, 0.7*M_PI, 0.8*M_PI, 0.9*M_PI, M_PI };

#define TERMS 4

int main(int argc, char **argv)
{
    double divisor[TERMS], sine;
    int a, t, angles = sizeof(angle)/sizeof(angle[0]);

    /* Initialize denominators of series terms */
    divisor[0] = 1;
    for(t=1;t<TERMS;t++)
        divisor[t] = -2*t*(2*t+1)*divisor[t-1];

    /* Compute sine of each angle */
    for(a=0;a<angles;a++)
    {
        sine = 0;
        /* Sum the terms of the series */
        for(t=0;t<TERMS;t++)

```

```

        sine += pow(angle[a],(2*t+1))/divisor[t];
        printf("sin(%lf) + %lf\n",angle[a],sine);
    }
}

```

In a parallel environment, we could assign each term to one task and just accumulate the results on a separate node. In fact, that is what the following example does.

```

/*****
 *
 * Series Evaluation - parallel version
 *
 * To compile:
 * mpcc -o series_parallel serial_parallel.c
 *
 *****/

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <mpi.h>

double angle[] = { 0.0, 0.1*M_PI, 0.2*M_PI, 0.3*M_PI, 0.4*M_PI,
                  0.5*M_PI, 0.6*M_PI, 0.7*M_PI, 0.8*M_PI, 0.9*M_PI, M_PI };

#define TERMS 8

int main(int argc, char **argv)
{
    double data, divisor, partial, sine;
    int a, t, angles = sizeof(angle)/sizeof(angle[0]);
    int me, tasks, term;

    MPI_Init(&argc,&argv);      /* Initialize MPI */
    MPI_Comm_size(MPI_COMM_WORLD,&tasks); /* How many parallel tasks are there? */
    MPI_Comm_rank(MPI_COMM_WORLD,&me);    /* Who am I? */

    term = 2*me+1;             /* Each task computes a term */
    /* Scan the factorial terms through the group members */
    /* Each member will effectively multiply the product of */
    /* the result of all previous members by its factorial */
    /* term, resulting in the factorial up to that point */
    if(me==0)
        data = 1.0;
    else
        data = -(term-1)*term;
    MPI_Scan(&data,&divisor,1,MPI_DOUBLE,MPI_PROD,MPI_COMM_WORLD);

    /* Compute sine of each angle */
    for(a=0;a<angles;a++)
    {
        partial = pow(angle[a],term)/divisor;
        /* Pass all the partials back to task 0 and */
        /* accumulate them with the MPI_SUM operation */
        MPI_Reduce(&partial,&sine,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
        /* The first task has the total value */
        if(me==0)
            printf("sin(%lf) + %lf\n",angle[a],sine);
    }
    MPI_Finalize();
    return(0);
}

```

With this approach, each task *i* uses its position in the `MPI_COMM_WORLD` communicator group to compute the value of one term. It first computes its

working value as  $2i+1$  and calculates the factorial of this value. Since  $(2i+1)!$  is  $(2i-1)! \times 2i \times (2i+1)$ , if each task could get the factorial value computed by the previous task, all it would have to do is multiply it by  $2i \times (2i+1)$ . Fortunately, MPI provides the capability to do this with the **MPI\_Scan** function. When **MPI\_Scan** is invoked on the first task in a communication group, the result is the input data to **MPI\_Scan**. When **MPI\_Scan** is invoked on subsequent members of the group, the result is obtained by invoking a function on the result of the previous member of the group and its input data.

Note that the MPI standard, as documented in *MPI: A Message-Passing Interface Standard, Version 1.1* available from the University of Tennessee, does not specify how the scan function is to be implemented, so a particular implementation does not have to obtain the result from one task and pass it on to the next for processing. This is, however, a convenient way of visualizing the scan function, and the remainder of our discussion will assume this is happening.

In our example, the function invoked is the built-in multiplication function, **MPI\_PROD**. Task 0 (which is computing  $1!$ ) sets its result to 1. Task 2 is computing  $3!$  which it obtains by multiplying  $2 \times 3$  by  $1!$  (the result of Task 0). Task 3 multiplies  $3!$  (the result of Task 2) by  $4 \times 5$  to get  $5!$ . This continues until all the tasks have computed their factorial values. The input data to the **MPI\_Scan** calls is made negative so the signs of the divisors will alternate between plus and minus.

Once the divisor for a term has been computed, the loop through all the angles ( $\theta$ ) can be done. The partial term is computed as:

$$\pm \frac{\theta^n}{n!}$$

Figure 4. Formula for partial term

Then, **MPI\_Reduce** is called which is similar to **MPI\_Scan** except that instead of calling a function on each task, the tasks send their raw data to Task 0, which invokes the function on all data values. The function being invoked in the example is **MPI\_SUM** which just adds the data values from all of the tasks. Then, Task 0 prints out the result.

## Duplication versus redundancy

In the “Matrix Inversion Program” on page 57, each task goes through the process of allocating the matrix and copying the initialization data into it. So why does not one task do this and send the result to all the other tasks? This example has a trivial initialization process, but in a situation where initialization requires complex time-consuming calculations, this question is even more important.

In order to understand the answer to this question and, more importantly, be able to apply the understanding to answering the question for other applications, you need to stop and consider the application as a whole. If one task of a parallel application takes on the role of initializer, two things happen. First, all of the other tasks must wait for the initializer to complete (assuming that no work can be done until initialization is completed). Second, some sort of communication must occur in order to get the results of initialization distributed to all the other tasks. This not only means that there’s nothing for the other tasks to do while one task is doing the initializing, there’s also a cost associated with sending the results out. Although

replicating the initialization process on each of the parallel tasks seems like unnecessary duplication, it allows the tasks to start processing more quickly because they don't have to wait to receive the data.

So, should all initialization be done in parallel? Not necessarily. You have to keep the big picture in mind when deciding. If the initialization is just computation and setup based on input parameters, each parallel task can initialize independently. Although this seems counter-intuitive at first, because the effort is redundant, for the reasons given above, it is the right answer. Eventually you will get used to it. However, if initialization requires access to system resources that are shared by all the parallel tasks (such as file systems and networks), having each task attempt to obtain the resources will create contention in the system and hinder the initialization process. In this case, it makes sense for one task to access the system resources on behalf of the entire application. In fact, if multiple system resources are required, you could have multiple tasks access each of the resources in parallel. Once the data has been obtained from the resource, you need to decide whether to share the raw data among the tasks and have each task process it, or have one task perform the initialization processing and distribute the results to all the other tasks. You can base this decision whether the amount of data increases or decreases during the initialization processing. Of course, you want to transmit the smaller amount.

So, the bottom line is that duplicating the same work on all the remote tasks (which is not the same as redundancy, which implies something can be eliminated) is not bad if:

- The work is inherently serial
- The work is parallel, but the cost of computation is less than the cost of communication
- The work must be completed before tasks can proceed
- Communication can be avoided by having each task perform the same work

---

## Chapter 8. Programming considerations for user applications in POE

This chapter documents various limitations, restrictions, and programming considerations for user applications written under the Parallel Environment (PE).

---

### Environment overview

As the end user, you are encouraged to think of the Parallel Operating Environment (POE) (also referred to as the **poe** command) as an ordinary (serial) command. It accepts redirected I/O, can be run under the **nice** and **time** commands, interprets command flags, and can be invoked in shell scripts.

An n-task parallel job running in the POE actually consists of the n user tasks, an equal number (n) of instances of the Parallel Environment **pmd** daemon (which is the parent process of the user's task), and the POE "home node" process in which the **poe** command runs. A **pmd** daemon is started by the POE home node on each machine on which each user task runs, and serves as the point of contact between the home node and the user's tasks.

The POE home node routes standard input, standard output and standard error streams between the home node and the user's tasks via the **pmd** daemon, using TCP/IP sockets for this purpose. The sockets are created when the POE home node starts the **pmd** daemon for each task of a parallel job. The POE home node and **pmd** also use the sockets to exchange control messages to provide task synchronization, exit status and signaling. These capabilities do not depend on the message passing library and are available to control any parallel program run by the **poe** command.

---

### User authentication

PE requires that the Partition Manager Daemon (PMD) on each remote node authenticate users before beginning any PE processing. The **poe** command that initiates remote processing therefore must obtain the user's password and distribute the password using the TCP connections to each PMD for authentication.

The **poe** command will use the `getpass()` service to prompt interactively for a password. When **poe** is called by a program, the calling program must present the password to POE by using an unnamed pipe. Programs must call **poe** as illustrated in the following example in order to successfully send the user's password to POE.

```

/* Example program demonstrating how programs must call the PE poe command in order
 * to pass the user's password for authentication by PMD on the target execution nodes.
 *
 * The parent process uses fork/exec to execute the poe program. The parent process will
 * establish an unnamed (kernel) pipe.
 *
 * The child process will dup the pipe's fd0 to its stdin. Then the child process exec's
 * the poe program. Now poe can read the password from the pipe.
 */
#define _XOPEN_SOURCE_EXTENDED 1
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <errno.h>

#define read 0
#define write 1
#define STDIN 0
#define STDOUT 1

int main() {
    char *pw = "testpw1"; /* Password */
    char *parms[] = {"parm1", "parm2"}; /* Sample parameters */
    int PWpipe[2]; /* Kernel pipe for parent to write to child */
    int pid; /* Process id */
    int status; /* Child process status */
    if (pipe(PWpipe) < 0 ) { /* Open the pipe for passing the password. */
        fprintf(stderr, "bad pipe: %s\n", strerror(errno));
        exit(1);
    }
    pid = fork(); /* Fork the child process. */
    if (pid) { /* Parent process */
        close(PWpipe[read]); /* Close read descriptor in parent process. */
        dup2(PWpipe[write], STDOUT); /* Dup the pipe's write descriptor to stdout*/
        puts(pw); /* and write the password to the pipe. */
        wait(&status); /* Wait for the child process to complete. */
    }
    else { /* child process */
        close(PWpipe[write]); /* Close write descriptor in child process. */
        dup2(PWpipe[read], STDIN); /* Dup the pipe's read descriptor to stdin */
        execvp("poe", parms); /* and execvp poe. */
    }
}

```

---

## Exit status

Exit status is a value between 0 and 255 inclusive. It is returned from POE on the home node reflecting the composite exit status of your parallel application, as follows:

- If `MPI_Abort(comm,nn>0)` is called, the exit status is `nn (mod 256)`.
- If all tasks terminate via `exit(MM>=0)` and `MM` is not equal to 1 and is `<128` for all nodes, then POE provides a synchronization barrier at the exit. The exit status is the largest value of `MM` from any parallel job (mod 256).
- If any task terminates by `exit(MM =1)`, then POE will immediately terminate the parallel job, as if `MPI_Abort(MPI_COMM_WORLD,1)` had been called.
- If any task terminates via a signal (for example, a segment violation), the exit status is `128+signal` and the entire job is immediately terminated.
- If POE terminates before the start of the user's application, the exit status is 1.
- If the user's application cannot be loaded or fails during MPI initialization (before the user's `main()` is called), the exit status is 255.

- You should explicitly call `exit(MM)` to set the desired exit code. A program exiting without an explicit exit value returns unpredictable status, and may result in causing premature termination of the parallel application.

---

## POE job-step function

The POE job-step function is intended for the execution of a sequence of separate yet inter-related dependent programs. Therefore, it provides you with a job control mechanism that allows both job-step progression and job-step termination. The job control mechanism is the program's exit code.

- Job-step progression:

POE continues the job-step sequence if the task exit code is 0 or in the range of 2 - 127.

- Job-step termination:

POE terminates the parallel job, and does not execute any remaining user programs in the job-step list if the task exit code is 1 or greater than 127.

- Default termination:

Any POE infrastructure detected failure (such as failure to open pipes to the child process or an exec failure to start the user's executable) terminates the parallel job, and does not execute any remaining user programs in the job-step queue.

---

## POE additions to the user executable

POE adds the following routines when your executable is compiled with `mpcc` or `mpCC`.

### Initialization user exits CEEBXITA and CEEBINT

POE provides user exits (CEEBXITA, CEEBINT) which are statically bound with the application executable. They trigger the POE initialization before invoking the application program `main()`. Therefore, the user should not code his/her own CEEBXITA/CEEBINT initialization/termination exit.

POE sets up its environment before the user `main()` program gains control. As a result, any program compiled with the POE compiler scripts must be run under the control of POE and is not suitable as a serial program.

If POE initialization fails, the parallel task is terminated with the exit code that is described above.

### Signal Handlers

POE installs signal handlers for most signals that cause program termination in order to notify the other tasks of termination. POE then causes the program to exit normally with a code of  $(128 + \text{signal})$ .

You can install your own signal handlers by using the `sigaction()` system call. If you use `sigaction()`, you can use either the `sa_handler` member or the `sa_sigaction` member in the `sigaction` structure to define the signal-handling function. If you use the `sa_sigaction` member, the `SA_SIGINFO` flag must be set.

**Note:** In the text that follows, handlers that use `sa_handler` (`SA_SIGINFO` is off) are referred to as *type-1* handlers. Handlers that use `sa_sigaction` (`SA_SIGINFO` is on) are referred to as *type-2* handlers.

POE handles three sets of signals:

1. Asynchronous signals handled by a special POE thread
  - SIGHUP
  - SIGINT
  - SIGQUIT
  - SIGTERM

The POE run-time environment creates a thread to handle these signals using **sigwait**. User can install their own signal handlers, which are called by the thread as **function(signo)** for type-1 handlers and as **function(signo,NULL,NULL)** for type-2 handlers.

If the user program decides to terminate, the user signal handler should call **pm\_child\_sig\_handler(signo,NULL,NULL)**; This allows POE to terminate the program in a defined way with exit code (128 + *signo*). The prototype for `pm_child_sig_handler` is defined in *pm\_util.h*.

2. Signals for which POE sets up signal handlers

For the following signals, POE installs type-2 signal handlers.

- SIGABRT
- SIGBUS
- SIGFPE
- SIGILL
- SIGSEGV
- SIGSYS

Users can install their own signal handlers, but they should save the address of the POE signal handler. If the user program decides to terminate, it should call the POE signal handler as **function(signo,NULL,NULL)**. If the user program decides not to terminate, it should just return to the interrupted code.

3. SIGIO

POE blocks SIGIO before calling your program. If you have registered a signal handler for SIGIO before **MPI\_Init()** is called, the function is added to the interrupt service thread and is executed as **function(SIGIO)** for type-1 handlers and as **function(SIGIO,NULL,NULL)** for type-2 handlers each time the service thread is dispatched. Although it is registered as a signal handler, the function is not required to be “signal safe” because it is executed on a thread. You can use pthread calls to communicate with other threads. MPI functions *cannot* be called in this handler.

After **MPI\_Finalize()** is called, your signal handler is restored, but you need to unblock SIGIO in order to receive subsequent SIGIO signals.

If you register or change the SIGIO signal handler after calling **MPI\_Init()**, your changes are ignored by the MPI library, but they are not undone by **MPI\_Finalize()**.

**Note:** Do not issue message-passing calls from signal handlers. Also, many library calls are not “signal safe”, and should not be issued from signal handlers. See function **sigaction()** in the *z/OS C/C++ Run-Time Library Reference* for a list of functions that signal handlers can call.

---

## Limitations in setting the thread stacksize

The main thread stacksize is the same as the stacksize used for non-threaded applications. If you write your own MPI reduce functions to use with nonblocking collective communications or a SIGIO handler that will be executed on one of the library service threads, you are limited to a stacksize of 64KB by default. To increase your thread stacksize, use the environment variable `MP_THREAD_STACKSIZE`.

---

## Do not hard code file descriptor numbers

Do not use hard coded file descriptor numbers beyond those specified by `STDIN`, `STDOUT` and `STDERR`.

POE opens several files and uses file descriptors as message passing handles. These are allocated before the user gets control, so the first file descriptor allocated to a user is unpredictable.

---

## POE gets control first and handles process initialization

POE sets up its environment via the entry point `mp_main()`. `mp_main()` sets up signal handlers and an `atexit` routine before calling your main program.

Users must not implement their own initialization exits to replace the `pe_CEEBINT.o` and `pe_CEEBXITA.o` modules provided by POE.

---

## Termination of a parallel job

POE provides for orderly termination of a parallel job, so that all tasks terminate at the same time. This is accomplished in the `atexit` routine registered at program initialization. For normal exits (codes 0, 2-127), the `atexit` routine sends a control message to the POE home node, and waits for a positive response. For abnormal exits and those which don't go through the `atexit` routine, the `pmd` daemon catches the exit code and sends a control message to the POE home node.

For normal exits, when POE gets a control message for every task, it responds to each node, allowing that node to exit normally with its individual exit code. The `pmd` daemon monitors the exit code and passes it back to the POE home node for presentation to the user.

For abnormal exits and those detected by `pmd`, POE sends a message to each `pmd` asking that it send a `SIGTERM` signal to its task, thereby terminating the task. When the task finally exits, `pmd` sends its exit code back to the POE home node and exits itself.

User-initiated termination of the POE home node via `SIGINT` (`EscChar-c`) causes a message to be sent to `pmd` asking that the appropriate signal be sent to the parallel task. Again, `pmd` waits for the task to die then terminates itself.

---

## Your program cannot run as root

To prevent uncontrolled root access to the entire parallel job computation resource, POE checks to see that the user is not root as part of its authentication.

---

## Forks are limited

If a task forks, only the thread that forked exists in the child task. Therefore, the message passing library will not operate properly. Also, if the forked child does not exec another program, it should be aware that an atexit routine has been registered for the parent which is also inherited by the child. In most cases, the atexit routine requests that POE terminate the task (parent). A forked child should terminate with an `_exit(0)` system call to prevent the atexit routine from being called. Also, if the forked parent terminates before the child, the child task will not be cleaned up by POE.

A forked child *must not* call the message passing library (the MPI subroutines).

---

## Shell execution

You can have POE run a shell script which is loaded and run on the remote nodes as if it were a binary file.

The program executed by POE on the parallel nodes does not run under a shell on those nodes. Redirection and piping of STDIO applies to the POE home node (`poe` binary), and not the user's code. If shell processing of a command line is desired on the remote nodes, invoke a shell script on the remote nodes to provide the desired preprocessing before the user's application is executed.

---

## Do not rewind stdin, stdout or stderr

The Partition Manager Daemon uses pipes to direct stdin, stdout and stderr to the user's program, therefore, do not rewind these files.

---

## Ensuring that string arguments are passed to your program correctly

Quotation marks, either single or double, used as argument delimiters are stripped away by the shell and are never "seen" by poe. Therefore, the quotation marks must be escaped to allow the quoted string to be passed correctly to the remote task(s) as one argument. For example, if you want to pass the following string to the user program (including the imbedded blank)

```
a b
```

then you need to enter the following:

```
poe user_program \"a b\"
```

`user_program` is passed the following argument as one token:

```
a b
```

Without the backslashes, the string would have been treated as two arguments (*a and b*).

---

## Network tuning considerations

Programs generating large volumes of STDOUT or STDERR may overload the home node. As described previously, standard output and standard error files generated by a user's program are piped to `pmd`, then forwarded to the poe binary via a TCP/IP socket. It is possible to generate so much data that the IP message

buffers on the home node are exhausted, the poe binary hangs and possibly the entire node may hang). Note that the option `-stdoutmode` (environment variable `MP_STDOUTMODE`) controls which output stream is displayed by the poe binary, but does not limit the standard output traffic received from the remote nodes, even if set to display the output of just one node.

The POE environment variable `MP_SNDBUF` can be used to override the default network settings for the size of the TCP buffers used.

If you have large volumes of standard I/O, work with your network administrator to establish appropriate TCP/IP tuning parameters. You may also want to examine if using named pipes is appropriate for your application.

---

## Standard I/O requires special attention

When your program runs on the remote nodes, it has no controlling terminal. `STDIN`, `STDOUT`, and `STDERR` are always piped.

If your MPI program processes `STDIN` from a large file on the home node, you must do one of the following:

- Invoke `MPI_Init()` before performing any `STDIN` processing, or
- Ensure that all `STDIN` has been processed (EOF) before invoking `MPI_Init()`.

This also includes programs which may not explicitly use MPI

If `STDIN` is piped (or redirected) to the `poe` binary (via ordinary pipes), then handle `STDIN` in the following way:

- If all `STDIN` is read by your program before `MPI_Init()` is called, set the environment variable `MP_HOLD_STDIN=NO`.
- If none of `STDIN` is read before `MPI_Init()` is called, set the environment variable `MP_HOLD_STDIN=YES`.
- If none of the above applies, it may not be possible to run your program correctly, you will have to devise some other mechanism for providing data to your program.

## STDIN/STDOUT piping example

The following two scripts show how `STDIN` and `STDOUT` can be piped directly between prior and post processing steps, bypassing the POE home node process. This example assumes that parallel task 0 is known or forced to be on the same node as the POE home node.

The script `compute_home` runs on the home node; the script `compute_parallel` runs on the parallel nodes (those running tasks 0 through `n-1`).

Note that the flag `—promptpw yes` is set. This causes poe to prompt for the password. Leaving the command flag out causes poe to expect the password to be piped.

```
compute_home:
#!/bin/ksh
# Example script compute home runs three processes:
# data generator creates/gets data and writes to stdout
# data processor is a parallel program that reads data
# from stdin, processes it in parallel, and writes
# the results to stdout.
# data_consumer reads data from stdin and summarizes it
#
```

```

set -o verbose
mkfifo poe_in_$$
mkfifo poe_out_$$
export MP_STDOUTMODE=0
export MP_STDINMODE=0
  data_generator >poe_in_$$ | \
  poe_compute_parallel -promptpw yes poe_in_$$ poe_out_$$ data_processor | \
  data_consumer <poe_out_$$
rc=$?
rm poe_in_$$
rm poe_out_$$
exit rc

compute_parallel:
#! /bin/ksh
# Example script compute_parallel is a Shell script that
# takes the following arguments:
# 1) name of input named pipe (stdin)
# 2) name of output named pipe (stdout)
# 3) name of program to be run (and arguments)
#
poe_in=$1
poe_out=$2
shift 2
$* <$poe_in >$poe_out

```

---

## Program and thread termination

MPI\_Finalize terminates the MPI service threads, but does not affect user-created threads. Use pthread\_exit to terminate any user-created threads, and exit(m) to terminate the main program (initial thread). The value of m is used to set POE's exit status as explained on "Exit status" on page 64.

---

## Other thread-specific considerations

### Order requirement for system includes

For threaded programs, z/OS requires that the system include <pthread.h> must be first with <stdio.h> or other system includes following it. <pthread.h> defines some conditional compile variables that modify the code generation of subsequent includes, particularly <stdio.h>. Please note that <pthread.h> is not required unless your file uses thread-related calls or data.

### MPI\_Init

Call MPI\_Init once per task not once per thread. MPI\_Init does not have to be called on the main thread, but MPI\_Init and MPI\_Finalize must be called on the same thread.

MPI calls on other threads must adhere to the MPI standard in regard to the following:

- A thread cannot make MPI calls until MPI\_Init has been called.
- A thread cannot make MPI calls after MPI\_Finalize has been called.
- Unless there is a specific thread protocol programmed, you cannot rely on any specific order or speed of thread processing.

PE MPI does not include the MPI\_Init\_thread subroutine at this time. A call to MPI\_Init with environment variable MP\_SINGLE\_THREAD set to **yes** is equivalent to a call to MPI\_Init\_thread specifying MPI\_THREAD\_FUNNELED. A call with MP\_SINGLE\_THREAD set to **no** is equivalent to using MPI\_THREAD\_MULTIPLE.

The default setting of `MP_SINGLE_THREAD` is **no**, so the default behavior of the library is `MPI_THREAD_MULTIPLE`. MPI-IO and MPI 1-sided communication will not operate if `MP_SINGLE_THREAD` is set to **yes**.

## Collective communications

Collective communications must meet the MPI standard requirement that all participating tasks execute collective communications on any given communicator in the same order. If collective communications calls are made on multiple threads, it is your responsibility to ensure the proper sequencing or to use distinct communicators.

---

## Reserved environment variables

Environment variables starting with `MP_` are intended for use by POE and should be set only as instructed in the documentation. POE also uses a handful of `MP_...` environment variables for internal purposes, which should not be interfered with.

---

## Message catalog considerations

POE assumes that `NLSPATH` contains the appropriate POE message catalogs, even if `LANG` is set to `"C"` or is unset.

---

## MPI-IO Requires Shared HFS To Be Used Effectively

The implementation of MPI-IO depends on all tasks running on a single file system. IBM Shared HFS for z/OS is able to present a single file system to all nodes of a S/390 Parallel Sysplex. Other shared file systems, as NFS for example, do not have the same rigorous management of file consistency when updates occur from more than one node.

MPI-IO can be used with most file systems as long as all tasks are on a single node. This single node approach may be useful in learning to use MPI-IO, but is not likely to be worthwhile in any production context.

Any production use of MPI-IO must be based on Shared HFS or another file system with equivalent functionality and consistency.

**Note:** It is possible to define the set of tasks which are participating in I/O with use of `MP_IONODEFILE`.

---

## Using Shared Memory

MPI programs with more than one task on the same computing node may benefit from using shared memory to send messages between intranode tasks.

This support includes the `MP_SHARED_MEMORY` environment variable. The default setting is **no**. Setting this variable to **yes** directs MPI to use a shared-memory protocol whenever two or more tasks of a job are executing on the same node.

For programs on which *all* tasks are on the same node, shared memory is used exclusively for all MPI communication (if `MP_SHARED_MEMORY=YES`).

The user may want to consider setting `MP_WAIT_MODE=poll` when using the MPI library with shared memory support enabled.

Programs that fail due to a signaled z/OS condition (such as a segmentation violation or user interrupt) may exit without releasing the shared memory segment on the node or nodes on which the tasks were running. Eventually the unreclaimed shared memory space will prevent jobs from being run, so the shared memory space must be reclaimed periodically by the user or by the system administrator. Message FOMO249 is issued if the job exit status is greater than 128 and the shared memory option is enabled:

```
FOM00249 Job job_id completed with exit status status.
        Use the ipcrm command to reclaim shared memory if necessary.
```

The *job\_id* is the key of the shared memory segment, which allows the shared memory segment to be located in the output of the **ipcs** command, and the corresponding segment ID removed by the **ipcrm** command. The following script shows one way this can be done. This script is intended to be run by the user who ran the POE job. The argument is the *job\_id* as reported by message FOMO0249.

```
#!/bin/sh
# Clean up the shared memory left by a failed POE job
# This command should be run on each node where the job had run
# Syntax: cleanup_shm < job_id >
#
# If called with no arguments, list segments owned by user and show syntax:
if [ $# -eq 0 ]
then
    echo "The following shared memory segments are owned by" $LOGNAME
    /bin/ipcs -m | head -3
    /bin/ipcs -m | /bin/grep $LOGNAME
    echo "Syntax: cleanup_shm < shared_memory_key >"
    exit 1
else
# User supplies memory key = POE Job ID
MEM_KEY=$1
set -- ` /bin/ipcs | /bin/grep $MEM_KEY `
if [ -z $2 ]
then
    echo "Shared memory key" $MEM_KEY "not in use"
    exit 0
fi
MEM_ID=$2
echo "Shared Memory Key =" $3
echo "Removing ID" $MEM_ID
/bin/ipcrm -m $MEM_ID
exit 0
fi
```

---

## Chapter 9. Debugging

Before continuing, let us stop and think about the basic process of creating a parallel program. Here are the steps, (which have been greatly abbreviated):

1. Create and compile program
2. Start PE
3. Execute the program
4. Verify the output
5. Optimize the performance.

As with any process, problems can arise in any one of these steps, and different tools are required to identify, analyze and correct the problem. Knowing the right tool to use is the first step in fixing the problem. The remainder of this chapter tells you about some of the common problems you might run into, and what to do when they occur. The sections in this chapter are labeled according to the *symptom* you might be experiencing.

---

### Messages

#### Message catalog errors

Messages are an important part of diagnosing problems, so it's essential that you not only have access to them, but that they are at the correct level. In some cases, you may get message catalog errors. This usually means that the message catalog could not be located or loaded. Check that your **NLSPATH** environment variable includes the path where the message catalog is located. Generally, the message catalog will be in **/usr/lib/nls/msg/C** (English version). The message catalogs in the Japanese version are located in: **/usr/lib/nls/msg/Ja\_JP**. For further information on NLS matters, see "National Language Support" on page xiv.

If the message catalogs are not in the proper place, or your environment variables are not set properly, your system administrator can probably help you. There is really no point in going on until you can read the real error messages!

The following are the Parallel Environment message catalogs :

- mpci\_err.cat
- pepdbx.cat
- pepedb.cat (English version only)
- pempl.cat
- pepoe.cat

**Note:** No pepedb.cat message catalog exists for the Japanese version.

#### Finding messages

There are a number of places that you can find messages:

- They are displayed on the home node when it is running POE (STDERR and STDOUT).
- If you set either the **MP\_PMDLOG** environment variable or the **-pmdlog** command-line option to yes, they are collected in the pmd log file of each task, in **/tmp** (STDERR and STDOUT).

## Logging POE errors to a file

You can also specify that diagnostic messages be logged to a file in **/tmp** on each of the remote nodes of your partition by using the **MP\_PMDLOG** environment variable. The log file is called **/tmp/mplog,pid.n**, where *pid* is the process id of the Partition Manager Daemon (pmd) that was started in response to the **poe** command, and *n* is the task number. This file contains additional diagnostic information about why the user connection was not made. If the file is not there, then pmd did not start. Check the **/etc/inetd.conf** and **/etc/services** entries and the executability of pmd for the root user ID again.

For more information about the **MP\_PMDLOG** environment variable, see “Appendix B. POE environment variables and command-line flags” on page 283. Should you require help from the IBM support Center in resolving a PE problem, you will probably be asked to generate these diagnostic logs.

The ability to generate diagnostic logs on each node is particularly useful for isolating the cause for abnormal termination, especially when the connection between the remote node and the home node Partition Manager is broken.

As with most PE environment variables, you can override **MP\_PMDLOG** with the corresponding command-line flag *-pmdlog*.

## Message format

Knowing which component a message is associated with can be helpful, especially when trying to resolve a problem. As a result, PE messages include prefixes that identify the related component. The message identifiers for the PE components are as follows.

**FOMOnnnn**  
Parallel Operating Environment

**FOMMnnnn**  
Message Passing Interface

**FOMOGnnnn**  
pedb debugger

**FOMOHnnnn**  
pdbx debugger

where:

- The first letters (FOMM, FOMO, FOMOG, FOMOH) identify the component that issued the message.
- *nnnn* identifies the sequence of the message in the group.

For more information about PE messages, see *z/OS UNIX System Services Messages and Codes*.

---

## Cannot compile a parallel program

Programs for the Parallel Environment must be compiled with the current release of **mpcc**. If the command you’re trying to use cannot be found, make sure the installation was successful and that your *PATH* environment variable contains the path to the compiler scripts. This command calls the C compiler, so you also need to make sure that the underlying compiler is installed and accessible. Your System Administrator should be able to assist you in verifying these things.

---

## Cannot start a parallel job

Once your program has been successfully compiled, you either invoke it directly or start the Parallel Operating Environment (POE) and then submit the program to it. In both cases, POE is started to establish communication with the parallel nodes. Problems that can occur at this point include:

- POE does not start
- or
- POE cannot connect to the remote nodes.

These problems can be caused by problems:

- on the home node (where you are trying to submit the job)
- on the remote parallel nodes
- or in the communication subsystem that connects them

You need to make sure that all the things POE expects to be set up really are. Here is what you do:

1. Make sure that you can execute POE. Type:

```
$ poe
```

If the result is `poe: FSUM7351 not found`, you do not have POE in your path. It might mean that POE is not installed, or that your path does not point to it. Check that the file `/bin/poe` exists and is executable, and that your PATH includes `/bin`.

2. Type:

```
$ env | grep MP_
```

Look at the settings of the environment variables beginning with `MP_`, (the POE environment variables). Check their values against what you expect, particularly `MP_HOSTFILE` (where the list of remote host names is to be found), `MP_RESD` (whether the Workload Manager is to be used to allocate remote hosts) and `MP_RMPOOL` (the pool from which Workload Manager is to allocate remote hosts) values. If they are all unset, make sure that you have a file named `host.list` in your current directory. This file must include the names of all the remote parallel hosts that can be used. There must be at least as many hosts available as the number of parallel processes you specified with the `MP_PROCS` environment variable.

3. Type:

```
$ poe -procs 1
```

You should get the prompt message FOMO0503.

If you do, POE has successfully loaded, established communication with the first remote host in your host list file, validated your use of that remote host, and is ready to go to work. If you type any command, for example, `date`, `ls`, or `env`, you should get a response when the command executes on the remote host.

If you get some other set of messages, then the message text should give you some idea of where to look. Some common situations include:

- Cannot connect with the remote host

The path to the remote host is unavailable. Check to make sure that you are trying to connect to the host you think you are. If you are using the Workload Manager to allocate nodes from a pool, you may want to allocate nodes from a known list instead.

Check the `/etc/services` file on your home node, to make sure that the Parallel Environment service is defined. Check the `/etc/services` and `/etc/inetd.conf` files on the remote host to make sure that the PE service is defined, and that the Partition Manager Daemon (**pmd**) program invoked by **inetd** on the remote node is executable.

- User not authorized on remote host

You need an ID on the remote host and your ID and password must be identical on the home node and all remote nodes.

- Other strangeness

On the home node, you can set or increase the **MP\_INFOLEVEL** environment variable (or use the **-infolevel** command-line option) to get more information out of POE while it is running. Although this won't give you any more information about the error, it will give you an idea of where POE was, and what it was trying to do when the error occurred. A value of 6 will give you more information than you could ever want.

---

## Cannot execute a parallel program

Once POE can be started, you'll need to consider the problems that can arise in running a parallel program, specifically initializing the message passing subsystem. The way to eliminate this initialization as the source of POE start-up problems is to run a program that does not use message passing. As discussed in "Running simple commands" on page 7, you can use POE to invoke any command or serial program on remote nodes. If you can get a command or simple program, like *Hello, World!*, to run under POE, but a parallel program does not, you can be pretty sure the problem is in the message passing subsystem. The message passing subsystem is the underlying implementation of the message passing calls used by a parallel program (in other words, an **MPI\_Send**). POE code that is linked into your executable by the **mpcc** or **mpCC** command initializes the message passing subsystem.

Use specific remote hosts in your host list file and do not use the Workload Manager (set **MP\_RESD=no**). If you do not have a small parallel program around, recompile `hello.c` as follows:

```
$ mpcc -o hello_p hello.c
```

and make sure that the executable is loadable on the remote host that you are using.

Type the following command, and then look at the messages on the console:

```
$ poe hello_p -procs 1 -infolevel 4
```

If the last message that you see looks like this:

```
Calling mpci_connect
```

and there are no further messages, there is an error in opening an UDP socket on the remote host. Check to make sure that the IP address of the remote host is correct, as reported in the informational messages printed out by POE, and perform any other IP diagnostic procedures that you know of.

```
If you get
Hello, World!
```

then the communication subsystem has been successfully initialized on the one node, and things ought to be looking good.

---

## The program runs but...

Once you have gotten the parallel application running, it would be nice if you were guaranteed that it would run correctly. Unfortunately, this is not the case. In some cases, you may get no output at all, and your challenge is to figure out why not. In other cases, you may get output that is just not correct and, again, you must figure out why it is not.

## Debugging your parallel program

An important tool in analyzing your parallel program is the PE parallel debugger (**pedb** or **pdbx**). In some situations, using the parallel debugger is no different than using a debugger for a serial program. In others, however, the parallel nature of the problem introduces some subtle and not-so-subtle differences which you should understand in order to use the debugger efficiently. While debugging a serial application, you can focus your attention on the single problem area. In a parallel application, not only must you shift your attention between the various parallel tasks, you must also consider how the interaction among the tasks may be affecting the problem.

### The simplest problem

The simplest parallel program to debug is one where all the problems exist in a single task. In this case, you can unhook all the other tasks from the debugger's control and use the parallel debugger as if it were a serial debugger. However, in addition to being the simplest case, it is also very rare.

### The next simplest problem

The next simplest case is one where all the tasks are doing the same thing and they all experience the problem that is being investigated. In this case, you can apply the same debug commands to all the tasks, advance them in lockstep and interrogate the state of each task before proceeding. In this situation, you need to be sure to avoid debugging-introduced deadlocks. These are situations where the debugger is trying to single-step a task past a blocking communication call, but the debugger has not stepped the sender of the message past the point where the message is sent. In these cases, control will not be returned to the debugger until the message is received, but the message will not be sent until control returns to the debugger. Get the picture?

### OK, the worst problem

The most difficult situation to debug and also the most common is where not all the tasks are doing the same thing and the problem spans two or more tasks. In these situations, you have to be aware of the state of each task, and the interrelations among tasks. You must ensure that blocking communication events either have been or will be satisfied before stepping or continuing through them. This means that the debugger has already executed the send for blocking receives, or the send will occur at the same time (as observed by the debugger) as the receive. Frequently, you may find that tracing back from an error state leads to a message from a task that you were not paying attention to. In these situations, your only choice may be to re-run the application and focus on the events leading up to the send.

## No output at all

### Should there be output?

If you're getting no output from your program and you think you ought to be, the first thing you should do is make sure you have enabled the program to send data back to you. If the `MP_STDOUTMODE` environment variable is set to a number, it is the number of the only task for which standard output will be displayed. If that task does not generate standard output, you will not see any.

### There should be output

If `MP_STDOUTMODE` is set appropriately, the next step is to verify that the program is actually doing something. Start by observing how the program terminates (or fails to). It will do one of the following things:

- Terminate without generating output other than POE messages.
- Fail to terminate after a **really** long time, still without generating output.

In the first case, you should examine any messages you receive (since your program is not generating any output, all of the messages will be coming from POE).

In the second case, you will have to stop the program yourself (<EscChar-c> should work).

One possible reason for lack of output could be that your program is terminating abnormally before it can generate any. POE will report abnormal termination conditions such as being killed, as well as non-zero return codes. Sometimes these messages are obscured in the blur of other errata, so it is important to check the messages carefully.

**Figuring out return codes:** It is important to understand POE's interpretation of return codes. If the exit code for a task is zero (0) or in the range of 2 to 127, then POE will make that task wait until all tasks have exited. If the exit code is 1 or greater than 127 (or less than 0), then POE will terminate the entire parallel job abruptly (with a `SIGTERM` signal to each task). In normal program execution, one would expect to have each program go through `exit(0)` or `STOP`, and exit with an exit code of 0.

However, if a task encounters an error condition (for example, a full file system), then it may exit unexpectedly. In these cases, the exit code is usually set to -1, but if you have written error handlers which produce exit codes other than 1 or -1, then POE's termination algorithm may cause your program to *hang* because one task has terminated abnormally, while the other tasks continue processing (expecting the terminated task to participate).

If the POE messages indicate the job was killed (either because of some external situation like low page space or because of POE's interpretation of the return codes), it may be enough information to fix the problem. Otherwise, more analysis is required.

## It hangs

If you've gotten this far and the POE messages and the additional checking by the message passing routines have been unable to shed any light on why your program is not generating output, the next step is to figure out whether your program is doing anything at all (besides not giving you output).

Let us look at the following example...it has got a bug in it.

```
/******  
*  
* Ray trace program with bug  
*  
* To compile:  
* mpcc -o rtrace_bug rtrace_bug.c  
*  
*  
* Description:  
* This is a sample program that partitions N tasks into  
* two groups, a collect node and N - 1 compute nodes.  
* The responsibility of the collect node is to collect the data  
* generated by the compute nodes. The compute nodes send the  
* results of their work to the collect node for collection.  
*  
* There is a bug in this code. Please do not fix it in this file!  
*  
*****/  
  
#include <mpi.h>  
  
#define PIXEL_WIDTH 50  
#define PIXEL_HEIGHT 50  
  
int First_Line = 0;  
int Last_Line = 0;  
  
int main(int argc, char *argv[])  
{  
    int numtask;  
    int taskid;  
  
    /* Find out number of tasks/nodes. */  
    MPI_Init( &argc, &argv);  
    MPI_Comm_size( MPI_COMM_WORLD, &numtask);  
    MPI_Comm_rank( MPI_COMM_WORLD, &taskid);  
  
    /* Task 0 is the coordinator and collects the processed pixels */  
    /* All the other tasks process the pixels */  
    if ( taskid == 0 )  
        collect_pixels(taskid, numtask);  
    else  
        compute_pixels(taskid, numtask);  
  
    printf("Task %d waiting to complete.\n", taskid);  
    /* Wait for everybody to complete */  
    MPI_Barrier(MPI_COMM_WORLD);  
    printf("Task %d complete.\n",taskid);  
    MPI_Finalize();  
    exit(0);  
}  
  
/* In a real implementation, this routine would process the pixel */  
/* in some manner and send back the processed pixel along with its*/  
/* location. Since we're not processing the pixel, all we do is */  
/* send back the location */  
compute_pixels(int taskid, int numtask)  
{  
    int section;  
    int row, col;  
    int pixel_data[2];  
    MPI_Status stat;  
    printf("Compute #%d: checking in\n", taskid);  
  
    section = PIXEL_HEIGHT / (numtask -1);
```

```

First_Line = (taskid - 1) * section;
Last_Line  = taskid * section;

for (row = First_Line; row < Last_Line; row ++)
    for ( col = 0; col < PIXEL_WIDTH; col ++)
    {
        pixel_data[0] = row;
        pixel_data[1] = col;
        MPI_Send(pixel_data, 2, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }
printf("Compute #%d: done sending. ", taskid);
return;
}

/* This routine collects the pixels. In a real implementation, */
/* after receiving the pixel data, the routine would look at the*/
/* location information that came back with the pixel and move */
/* the pixel into the appropriate place in the working buffer */
/* Since we aren't doing anything with the pixel data, we don't */
/* bother and each message overwrites the previous one      */
collect_pixels(int taskid, int numtask)
{
    int pixel_data [2] ;
    MPI_Status stat;
    int mx = PIXEL_HEIGHT * PIXEL_WIDTH;

    printf("Control #%d: No. of nodes used is %d\n", taskid,numtask);
    printf("Control: expect to receive %d messages\n", mx);

    while (mx > 0)
    {
        MPI_Recv(pixel_data, 2, MPI_INT, MPI_ANY_SOURCE,
                MPI_ANY_TAG, MPI_COMM_WORLD, &stat);
        mx--;
    }
    printf("Control node #%d: done receiving. ",taskid);
    return;
}

```

This example was taken from a ray tracing program that distributed a display buffer out to server nodes. The intent is that each task, other than Task 0, takes an equal number of full rows of the display buffer, processes the pixels in those rows, and then sends the updated pixel values back to the client. In the real application, the task would compute the new pixel value and send it as well, but in this example, we're just sending the row and column of the pixel. Because the client is getting the row and column location of each pixel in the message, it doesn't care which server each pixel comes from. The client is Task 0 and the servers are all the other tasks in the parallel job.

This example has a functional bug in it. With a little bit of analysis, the bug is probably easy to spot and you may be tempted to fix it right away. *Please do not !*

When you run this program, you get the output shown below. Notice that we are using the **-g** option when we compile the example. We are cheating a little bit, because we know that there is going to be a problem, so we are compiling with debug information turned on right away.

```

$ mpcc -g -o rtrace_bug rtrace_bug.c
$ rtrace_bug -procs 4 -labelio yes
1:Compute #1: checking in
0:Control #0: No. of nodes used is 4
1:Compute #1: done sending. Task 1 waiting to complete.
2:Compute #2: checking in
3:Compute #3: checking in

```

```

0:Control: expect to receive 2500 messages
2:Compute #2: done sending. Task 2 waiting to complete.
3:Compute #3: done sending. Task 3 waiting to complete.
^C
ERROR: FOM00250 task 1: Interrupt
ERROR: FOM00250 task 2: Interrupt
ERROR: FOM00250 task 3: Interrupt
ERROR: FOM00250 task 0: Interrupt

```

No matter how long you wait, the program will not terminate until you press `<EscChar-c>`.

So, we suspect that the program is hanging somewhere. We know that it starts executing because we get some messages from it. It could be a logical hang or it could be a communication hang.

## Let us attach the debugger

Let us use the debugger to find out why the program is hanging. The best way to diagnose this problem is to attach the debugger directly to our POE job.

Set z/OS run-time option TEST, start POE and run `rtrace_bug`:

```

$ export _CEE_RUNOPTS='test(all)'
$ rtrace_bug -procs 4 -labelio yes

```

To attach the debugger, we first need to get the process id (pid) of the POE job. You can do this with the z/OS `ps` command:

```

$ ps -ef | grep poe

smith 24152 20728 0 08:25:22 pts/0 0:00 poe

```

Next, we will need to start the debugger in attach mode. Before starting the debugger we have to set the environment variable `_BPX_PTRACE_ATTACH`. Note that we can use either the `pdbx` or the `pedb` debugger. In this next example, we will use `pedb`, which we will start in attach mode by using the `-a` flag and the process identifier (pid) of the POE job:

```

$ export _BPX_PTRACE_ATTACH=yes
$ pedb -a 24152

```

After starting the debugger in attach mode, the `pedb` Attach dialog window appears:

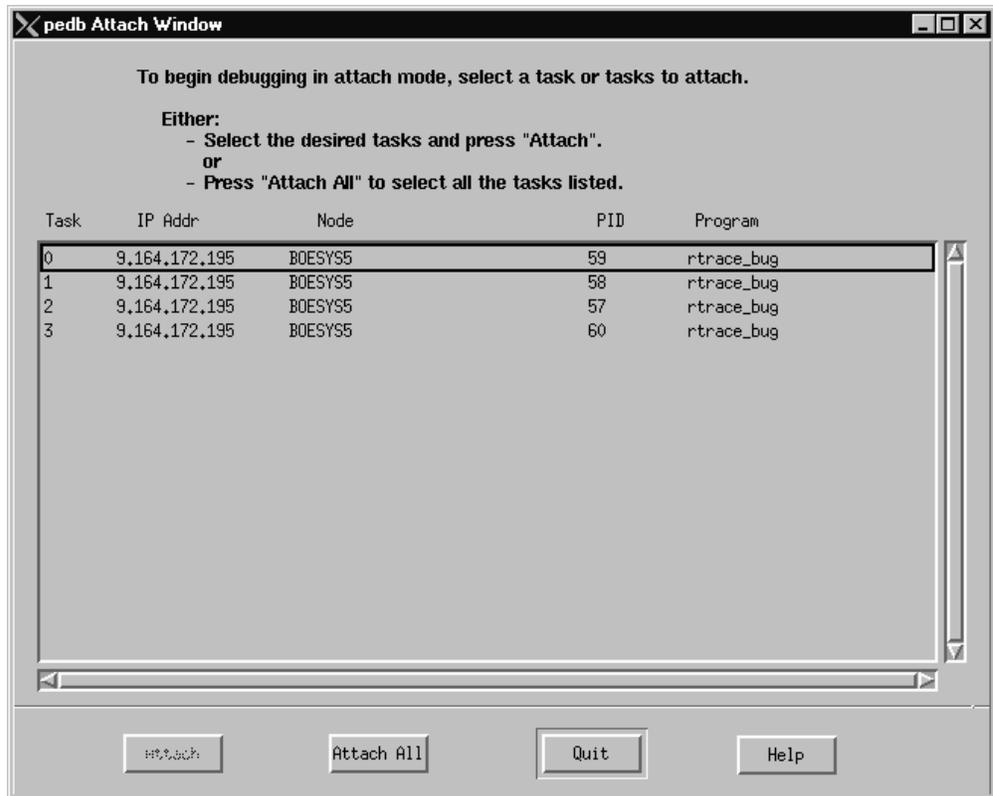


Figure 5. Attach Dialog window

The Attach Dialog window contains a list of task numbers and other information that describes the POE application. It provides information for each task in the following fields:

**Task** The task number

**IP** The IP address of the node on which the task or application is running

**Node** The name, if available, of the node on which the task or application is running

**PID** The process identifier of the selected task

**Program**

The name of the application and arguments, if any. These may be different if your program is MPMD.

At the bottom of the window there are two buttons (other than **Quit** and **Help**):

**Attach** Causes the debugger to attach to the tasks that you selected. This button remains grayed out until you make a selection.

**Attach All**

Causes the debugger to attach to **all** the tasks listed in the window. You do not have to select any specific tasks.

Next, select the tasks to which you want to attach. You can either select all the tasks by pressing the **Attach All** button, or you can select individual tasks, by pressing the **Attach** button. In our example, since we don't know which task or set of tasks is causing the problem, we'll attach to all the tasks by pressing the **Attach All** button.

## PLACE

the mouse cursor on the **Attach All** button.

## PRESS

the left mouse button.

- The Attach Dialog window closes and the debugger main window appears:

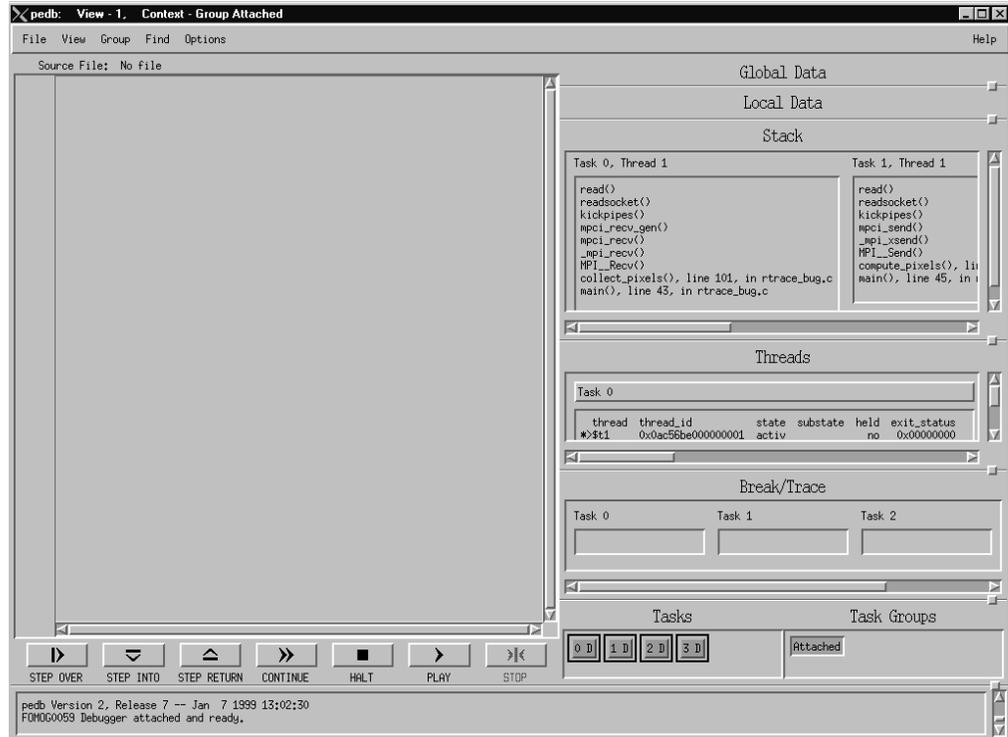


Figure 6. pedb main window

Since our code is hung in low level routines, the initial main window only provides stack traces. To get additional information for a particular task, double click on the highest line in the stack trace that has a line number and a file name associated with it. This indicates that the source code is available.

For task 0, in our example, this line in the stack trace is:

```
collect_pixels(), line 101 in rtrace_bug.c
```

Clicking on this line causes the local data to appear in the *Local Data* area and the source file (the **collect\_pixels** function) to appear in the *Source File* area. In the source for **collect\_pixels**, line 101 is highlighted. Note that the function name and line number **within** the function your program last executed appears here. (in this case, it was function **MPI\_Recv()** on line 101).

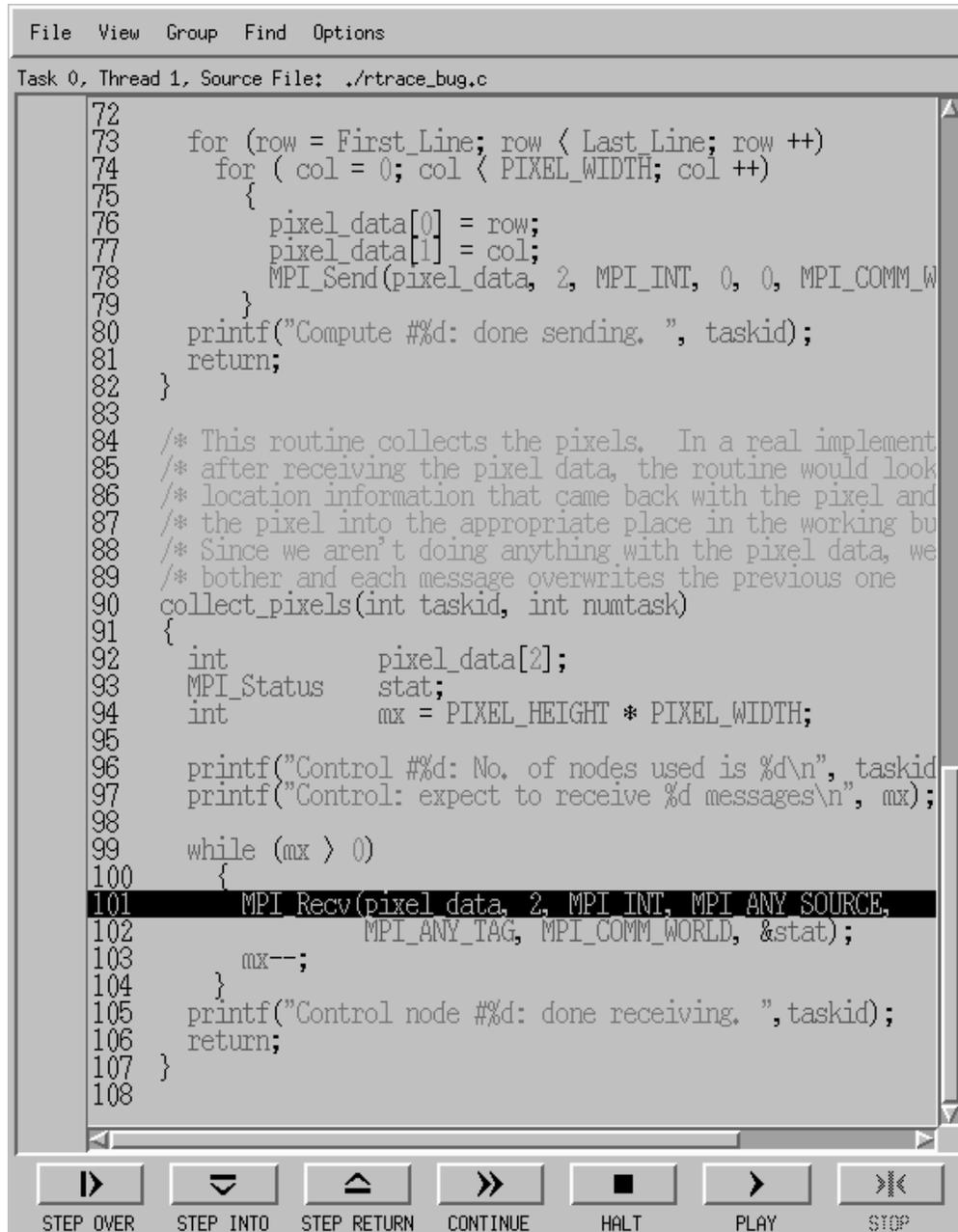


Figure 7. Getting additional information about a task

**PLACE**

the mouse cursor on the Task 0 label (not the box) in the Global Data area.

**PRESS**

- the right mouse button.
- a pop-up menu appears.

**SELECT**

- the Show All option.
- All the global variables for this are tracked and displayed in the window below the task button.

Repeat the steps above for each task.

Now you can see that task 0 is stopped on an **MPI\_Recv()** call. When we look at the Local Data values, we find that **mx** is still set to 100, so task 0 thinks it's still going to receive 100 messages. Now, let us look at what the other tasks are doing.

To get information on task 1, go to its stack window and double click on the highest entry that includes a line number. In our example, this line is:

```
main(argc = 1, argv = 0x2ff22a74), line 43, in rtrace_bug.c
```

Task 1 has reached an **MPI\_Barrier()** call. If we quickly check the other tasks, we see that they have all reached this point as well. So... the problem is solved. Tasks 1 through 3 have completed sending messages, but task 0 is still expecting to receive more. Task 0 was expecting 2500 messages, but only got 2400, so it is still waiting for 100 messages. Let us see how many messages each of the other tasks are sending. To do this, we will look at the global variables **First\_Line** and **Last\_Line**. We can get the values of **First\_Line** and **Last\_Line** for each task by selecting them in the Global Data area.

**PLACE**

the mouse cursor over the desired task number label (not the box) in the Global Data area.

**PRESS**

the right mouse button.

- a pop-up menu appears.

**SELECT**

the Show All option.

- The **First\_Line** and **Last\_Line** variables are tracked and displayed in the window below the task button.

Repeat the steps above for each task.

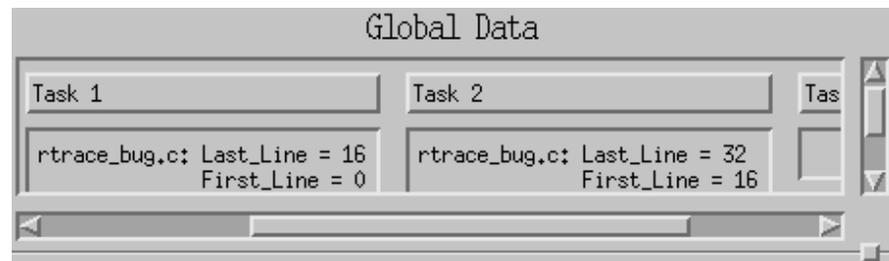


Figure 8. Global Data window

As you can see...

- Task 1 is processing lines 0 through 16
- Task 2 is processing lines 16 through 32
- Task 3 is processing lines 32 through 48.

So what happened to lines 48 and 49? Since each row is 50 pixels wide, and we are missing 2 rows, that explains the 100 missing messages. As you've probably already figured out, the division of the total number of lines by the number of tasks is not integral, so we lose part of the result when it's converted back to an integer. Where each task is supposed to be processing 16 and two-thirds lines, it is only handling 16.

## Fix the problem

So how do we fix this problem permanently? As we mentioned above, there are many ways:

- We could have the last task always go to the last row as we did in the debugger.
- We could have the program refuse to run unless the number of tasks are evenly divisible by the number of pixels (a rather harsh solution).
- We could have tasks process the complete row when they have responsibility for half or more of a row.

In our case, since Task 1 was responsible for 16 and two thirds rows, it would process rows 0 through 16. Task 2 would process 17-33 and Task 3 would process 34-49. The way we are going to solve it is by creating blocks, with as many rows as there are servers. Each server is responsible for one row in each block (the offset of the row in the block is determined by the server's task number). The fixed code is shown in the following example. Note that this is only part of the program.

```
/******  
*  
* Ray trace program with bug corrected  
*  
* To compile:  
* mpcc -o rtrace_good rtrace_good.c  
*  
*  
* Description:  
* This is part of a sample program that partitions N tasks into  
* two groups, a collect node and N - 1 compute nodes.  
* The responsibility of the collect node is to collect the data  
* generated by the compute nodes. The compute nodes send the  
* results of their work to the collect node for collection.  
*  
* The bug in the original code was due to the fact that each processing  
* task determined the rows to cover by dividing the total number of  
* rows by the number of processing tasks. If that division was not  
* integral, the number of pixels processed was less than the number of  
* pixels expected by the collection task and that task waited  
* indefinitely for more input.  
*  
* The solution is to allocate the pixels among the processing tasks  
* in such a manner as to ensure that all pixels are processed.  
*  
*****/  
  
compute_pixels(int taskid, int numtask)  
{  
    int offset;  
    int row, col;  
    int pixel_data[2];  
    MPI_Status stat;  
  
    printf("Compute #%d: checking in\n", taskid);  
  
    First_Line = (taskid - 1);      /* First n-1 rows are assigned */  
    /* to processing tasks      */  
    offset = numtask - 1;          /* Each task skips over rows      */  
    /* processed by other tasks */  
  
    /* Go through entire pixel buffer, jumping ahead by numtask-1 each time */  
    for (row = First_Line; row < PIXEL_HEIGHT; row += offset)  
        for (col = 0; col < PIXEL_WIDTH; col ++)  
        {  
            pixel_data[0] = row;  
            pixel_data[1] = col;  
            MPI_Send(pixel_data, 2, MPI_INT, 0, 0, MPI_COMM_WORLD);  
        }  
}
```

```

    }
    printf("Compute #d: done sending. ", taskid);
    return;
}

```

This program is the same as the original one except for the loop in **compute\_pixels**. Now, each task starts at a row determined by its task number and jumps to the next block on each iteration of the loop. The loop is terminated when the task jumps past the last row (which will be at different points when the number of rows is not evenly divisible by the number of servers.)

### What is the hang-up?

The symptom of the problem in the **rtrace\_bug** program was a hang. Hangs can occur for the same reasons they occur in serial programs (in other words, loops without exit conditions). They may also occur because of message passing deadlocks or because of some subtle differences between the parallel and sequential environments.

However, sometimes analysis under the debugger indicates that the source of a hang is a message that was never received, even though it's a valid one, and even though it appears to have been sent. In these situations, the problem is probably due to lost messages in the communication subsystem. This is especially true if the lost message is intermittent or varies from run to run. This is either the program's fault or the environment's fault. Before investigating the environment, you should analyze the program's *safety* with respect to MPI. A *safe* MPI program is one that does not depend on a particular implementation of MPI.

Although MPI specifies many details about the interface and behavior of communication calls, it also leaves many implementation details unspecified (and it doesn't just omit them, it specifies that they are unspecified.) This means that certain uses of MPI may work correctly in one implementation and fail in another, particularly in the area of how messages are buffered. An application may even work with one set of data and fail with another in the same implementation of MPI. This is because, when the program works, it has stayed within the limits of the implementation. When it fails, it has exceeded the limits. Because the limits are unspecified by MPI, both implementations are valid. MPI *safety* is discussed further in "Appendix D. MPI safety" on page 293.

Once you have verified that the application is *MPI-safe*, your only recourse is to blame lost messages on the environment. Use the standard network analysis tools to diagnose the problem.

Message passing between lots of remote hosts can tax the underlying IP system. Make sure that you look at all the remote nodes, not just the home node.

### Other hang-ups

One final cause for no output is a problem on the home node (POE is hung). Normally, a *hang* is associated with the remote hosts waiting for each other, or for a termination signal. POE running on the home node is alive and well, waiting patiently for some action on the remote hosts. If you type **<EscChar-c>** on the POE console, you will be able to successfully interrupt and terminate the set of remote hosts. See "Appendix A. Parallel Environment commands" on page 185 for information on the **poekill** command.

There are situations where POE itself can hang. Usually these are associated with large volumes of input or output. Remember that POE normally gets standard output from each node; if each task writes a large amount of data to standard output, it may chew up the IP buffers on the machine running POE, causing it (and all the other processes on that machine) to block and hang. The only way to know that this is the problem is by seeing that the rest of the home node has hung. If you think that POE is hung on the home node, your only solution may be to kill POE there. Press <EscChar-c> several times, or use the command **kill -9**.

## Bad output

Bad output includes unexpected error messages. After all, who expects error messages or bad results (results that are not correct).

### Error messages

The causes of error messages are tracked down and corrected in parallel programs using techniques similar to those used for serial programs. One difference, however, is that you need to identify which task is producing the message, if it is not coming from all tasks. You can do this by setting the **MP\_LABELIO** environment variable to **yes**, or using the **-labelio yes** command-line parameter. Generally, the message will give you enough information to identify the location of the problem.

You may also want to generate *more* error and warning messages by setting the **MP\_EUIDEVELOP** environment variable to **yes**. when you first start running a new parallel application. This will give you more information about the things that the message passing library considers errors or unsafe practices.

### Bad results

Bad results are tracked down and corrected in a parallel program in a fashion similar to that used for serial programs. The process, as we saw in the previous debugging exercise, can be more complicated because the processing and control flow on one task may be affected by other tasks. In a serial program, you can follow the exact sequence of instructions that were executed and observe the values of all variables that affect the control flow. However, in a parallel program, both the control flow and the data processing on a task may be affected by messages that are sent from other tasks. For one thing, you may not have been watching those other tasks. For another, the messages could have been sent a long time ago, so it's very difficult to correlate a message that you receive with a particular series of events.

---

## Debugging and Threads

So far, we've talked about debugging normal old serial or parallel programs, but you may want to debug a threaded program (or a program that uses threaded libraries). If this is the case, there are a few things you should consider.

Before you do anything else, you first need to understand the environment you're working in. You have the potential to create a multi-threaded application, using a multi-threaded library, that consists of multiple distributed tasks. As a result, finding and diagnosing bugs in this environment may require a different set of debugging techniques that you're not used to using. Here are some things to remember.

When you attach to a running program, all the tasks you selected in your program will be stopped at their current points of execution. Typically, you want to see the current point of execution of your task. This stop point is the position of the

program counter, and may be in any one of the many threads that your program may create OR any one of the threads that the MPI library creates. With non-threaded programs it was adequate to just travel up the program stack until you reached your application code (assuming you compiled your program with the `-g` option). But with threaded programs, you now need to traverse across other threads to get to your thread(s) and then up the program stack to view the current point of execution of your code.

The MPI library itself will create a set of threads to process message requests. When you attach to a program that uses the MPI library, all of the threads associated with the POE job are stopped, including the ones created and used by MPI.

It's important to note that to effectively debug your application, you must be aware of how threads are dispatched. When a task is stopped, all threads are also stopped. Each time you issue an execution command such as **step over**, **step into**, **step return**, or **continue**, all the threads are released for execution until the next stop (at which time they are stopped, even if they haven't completed their work). This stop may be at a breakpoint you set or the result of a step. A single step over an MPI routine may prevent the MPI library threads from completely processing the message that is being exchanged.

For example, if you wanted to debug the transfer of a message from a send node to a receiver node, you would step over an `MPI_SEND()` in your program on task 1, switch to task 2, then step over the `MPI_RECV()` on task 2. Unless the MPI threads on task 1 and 2 have the opportunity to process the message transfer, it will appear that the message was lost. Remember... the window of opportunity for the MPI threads to process the message is brief, and is only open during the **step over**. Otherwise, the threads will be stopped. Longer-running execution requests, of both the sending and receiving nodes, allow the message to be processed and, eventually, received.

For more information on debugging threaded and non-threaded MPI programs with the PE debugging tools, (**pdbx** and **pedb**), see "Chapter 10. Using the pdbx debugger" on page 93 and "Chapter 11. Using the pedb debugger" on page 127.

For more information on MPI library, see *z/OS UNIX System Services Parallel Environment: MPI Programming and Subroutine Reference*.

---

## Keeping an eye on progress

Often, once a program is running correctly, you would like to monitor its progress. Frequently, in a sequential program, this is done by printing to standard output. However, if you remember from Chapter 3, standard output from all the tasks is interleaved, and it is difficult to follow the progress of just one task. If you set the `MP_STDOUTMODE` environment variable to **ordered**, you can not see how the progress of one task relates to another. In addition, normal output is not a blocking operation. This means that a task that writes a message will continue processing so that by the time you see the message, the task is well beyond that point. This makes it difficult to understand the true state of the parallel application, and it is especially difficult to correlate the states of two tasks from their progress messages. One way to synchronize the state of a parallel task with its output messages is to use the *Program Marker Array* (**pmarray**).

The *Program Marker Array* consists of two components: the display function, **pmarray**, and the instrumentation call, **mpc\_marker**. When **pmarray** is running, it shows a display that looks like Figure 9, below.

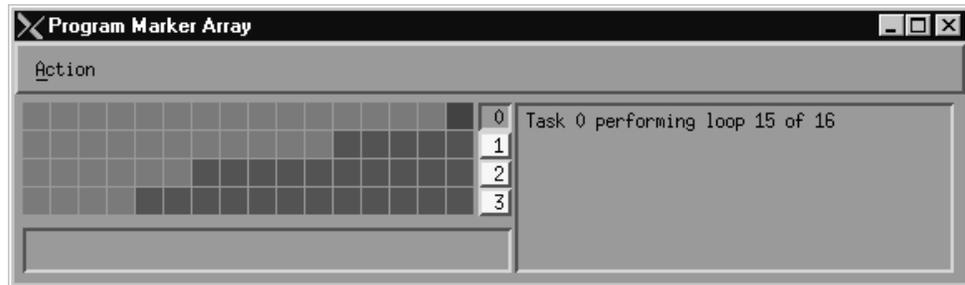


Figure 9. Program Marker Array

Each row of colored squares is associated with one task, which can change the color of any of the lights in its row with the **mpc\_marker** call. The declaration looks like this:

```
void mpc_marker(int light, int color, char *str)
```

This call accepts two integer values and a character string. The first parameter, **light**, controls which light in the **pmarray** is being modified. You can have up to 100 lights for each task. The second parameter, **color**, specifies the color to which you are setting the light. There are 100 colors available. The third parameter is a string of up to 80 characters that is a message shown in the text area of the **pmarray** display.

Before you start the parallel application, you need to tell **pmarray** how many lights to use, as well as how many tasks there will be. You do this with the **MP\_PMLIGHTS** and the **MP\_PROCS** environment variables.

```
$ export MP_PROCS=4
$ export MP_PMLIGHTS=16
```

If the parallel application is started from an *X-Windows* environment where **pmarray** is running, the output square of **pmarray**, for the task that made the call in the position specified by the **light** parameter, changes to the color specified by the **color** parameter. The character string is displayed in a text output region for the task. In addition to providing a quick graphical representation of the progress of the application, the output to **pmarray** is synchronized with the task that generates it. The task will not proceed until it has been informed that the data has been sent to **pmarray**. This gives you a much more current view of the state of each task.

The example below shows how **pmarray** can be used to track the progress of an application. This program does not do anything useful, but there is an inner loop that is executed 16 times, and an outer loop that is executed based on an input parameter. On each pass through the inner loop, the **mpc\_marker** call is made to color each square in the task's **pmarray** row according to the color of the index for the outer loop. On the first pass through the inner loop, each of the 16 squares will be colored with color 0. On the second pass, they will be colored with color 1. On each pass through the outer loop, the task will be delayed by the number of seconds equal to its task number. Thus, task 0 will quickly finish, but task 4 will take a while to finish. The color of the squares for a task indicates how far they are

through the outer loop. The square that is actually changing color is the position in the inner loop. In addition, a text message is updated on each pass through the outer loop.

```

/*****
 *
 * Demonstration of use of pmarray
 *
 * To compile:
 * mpcc -g -o use_pmarray use_pmarray.c
 *
 *****/

#include<stdlib.h>
#include<stdio.h>
#include<mpi.h>
#include<time.h>

int main(int argc, char **argv)
{
    int i, j;
    int inner_loops = 16, outer_loops = 0;
    int me;
    char buffer[256];
    time_t start, now;

    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&me);

    if(argc>1) outer_loops = atoi(argv[1]);
    if(outer_loops<1) outer_loops = 16;

    for(i=0;i<outer_loops;i++)
    {
        /* Create message that will be shown in pmarray text area */
        sprintf(buffer,"Task %d performing loop %d of %d",me,i+1,outer_loops);
        printf("%s\n",buffer);
        for(j=0;j<inner_loops;j++)
        {
            /* pmarray light shows which outer loop we are in */
            /* color of light shows which inner loop we are in */
            /* text in buffer is created in outer loop */
            mpc_marker(i,5*j,buffer);
        }

        /* Pause for a number of seconds determined by which */
        /* task this is. sleep(me) cannot be used because */
        /* underlying communication mechanism uses a regular */
        /* timer interrupt that interrupts the sleep call */
        /* Instead, we'll look at the time we start waiting */
        /* and then loop until the difference between the */
        /* time we started and the current time is equal to */
        /* the task id */
        time(&start);
        time(&now);
        while(difftime(now,start)<(double)me)
        {
            time(&now);
        }
    }
    MPI_Finalize();
    return 0;
}

```

Before running this example, you need to start **pmarray**, telling it how many lights to use. You do this with the **MP\_PMLIGHTS** environment variable.

In our example, if we wanted to run the program with eight outer loops, we would set **MP\_PMLIGHTS** to **8** before running the program.

Although it is not as freeform as print statements, or as extensible, **pmarray** allows you to send three pieces of information (the light number, the color, and the text string) back to the home node for presentation. It also ensures that the presentation is synchronized as closely to the task state as possible. We recommend that if you use **pmarray** for debugging, you define a consistent strategy for its use in your application. For example, you may want to use color to indicate state (initializing, active, disabled, terminated), and light number to indicate module or subsystem. You can configure **pmarray** with as many lights as will fit on the display.

---

## Chapter 10. 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 this manual 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.
- 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 S/390 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 94 and Table 2 on page 94 outlines the `pdbx` subcommands that are 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 commands" on page 185.

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 that are entered at the `pdbx` command prompt.

`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.

For a description of the **pdbx** subcommands, see “Appendix A. Parallel Environment commands” on page 185.

While working with PE, you can display the online information by using the man command. For example enter

```
man pdbxalias
```

for information on the **pdbx alias** subcommand. For information on other **pdbx** subcommands, enter **man** and the name of the subcommand prefixed by **pdbx**.

You can set the command context on a single task or a group of tasks as described in “Setting command context” on page 106.

Table 1. Context insensitive **pdbx** subcommands

Context insensitive <b>pdbx</b> 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 121
attach <[all   task_list]>	Attach the debugger to some or all the tasks of a given <b>poe</b> job.	“Attach mode” on page 99
detach	Detach <b>pdbx</b> from all tasks that were attached. This subcommand causes the debugger to exit but leaves the <b>poe</b> application running.	“Exiting <b>pdbx</b> ” on page 124
dhhelp [dbx_command]	Display a brief list of <b>dbx</b> commands or help information about them.	“Accessing help for <b>dbx</b> subcommands” on page 120
group <action> [group_name] [task_list]	Manipulate groups. The actions are <b>add</b> , <b>change</b> , <b>delete</b> , and <b>list</b> . 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 103
help [subject]	Display a list of <b>pdbx</b> commands and topics or help information about them.	“Accessing help for <b>pdbx</b> subcommands” on page 120
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 106
quit	End a <b>pdbx</b> session.	“Exiting <b>pdbx</b> ” on page 124
source <cmd_file>	Execute <b>pdbx</b> subcommands from a specified file. <b>Note:</b> The file may contain context sensitive commands.	“Reading subcommands from a command file” on page 122
tasks [long]	Display information about all the tasks in the partition.	“Displaying tasks and their states” on page 102
unalias alias_name	Remove a command alias specified by the <b>alias</b> subcommand.	“Creating, removing, and listing command aliases” on page 121

Table 2. Context sensitive **pdbx** subcommands

Context sensitive <b>pdbx</b> subcommands		
This subcommand:	Is used to:	For more information see:

Table 2. Context sensitive *pdbx* subcommands (continued)

Context sensitive <i>pdbx</i> subcommands		
delete <[event_list   *   all]>	Remove breakpoints and tracepoints set by the <b>stop</b> and <b>trace</b> 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 <i>pdbx</i> events” on page 114
<i>dbx</i> < <i>dbx_command</i> >	Issue a <b>dbx</b> subcommand directly to the <b>dbx</b> sessions running on the remote nodes. This subcommand is not intended for casual use. It must be used with caution, because it circumvents the <b>pdbx</b> server which normally manages communication between the user and the remote <b>dbx</b> sessions. It enables experienced <b>dbx</b> users to communicate directly with remote <b>dbx</b> sessions, but can cause problems as <b>pdbx</b> will have no knowledge of the communication that transpired. <b>Note:</b> In addition to the <b>pdbx</b> subcommands shown in this table, you can use most of the <b>dbx</b> subcommands. The <b>dbx</b> subcommands are all context sensitive. The only <b>dbx</b> subcommands that you cannot use are <b>clear</b> , <b>detach</b> , <b>edit</b> , <b>multproc</b> , <b>prompt</b> , <b>run</b> , <b>rerun</b> , <b>screen</b> , and the <b>sh</b> subcommand with no arguments.	the online PE manual page for <b>pdbx</b> . This manual page also appears in “Appendix A. Parallel Environment commands” on page 185.
hook	Regain control over an unhooked task.	“Unhooking and hooking tasks” on page 116
list [line_number   line_number, line_number   procedure]	Display lines of the current source file, or of a procedure.	“Displaying source” on page 119
load <program> [program_arguments]	Load a program on each node in the current context. This can only be issued once per task per <b>pdbx</b> session. <b>pdbx</b> will look for the program in the current directory unless a relative or absolute path name is specified.	“Loading the partition with the load subcommand” on page 101
print <[expression]>	Print the value of an expression.	“Viewing program variables” on page 117
status [all]	Display a list of breakpoints and tracepoints set by the <b>stop</b> and <b>trace</b> subcommands in the current context. If “all” is specified, all events, regardless of context are shown.	“Checking event status” on page 115
stop	Set a breakpoint for tasks in the current context. Breakpoints are stopping places in your program that halt execution.	“Setting breakpoints” on page 111
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 112
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 116
where	Display a list of active procedures and functions.	“Viewing program call stacks” on page 117
<EscChar-c>	Regain debugger control when some tasks in the current context are running. This causes a <b>pdbx</b> subset prompt to be displayed, which allows a subset of the <b>pdbx</b> function to be performed.	“Context switch when blocked” on page 108

## 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.

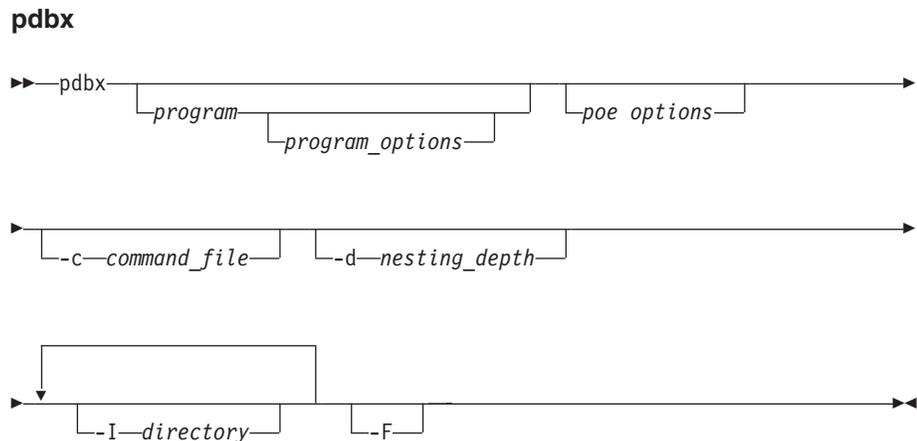
When the application is started using **pdbx** in normal mode, debugger control of the application is given to the user by default at the first executable source line within the main routine. If the file containing the main routine is not compiled with **-g** the debugger will exit. The environment variable **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. On POE environment variables and command-line flags refer to "Appendix B. POE environment variables and command-line flags" on page 283.

### 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 path name is specified.

ENTER



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

```
FOM00504 Partition loaded ...
```

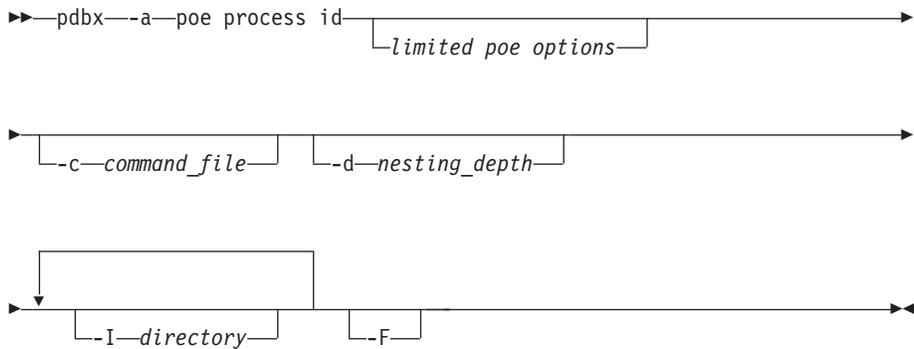
You will then see the **pdbx** prompt:

```
pdbx(a11)
```

The prompt shows the command context all. For more information see “Setting command context” on page 106.

ENTER

**pdbx**



- This starts **pdbx** in attach mode. See “Attach mode” on page 99 for more information.

ENTER



- 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 3. 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 by 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 “Appendix B. POE environment variables and command-line flags” on page 283.

**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 3. Debugger option flags (*pdbx*)

Use this flag:	To:	For example:
<b>-a</b>	Attach to a running <b>poe</b> job by specifying its process id. This must be executed from the node where the <b>poe</b> 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 <b>pdbx</b> debugger to an already running <b>poe</b> job.  ENTER <b>pdbx -a</b> <poe_process_id>

Table 3. Debugger option flags (pdbx) (continued)

Use this flag:	To:	For example:
<b>-c</b>	<p>Read <b>pdbx</b> startup commands from the specified <i>commands_file</i>. The commands stored in the specified file are executed before command input is accepted from the keyboard.</p> <p>If the <b>-c</b> flag is not used, the <b>pdbx</b> debug program attempts to read startup commands from the file <b>·pdbxinit</b>. To find this file, it first looks in the current directory, and then in the user's home directory.</p> <p>In a <b>pdbx</b> session, you can also read commands from a file using the <b>source</b> subcommand. "Reading subcommands from a command file" on page 122 describes how to use this subcommand in <b>pdbx</b>.</p>	<p>To start the <b>pdbx</b> debugger and read startup commands from a file called <b>start.cmd</b>:</p> <p>ENTER  <b>pdbx -c start.cmd</b></p>
<b>-d</b>	<p>Set the limit for the nesting of program blocks. The default nesting depth limit is 1000. This flag is passed to <b>dbx</b> unmodified.</p>	<p>To specify a nesting depth limit:</p> <p>ENTER  <b>pdbx -d nesting depth</b></p>
<b>-F</b>	<p>This flag can be used to turn off <i>lazy reading</i> mode. Turning lazy reading mode off forces the remote <b>dbx</b> sessions to read all symbol table information at startup time. By default, lazy reading mode is on.</p> <p>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 <b>dbx</b> sessions. Because all symbol table information is not read at <b>dbx</b> 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 <b>whereis</b> command, where instances of the specified local variable may not be found until the other files containing these instances are somehow referenced.</p>	<p>To start the <b>pdbx</b> debugger and read all symbol table information:</p> <p>ENTER  <b>pdbx -F</b></p>
<b>-h</b>	<p>Write the <b>pdbx</b> usage to <b>STDERR</b> then exit. This includes <b>pdbx</b> command-line syntax and a description of <b>pdbx</b> options.</p>	<p>ENTER  <b>pdbx -h</b></p>
<b>-I</b> (upper case i)	<p>Specify a directory to be searched for an executable's source files. This flag must be specified multiple times to set multiple paths. (Once <b>pdbx</b> is running, this list can be overridden on a group or single node basis with the <b>use</b> command.)</p>	<p>To add <i>directory1</i> to the list of directories to be searched when starting the <b>pdbx</b> debugger:</p> <p>ENTER  <b>pdbx -I dir1</b></p> <p>You can add as many directories as you like to the directory list in this way. For example, to add two directories:</p> <p>ENTER  <b>pdbx -I dir1 -I dir2</b></p>

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 *z/OS UNIX System Services Command Reference* and *z/OS UNIX System Services Programming Tools*.

## 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 that is 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
```

**Note:** When using the attach feature, some environment variables must be set before. See “Let us attach the debugger” on page 81 for a detailed description.

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 that is 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 by using the task numbering of the original **poe** application partition.

## Attach screen

Figure 10 shows a sample **pdbx** Attach screen.

ATTENTION: FOM0H9049 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 10. 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 103 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 that is 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 that is not compiled with **-g**; however, the information provided is limited to a stack trace.

You can also attach **pdbx** to an application that is 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 that 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 that 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 C. Command-line flags for Normal or Attach Mode” on page 291 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 incorrect 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 that 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 path name 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:
<p><b>CHECK</b></p> <p>the <b>pdbx</b> command prompt to make sure the command context is on all tasks. The context <i>all</i> is the default when you start the <b>pdbx</b> debugger, so the prompt should read:</p> <pre>pdbx(all)</pre> <p>If the command context is not set on <i>all</i> tasks, reset it. To do this:</p> <p><b>ENTER</b></p> <pre>on all</pre> <p>Once the command context is on all tasks:</p> <p><b>ENTER</b></p> <pre>load program [program_options]</pre> <ul style="list-style-type: none"> <li>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.</li> </ul> <p><b>Note:</b> The example above has the same effect as putting the program name and options on the command line.</p>	<p><b>SET</b> 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 <i>program1</i> on task 0 and a program named <i>program2</i> on tasks 1 through 4, you would:</p> <p><b>ENTER</b></p> <pre>on 0</pre> <ul style="list-style-type: none"> <li>The debugger sets the command context on task 0</li> </ul> <p><b>ENTER</b></p> <pre>load program1 [program_options]</pre> <p>The debugger loads <i>program1</i> on task 0.</p> <p><b>ENTER</b></p> <pre>group add groupa 1-4</pre> <ul style="list-style-type: none"> <li>The debugger creates a task group named <i>groupa</i> consisting of tasks 1 through 4.</li> </ul> <p><b>ENTER</b></p> <pre>on groupa</pre> <p>The debugger sets the command context on tasks 1 through 4.</p> <p><b>ENTER</b></p> <pre>load program2 [program_options]</pre> <ul style="list-style-type: none"> <li>The debugger loads <i>program2</i> onto tasks 1 through 4, and the message "Partition loaded..." displays. The parallel program runs up to the first executable statement and stops.</li> </ul>

## 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 4 on page 106 for information on task states). If you specify "long" after the command, it also displays the name, IP address, and job manager number that are 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 that 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* that consists 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 <i>task_list</i> can be:	For example, to add the following task(s) to <i>groupa</i> :	You would ENTER:	• The following message displays:
a single task	task 6	<b>group add</b> <i>groupa</i> 6	1 task was added to group "groupa".
a collection of tasks	tasks 6, 8, and 10	<b>group add</b> <i>groupa</i> 6 8 10	3 tasks were added to group "groupa".
a range of tasks	tasks 6 through 10	<b>group add</b> <i>groupa</i> 6:10	5 tasks were added to group "groupa".
a range of tasks	tasks 6 through 10	<b>group add</b> <i>groupa</i> 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 <i>task_list</i> can be:	For example, to delete the following from <i>groupa</i> :	You would ENTER:	• The following message displays:
a single task	task 6	<b>group delete</b> <i>groupa</i> 6	Task: 6 was successfully deleted from group "groupa".
a collection of tasks	task 6, 8, and 10	<b>group delete</b> <i>groupa</i> 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	<b>group delete</b> <i>groupa</i> 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	<b>group delete</b> <i>groupa</i> 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 that 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:	• The following message displays:
list all the task groups.	<b>group list</b>	The debugger displays a list of all existing task groups and their members. An example of such a list is shown below.  <pre> 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 </pre>
list all the members of a single task group	<b>group list</b> <i>oddTasks</i>	The debugger displays a list of all the members of task group <i>oddTasks</i> .  <pre> 1:D    3:U    5:D    7:D    9:D   11:R </pre>

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 4. 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 11 on page 106 illustrates the relationship between the **pdbx** debugger, which runs on the home task, and the various **dbx** processes that run 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”.

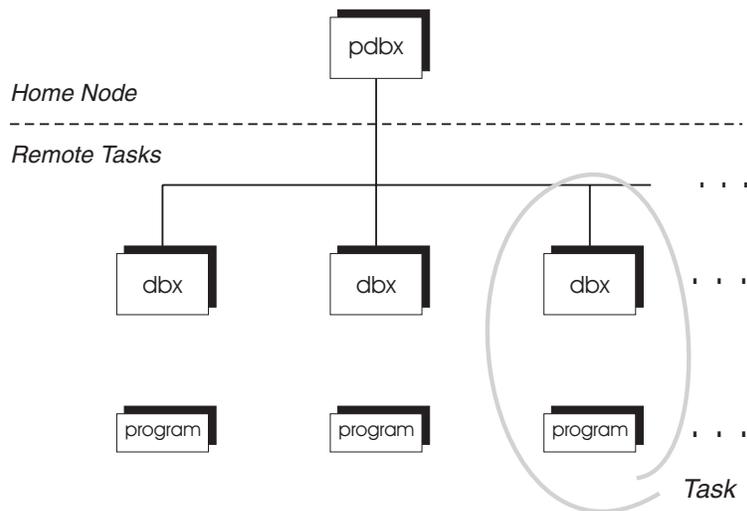


Figure 11. Relationship between home node (pdbx) and remote tasks (dbx processes)

### 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(a11)
```

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 that you have a parallel program that is 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.

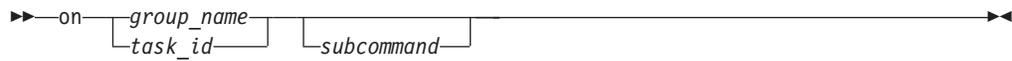
```
pdbx(all)
```

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 108 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 <EscChar-c> followed by the **back** or the **on** command, to issue further **pdbx** commands in the original context. The syntax is:



For example, assume that 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 that are 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 <EscChar-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 **<EscChar-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 93).

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 121
back	Return to a <b>pdbx</b> prompt.	“Returning to a pdbx prompt” on page 110
group <action> [group_name] [task_list]	Manipulate groups. The actions are <b>add</b> , <b>change</b> , <b>delete</b> , and <b>list</b> . 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 103
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 <b>pdbx</b> prompt.	“Interrupting tasks” on page 112
help [subject]	Display a list of <b>pdbx</b> commands and topics or help information about them.	“Accessing help for pdbx subcommands” on page 120
on <[group   task]>	Set the current context for later subcommands. This command always returns to a <b>pdbx</b> prompt.	“Setting command context” on page 106
source <cmd_file>	Execute subcommands stored in a file. <b>Note:</b> The file may contain context sensitive commands.	“Reading subcommands from a command file” on page 122
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 115
tasks [long]	Display processes (tasks) and their states.	“Displaying tasks and their states” on page 102
quit	Exit the <b>pdbx</b> program and kill the application.	“Exiting pdbx” on page 124
unalias alias_name	Remove a previously defined alias.	“Creating, removing, and listing command aliases” on page 121

This subset command:	Is used to:	For more information see:
<EscChar-c>	Has no effect, except to display the following message: Typing EscChar-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" on page 108

**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 <EscChar-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 by 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 that are 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**.

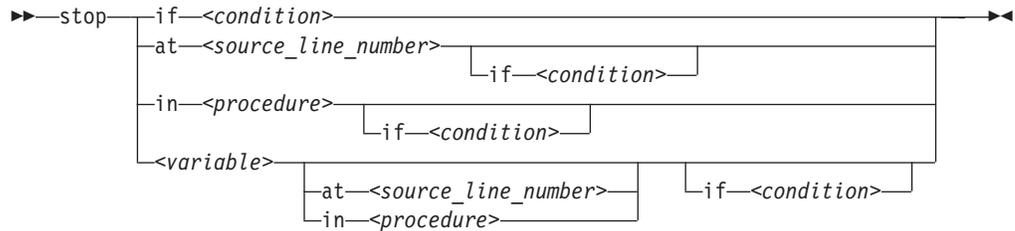
The next few pages should act as an introduction to breakpoints and tracepoints if you are unfamiliar with **dbx**.

Refer to *z/OS UNIX System Services Command Reference* and *z/OS UNIX System Services Programming Tools* for more information on **dbx** 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:



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 122.

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 that is 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: F0M0H2081 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 by 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 113 for more information.
  3. For further details on the **stop** subcommand, refer to its use on the **dbx** command as described in *z/OS UNIX System Services Command Reference* and *z/OS UNIX System Services Programming Tools*.

## 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 <EscChar-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.

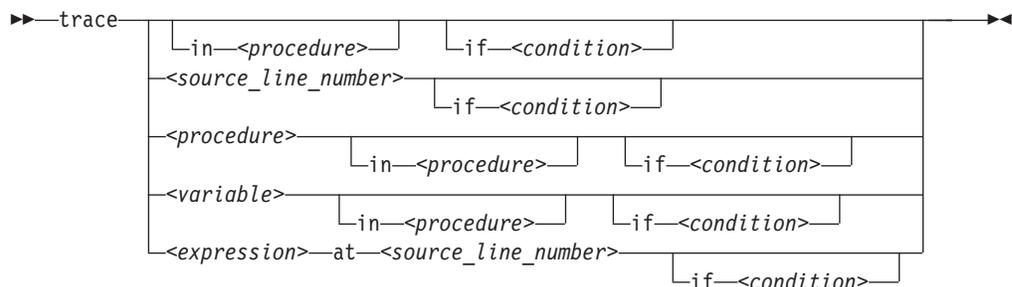
The **halt all** command at the **pdbx** prompt affects tasks outside of the current context. Messages at the prompt show the task numbers that are and are not interrupted, but the **pdbx** prompt returns immediately because the state of the tasks in the current context is unchanged.

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:



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 122.

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: F0M0H2081 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 *z/OS UNIX System Services Command 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:



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
FOM0H2062 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
main(), line 144 in "send1.c"
```

### Viewing program variables

The **print** subcommand prints the value of a list of expressions, specified by the *expression* parameters.

The syntax of this context sensitive subcommand is:

```
▶▶—print—expression—▶▶
```

See “Specifying expressions” on page 122 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
```

Refer to *z/OS UNIX System Services Programming Tools* 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.

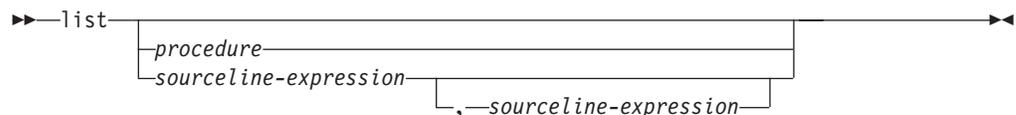
All lines from the first line number specified to the second line number specified, inclusive, are then displayed, provided these lines fit in the list window.

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:



## 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 _____▶▶
      |_____|
      |subcommand|
      |_____|
      |topic|
```

Another command to get help for the **pdbx** subcommands, is the **man** command.

Type

```
man pdbxalias
```

to view help for the **pdbx alias** subcommand. Generally, help for the **pdbx** subcommands can be displayed with the **man** command and the subcommand name prefixed by **pdbx**.

### 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_subcommand>|
```



►►—`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—commands_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 that 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 3 on page 97. 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. The following operators are valid:

Arithmetic operators

<b>+</b>	Addition
<b>-</b>	Subtraction
<b>-</b>	Negation
<b>*</b>	Multiplication
<b>/</b>	Floating point division
<b>div</b>	Integer division
<b>mod</b>	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 commands” on page 185 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 may 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 11. 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 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 this manual 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.
- 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**.

---

### Setting up the debugger environment

#### Setting up your X-Window environment

If you are already running X-Windows applications, you can probably skip this step. However, you might check that the **pedb** X defaults file has been changed by your System Administrator. For details, refer to *z/OS UNIX System Services Planning*.

First you must make sure that an X-server is running on the workstation on which you want to display the **pedb** Window. On UNIX machines the X-server is usually started by default at boot time. On Personal Computers you may have to start it yourself (for example, for OS/2 there is an X-server called 'PMX Server'). Next, you must give permission to the z/OS system (the X-client machine) to display something on the screen of the workstation. This is typically done with the **xhost** command on a OS/2 workstation:

```
ENTER
      xhost z/OS-machine-name
```

The *z/OS-machine-name* can be either a host name or an IP address.

To complete the X-Windows setup you must tell `pedb` (the X-client) where to display its window. This is done using the `DISPLAY` environment variable:

```
ENTER
      export DISPLAY workstation-name:0
```

The *workstation-name* can be either a host name or an IP address.

## Be aware of storage requirements

`pedb` is a rather large application which requires sufficient storage available. When using the OMVS shell, a minimum TSO logon region size 50MB is recommended.

In case you run out of space while being logged in to the UNIX shell via `rlogin` or `telnet`, you may ask your System Administrator to increase the value of `MAXASIZE` defined in `SYS1.PARMLIB (BPXPRMxx)`.

## Setting up the `pedb.ad` file

It is necessary to set up the `pedb.ad` file. See “Customizing `pedb` resources” on page 183 for information on this matter.

---

## 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.

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. On POE environment variables and command-line flags refer to “Appendix B. POE environment variables and command-line flags” on page 283.

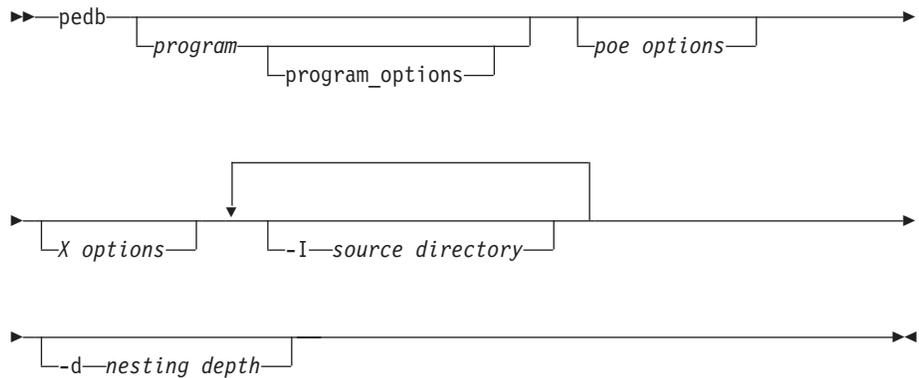
## Normal mode

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**

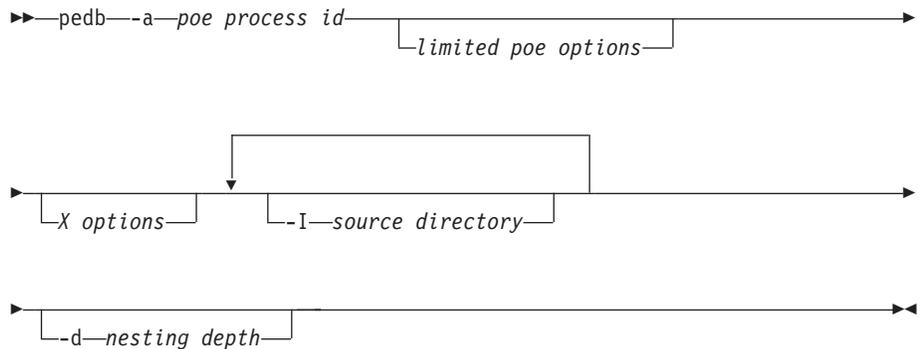


• This starts **pedb**. You will see the **pedb** main window open. If you specified a *program*, it is loaded on each node of your partition and you see the message:

FOMOG0101 Your program has been loaded.

ENTER

**pedb**



• This starts **pedb** in attach mode. See “Attach mode” on page 130 for more information.

ENTER



• 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 5 on page 130. 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 “Appendix B. POE environment variables and command-line flags” on page 283.

Table 5. Debugger option flags (*pedb*)

Use this flag:	To:	For example:
<b>-a</b>	Attach to a running <b>poe</b> job by specifying its process id. This must be executed from the node where the <b>poe</b> 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 <b>pedb</b> in attach mode: <b>ENTER</b> <b>pedb -a &lt;poe PID&gt;</b>
<b>-d</b>	Set the limit for the nesting of program blocks. The default nesting depth limit is 25.	To specify a nesting depth limit: <b>ENTER</b> <b>pedb -d nesting.depth</b>
<b>-h</b>	Write the <b>pedb</b> usage to STDERR then exit. This includes <b>pedb</b> command-line syntax and a description of <b>pedb</b> flags.	To write the <b>pedb</b> usage to STDERR: <b>ENTER</b> <b>pedb -h</b>
<b>-I</b> (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 <b>pedb</b> is running, this list can also be updated using the Update Source Path window.)	To add <i>directory1</i> to the list of directories to be searched when starting the <b>pedb</b> debugger: <b>ENTER</b> <b>pedb -I dir1</b>  You can add as many directories as you like to the directory list in this way. For example, to add two directories: <b>ENTER</b> <b>pedb -I dir1 -I dir2</b>

## 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
```

**Note:** When using the attach feature, some environment variables must be set before. See “Let us attach the debugger” on page 81 for a detailed description.

**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 12 shows the **pedb** Attach window.

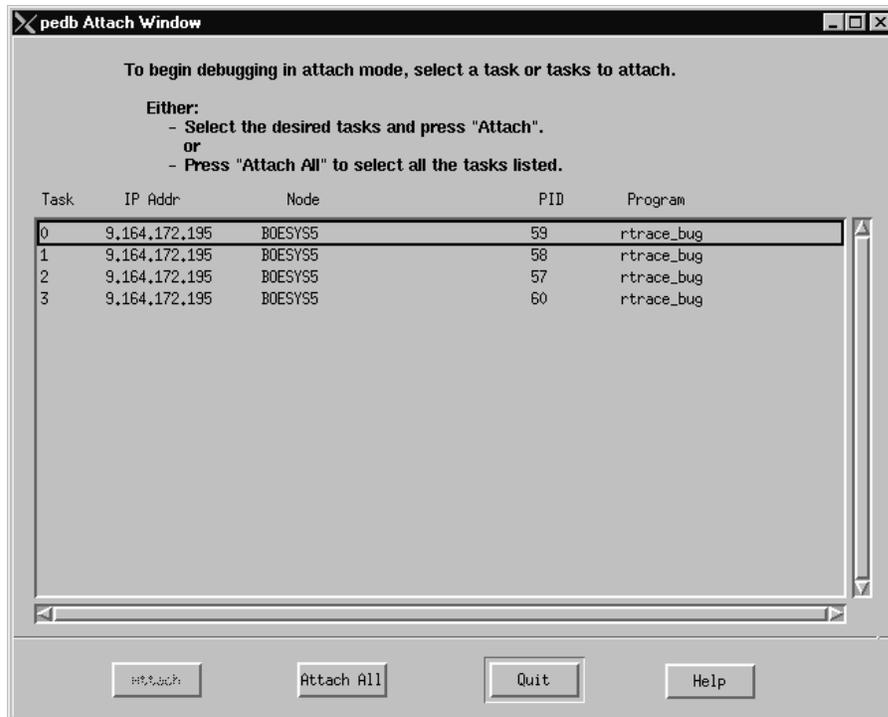


Figure 12. pedb Attach window

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

## ENTER

**password** on the shell where pedb has been started

- 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 that 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 that 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 C. Command-line flags for Normal or Attach Mode” on page 291 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 that 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 13 on page 135. 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 in Figure 15 on page 138.

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.

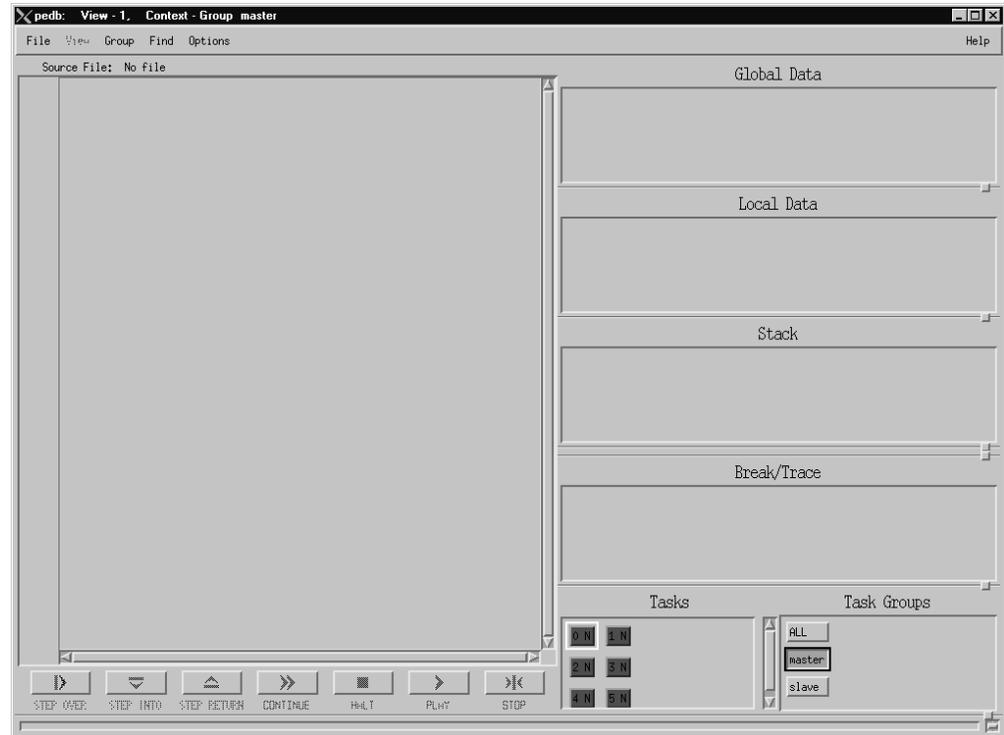


Figure 13. *pedb* main window before the partition is loaded

If you did not 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.

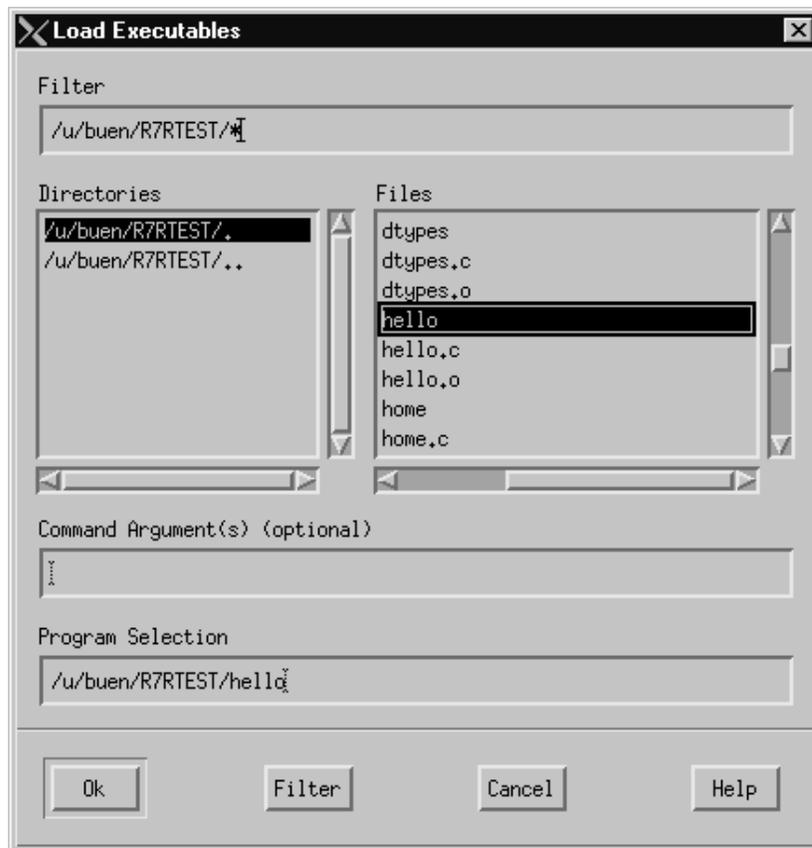


Figure 14. Load Executables window

## Program search path

Like **POE**, **pedb** uses the normal shell search path that is established by the environment variable **PATH** if you do not 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 179 for information on the source code search path.

To load the same executable for all tasks (SPMD):	To load different executables (MPMD):
<p><b>CHECK</b> the title bar of the <b>pedb</b> window to make sure the context is set to all tasks. This is the default when you start the <b>pedb</b> debugger. If the context is not on the task group <i>ALL</i>, reset it.</p> <p>To set the context on all tasks:</p> <p><b>PRESS</b> the task group button labeled <i>ALL</i> in the Task Groups Area.</p> <p>Once the context is set on all tasks:</p> <p><b>SELECT</b> <b>File → Load Executables ...</b> • The Load Executables window opens. This window allows you to choose the appropriate directory and select the corresponding executable file.</p> <p><b>TYPE IN</b> any command-line arguments to the executable selected for loading.</p> <p><b>SELECT</b> the directory and executable file you want to load by clicking on each name. Double clicking on the file name automatically loads the program.</p> <p><b>PRESS OK</b> • The Load Executables window closes, and the specified program is loaded for all tasks. Each task stops at the first executable source line. <b>MP_DEBUG_INITIAL_STOP</b> can be set to override the default of the first executable source line in <i>main()</i>. Set <b>MP_DEBUG_INITIAL_STOP</b> to the <i>file:linenumber</i>.</p> <p><b>DISPLAYS MESSAGE</b> FOMOG0101 Your program has been loaded.</p>	<p>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:</p> <p><b>PRESS</b> the task button labeled <i>0</i> in the Task Area.</p> <p><b>SELECT</b> <b>File → Load Executables ...</b> • The Load Executables window opens. This window allows you to choose the appropriate directory and select the corresponding executable file.</p> <p><b>SELECT</b> the directory and executable file you want to load by clicking on each name. Double clicking on the file name automatically loads the program.</p> <p><b>TYPE IN</b> the command-line arguments to the executable selected for loading.</p> <p><b>PRESS OK</b> • The Load Executables window closes and the debugger loads the program for task 0.</p> <p>Create a group, say <i>group1</i>, containing tasks 1 through 4.</p> <p><b>PRESS</b> the task group button labeled <i>group1</i> to set the context.</p> <p><b>SELECT</b> <b>File → Load Executables ...</b> • The Load Executables window opens.</p> <p><b>SELECT</b> the directory and executable file you want to load by clicking on each name. Double clicking on the file name automatically loads the program.</p> <p><b>TYPE IN</b> any command-line arguments to the executable selected for loading.</p> <p><b>PRESS OK</b> • The Load Executables window closes and the debugger loads the program for tasks 1 through 4. Each tasks stops at the first executable statement. <b>MP_DEBUG_INITIAL_STOP</b> can be set to override the default of the first executable source line in <i>main()</i>. Set <b>MP_DEBUG_INITIAL_STOP</b> to the <i>file:linenumber</i>.</p> <p><b>DISPLAYS MESSAGE</b> FOMOG0101 Your program has been loaded.</p>

## The pedb window with a partition loaded

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

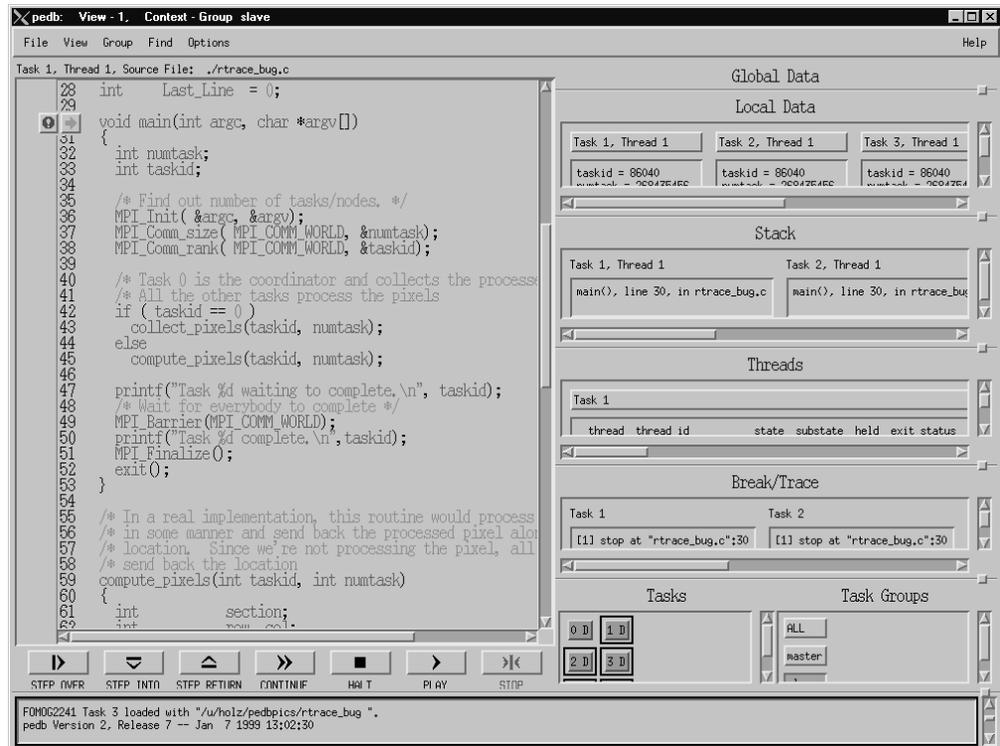


Figure 15. pedb main window after the partition is loaded

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
  - Tools
  - 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 156. 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 157 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 161 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 156 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 144.
- 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 140 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 13 on page 135, 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 that 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" 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.

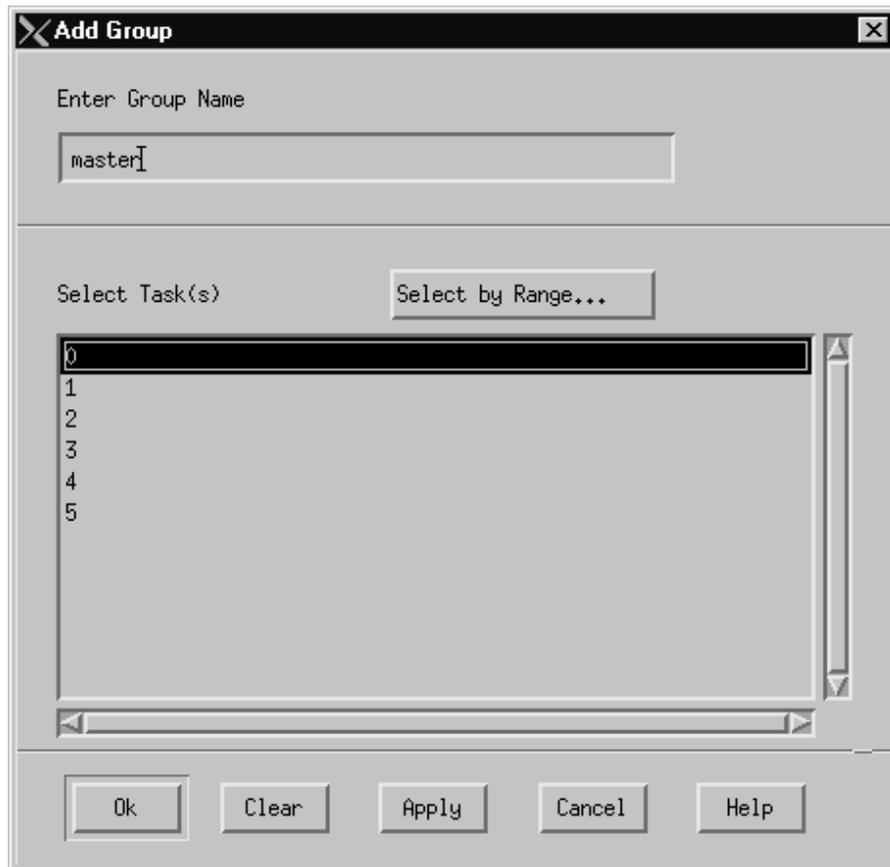


Figure 16. Add Group window

**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 pressing the left mouse button.

or

a range of tasks by pressing 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 deselect 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:
tasks 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 6. 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 <b>pedb</b> .
R	White	Running	The task is in control and running.
P	MediumSeaGreen	Playing	The <b>Play</b> button has been pressed. The task is switching between Playing and Running, with some limited function.

Table 6. Status codes (continued)

Code	Default Color	Status	Description
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 <b>hidden_exit</b> breakpoint that was set by <b>libdbx</b> .
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 7. 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 <i>over</i> subroutines and functions.	Step Over
Manually step the execution of tasks in the current context, by a line of source code, stepping <i>into</i> 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 <i>running</i> tasks. The <b>Halt</b> 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
<b>Note:</b> To modify the function of the <b>Play</b> button, refer to "Customizing the Play control button" on page 152.	

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. 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 that 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 150.

**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.c":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. The following operators are valid:

#### Arithmetic operators

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

#### Relational and logical operators

<b>&lt;</b>	Less than
<b>&gt;</b>	Greater than
<b>&lt;=</b>	Less than or equal to
<b>&gt;=</b>	Greater than or equal to
<b>==</b>	Equal to
<b>=</b>	Equal to
<b>!=</b>	Not equal to
<b>&lt; &gt;</b>	Not equal to
<b>  </b>	Logical OR
<b>or</b>	Logical OR
<b>&amp;&amp;</b>	Logical AND
<b>and</b>	Logical AND

#### Bitwise operators

<b>bitand</b>	Bitwise AND
<b> </b>	Bitwise OR
<b>xor</b>	Bitwise exclusive OR
<b>~</b>	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] if x > 19 { stop } at "ptst4.c":22
```

*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 **\$current**. 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 `$current = $t1` after selecting line 234 in your source would result in the program stopping when the thread labeled `$t1` encountered line 234. To state this another way, the breakpoint is triggered only when thread 1 encounters the breakpoint. Refer to *z/OS UNIX System Services Programming Tools* 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 8 on page 150.

### 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 8. Removing breakpoints

To remove the breakpoint for a single task:	To remove all breakpoints at the same line of code:
<p><b>PLACE</b> the mouse cursor over the breakpoint's event message for that task in the Break/Trace Area.  <b>Note:</b> The task must be in the current context for the event message to be displayed.</p> <p><b>PRESS</b> the left mouse button</p> <ul style="list-style-type: none"> <li>• The breakpoint's event message highlights to show that you have selected the breakpoint.</li> </ul> <p><b>PRESS</b> the right mouse button.</p> <ul style="list-style-type: none"> <li>• A selection menu appears.</li> </ul> <p><b>SELECT</b>  <b>Delete</b></p> <ul style="list-style-type: none"> <li>• The breakpoint's event message disappears to show that the debugger has removed the breakpoint for the task.</li> </ul>	<p><b>PLACE</b> the mouse cursor over the stop marker at that line of code.</p> <p><b>PRESS</b> the right mouse button.</p> <ul style="list-style-type: none"> <li>• A selection menu appears.</li> </ul> <p><b>SELECT</b>  <b>Delete</b></p> <ul style="list-style-type: none"> <li>• 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 <i>highlighted</i> in the Break/Trace Area.</li> </ul>

**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 152 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:
<p><b>PLACE</b> the cursor over the <b>Play</b> control button.</p> <p><b>PRESS</b> the right mouse button.</p> <ul style="list-style-type: none"> <li>• The <b>Play</b> menu appears.</li> </ul> <p><b>SELECT</b> <b>Change Play Command</b></p> <ul style="list-style-type: none"> <li>• Another menu appears with the command choices.</li> </ul> <p><b>SELECT</b> the command you want the debugger to execute repeatedly when you press the <b>Play</b> control button.</p> <ul style="list-style-type: none"> <li>• The menu closes, and the <b>Play</b> control button is set to execute the command you specified.</li> </ul>	<p><b>PLACE</b> the cursor over the <b>Play</b> control button.</p> <p><b>PRESS</b> the right mouse button.</p> <ul style="list-style-type: none"> <li>• The <b>Play</b> menu appears.</li> </ul> <p><b>SELECT</b> <b>Change Play Delay</b></p> <ul style="list-style-type: none"> <li>• The Change Play Delay window opens.</li> </ul> <p><b>FOCUS</b> on the text entry field in this window.</p> <p><b>TYPE IN</b> The new delay time in tenths of seconds.</p> <p><b>PRESS</b> <b>OK</b></p> <ul style="list-style-type: none"> <li>• The Change Play Delay window closes, and the <b>Play</b> control button is set to execute its command with the new delay you specified.</li> </ul>

**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.c":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 147 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 149 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.c":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 9.

*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 9. Removing tracepoints

To remove the tracepoint for a single task:	To remove all tracepoints at the same line of code:
<p><b>PLACE</b> the mouse cursor over the tracepoint's event message for that task in the Break/Trace Area. <b>Note:</b> The task must be in the current context for the event message to be displayed.</p> <p><b>PRESS</b> the left mouse button</p> <ul style="list-style-type: none"> <li>• The tracepoint's event message highlights to show that you have selected the tracepoint.</li> </ul> <p><b>PRESS</b> the right mouse button.</p> <ul style="list-style-type: none"> <li>• A selection menu appears.</li> </ul> <p><b>SELECT</b> <b>Delete</b></p> <ul style="list-style-type: none"> <li>• The tracepoint's event message disappears to show that the debugger has removed the tracepoint for the task.</li> </ul>	<p><b>PLACE</b> the mouse cursor over the trace marker at that line of code.</p> <p><b>PRESS</b> the right mouse button.</p> <ul style="list-style-type: none"> <li>• The <b>Break/Trace</b> menu appears.</li> </ul> <p><b>SELECT</b> <b>Delete</b></p> <ul style="list-style-type: none"> <li>• The debugger removes all tracepoints for the tasks in the current context. The trace marker disappears as well as the event strings <i>highlighted</i> in the Break/Trace Area.</li> </ul>

**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 that 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, 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:** The compiler may have optimized 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

## 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 17. 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 18. 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 19 on page 160 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 160 are also available in this window.

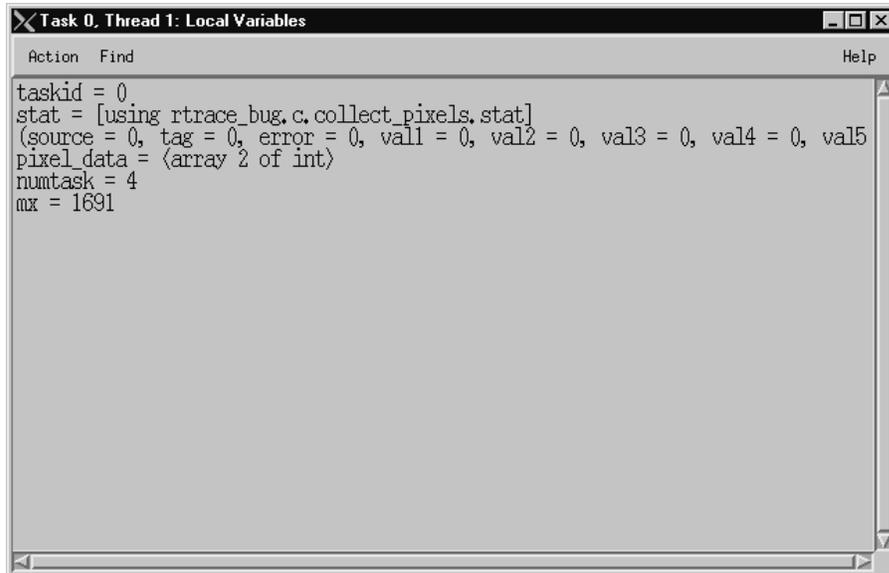


Figure 19. Variable Viewer window

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

**PLACE**

the mouse cursor over the **Action** pull-down on the menu bar.

**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 **-I** flag, or the Source Path Window.

#### SELECT

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

or

multiple, contiguous variables by pressing 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 deselect 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 157 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.

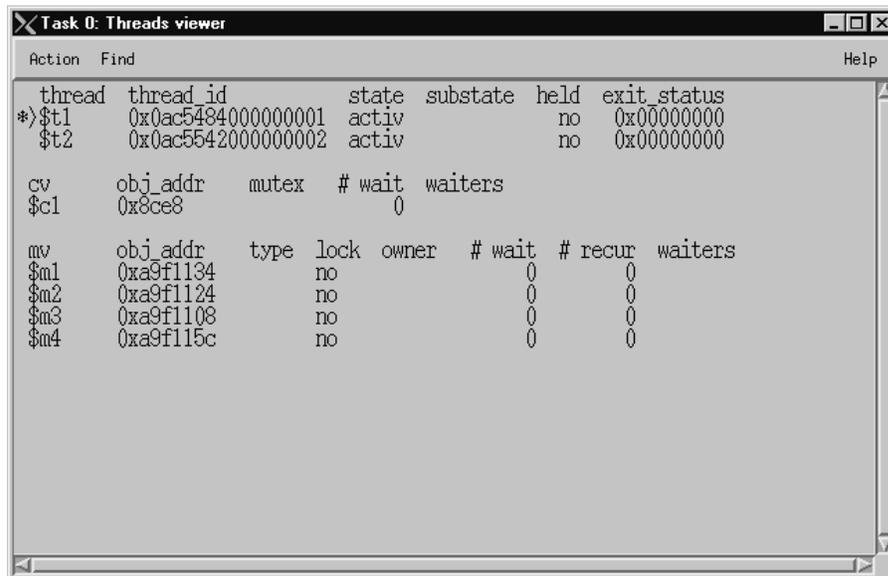


Figure 20. Threads Viewer window

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.

### 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 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 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 on page 162 and on page 162 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 on page 163. 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 redisplay 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 180.

**Note:** The debugger will display the existence and status of internal PE threads even though you may not have explicitly coded threads.

*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

Scalars, 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 167 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.

**Note:** When changing floating points values, always use a "." (dot). For example type "10.0" instead of "10".

*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 21.

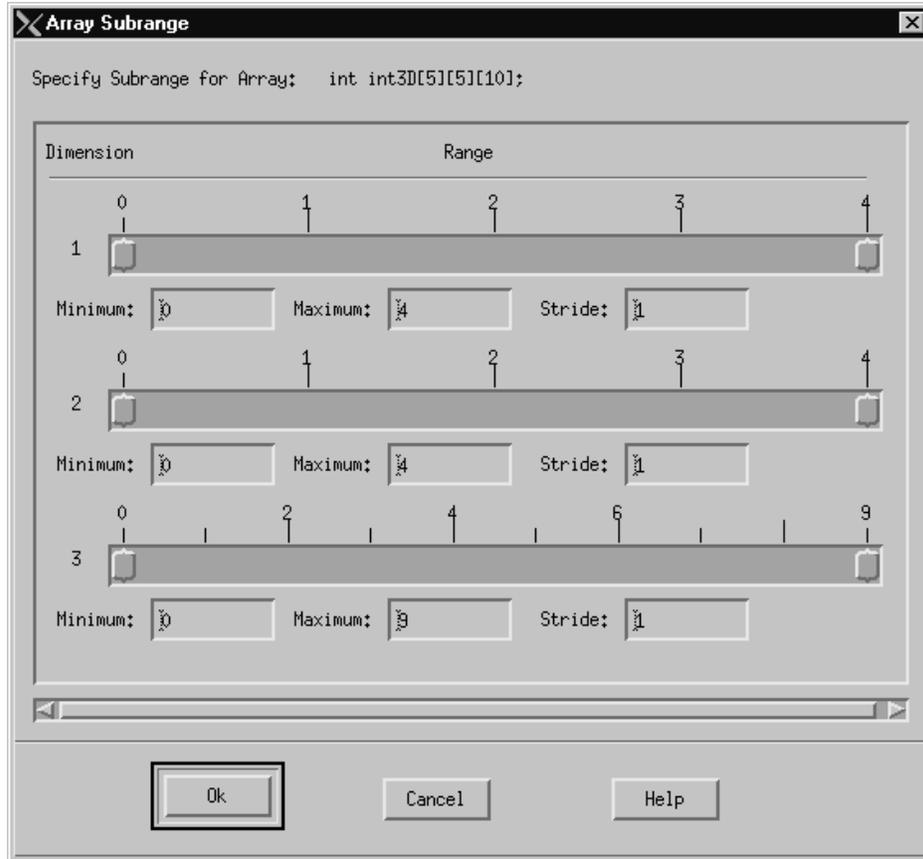


Figure 21. Array Subrange window

*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 and 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.

## Viewing MPI Request Queues

This section describes the **pedb** debugger's *message queue* facility. Part of the **pedb** debugger interface, the message queue viewing feature is designed to help you debug Message Passing Interface (MPI) applications by showing internal message request queue information. This feature allows you to view:

- A summary of the number of active messages for each task in the application. You can select criteria for the summary information based on message type and source, destination, and tag filters.
- Message queue information for a specific task.
- Detailed information concerning a specific message.

**Initial Error and Warning Messages:** It is possible that there could be problems when **pedb** does its initial checking. For example:

- The version of MPI being used may not be supported by the version of the debugger.

If this problem is discovered, an error message will appear, and the message queue debugging window will not appear, or will close.

When you start the message queue facility, it is possible that MPI has not initialized yet. If this is true, the initial message queue window will indicate that there is no data. The following describes this case in more detail.

During the initial checking, the debugger also determines the mode in which the MPI application is running. If it is not running in "debug" mode, the data will not include information on blocking messages. Debug mode is achieved by setting the **MP\_EUIDEVELOP** environment variable to **DEB**. For more information on

**MP\_EUIDEVELOP**, refer to 279. If the application is not running in debug mode, a warning message will be displayed along with the initial message queue debugging window.

Requesting information for any of the message queue windows causes the cursor to change to a “stopwatch” Further requests are disabled until the current request has finished. While the stopwatch is showing, the **pedb** main window is disabled. If a problem occurs, or a request is taking too much time, the message queue windows can be cancelled by pressing the **Cancel** button on the Application Message Queues window. This closes all message queue windows and enables the **pedb** main window. Pressing the **Cancel** button on the Application Message Queues window will always have the effect of closing all the message queue debugging windows.

**Note:** You can customize the colors used in any of the message queue windows. You can define these resources in the **pedb** X defaults file `/usr/lpp/tcpip/X11R6/lib/X11/app-defaults/Pedb`. For further information see “Customizing pedb resources” on page 183.

**Application Message Queues Window:** To start the message queue facility, from the **pedbmain** menu area:

**SELECT**

**Tools → Message Queues**

- The Application Message Queues window opens.

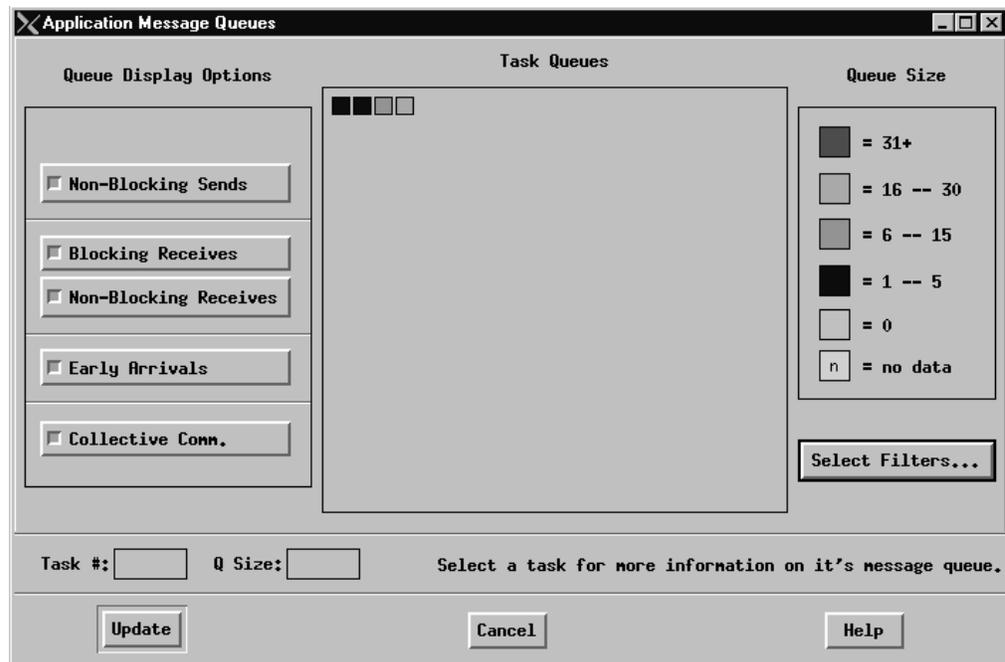


Figure 22. Application Message Queues window

The Application Message Queues window displays message queue summary information and provides a starting point for additional message queue debugging features.

The center of the Application Message Queues window contains a set of buttons representing all the tasks defined for the MPI application. The color of each button indicates the approximate size of the queue. The interpretation of the colors is given in the Queue Size scale on the right side of the window.

To further assist in interpreting these buttons, there is a message area at the bottom of the window containing the task identifier and the actual queue size. These values are filled in automatically when you move the cursor over one of the task buttons. On the right side of this area, informational messages appear at appropriate times.

This window also contains a list of Queue Display Options on the left side, and a Select Filters... button on the right, under the Queue Size scale. The criteria for selecting summary information can be modified by selecting these options and filters. Once you set option and filter information, it remains set for future updates until you change it.

**Note:** You should use caution when leaving criteria set, since this can incur considerable overhead, especially in the case of filters.

See “Selecting Options” on page 171, and “Select Filters Window” on page 171 for more information.

When you first open this window, it contains summary information that includes all types of messages, including *early arrivals*. Early arrivals refer to messages that have arrived at a task, but have no posted receives to accept them.

A button with a tan colored (default color) background and labeled with an “n” indicates there is currently no data available for that task. There could be many reasons for no available data. Basically, there is no data unless a task is in “debug” state, as indicated by the task buttons at the bottom of the **pedb** main window. Also, no data is available unless a task is in the current debugger context. The reason for a task not having any data is displayed in the right side of the message area when you move the cursor over the task button.

Pressing the Cancel button on this window causes all message queue windows to close.

*Selecting Options:* On the left side of the window there is a series of toggle buttons representing different categories of messages. When you first open the window, all the toggles are selected indicating the summary information is being collected for all categories. You can select or deselect any combination of these options for future summary information. Once you make the selections, you can retrieve the summary information by pressing the Update button at the bottom of the window.

**Select Filters Window:** Another way to set the criteria for gathering summary information is by using the Select Filters window. To open this window:

**PRESS**

**Select Filters...**

- The Select Filters window opens.

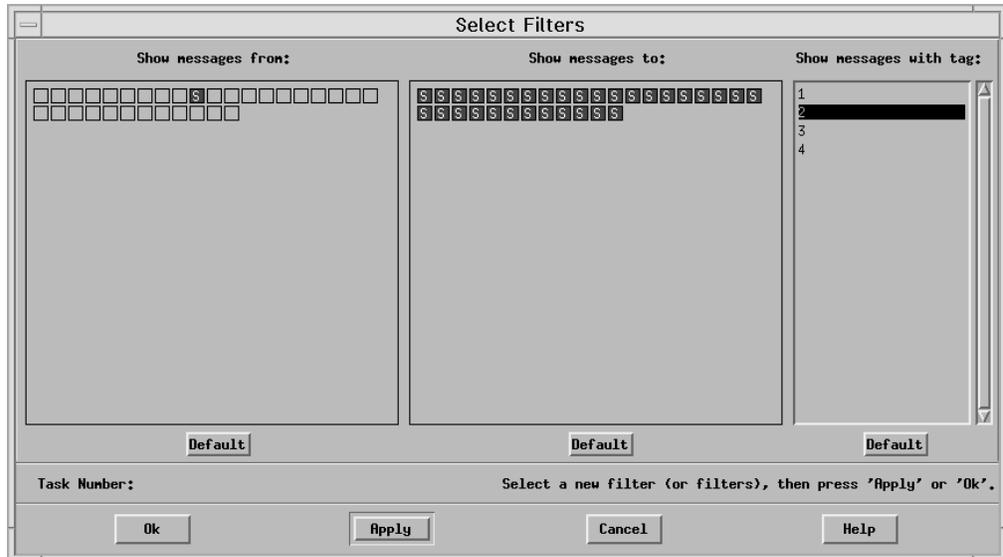


Figure 23. Select Filters window

The Select Filters window is used to set source, destination, and tag values as selection criteria for message queue summary information.

This window consists of three categories of filters you can set:

- Show messages from: (source area)
- Show messages to: (destination area)
- Show messages with tag:

Both the source and destination areas contain a set of buttons representing tasks defined for the application. The corresponding task numbers are displayed in the bottom left hand side of the window when you move the cursor over the buttons. You can select one task by pressing the task button. You can select multiple contiguous tasks by pressing and holding the left mouse button and moving the pointer across the desired tasks. Also, you can select multiple non-contiguous tasks by holding down the **Ctrl** key and selecting individual or contiguous tasks.

The “Show messages with tag:” list on the right side of the window is created by extracting all the unique tags from the message records of the available tasks. The way you select tags is similar to the way you select the other filters, except lines of the tag list are selected instead of task buttons.

You can select one or more values from each category. If you do not want a specific setting, press the default button to select all possible values. Any value found in a message is acceptable and meets the criteria. You should select at least one value for each category. In other words, a message record satisfies the filter criteria if it has one source value, one destination value, and one tag value. The window opens displaying the current settings.

**Note:** Once the filters are set, you need to go back to the Application Message Queues window to request an update.

**Scale Range Setting Window:** The default queue size ranges on the Application Message Queues window may not be appropriate for all MPI applications. You can adjust the top three ranges to more reasonable sizes. To modify the range for a particular button:

## SELECT

the button with the mouse to open a Scale Range Setting window.

The color of the small square in this window corresponds to the color of the range button, to help you keep track of which button is being changed. To change the minimum value:

## ENTER

the new value.

## PRESS

### OK

- The minimum value of this range and the maximum value of the previous lower range is adjusted. If you attempt to set the minimum value lower than the previous range minimum value plus one, you will get an error message. You are responsible for not defining overlapping maximum ranges.

**Task Message Queue Window:** You can also get further information on a particular task's message queue by opening the Task Message Queue window. On the Application Message Queues window:

## SELECT

the task button for the desired task with the mouse.

- The Task Message Queue window opens. The number of the task the cursor is on and the actual queue size is displayed in the lower left side of the Application Message Queues window.

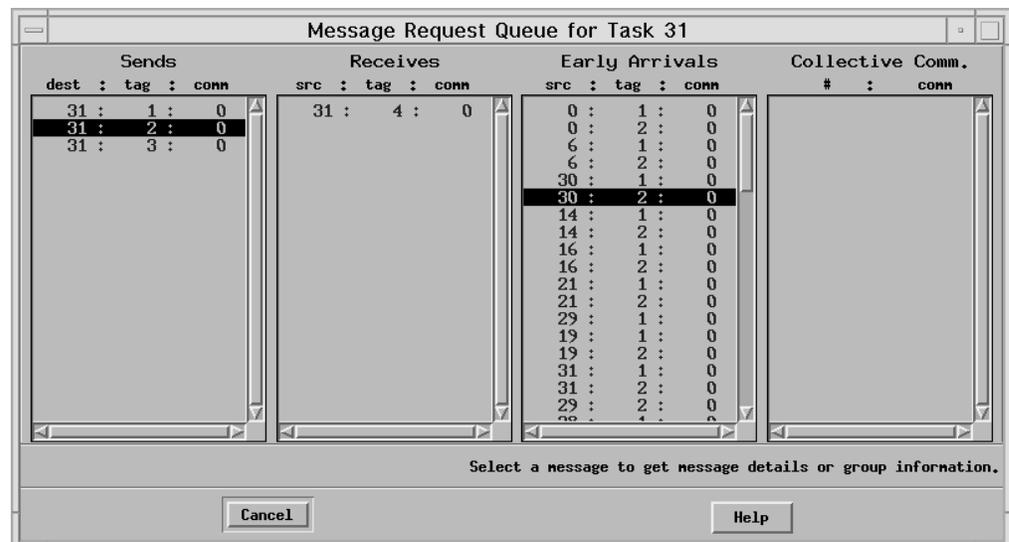


Figure 24. Task Message Queue window

The Task Message Queue window gives a list of the message request records and early arrival records for the task. This window displays the queue of currently active messages for this task. The information is separated into four categories:

- Sends
- Receives
- Early Arrivals
- Collective Communications.

Each entry represents a unique message. Entries in the first three categories provide the tag, communicator, and source or destination of the message. The source and destination is given in terms of task id. Entries in the Collective Communications column contain an arbitrary message index and the communicator. Blocking message entries are printed in red and nonblocking messages are printed in blue.

The early arrival messages are unique in that they represent messages sent by a task that have arrived before a receive has been posted to accept them.

From this window you can get additional message details or message communicator and group data. To view this data:

**SELECT**

a particular entry with the left mouse button.

**HOLD DOWN**

the right mouse button.

- The Message Data menu opens.

**SELECT**

**Message Details**

or

**Group Info**

**Message Details Window:** To open the Message Details window:

**SELECT**

**Tools → Message Queues** from the **pedb** main window to get the Application Message Queues window.

**SELECT**

the desired task button to get the Task Message Queue window.

**SELECT**

the desired message entry.

**HOLD**

the right mouse button to get the Message Data menu.

**SELECT**

**Message Details** from the Message Data menu.

This window displays details for a specific message. There are four basic formats to this window corresponding to point to point, send/receive, collective communication, and early arrival messages. The following figures show examples of each window.

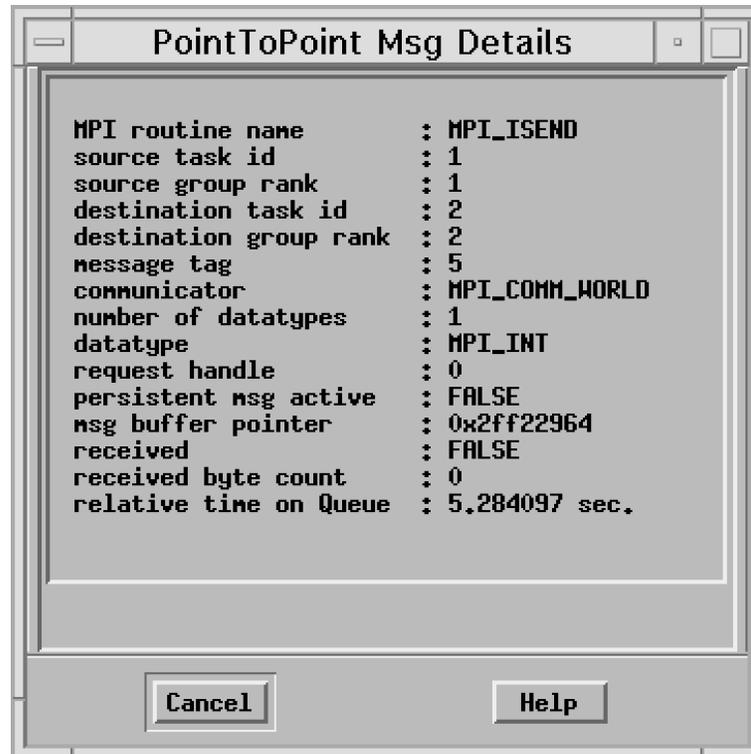


Figure 25. Point to Point Message Details window

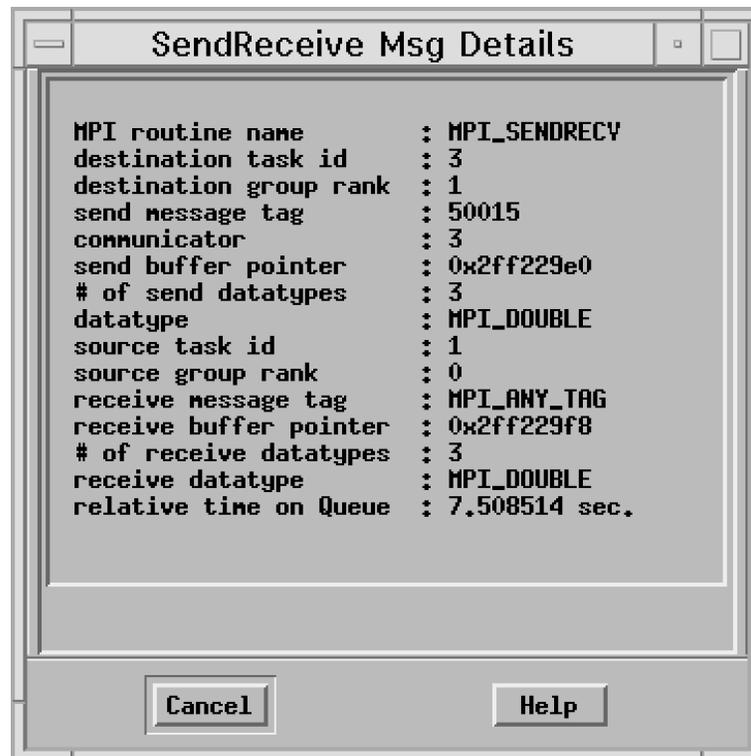


Figure 26. Send/Receive Message Details window



Figure 27. Collective Communications Details window

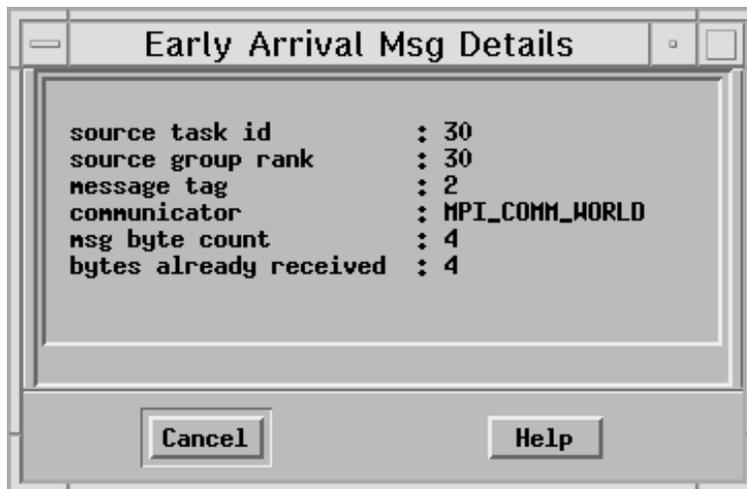


Figure 28. Early Arrival Message Details window

**Message Group Information Window:** To open the Message Group Information window:

**SELECT**

Tools → Message Queues from the **pedb** main window to get the Application Message Queues window.

**SELECT**

the desired task button to get the Task Message Queue window.

**SELECT**  
the desired message entry.

**HOLD**  
the right mouse button to get the Message Data menu.

**SELECT**  
**Group Info** from the Message Data menu.

This window displays information about the selected message followed by the task id, rank, and local communicator of the group members. The format of the window varies based on whether the communicator is an inter-communicator or an intra-communicator. If it is an intra-communicator, the window displays information about both the local and remote groups.

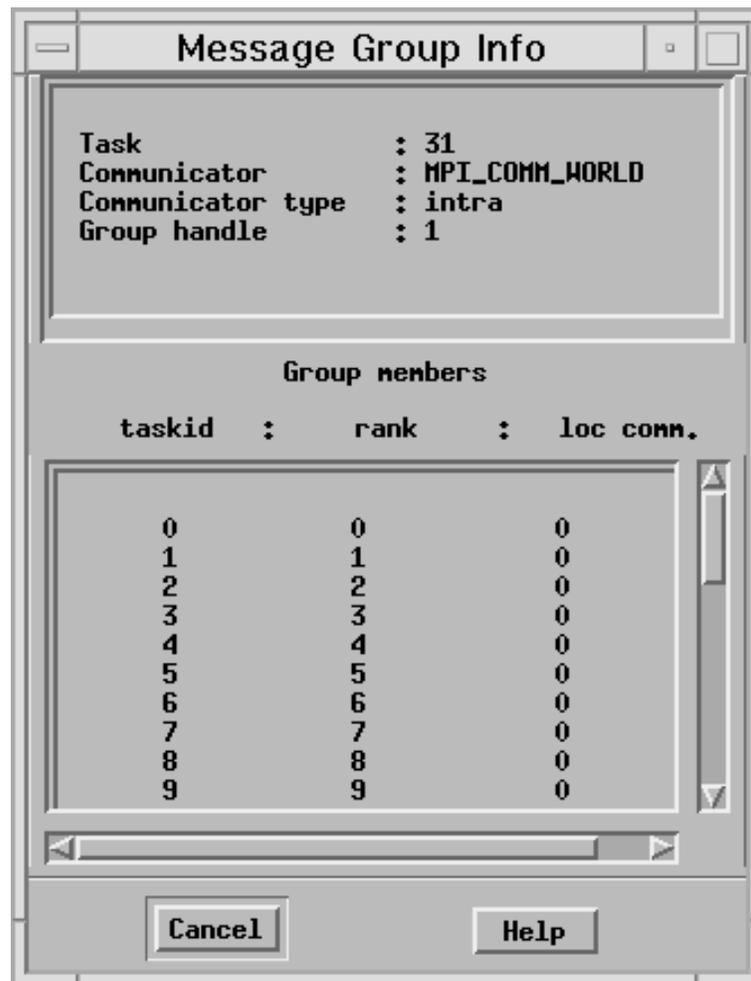


Figure 29. Message Group Information window

**Updating Message Queue Information:** Message queue information is retrieved from the task (executable) when it has been stopped by the debugger and is in **pedb** "debug" state. The task buttons at the bottom of the **pedb** main window indicate "debug" state. When the task is executed by the debugger, by stepping, etc., the message queue could potentially change. Therefore, it is necessary to update the message queue information when the task returns to "debug" state.

If the message queue debugging features are currently being used, the debugger automatically updates the message queue windows when the task returns to “debug” state after being executed. The procedure for updating is as follows:

- The Application Message Queues window is updated using the current option and filter settings.
- The tags in the Select Filters window are updated to list the current set of tags in use.
- Any Task Message Queue windows that are open for tasks in the current debugger context are updated with the current list of messages for the task. This is assuming the tasks are in “debug” state.
- Any Task Message Queue windows for tasks not in the current context, or not available, are not updated and are closed.
- Any Message Details or Message Group Information windows are closed. This is because there is no absolute way to determine if the message is still on the queue.

### 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 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 this manual 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 start-up.

#### **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** pull-down 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 pull-down and from buttons in the Find window. Accelerators <EscChar-f>, <EscChar-n>, <EscChar-p>, and <EscChar-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 pull-down 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 30. The Find window title will indicate if you are using **Find** from the source code window or the Variable Viewer.

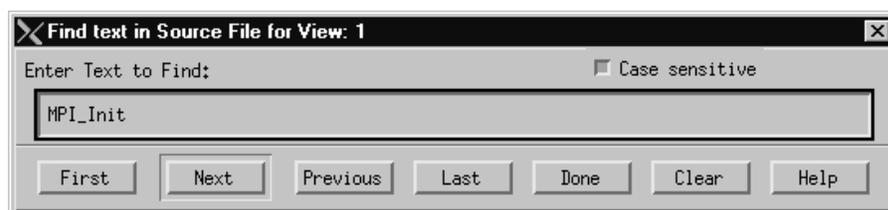


Figure 30. Find window

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 pull-down 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 online help information for using **Find**.

If the search fails, a message is displayed in the information area indicating the search direction and the actual string that are 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 164.

**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 10 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.

Table 10. 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.

Table 10. Default color scheme (continued)

Resource	Language element
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).

#### 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:*.

#### 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/lib/X11/app-defaults/Pedb* (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.

**Note:** If the **pedb** X defaults file is not defined, ask your System Administrator to install it. You find samples in */samples/pe/pedb/Pedb.ad*. For details, see *z/OS UNIX System Services Planning*.

**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 has not 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.

---

## Appendix A. Parallel Environment commands

This appendix contains the manual pages for the PE commands discussed throughout this book. Each manual page is organized into the sections listed below. The sections always appear in the same order, but some appear in all manual pages while others are optional.

### **Purpose**

Provides the name of the command described in the manual page, and a brief description of its purpose.

### **Format**

Includes a diagram that summarizes the command syntax, and provides a brief synopsis of its use and function. If you are unfamiliar with the typographic conventions used in the syntax diagrams, see “Document conventions” on page xii.

**Flags** Lists and describes any required and optional flags for the command.

**Usage** Describes the command more fully than the **Purpose** and **Format** sections.

### **Environment Variables**

Lists and describes any applicable environment variables.

### **Examples**

Provides examples of ways in which the command is typically used.

**Files** Lists and describes any files related to the command.

### **Related Information**

Lists commands, functions, file formats, and special files that are employed by the command, that have a purpose related to the command, or that are otherwise of interest within the context of the command.



- 136 error renaming temp file to file name
- 137 input file is empty
- 138 invalid block size
- 139 error allocation storage

## System Environment

### MP\_BLKSIZE

Specifies the data block size (in bytes) used to copy the data. Valid values are between 1 and 8,000,000 bytes. The default is 100,000 bytes.

## Examples

To copy a file from your current directory to the current directory on the first 8 systems specified in the myHosts.list file, enter:

```
mcp filename -procs 8 -hostfile myHosts.list
```

To copy a file from your current directory to the /tmp directory on the first 8 systems specified in the myHosts.list file, enter:

```
mcp filename /tmp -procs 8 -hostfile myHosts.list
```

To copy a file from your current directory to a different file name on the first 8 systems specified in the myHosts.list file, enter:

```
mcp filename /tmp/newfilename -procs 8 -hostfile myHosts.list
```



**mcpgather** is a POE program and, therefore, all POE options are available. You can set POE options with either command-line flags or environment variables. The number of nodes to copy the file from (**-procs**) is the POE option of most interest.

Return codes are:

- 129  
invalid number of arguments specified
- 130  
invalid option flag specified
- 131  
unable to resolve input file name(s)
- 132  
could not open input file for read
- 133  
no room on destination node's file system
- 134  
error opening file output file
- 135  
error creating output file
- 136  
error writing to output file
- 137  
MPI\_Send of data failed
- 138  
final MPI\_Send failed
- 139  
MPI\_Recv failed
- 140  
invalid block size
- 141  
error allocating storage
- 142  
total number of tasks must be greater than one

## System Environment

### MP\_BLKSIZE

Specifies the data block size (in bytes) used to copy the data. Valid values are between 1 and 8,000,000 bytes. The default is 100,000 bytes.

## Examples

1. You can copy a single file from all tasks into the destination directory. For example, enter:

```
mcpgather -a hello_world /tmp -procs 4
```

This will copy the file *hello\_world* (assuming it is a file and not a directory) from tasks 0 through 3 as to task 0:

## mcpgath

```
From task 0: /tmp/hello_world.0
From task 1: /tmp/hello_world.1
From task 2: /tmp/hello_world.2
From task 3: /tmp/hello_world.3
```

2. You can specify any number of files as source files. The destination directory must be the last item specified before any POE flags. For example:

```
mcpgath -a file1.a file2.a file3.a file4.a file5.a /tmp -procs 4
```

will take *file1.a* through *file5.a* from the local directory on each task and copy them back to task 0. All files specified must exist on all tasks involved. The file distribution will be as follows:

```
From Task 0: /tmp/file1.a.0
From Task 1: /tmp/file1.a.1
From Task 2: /tmp/file1.a.2
From Task 3: /tmp/file1.a.3
From Task 0: /tmp/file2.a.0
From Task 1: /tmp/file2.a.1
From Task 2: /tmp/file2.a.2
From Task 3: /tmp/file2.a.3
From Task 0: /tmp/file3.a.0
From Task 1: /tmp/file3.a.1
From Task 2: /tmp/file3.a.2
From Task 3: /tmp/file3.a.3
From Task 0: /tmp/file4.a.0
From Task 1: /tmp/file4.a.1
From Task 2: /tmp/file4.a.2
From Task 3: /tmp/file4.a.3
From Task 0: /tmp/file5.a.0
From Task 1: /tmp/file5.a.1
From Task 2: /tmp/file5.a.2
From Task 3: /tmp/file5.a.3
```

3. You can specify wildcard values to expand into a list of files to be gathered. For this example, assume the following distribution of files before calling **mcpgath**:

```
Task 0 contains file1.a and file2.a
Task 1 contains file1.a only
Task 2 contains file1.a, file2.a, and file3.a
Task 3 contains file4.a, file5.a, and file6.a
```

Enter:

```
mcpgath -a "file*.a" /tmp -procs 4
```

This will pass the wildcard expansion to each task, which will resolve into the list of locally existing files to be copied. This will result in the following distribution of files on task 0:

```
From Task 0: /tmp/file1.a.0
From Task 0: /tmp/file2.a.0
From Task 1: /tmp/file1.a.1
From Task 2: /tmp/file1.a.2
From Task 2: /tmp/file2.a.2
From Task 2: /tmp/file3.a.2
From Task 3: /tmp/file4.a.3
From Task 3: /tmp/file5.a.3
From Task 3: /tmp/file6.a.3
```

4. You can specify a directory name as the source, from which the files to be gathered are found. For this example, assume the following distribution of files before calling **mcpgath**:

```
Task 0 /test contains file1.a and file2.a
Task 1 /test contains file1.a only
Task 2 /test contains file1.a and file3.a
Task 3 /test contains file2.a, file4.a, and file5.a
```

Enter:

```
mcpgather -a /test /tmp -procs 4
```

This results in the following file distribution:

```
From Task 0: /tmp/file1.a.0  
From Task 0: /tmp/file2.a.0  
From Task 1: /tmp/file1.a.1  
From Task 2: /tmp/file1.a.2  
From Task 2: /tmp/file3.a.2  
From Task 3: /tmp/file2.a.3  
From Task 3: /tmp/file4.a.3  
From Task 3: /tmp/file5.a.3
```



quotes, otherwise they will be expanded locally on the task from where the command is issued, which may not produce the intended file name resolution.

**mcp<sub>scat</sub>** is a POE program and, therefore, all POE options are available. You can set POE options with either command-line flags or environment variables. The number of nodes to copy the file to (**-procs**) is the POE option of most interest.

Return codes are:

129	invalid number of arguments specified
130	invalid option flag specified
131	unable to resolve input file name(s)
132	could not open input file for read
133	no room on destination node's file system
134	error opening file output file
135	error creating output file
136	<b>MPI_Send</b> of data failed
137	final <b>MPI_Send</b> failed
138	<b>MPI_Recv</b> failed
139	failed opening temporary file
140	failed writing temporary file
141	error renaming temp file to filename
142	input file is empty (zero byte file size)
143	invalid blocksize
144	error allocating storage
145	number of tasks and files do not match
146	not enough memory for list of file names

## System Environment

### MP\_BLKSIZE

Specifies the data block size (in bytes) used to copy the data. Valid values are between 1 and 8,000,000 bytes. The default is 100,000 bytes.

## Examples

1. You can copy a single file to all tasks into the destination directory. For example, enter:

```
mcpscat filename /tmp -procs 4
```

This will take the file and distribute it to tasks 0 through 3 as */tmp/filename*.

2. You can specify any number of files as source files. The destination directory must be the last item specified before any POE flags. For example:

```
mcpscat file1.a file2.a file3.a file4.a file5.a /tmp -procs 4
```

will take *file1.a* through *file5.a* from the local directory and copy them in a round robin order to tasks 0 through 3 into */tmp*. The file distribution will be as follows:

## mcpscat

```
Task 0: /tmp/file1.a  
Task 1: /tmp/file2.a  
Task 2: /tmp/file3.a  
Task 3: /tmp/file4.a  
Task 0: /tmp/file5.a
```

3. You can specify the source files to copy in a file. For example:

```
mcpscat -f file.list /tmp -procs 4
```

will produce the same results as the previous example if as *file.list* contains five lines with the file names *file1.a* through *file5.a* in it.

4. You can specify wildcard values to expand into a list of files to be scattered.

Enter:

```
mcpscat "file*.a" /tmp -procs 4
```

Assuming Task 0 contains *file1.a*, *file2.a*, *file3.a*, *file4.a*, and *file5.a* in its home directory, this will result in a similar distribution as in the previous example.

5. You can specify a directory name as the source, from which the files to be scattered are found. Assuming */test* contains *myfile.a*, *myfile.b*, *myfile.c*, *myfile.d*, *myfile.f*, and *myfile.g* on Task 0, enter:

```
mcpscat /test /tmp -procs 4
```

This results in the following file distribution:

```
Task 0: /tmp/myfile.a  
Task 1: /tmp/myfile.b  
Task 2: /tmp/myfile.c  
Task 3: /tmp/myfile.d  
Task 0: /tmp/myfile.f  
Task 1: /tmp/myfile.g
```

## mpcc/mpCC

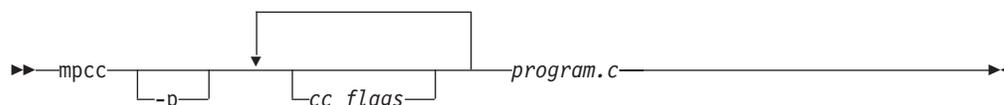
### Purpose

**mpcc** – Invokes a shell script to compile C programs.

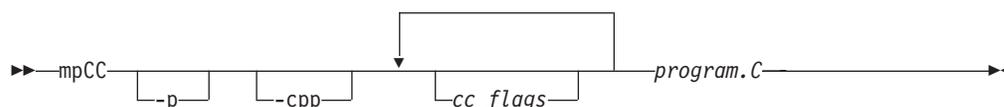
**mpCC** – Invokes a shell script to compile C++ programs.

### Format

#### mpcc



#### mpCC



The **mpcc** shell script compiles C programs and the **mpCC** compiles C++ programs. During compilation both are linking in the Partition Manager and the Message Passing Interface (MPI). By default **mpcc** and **mpCC** link to the MPI C-bindings.

**-p** Enables MPI nameshift profiling. Specify **-p** if your program profiles MPI functions using the name-shifted MPI interface (PMPI\_XXX).

#### **-cpp**

sets the compile flag `_MPI_CPP_BINDINGS`. This must be set in order to force the linkage of the MPI-2 C++-bindings. Option **-cpp** is only valid for command **mpCC**.

### Flags

Any of the compiler flags normally accepted by the **c89** command can also be used on **mpcc** as well as any **c++** flags can be used on **mpCC**. For a complete listing of these flag options, refer to the *z/OS UNIX System Services Command Reference*. Typical options to **mpcc** and **mpCC** include:

**-v** causes a pseudo-JCL to be written to stdout.

**-g** produces an object file with information in symbolic form. This information is used by symbolic debuggers like dbx, pdbx, and pedb (C-only).

**-o** names the executable.

#### **-I (upper-case i)**

names directories searched for additional include files.

## mpcc/mpCC

### Usage

The **mpcc** shell script calls the z/OS **c89** and **mpCC** calls the z/OS **c++** compiler. In addition, the Partition Manager and Message Passing Interface are automatically linked in. The script creates an executable that dynamically binds with the message passing libraries.

Compiler flags are passed by **mpcc** (**mpCC**) to the **c89** (**c++**) command, so any of the **c89** (**c++**) options can be used on the **mpcc** (**mpCC**) shell script. The POE and MPI library implementation is dynamically linked when you invoke the executable.

### System Environment

#### MP\_PREFIX

sets an alternate path to the scripts library. If not set or NULL, the standard path */usr* is used. If this environment variable is set, then all libraries are prefixed by *\$MP\_PREFIX*. For example, if **MP\_PREFIX** is set to */myPath*, the directory */myPath/include* is searched for header files, and */myPath/lib* for libraries.

#### MP\_VERBOSE

causes **mpcc** and **mpCC** to echo the compile command if set to *YES*.

### Examples

To compile the C program *xyz.c*, and produce an executable *xyz*, enter:

```
mpcc -o xyz xyz.c
```

To compile a C++ program *xyz*, enter:

```
mpCC -o xyz xyz.C
```

### Files

When you compile a program using **mpcc** or **mpCC**, the following files are automatically selected:

- /usr/lib/ppe.x* (definition side deck for the Parallel Environment for z/OS)
- /usr/lib/pe\_CEEBXITA.o* (assembler initialization user exit)
- /usr/lib/pe\_CEEBINT.o* (HLL initialization user exit)

### Related Information

commands **c89**, **c++**, **dbx**



## pdbx

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.

**-h** Writes the **pdbx** usage to STDERR then exits. This includes **pdbx** command-line syntax and a description of **pdbx** options.

**-I (upper-case i)**

Specifies 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** subcommand.)

**-tmpdir**

This *POE option* specifies the directory to which the individual startup files (*.pdbxinit.process\_id.task\_id*) are written for each **dbx** task. For more information on *.pdbxinit* see Table 3 on page 97 and "Reading subcommands from a command file" on page 122. If not set, and if its associated environment variable **MP\_TMPDIR** is not set, the default location is */tmp*.

## Usage

**pdbx** is the Parallel Environment's command line debugger for parallel programs. It is based, and built, on the debugging tool **dbx**.

**pdbx** supports most of the familiar **dbx** subcommands, as well as additional **pdbx** subcommands. For online help for the subcommands type **man pdbxsubcommand**, e.g. **man pdbxalias** for help on the **pdbx alias** subcommand.

To use **pdbx** for interactive debugging you first need to compile the program and set up the execution environment as you would to invoke a parallel program with the **poe** command. Your program should be compiled with the **-g** flag in order to produce an object file with symbol table references. For more information on the **-g** option, refer to *z/OS UNIX System Services Command Reference*.

**pdbx** maintains **dbx's** command line interface and subcommands. When you invoke **pdbx**, the **pdbx** command prompt displays to mark the start of a **pdbx** session.

When using **pdbx**, you should keep in mind that **pdbx** subcommands can either be context sensitive or context insensitive. In **pdbx**, context refers to a setting that controls which task(s) receive the subcommands entered at the **pdbx** command prompt. A default command context is provided which contains all tasks in your partition. You can, however, set the command context on a single task or a group of tasks you define. Context sensitive subcommands, when entered, only affect those tasks in the current command context. Context insensitive subcommands are not affected by the command context setting.

If you are already familiar with **dbx**, you should be aware that some **dbx** subcommands behave somewhat differently in **pdbx**. Be aware that:

- all the **dbx** subcommands are context sensitive in **pdbx**. If you use the **stop** subcommand, for example, it will only set breakpoints for the tasks in the current context. Tasks outside the current context are not affected.
- redirection from **dbx** subcommands is not supported.
- you cannot use the subcommands **clear**, **edit**, **multproc**, **prompt**, **run**, **rerun**, and the **sh** subcommand with no arguments.
- since **pdbx** runs in the Parallel Operating Environment, output from the parallel tasks may not be ordered. You can force task ordering, however, by setting the output mode to *ordered* using the **MP\_STDOUTMODE** environment variable or the **-stdoutmode** flag when invoking your program with **pdbx**.

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

- 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.

At the **pdbx** subset prompt, 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 STDIN is not given to the application. Most commands at the **pdbx** subset prompt produce information about the application and then produce another **pdbx** subset prompt. The exceptions are the **halt**, **back**, **on**, and **quit** commands. For more information, see “Context switch when blocked” on page 108.

## System Environment

Because the **pdbx** command runs in the Parallel Operating Environment, it interacts with the same environment variables associated with the **poe** command. See the **poe** command for a description of these environment variables. As indicated by the syntax statements, you are also able to specify **poe** command-line options when invoking **pdbx**. Using these options will override the setting of the corresponding environment variable, as is the case when invoking a parallel program with the **poe** command. Variables specific to **pdbx** are:

### HOME

During **pdbx** initialization, **pdbx** uses this environment variable to search for two special initialization files. First, **pdbx** searches for `.pdbxinit` in the user’s current directory. If the file is not found, **pdbx** checks the file `$HOME/.pdbxinit`.

### MP\_DBXPROMPTMOD

The **dbx** prompt `\n(dbx)` is used by **pdbx** as an indicator denoting that a **dbx** subcommand has completed. This environment variable can be used

## pdbx

to modify the prompt. Any value assigned to **MP\_DBXPROMPTMOD** will have a "." prepended and then be inserted in the `\n(dbx)` prompt between the "x" and the ").". This environment variable is needed in rare situations when the string `\n(dbx)` is present in the output of the application being debugged. For example, if **MP\_DBXPROMPTMOD** is set to `unique157`, the prompt would be `\n(dbx.unique157)`.

### MP\_TMPDIR

This environment variable specifies the directory to which the individual startup files (`.pdbxinit.process_id.task_id`) are written for each **dbx** task. This is frequently local, but may be a shared directory. If not set, and if its associated command-line flag **tmpdir** is not used, the default location is `/tmp`.

### MP\_DEBUG\_INITIAL\_STOP

This environment variable redefines the initial stop point in **pdbx** (overriding the stop in main). It can be set to `sourcefile:linenumber`, where `sourcefile` is a file containing source code of the program to be executed. Typically, the source file name ends with the `.c` or `.C` suffix. `linenumber` is a line number in this file. This line must contain executable code, not data declarations. It cannot be a comment, blank, or continuation line.

If no `linenumber` is specified (and the colon is omitted), the `sourcefile` field is taken to be a function or subroutine name, and a "stop in" is performed on entry to the function.

If **MP\_DEBUG\_INITIAL\_STOP** is undefined, the default stop location will be the first executable line in the function main.

## Examples

To start **pdbx**, first set up the execution environment as you would for the **poe** command, and then enter:

```
pdbx
```

After initialization, you should see the prompt:

```
pdbx(a11)
```

## Files

`.pdbxinit` (Initial commands for **pdbx** in `./` or `$HOME`)

`.pdbxinit.process_id.task_id` (Initial commands for the individual **dbx** tasks)

For more information on `.pdbxinit` see Table 3 on page 97 and "Reading subcommands from a command file" on page 122.

**Note:** The following temporary files are created during the execution of **pdbx** in attach mode:

- `/tmp/.pdbx.<poe-pid>.host.list` - a temporary host list file containing information needed to attach to tasks on remote nodes.
- `/tmp/.pdbx.<pdbx-pid>.menu` - a temporary file to hold the attach task menu. Both of these files are removed before the debugger exits.

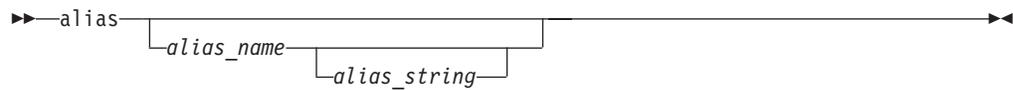
## Related Information

commands **mpcc**, **mpCC** and **dbx** For information on **dbx**, see *z/OS UNIX System Services Command Reference*.

**Note:** In order to display the description of a `pdbx` subcommand you must prefix the subcommand with `pdbx`. For example, to view the description of the `pdbx alias` subcommand, you must enter: **`man pdbxalias`**.

## pdbx alias subcommand

### Format



The **alias** subcommand creates aliases for **pdbx** subcommands. The *alias\_name* parameter is the alias being created. The *alias\_string* is the **pdbx** subcommand for which you wish you define an alias, and is a single **pdbx** subcommand. If used without parameters, the **alias** subcommand displays all current aliases. If only *alias\_name* is specified, it lists the alias name and the alias string that is assigned to it. This subcommand is context insensitive.

A number of default aliases are provided by **pdbx**. They are:

<b>t</b>	where
<b>j</b>	status
<b>st</b>	stop
<b>s</b>	step
<b>x</b>	registers
<b>q</b>	quit
<b>p</b>	print
<b>n</b>	next
<b>m</b>	map
<b>l</b>	list
<b>h</b>	help
<b>d</b>	delete
<b>c</b>	cont
<b>th</b>	thread
<b>mu</b>	mutex
<b>cv</b>	condition
<b>active</b>	tasks
<b>threads</b>	thread

Apart from these, aliases are only known during the current **pdbx** session. They are not saved between **pdbx** sessions, and are lost upon exiting **pdbx**.

**Note:** One method for reusing aliases is to define them in *.pdbxinit* to allow them to be created for each **pdbx** execution. The default aliases are available after the partition has been loaded.

Aliases can also be removed using the **unalias** subcommand for the **pdbx** command.

1. If you have two task groups defined in your **pdbx** session called "master" and "workers", and you wish to define aliases to easily qualify each, enter:

```
alias mas on master
alias w on workers
```

This will allow you to switch the command context between the master and workers groups by typing:

```
mas
```

to switch context to the “master” group, or:

```
w
```

to switch context to the “workers” group.

2. To display the string that has been defined for the alias “p”, enter:

```
alias p
```

3. To list all aliases currently defined, enter:

```
alias
```

Related to this subcommand is the **pdbx unalias** subcommand.

---

## pdbx assign subcommand

### Format

▶▶—assign—<variable>—=—<expression>—◀◀

The **assign** subcommand assigns the value of an expression to a variable.

1. To assign a value of 5 to the x variable:  
`pdbx(all) assign x = 5`
2. To assign the value of the y variable to the x variable:  
`pdbx(all) assign x = y`
3. To assign the character value 'z' to the z variable:  
`pdbx(all) assign z = 'z'`
4. To assign the "Hello World" string to a character pointer Y:  
`pdbx(all) assign Y = "Hello World"`
5. To disable type checking, activate the set variable \$unsafeassign:  
`pdbx(all) set $unsafeassign`

---

## pdbx attach subcommand

### Format

```
▶▶ attach [all | <task_list>] ▶▶
```

The **attach** subcommand is used to attach the debugger to some or all the tasks of a given **poe** job.

Individual tasks are separated by spaces. A range of tasks may be separated by a dash or a colon. For example, the command **attach 2 4 5-7** would mean to attach to tasks 2,4,5,6, and 7.

---

## pdbx back subcommand

### Format

▶▶—back—◀◀

The **back** command returns you to a **pdbx** prompt when you were already at a **pdbx** subset prompt. You can use the command if you want the application to continue as it was before <EscChar-c> was issued. Also, you can use it at the **pdbx** subset prompt if all of the nodes are checked into “debug ready” state, and you want to do full **pdbx** processing.

The **back** command is only valid at the **pdbx** subset prompt.

---

## pdbx case subcommand

### Format



The **case** subcommand changes how **pdbx** interprets symbols. Use this command if a symbol needs to be interpreted in a way not consistent with the current language.

Entering the **case** subcommand with no parameters displays the current case mode. The parameters include:

**default**

Varies with the current language.

**mixed** Causes symbols to be interpreted as they actually appear.

**lower** Causes symbols to be interpreted as lowercase.

**upper** Causes symbols to be interpreted as uppercase.

---

## pdbx catch subcommand

### Format



The **catch** subcommand with no arguments prints all signals currently being caught. If a signal is specified, **pdbx** will trap the signal before it is sent to the program. This is useful when the program being debugged has signal handlers.

When the program encounters a signal that is being caught to the debugger, a message stating which signal was detected is shown, and the **pdbx** prompt is displayed. To have the program continue and process the signal, issue the **cont** subcommand with the **signal** option. Other execution control commands and the **cont** subcommand without the **signal** option will cause the program to behave as if it had never encountered the signal.

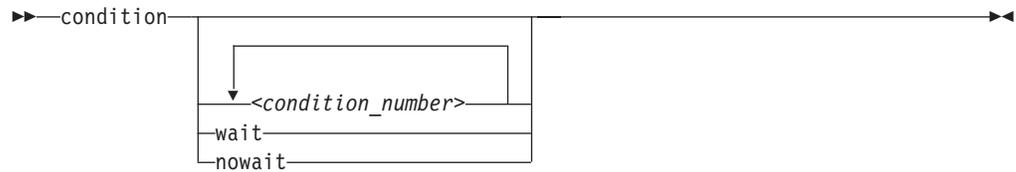
A signal may be specified by number or name. Signal names are by default case insensitive and the "SIG" prefix is optional.

By default all signals are caught except SIGHUP, SIGKILL, SIGALRM, SIGCHLD, SIGDUMP and SIGIO.

Related to this subcommand are the **ignore** and **cont** subcommands.

## pdbx condition subcommand

### Format



The **condition** subcommand displays the current state of all known conditions in the process. Condition variables to be listed can be specified through the `<condition_number>` parameters, or all condition variables will be listed. Users can also choose to display only condition variables with or without waiters by using the **wait** or **nowait** options.

The information listed for each condition is as follows:

**cv**        Indicates the symbolic name of the condition variable, in the form `$condition_number`.

**obj\_addr**

Indicates the memory address of the condition variable.

**num\_wait**

Indicates the number of threads waiting on the condition variable.

**waiters**

Lists the user threads which are waiting on the condition variable.

Related to this subcommand is the **thread** subcommand.

---

## **pdbx cont subcommand**

### **Format**



The **cont** subcommand allows execution to continue from where the program last stopped, until either the program finishes or another breakpoint is reached. If a signal is specified, it is given to the program, and the process continues as though it received the signal. If a signal is not specified, the process continues as though it had not been stopped and did not get the signal.

Related to this subcommand are the **catch**, **ignore**, **step**, **stepi**, **next**, and **nexti** subcommands.

---

## pdbx dbx subcommand

### Format

▶▶—*dbx*—*dbx\_subcommand*—▶▶

The **dbx** subcommand is context sensitive and will pass the specified *dbx\_subcommand* directly to the **dbx** running on each task in the current context with no **pdbx** intervention. The specified *dbx\_subcommand* can be any valid **dbx** subcommand.

**Note:** The **pdbx** command uses **dbx** to access tasks on individual nodes. In many cases, **pdbx** saves and requires its own state information about the tasks. Some **dbx** commands will circumvent the ability of **pdbx** to maintain accurate state information about the tasks being debugged. Therefore, use the **dbx** subcommand with caution. In general, **dbx** subcommands used to display information will have no adverse side effects. The **dbx** subcommands **clear**, **edit**, **multproc**, **prompt**, **run**, **rerun**, and the **sh** subcommand with no arguments are currently unsupported under **pdbx** and should not be used.

To display the events that the **dbx** running as task 1 recognizes, enter:  
on 1 dbx status

Related to this subcommand is the **dbx** command.

## pdbx delete subcommand

### Format



The **delete** subcommand removes events (breakpoints and tracepoints) of the specified event numbers. An event list can be specified in the following manner. 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 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 context from which the event was created. Event numbers are unique to the context in which they were set. Keep in mind that, in order to remove an event, the context must be on the appropriate task or task group.

Assume the command context is set 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:

```

on 1
delete 0
on all
delete 0,1

```

OR

```

on 1
delete 0
on all
delete *

```

OR

```

delete all

```

Related to this subcommand are the **pdbx status**, **stop**, and **trace** subcommands.

---

## **pdbx detach subcommand**

### **Format**

▶▶—detach—◀◀

The **detach** subcommand detaches **pdbx** from all tasks that were attached. This subcommand causes the debugger to exit but leaves the **poe** application running.

---

## **pdbx dhelp subcommand**

### **Format**

▶▶—dhelp—▶▶  
└─<dbx\_subcommand>┘

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

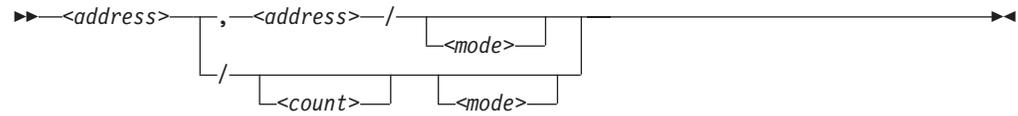
If you type **dhelp** with an argument, information will be displayed about that command.

**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.

Related to this subcommand is the **pdbx help** subcommand.

## pdbx display memory subcommand

### Format



The **display memory** subcommand, which does not have a keyword to initiate the command, displays a portion of memory controlled by the address(es), count(s) and mode(s) specified.

If an address is specified, the display contents of memory at that address is printed. If more than one address or count locations are specified, display contents of memory starting at the first *<address>* up to the second *<address>* or until *<count>* items are printed. If the address is ".", the address following the one most recently printed is used. The mode specifies how memory is to be printed. If it is omitted the previous mode specified is used. The initial mode is "X".

The following modes are supported:

- i** print the machine instruction
- d** print a short word in decimal
- D** print a long word in decimal
- o** print a short word in octal
- O** print a long word in octal
- x** print a short word in hexadecimal
- X** print a long word in hexadecimal
- b** print a byte in octal
- c** print a byte as a character
- h** print a byte in hexadecimal
- s** print a string (terminated by a null byte)
- f** print a single precision float number
- g** print a double precision float number
- q** print a quad precision float number

---

## pdbx down subcommand

### Format

▶▶—down—count—▶▶

The **down** subcommand moves the current function down the stack the number of levels specified by *count*. The current function is used for resolving names. The default for the *count* parameter is one.

The **up** and **down** subcommands can be used to navigate through the call stack. Using these subcommands to change the current function also causes the current file and local variables to be updated to the chosen stack level.

Related to this subcommand are the **up**, **print**, **dump**, **func**, **file**, and **where** commands.



---

## pdbx file subcommand

### Format

▶▶—file—▶▶  
    └──file──┘

The **file** subcommand changes the current source file to the file specified by the *file* parameter. It does not write to that file. The *file* parameter can specify a full path name to the file. If the parameter does not specify a path, the **pdbx** program tries to find the file by searching the use path. If the parameter is not specified, the **file** subcommand displays the name of the current source file. The **file** subcommand also displays the full or relative path name of the file if the path is known.

Related to this subcommand is the **func** subcommand.

---

## pdbx func subcommand

### Format

►►—func—procedure—►►

The **func** command changes the current function to the procedure or function specified by the *procedure* parameter. If the *procedure* parameter is not specified, the default current function is displayed. Changing the current function implicitly changes the current source file to the file containing the new function. The current scope used for name resolution is also changed.

Related to this subcommand is the **file** subcommand.

## **pdbx goto subcommand**

### **Format**

▶▶ goto [ "<file name>" -: ] <line\_number> ▶▶

The **goto** subcommand causes the specified source line to be run next. Normally, the source line must be in the same function as the current source line. To override this restriction, use the **set** subcommand with the **\$unsafegoto** flag.

---

## **pdbx gotoi subcommand**

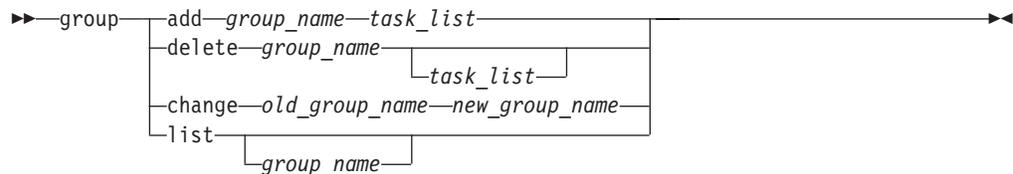
### **Format**

▶▶—*gotoi*—*address*—————▶▶

The **gotoi** subcommand changes the program counter address to the address specified by the *address* parameter.

## pdbx group subcommand

### Format



The **group** subcommand groups individual tasks under a common name for easier setting of command context. It can add or delete a group, add or delete tasks from a group, change the name of a group, list the tasks in a group, or list all groups. This subcommand is context insensitive.

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.

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. Individual task identifiers and ranges can also be combined in creating the desired *task\_list*.

**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.

The **add** action adds one or more tasks to a new or existing task group. The *task\_list* specified is a list of task identifiers to be included in the new or existing group.

The **delete** action deletes an existing task group, or deletes one or more tasks from an existing task group. The *task\_list*, if specified, is a list of task identifiers to be deleted from the new or existing group.

The **change** action changes the name of a task group from *old\_group\_name* to *new\_group\_name*.

The **list** action displays the task members for the *group\_name* specified, or for all task groups. The task identifiers will be followed by a one-letter status indicator.

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.

Consider an application running as five tasks numbered 0 through 4.

1. To create a task group "first" containing task 0, enter:

```
group add first 0
```

The **pdbx** debugger responds with:

```
1 task was added to group "first".
```

2. To create a task group "rest" containing tasks 1 through 4, enter:

```
group add rest 1:4
```

The **pdbx** debugger responds with:

```
4 tasks were added to group "rest".
```

3. To change the name of the default group "all" to "johnny", enter:

```
group change all johnny
```

The **pdbx** debugger responds with:

```
Group "all" has been renamed to "johnny"
```

4. To list all of the groups and the tasks they contain, enter:

```
group list
```

The **pdbx** debugger responds with:

```
johnny  0:D  1:D  2:D  3:D  4:D
first   0:D
rest    1:D  2:D  3:D  4:D
```

5. To delete the group "first", enter:

```
group delete first
```

To delete members 1, 2 and 3 from group "rest", enter:

```
group delete rest 1 2 3
```

or

```
group delete rest 1-3
```

The **pdbx** debugger responds with:

```
Task: 1 was successfully deleted from group "rest".
```

```
Task: 2 was successfully deleted from group "rest".
```

```
Task: 3 was successfully deleted from group "rest".
```

6. To list all of the groups and the tasks they contain, enter:

```
group list
```

The **pdbx** debugger responds with:

```
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
```

Related to this subcommand is the **pdbx on** subcommand.

---

## pdbx halt subcommand

### Format

▶▶ halt  ▶▶

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 **<EscChar-c>**. This command works at the **pdbx** prompt and **pdbx** subset prompt. If you specify “all” with the 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.

Related to this subcommand are the **pdbx tasks** and **group list** subcommands.

---

## **pdbx help subcommand**

### **Format**

▶▶ help [subcommand] [topic] ▶▶

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.

Related to this subcommand is the **pdbx dhelp** subcommand

---

## pdbx hook subcommand

### Format

▶—hook—◀

The **hook** subcommand allows you to reestablish control over all tasks in the current command context that have been unhooked using the **unhook** subcommand. This subcommand is context sensitive.

1. To reestablish control over task 2 if it has been unhooked, enter:  
on 2 hook  
  
or  
on 2  
hook
2. To reestablish control over all unhooked tasks in the task group “rest”, enter:  
on rest hook  
  
or  
on rest  
hook

Listing the members of the task group “all” using the **list** action of the **group** subcommand will allow you to check which tasks are hooked and which are unhooked. Enter:

```
group list all
```

The **pdbx** debugger will display a list similar to the following:

```
0:D   1:U   2:D   3:D
```

Tasks marked with the letter D next to them are “debug ready”, hooked tasks. In this case, tasks 0, 2, and 3 are “debug ready”. Tasks marked with the letter U are unhooked. In this case, task 1 is unhooked.

Related to this subcommand are the **dbx detach** subcommand and the **pdbx unhook** subcommand.

---

## pdbx ignore subcommand

### Format

```
▶▶ ignore [ <signal_number> | <signal_name> ] ▶▶
```

The **ignore** subcommand with no arguments prints all signals currently being ignored. If a signal is specified, **pdbx** stops trapping the signal before it is sent to the program.

A signal may be specified by number or name. Signal names are by default case insensitive and the “SIG” prefix is optional.

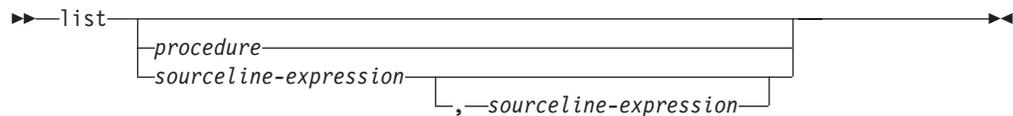
All signals except SIGHUP, SIGKILL, SIGALRM, SIGCHLD, and SIGIO are trapped by default.

The **pdbx** debugger cannot ignore the SIGTRAP signal if it comes from a process outside of the program being debugged.

Related to this subcommand is the **catch** subcommand.

## pdbx list subcommand

### Format



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.

To list the lines 1 through 10 in the current file, enter:

```
list 1,10
```

To list 10, or *\$listwindow*, lines around the main procedure, enter:

```
list main
```

To list 11 lines around the current line, enter:

```
list $-5,$+5
```

To list the next source line to be executed, issue:

```
pdbx(all) list $
0:  4      char johnny = 'h';
1:  4      char johnny = 'h';
```

To just show 1 task, since both are at the same source line:

```
pdbx(all) on 0 list $
0:  4      char johnny = 'h';
```

To create an alias to list just task 0:

```
pdbx(all) alias l0 on 0 list
```

To list line 5:

```
pdbx(all) l0 5
0: 5 char jessie = 'd';
```

To list lines around the procedure sub:

```
pdbx(all) l0 sub
0: 21
0: 22 /* return ptr to sum of parms, calc and sub1 */
0: 23 int *sub(char *s, int a, int k)
0: 24 {
0: 25     int *tmp;
0: 26     int it = 0;
0: 27     int i, j;
0: 28
0: 29     /* test calc */
0: 30     i = 1;
0: 31     j = i*2;
```

To change the next line to be listed to line 25:

```
pdbx(all) move 25
```

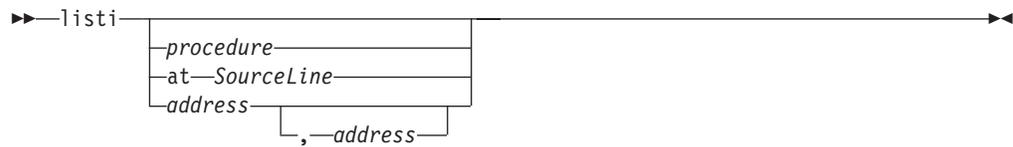
To list the next line to be listed minus two:

```
pdbx(all) l0 @-2
0: 23 int *sub(char *s, int a, int k)
```

Related to this subcommand is the **dbx list** subcommand.

## pdbx listi subcommand

### Format



The **listi** subcommand displays a specified set of instructions from the current program counter, depending on whether you specify procedure, source line, or address.

The **listi** subcommand with the *procedure* parameter lists instructions from the beginning of the specified procedure until the **list** window is filled.

Using the **at** *SourceLine* flag with the **listi** subcommand displays instructions beginning at the specified source line and continuing until the **list** window is filled. The *SourceLine* variable can be specified as an integer, or as a file name string followed by a : (colon) and an integer.

Specifying a beginning and ending address with the **listi** subcommand, using the *address* parameters, displays all instructions between the two addresses.

If the **listi** subcommand is used without flags or parameters, the next **\$listwindow** instructions are displayed. To change the current size of the **list** window, use the **set \$listwindow=Value** command.

---

## pdbx load subcommand

### Format

►►—load—*program*——*program\_options*—►►

The **load** subcommand loads the specified application *program* to be debugged on the task(s) in the current context. You can optionally specify *program\_options* to be passed to the application program. **pdbx** will look for the program in the current directory unless a relative or absolute path name is specified. The **load** subcommand is context sensitive. All tasks in the partition must have an application program loaded before other context sensitive subcommands can be issued. This subcommand enables you to individually or selectively load programs. If you wish to load the same program on all tasks in the partition, the name of the program can be passed as an argument to the **pdbx** command at startup.

To load the program “mpprob1” on all tasks in the current context, enter:  
load mpprob1

## **pdbx map subcommand**

### **Format**

▶▶ map ◀◀

The **map** subcommand displays characteristics for each loaded portion of the application. This information includes the name, text origin, text length, data origin, and data length for each loaded module.

## pdbx mutex subcommand

### Format



The **mutex** subcommand displays the current status of all known mutual exclusion locks in the process. Mutexes to be listed can be specified through the `<number>` parameter, or all mutexes will be listed. Users can also choose to display only locked or unlocked mutexes by using the **lock** or **unlock** options.

The information listed for each mutex is as follows:

**mutex** Indicates the symbolic name of the mutex, in the form `$mmutex_number`.

**type** Indicates the type of the mutex: non-rec (non recursive), recursi (recursive) or fast.

**obj\_addr** Indicates the memory address of the mutex.

**lock** Indicates the lock state of the mutex: yes if the mutex is locked, no if not.

**owner** If the mutex is locked, indicates the symbolic name of the user thread which holds the mutex.

Related to this subcommand is the **thread** subcommand.

---

## pdbx next subcommand

### Format

```
▶▶ next _____ ◀◀  
      └──number──┘
```

The **next** subcommand runs the application program up to the next source line. The *number* parameter specifies the number of times the subcommand runs. If the *number* parameter is not specified, **next** runs once only.

The difference between this and the **step** subcommand is that if the line contains a call to a procedure or function, **step** will stop at the beginning of that block, while **next** will not.

If you use the **next** subcommand in a multi-threaded application program, all the user threads run during the operation, but the program continues execution until the running thread reaches the specified source line. By default, breakpoints for all threads are ignored during the **next** command. This behavior can be changed using the **\$catchbp** set variable. If you wish to step the running thread only, use the **set** command to set the variable *\$hold\_next*. Setting this variable may result in deadlock, since the running thread may wait for a lock held by one of the blocked threads.

Related to this subcommand are the **nexti**, **step**, **stepi**, **return**, **cont**, and **set** subcommands.

---

## pdbx nexti subcommand

### Format

```
nexti [number]
```

The **nexti** subcommand runs the application program up to the next instruction. The *number* parameter specifies the number of times the subcommand will run. If the *number* parameter is not specified, **nexti** runs once only.

The difference between this and the **stepi** subcommand is that if the line contains a call to a procedure or function, **stepi** will stop at the beginning of that block, while **nexti** will not.

If you use the **nexti** subcommand in a multi-threaded application program, all the user threads run during the operation, but the program continues execution until the running thread reaches the specified machine instruction. If you wish to step the running thread only, use the **set** command to set the variable *\$hold\_next*. Setting this variable may result in deadlock since the running thread may wait for a lock held by one of the blocked threads.

Related to this subcommand are the **next**, **step**, **stepi**, **return**, **cont**, and **set** subcommands.

## pdbx on subcommand

### Format

```

▶▶ on [group_name] [task_id] [subcommand] ▶▶

```

The **on** subcommand sets the current command context used to direct subsequent subcommands at a specific task or group of tasks. The context can be set on a task group (by specifying a *group\_name*) or on a single task (by specifying a *task\_id*).

When a context sensitive *subcommand* is specified, it is directed to the given context without changing the current command context. Thus, specifying the optional *subcommand* enables you to temporarily deviate from the command context.

**Note:** The **pdbx** prompt will be presented after all of the tasks in the temporary context have completed the specified command. It is possible using **<EscChar-c>** followed by the **back** or the **on** command to issue further **pdbx** commands in the original context.

By using the **on** and **group** subcommands, the number of subcommands issued and the amount of debug data displayed can be tailored to manageable amounts.

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. Thus, 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 states are in the *debug ready* state, the **pdbx** prompt is displayed immediately.

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 108 for more information on the **pdbx** subset prompt.

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

Assume you have an application running as 15 tasks, and the output of the **group list** subcommand lists the existing task groups as:

```

all          0:D    1:U    2:D    3:D    4:D    5:D    6:U    7:D
             8:D    9:D   10:R   11:R   12:R   13:U   14:U
johnny       0:D
jessica      2:D    3:D    8:D
un           1:U    6:U   13:U   14:U
run          10:R   11:R   12:R
deb          2:D    3:D    4:D    5:D    8:D    9:D

```

- To add a breakpoint for task 0, enter:  
on johnny stop at 31

The **pdbx** debugger responds with:

```
johnny:[0] stop at "ring.c":31
```

- To add breakpoints for all of the tasks in the task group “jessica”, enter:  
on jessica stop in ring

The **pdbx** debugger responds with:

```
jessica:[0] stop in ring
```

- To switch the current context to the task group “johnny”, enter:  
on johnny

The **pdbx** debugger responds with the prompt:

```
pdbx(johnny)
```

- To add a conditional breakpoint for all tasks in the current context, enter:  
stop at 48 if len < 1

The **pdbx** debugger responds with:

```
johnny:[1] stop at "ring.c":48 if len < 1
```

- To view the events that have been set on the task group “jessica”, enter:  
on jessica status

The **pdbx** debugger responds with:

```
jessica:[0] stop in ring
```

- To add a tracepoint for task 2, enter:  
on 2

The **pdbx** debugger responds with the prompt:

```
pdbx(2)
```

Then, enter:

```
trace 57
```

The **pdbx** debugger responds with:

```
2:[0] trace "ring.c":57
```

- To view all of the events that have been set, enter:  
status all

The **pdbx** debugger responds with:

```
2:[0] trace "ring.c":57
johnny:[0] stop at "ring.c":48
johnny:[1] stop at "ring.c":56 if len < 1
jessica:[0] stop in ring
```

Related to this subcommand is the **pdbx group** subcommand.

---

## **pdbx print subcommand**

### **Format**

▶▶—print—*expression*————▶▶

The **print** subcommand prints the value of a list of expressions, specified by the *expression* parameters.

To display the value of x and the value of y shifted left two bits, enter:

```
print x, y << 2
```

Related to this subcommand are the **dbx assign** and the **pdbx set** subcommands.

---

## **pdbx quit subcommand**

### **Format**

▶▶—quit—◀◀

The **quit** subcommand terminates all program tasks, and ends the **pdbx** debugging session. The **quit** subcommand is context insensitive and has no parameters.

Quitting a debug session in attach mode causes the debugger and all the members of the original **poe** application partition to exit.

To exit the **pdbx** debug program, enter:

quit

---

## pdbx registers subcommand

### Format

▶—registers—◀

The **registers** subcommand displays the values of general purpose registers, system control registers, floating-point registers, and the current instruction register, such as the program status word (PSW) for z/OS.

Registers can be displayed or assigned to individually by using the following predefined register names:

**\$r0 through \$r31**

for the general purpose registers.

**\$fr0 through \$fr31**

for the floating point registers.

By default, the floating-point registers are not displayed. To display the floating-point registers, use the **unset \$noflregs** command.

**Notes:**

1. The register value may be set to the 0xdeadbeef hexadecimal value. The 0xdeadbeef hexadecimal value is an initialization value assigned to general purpose registers at process initialization.
2. The **registers** command cannot display registers if the current thread is in kernel mode.

---

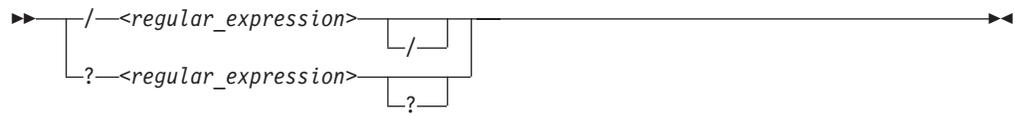
## **pdbx return subcommand**

### **Format**

▶▶—return—└──*procedure*──┘▶▶

The **return** subcommand causes the program to execute until a return to the procedure, specified by the *procedure* parameter, is reached. If the *procedure* parameter is not specified, execution ceases when the current procedure returns.

---

**pdbx search subcommand****Format**

The search forward (`/`) or search backward (`?`) subcommands allow you to search in the current source file for the given `<regular_expression>`. Both forms of search wrap around. The previous regular expression is used if no regular expression is given to the current command.

---

## **pdbx set subcommand**

### **Format**

```
▶▶ set [variable] [variable=expression] ▶▶
```

The **set** subcommand defines a value for the set variable. The value is specified by the *expression* parameter. The set variable is specified by the *variable* parameter. The name of the variable should not conflict with names in the program being debugged. A variable is expanded to the corresponding expression within other commands. If the **set** subcommand is used without arguments, the currently set variables are displayed.

Related to this subcommand is the **unset** subcommand.

## pdbx sh subcommand

### Format

▶▶—sh—*<command>*—▶▶

The **sh** subcommand passes the command specified by the *command* parameter to the shell on the remote task(s) for execution. The **SHELL** environment variable determines which shell is used. The default is the omvs shell (sh).

**Note:** The **sh** subcommand with no arguments is not supported.

To run the **ls** command on all tasks in the current context, enter:

```
sh ls
```

To display contents of the *foo.dat* data file on task 1, enter:

```
on 1 cat foo.dat
```

---

## pdbx skip subcommand

### Format

▶▶ skip number ▶▶

The **skip** subcommand continues execution of the program from the current stopping point, ignoring the next breakpoint. If a *number* variable is supplied, **skip** ignores that next amount of breakpoints.

Related to this subcommand is the **cont** subcommand.

---

## **pdbx source subcommand**

### **Format**

▶—source—*commands\_file*—▶

The **source** subcommand reads **pdbx** subcommands from the specified *commands\_file*. The *commands\_file* should reside on the node where **pdbx** was issued and can contain any commands that are valid on the **pdbx** command line. The **source** subcommand is context insensitive.

To read **pdbx** subcommands from a file named "jessica", enter:

```
source jessica
```

Related to this subcommand is the **dbx source** subcommand.

## pdbx status subcommand

### Format

```
▶ status [all] ▶
```

A list of **pdbx** events (breakpoints and tracepoints) can be displayed by 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.

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

Assume the following commands have been issued, setting various breakpoints and tracepoints.

```
on all
stop at 19
trace 21
on 0
trace foo at 21
on 1
stop in func
```

To display a list of breakpoints and tracepoints for tasks in the current “task 1” context, enter:

```
status
```

The **pdbx** debugger responds with lines of status like:

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

Notice that the status from the “task 0” context does not get displayed since the context is on “task 1”. Also notice that event 0 is unique for the “task 1” context and the “group all” context.

To see an example of **status all**, enter:

```
status all
```

The **pdbx** debugger responds with:

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

Related to this subcommand are the **pdbx stop**, **trace**, and **delete** subcommands.

---

## pdbx step subcommand

### Format

▶▶ step  ▶▶

The **step** subcommand runs source lines of the program. You specify the number of lines to be executed with the *number* parameter. If this parameter is omitted, the default is a value of 1.

The difference between this and the **next** subcommand is that if the line contains a call to a procedure or function, **step** will enter that procedure or function, while **next** will not.

If you use the **step** subcommand on a multi-threaded program, all the user threads run during the operation, but the program continues execution until the interrupted thread reaches the specified source line. By default, breakpoints for all threads are ignored during the **step** command. This behavior can be changed using the **\$catchbp** set variable.

If you wish to step the interrupted thread only, use the **set** subcommand to set the variable *\$hold\_next*. Setting this variable may result in debugger induced deadlock, since the interrupted thread may wait for a lock held by one of the threads blocked by *\$hold\_next*.

**Note:** Use the *\$stepignore* variable of the **set** subcommand to control the behavior of the **step** subcommand. The *\$stepignore* variable enables **step** to step over large routines for which no debugging information is available.

Related to this subcommand are the **stepi**, **next**, **nexti**, **return**, **cont**, and **set** commands.

---

## pdbx stepi subcommand

### Format

▶▶—stepi—┐  
          └─*number*—┘▶▶

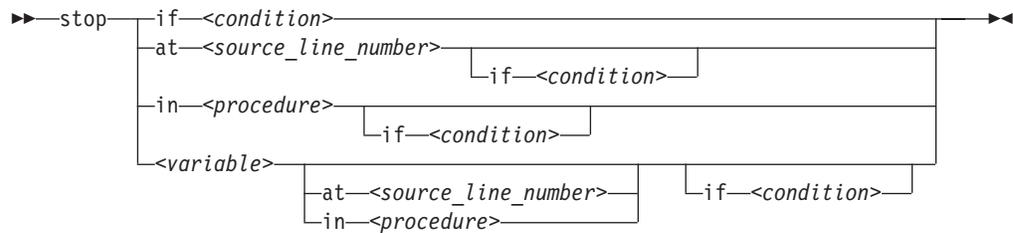
The **stepi** subcommand runs instructions of the program. You specify the number of instructions to be executed with the *number* parameter. If the parameter is omitted, the default is 1.

If used on a multi-threaded program, the **stepi** subcommand steps the interrupted thread only. All other user threads remain stopped.

Related to this subcommand are the **step**, **next**, **nexti**, **return**, **cont**, and **set** subcommands.

## pdbx stop subcommand

### Format



The **stop** subcommand sets stopping places called “breakpoints” for tasks in the current context. Use it to mark these stopping places, and then run the program. When the tasks reach a breakpoint, execution stops and the state of the program can then be examined. The **stop** subcommand is context sensitive.

Use the **status** subcommand to display a list of breakpoints that have been set for tasks in the current context. Use the **delete** subcommand to remove breakpoints.

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 122.

#### Notes:

1. The **pdbx** debugger will not attempt to set a breakpoint at a line number when in a group context if the group members (tasks) have different current source files.
2. When specifying variable names as arguments to the **stop** subcommand, fully qualified names should be used. This should be done because, when a **stop** subcommand is issued, a parallel application could be in a different function on each node. This may result in ambiguity in variable name resolution. Use the **which** subcommand to get the fully qualified name for a variable.

To set a breakpoint at line 19 of a program, enter:

```
stop at 19
```

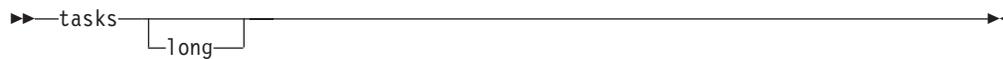
The **pdbx** debugger responds with a message like:

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

Related to this subcommand are the **dbx stop** and **which** subcommands, and the **pdbx trace**, **status**, and **delete** subcommands.

## pdbx tasks subcommand

### Format



With the **tasks** subcommand, you display information about all the tasks in the partition. Task state information is always displayed. 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.

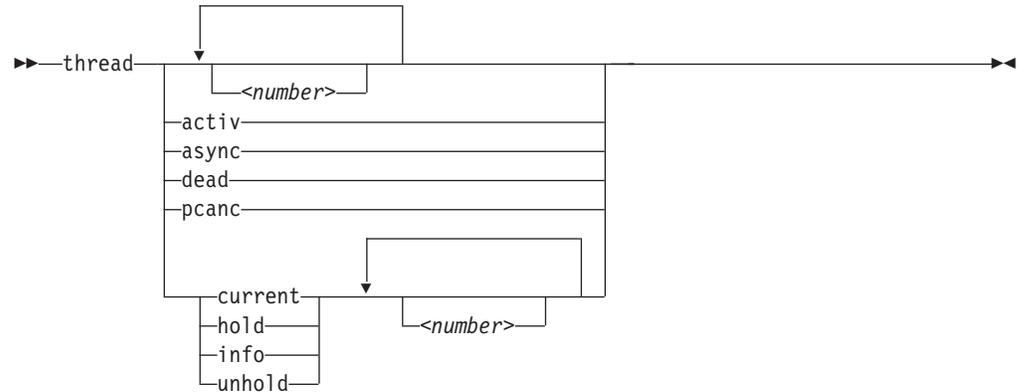
```
pdbx(others) tasks
 0:D    1:D    2:U    3:U    4:R    5:D    6:D    7:R
```

```
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
```

Related to this subcommand is the **pdbx group** subcommand.

## pdbx thread subcommand

### Format



The **thread** subcommand displays a list of active threads for the application program. All active threads are listed unless you use the **number** parameter to specify the threads you want listed. You can also select threads by their states using the **activ**, **async**, **dead**, or **pcanc** options.

You can use the **info** option to display full information about a thread, and threads can be held or unheld with the **hold** and **unhold** options. The focus thread is defaulted to the running thread: **dbx** uses it as the context for normal **dbx** subcommands such as **register**. You can use the **current** option to switch the **dbx** focus thread.

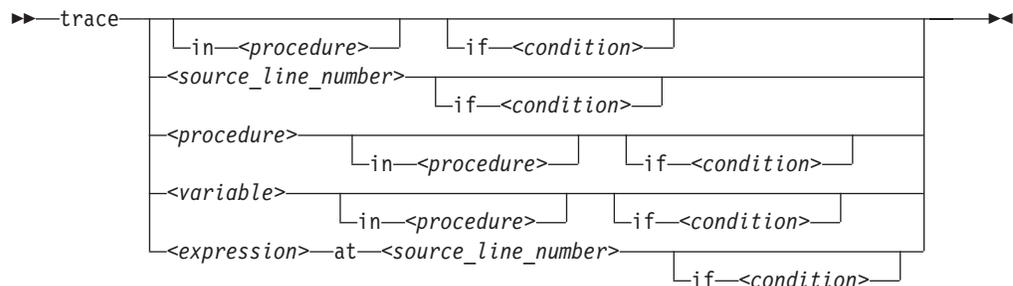
The displayed thread ("**>**") is the thread that is used by other **pdbx** commands that are thread specific such as:

**down**  
**dump**  
**file**  
**func**  
**list**  
**listi**  
**print**  
**registers**  
**up**  
**where**

Related to this subcommand is **mutex** subcommand.

## pdbx trace subcommand

### Format



The **trace** subcommand sets tracepoints for tasks in the current context. These tracepoints will cause tracing information for the specified *procedure*, *function*, *sourceline*, *expression* or *variable* to be displayed when the program runs. The **trace** subcommand is context sensitive.

Use the **status** subcommand to display a list of tracepoints that have been set in the current context. Use the **delete** subcommand to remove tracepoints.

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* or *procedure* argument.

Specify the *<condition>* argument using the syntax described by “Specifying expressions” on page 122.

#### Notes:

1. The **pdbx** debugger will not attempt to set a tracepoint at a line number when in a group context if the group members (tasks) have different current source files.
2. When specifying variable names as arguments to the **trace** subcommand, fully qualified names should be used. This should be done because, when a **trace** subcommand is issued, a parallel application could be in a different function on each node. This may result in ambiguity in variable name resolution. Use the **which** subcommand to get the fully qualified name for a variable.

To set a tracepoint for the variable “foo” at line 21 of a program, enter:

```
trace foo at 21
```

The **pdbx** debugger responds with a message like:

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

Related to this subcommand are the **dbx trace** and **which** subcommands, and the **pdbx stop**, **status**, and **delete** subcommands.

## pdbx unalias subcommand

### Format

►► `unalias` *alias\_name* ◀◀

The **unalias** subcommand removes **pdbx** command aliases. The *alias\_name* specified is any valid alias that has been defined within your current **pdbx** session. The **unalias** subcommand is context insensitive.

To remove the alias “p”, enter:

```
unalias p
```

Related to this subcommand is the **pdbx alias** subcommand.

---

## pdbx unhook subcommand

### Format

▶—unhook—◀

The **unhook** subcommand enables you to unhook tasks. Unhooking allows tasks to run without intervention from the **pdbx** debugger. You can later reestablish control over unhooked tasks using the **hook** subcommand. The **unhook** subcommand is similar to the **detach** subcommand in **dbx**. It is context sensitive and has no parameters.

1. To unhook task 2, enter:  
on 2 unhook  
  
or  
on 2  
unhook
2. To unhook all the tasks in the task group “rest”, enter:  
on rest unhook  
  
or  
on rest  
unhook

Listing the members of the task group “all” using the **list** action of the **group** subcommand will allow you to check which tasks are hooked, and which are unhooked. Enter:

```
group list all
```

The **pdbx** debugger will display a list similar to the following:

```
0:D   1:U   2:D   3:D
```

Tasks marked with the letter U next to them are unhooked tasks. In this case, task 1 is unhooked. Tasks marked with the letter D are “debug ready”, hooked tasks. In this case, tasks 0, 2, and 3 are hooked.

Related to this subcommand is the **dbx detach** subcommand and the **pdbx hook** subcommand.

## **pdbx unset subcommand**

### **Format**

▶▶—unset—*name*—————▶▶

The **unset** subcommand removes the set variable associated with the specified *name*.

Related to this subcommand is the **set** subcommand.

---

## pdbx up subcommand

### Format



```
▶▶ up [count] ◀◀
```

The **up** subcommand moves the current function up the stack the number of levels you specify with the *count* parameter. The current function is used for resolving names. The default for the *count* parameter is 1.

The **up** and **down** subcommands can be used to navigate through the call stack. Using these subcommands to change the current function also causes the current file and local variables to be updated to the chosen stack level.

Related to this subcommand are the **down**, **print**, **dump**, **func**, **file**, and **where** subcommands.

---

## **pdbx use subcommand**

### **Format**



The **use** subcommand sets the list of directories to be searched when the **pdbx** debugger looks for source files. If the subcommand is specified without arguments, the current list of directories to be searched is displayed.

The @ (at sign) is a special symbol that directs **pdbx** to look at the absolute path name information in the object file, if it exists. If you have a relative directory called @ to search, you should use `./@` in the search path.

The **use** subcommand uses the + (plus sign) to add more directories to the list of directories to be searched. If you have a directory named +, specify the absolute path name for the directory (for example, `./+` or `/tmp/+`).

Related to this subcommand are the **file** and **list** subcommands.

---

## **pdbx whatis subcommand**

### **Format**

▶▶ `whatis <name>` ◀◀

The **whatis** subcommand displays the declaration of what you specify as the *name* parameter. The *name* parameter can designate a variable, procedure, or function name, optionally qualified with a block name.

Related to this subcommand are the **whereis** and **which** subcommands.

---

## pdbx where subcommand

### Format

▶▶ where ◀◀

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

```
pdbx(all) where
init_trees(), line 23 in "funcs5.c"
colors(depth = 30, str = "This is it"), line 61 in "funcs5.c"
newmain(), line 59 in "funcs2.c"
f6(), line 25 in "funcs2.c"
main(argc = 1, argv = 0x2ff21c58), line 125 in "funcs.c"
```

Related to this subcommand are the **dbx up** and **down** subcommands.

---

## pdbx whereis subcommand

### Format

▶▶—whereis—*identifier*————▶▶

The **whereis** subcommand displays the full qualifications of all the symbols whose names match the specified *identifier*. The order in which the symbols print is not significant.

Related to this subcommand are the **whatis** and **which** commands.

## **pdbx which subcommand**

### **Format**

▶—*which*—*identifier*—▶

The **which** subcommand displays the full qualification of the given *identifier*. The full qualification consists of a list of the outer blocks with which the *identifier* is associated.

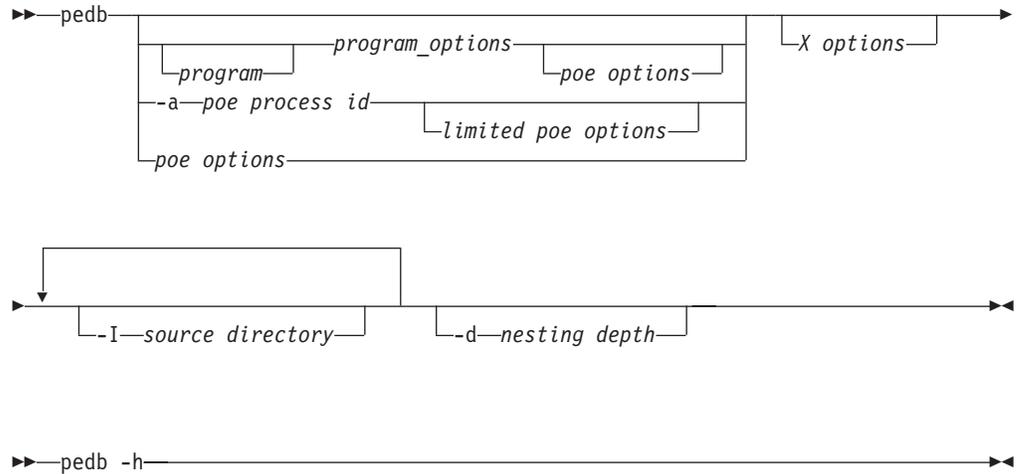
Related to this subcommand are the **whatis** and **whereis** subcommands.

## pedb

## Purpose

**pedb** – Invokes the **pedb** debugger, which is the X-Windows interface of the PE debugging facility.

## Format

**pedb**

The **pedb** command invokes the **pedb** debugger, which is the X-Windows interface of the PE debugging facility.

## Flags

The **pedb** command accepts standard X-Windows flags. Because the **pedb** command runs in the Parallel Operating Environment, it also accepts the flags that are supported by the **poe** command. See the **poe** manual page for a description of these POE options. Additional **pedb** flags are:

- a Attaches 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.
- d Sets the limit for the nesting of program blocks. The default nesting depth limit is 25.
- h Writes the **pedb** usage to STDERR. This includes **pedb** command-line syntax and a description of **pedb** flags.
- I (upper-case i) Specifies 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.)

## Usage

The **pedb** command invokes the X-Windows interface of the PE debugging facility. It runs in the Parallel Operating Environment.

To use **pedb** for interactive debugging, you first need to compile the program and set up the execution environment as you would to invoke a parallel program with the **poe** command. Your program should be compiled with the **-g** flag in order to produce an object file with symbol table references. For more information on the **-g** option, refer to *z/OS UNIX System Services Command Reference*.

## System Environment

Because the **pedb** command runs in the Parallel Operating Environment, it interacts with most of the same environment variables that are associated with the **poe** command. As indicated by the syntax statements, you are also able to specify **poe** command-line options when invoking **pedb**. Using these options will override the setting of the corresponding environment variable, as is the case when invoking a parallel program with the **poe** command.

### MP\_DEBUG\_INITIAL\_STOP

Determines the initial breakpoint in the application where **pdbx** or **pedb** get control. **MP\_DEBUG\_INITIAL\_STOP** should be specified as *file\_name:line\_number*. The *line\_number* is the number of the line within the source file *file\_name* where *file\_name* has been compiled with **-g**. The line number has to be one that defines executable code, i.e. a line which the debugger accepts as breakpoint. Another valid string for **MP\_DEBUG\_INITIAL\_STOP** would be the *function\_name* of the desired initial stopping point in the debugger. The default is to stop at the first executable source line in the main code. This environment variable has no associated command-line flag.

### MP\_DEBUG\_LOG

Determines the level of diagnostic messages that are written to `$MP_TMPDIR/debug_log.pid.taskid`. Typically, this environment variable is used only under the direction of the IBM Support Center in resolving a PE-related problem. This environment variable has no associated command-line flag.

### MP\_TMPDIR

The temporary directory to which the debug log file is written if **MP\_DEBUG\_LOG** is set. **pedb** also uses this as the location for some work files. The default value is */tmp*. The value of this environment variable can be overridden using the **-tmpdir** flag.

The command **pedb** uses the debugging engine of **dbx**. Therefore environment settings for **dbx** are inherited by **pedb**.

## Examples

To start the **pedb** debugger, enter:

```
pedb weather temperate asia -procs 8 -labelio yes
```

In attach mode the following environment variables must be set:

```
_CEE_RUNOPTS="test(a11)"
_BPX_PTRACE_ATTACH=yes
```

This will invoke **pedb** running the weather application on a partition containing 8 nodes with all program output labeled by task id.

The **pedb** window automatically opens to mark the start of the debug session.

## Files

host.list (Default host list file)

/usr/lib/X11/app-defaults/Pedb (Xdefaults file)

/samples/pe/pedb/Pedb.ad (sample Xdefaults file)

## Related Information

command **dbx**, **poe**

---

## pmarray

### Purpose

**pmarray** – Starts the Program Marker Array, which is an X-Windows tool for monitoring a parallel executable's run.

### Format

**pmarray**

▶▶—pmarray—————▶▶

The **pmarray** command starts the Program Marker Array X-Windows tool prior to invoking **poe**. This tool is used for run-time monitoring.

### Flags

None.

### Usage

The **pmarray** command starts the Program Marker Array. This X-Windows run-time monitoring tool consists of a number of small squares, or lights. Each task in a parallel program has its own row of lights. Using calls to the Parallel Utility Functions enables a parallel program to control the appearance of the Program Marker Array Window. Calls to `mpc_marker` enable a program to color lights on, and/or send output strings to the Window. Calls to `mpc_nlights` enable a program to determine the number of lights displayed per task row.

### System Environment

This command responds to the following environment variables:

#### MP\_PMLIGHTS

Indicates the number of lights displayed per row on the Array. When you invoke **poe**, you may override **MP\_PMLIGHTS** using the **-pmlights** flag. The array will then redisplay, showing the new number of lights per row.

#### MP\_PROCS

Indicates the number of program tasks. The **pmarray** command sets the number of task rows displayed in the Array equal to the value of **MP\_PROCS**. When you invoke **poe**, you may override **MP\_PROCS** using the **-procs** flag. The Array will then redisplay showing the new number of rows.

#### MP\_USRPORT

Indicates the port id used by the Partition Manager to connect to the Array. By default, the Partition Manager connects to the Array using a socket assigned to port 9999. If you get an error message indicating that the port is in use, specify a different port. Standard TCP/IP practice suggests using ports greater than 5000 and less than 10000.

### Examples

To start the Program Marker Array program as a background process, and open its window, enter:

pmarray &

## **Files**

/usr/lib/x11/app-defaults/PMarray (Xdefaults file)

/samples/pe/marker/PMarray.ad (sample Xdefaults file)

## **Related Information**

Subroutines: **mpc\_marker**, **mpc\_nlights**.

## Purpose

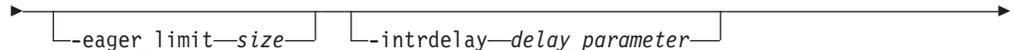
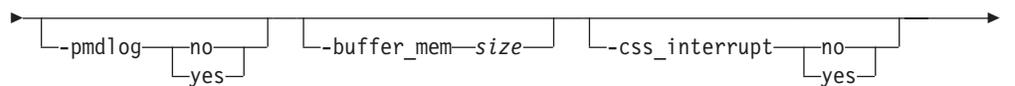
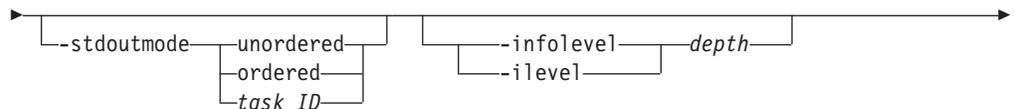
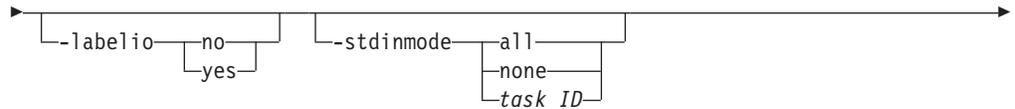
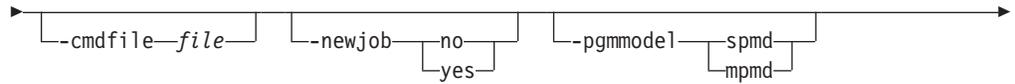
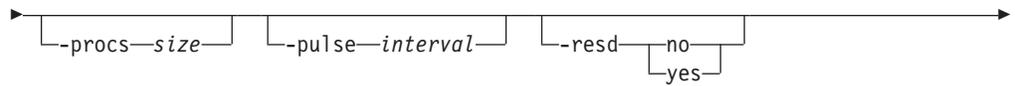
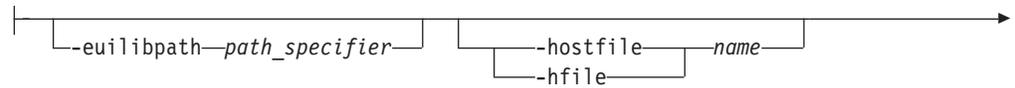
**poe** – Invokes the Parallel Operating Environment (POE) for loading and executing programs on remote processor nodes.

## Format

### poe



### flags:





particular flag sets, refer to the description of its associated environment variable in the Environment variables section. The following flags are grouped by function.

The following Partition Manager control flags override the associated environment variables.

**-euilibpath**  
MP\_EUILIBPATH  
**-hostfile** or **-hfile**  
MP\_HOSTFILE  
**-procs**  
MP\_PROCS  
**-pulse**  
MP\_PULSE  
**-resd**  
MP\_RESD  
**-rmpool**  
MP\_RMPOOL  
**-savehostfile**  
MP\_SAVEHOSTFILE  
**-wlm\_enclave**  
MP\_WLM\_ENCLAVE

The following Job Specification flags override the associated environment variables.

**-cmdfile**  
MP\_CMDFILE  
**-newjob**  
MP\_NEWJOB  
**-pgmmodel**  
MP\_PGMMODEL

The following I/O control flags override the associated environment variables.

**-labelio**  
MP\_LABELIO  
**-stdinmode**  
MP\_STDINMODE  
**-stdoutmode**  
MP\_STDOUTMODE

The following generation of diagnostic information flags override the associated environment variables.

**-infolevel** or **-ilevel**  
MP\_INFOLEVEL  
**-pmdlog**  
MP\_PMDLOG

The following Message Passing flags override the associated environment variables.

**-buffer\_mem**  
MP\_BUFFER\_MEM  
**-css\_interrupt**  
MP\_CSS\_INTERRUPT  
**-eager\_limit**  
MP\_EAGER\_LIMIT  
**-intrdelay**  
MP\_INTRDELAY

```

-ionodefile
    MP_IONODEFILE
-max_typedepth
    MP_MAX_TYPEDEPTH
-polling_interval
    MP_POLLING_INTERVAL
-shared_memory
    MP_SHARED_MEMORY
-single_thread
    MP_SINGLE_THREAD
-thread_stacksize
    MP_THREAD_STACKSIZE
-use_flow_control
    MP_USE_FLOW_CONTROL
-wait_mode
    MP_WAIT_MODE

```

The following miscellaneous flags override the associated environment variables:

```

-euidevelop
    MP_EUIDEVELOP
-pmlights
    MP_PMLIGHTS
-tmpdir
    MP_TMPDIR
-usrport
    MP_USRPORT

```

The command line flag:

```

-promptpw
    overrides the default password retrieval mechanism. If you specify yes, poe
    prompts for the password. If you specify no, poe reads the password from
    STDIN, regardless of whether poe is invoked interactively or through a
    program.
-h
    If you specify the flag -h, the manual page for poe is displayed.

```

## Usage

The **poe** command invokes the Parallel Operating Environment for loading and executing programs on remote nodes. You can enter it at your home node to:

- load and execute an SPMD program on all nodes of your partition.
- individually load the nodes of your partition with an MPMD job.
- load and execute a series of SPMD and MPMD programs, in individual job steps, on the same partition.
- run non-parallel programs on remote nodes.

The operation of POE is influenced by a number of POE environment variables. The flag options on this command are each used to temporarily override one of these environment variables. User *program options* can be freely interspersed with the flag options, and *additional options* not to be parsed by POE can be placed after a *fence string* defined by the **MP\_FENCE** environment variable. If no *program* is specified, POE will either prompt you for programs to load, or, if the **MP\_CMDFILE** environment variable is set, will load the partition using the specified commands file.

The environment variables and flags that influence the operation of this command fall into distinct categories of function. They are:

- **Partition Manager control.** The environment variables and flags in this category determine the method of node allocation, message passing mechanism, and the PULSE monitor function.
- **Job specification.** The environment variables and flags in this category determine whether or not the Partition Manager should maintain the partition for multiple job steps, whether commands should be read from a file or STDIN, and how the partition should be loaded.
- **I/O control.** The environment variables and flags in this category determine how I/O from the parallel tasks should be handled. These environment variables and flags set the input and output modes, and determine whether or not output is labeled by task id.
- **Generation of diagnostic information.** The environment variables and flags in this category enable you to generate diagnostic information that may be required by the IBM Support Center in resolving PE-related problems.
- **Message Passing Interface.** The environment variables and flags in this category enable you to specify values for tuning message passing applications.
- **Miscellaneous.** The additional environment variables and flags in this category enable additional error checking.

## System Environment

The environment variable descriptions in this section are grouped by function.

The following environment variables are associated with Partition Manager control.

### MP\_EUILIBPATH

Determines the path to the message passing and communication subsystem libraries. This only needs to be set if the libraries are moved. Valid values are any path specifier. The value of this environment variable can be overridden using the **-euilibpath** flag.

### MP\_HOSTFILE

Determines the name of a host list file for node allocation. Valid values are any file specifier or "NULL" or "". If not set, the default is *./host.list* in your current directory. The value of this environment variable can be overridden using the **-hostfile** or **-hfile** flags.

Task allocation is controlled via the pool list and pin list. The pool list contains all systems to which non-pinned tasks may be automatically allocated. The pin list contains systems with associated ("pinned") tasks which should be explicitly allocated to the respective systems.

If **MP\_HOSTFILE** is set to "NULL" or "" the pool list contains all systems in the sysplex (possibly constrained by **MP\_RMPOOL**) and the pin list is empty. Otherwise the host list file specifies both the pin list and the pool list.

Host list file entries are lines containing a system name possibly followed by a list of task numbers separated by blanks. The systems in a host list file not followed by task numbers make up the pool list. The systems in the host list file followed by at least one task number make up the pin list. The system names used in the host list file are either TCP host names or MVS system names. If **MP\_RESD** is set to **yes**, MVS host names must be used.

**MP\_PROCS**

Determines the number of program tasks. Valid values are any number from 1 to 512. If not set, the default is 1. The value of this environment variable can be overridden using the **-procs** flag.

**MP\_PULSE**

The interval (in seconds) at which POE checks the remote nodes to ensure that they are communicating with the home node. The default interval is 600 seconds (10 minutes). To disable the pulse function, specify an interval of 0 (zero) seconds. The pulse function is automatically disabled when running the **pdbx** and **pedb** debuggers (see “Chapter 10. Using the pdbx debugger” on page 93 and “Chapter 11. Using the pedb debugger” on page 127). You can override the value of this environment variable with the **-pulse** flag.

**MP\_REMOTEDIR**

Specifies the name of the current directory to be used on the remote nodes. By default, the current directory is the directory on the home node at the time POE is run.

**MP\_RESD**

Determines the automated node selection scheme that maps task numbers to systems in the pool list. The content of the pool list can be controlled with **MP\_HOSTFILE**. Valid values for **MP\_RESD** are either **yes** or **no**. If **MP\_RESD** is set to **no**, a round robin allocation scheme is used. **MP\_RESD** defaults to **no**. The value of this environment variable can be overridden using the **-resd** flag.

**MP\_RMPOOL**

You can set this to the name of a WLM scheduling environment. This causes POE to automatically allocate those nodes from the systems in the sysplex, which have the resources in the correct state for that environment. This option is not needed if you want POE to automatically allocate from all systems in the sysplex, or from the set of systems in the host list file. **MP\_RMPOOL** is used only if **MP\_HOSTFILE** is explicitly set to an empty string or “NULL”. It is ignored otherwise, and there is no default value. The value of this environment variable can be overridden using the **-rmpool** flag.

**MP\_SAVEHOSTFILE**

The name of an output host list file to be generated by the Partition Manager. Valid values are any relative or absolute path name. The value of this environment variable can be overridden using the **-savehostfile** flag.

**MP\_TIMEOUT**

Controls the length of time POE waits before abandoning an attempt to connect to the remote nodes. The default is 150 seconds. **MP\_TIMEOUT** also changes the length of time the communication subsystem will wait for a connection to be established during message passing initialization.

**MP\_WLM\_ENCLAVE**

Determines whether a Workload Manager (WLM) Multi-System enclave will be used. Valid values are **yes** and **no**. If **MP\_WLM\_ENCLAVE** is set to **yes** all tasks of the parallel program belong to the same WLM multi-system enclave. This option can only be used on a Parallel Sysplex. If not set, the default is **no**. The environment variable can be overridden using the **-wlm\_enclave** flag.

The following environment variables are associated with Job Specification.

**MP\_CMDFILE**

Determines the name of a POE commands file that is used to load the nodes of your partition. If set, POE will read the commands file rather than STDIN. Valid values are any file specifier. The value of this environment variable can be overridden using the **-cmdfile** flag.

**MP\_NEWJOB**

Determines whether or not the Partition Manager maintains your partition for multiple job steps. Valid values are **yes** or **no**. If not set, the default is **no**. The value of this environment variable can be overridden using the **-newjob** flag.

**MP\_PGMMODEL**

Determines the programming model you are using. Valid values are **spmd** or **mpmd**. If not set, the default is **spmd**. The value of this environment variable can be overridden using the **-pgmmodel** flag.

The following environment variables are associated with I/O Control.

**MP\_LABELIO**

Determines whether or not output from the parallel tasks is labeled by task id. Valid values are **yes** or **no**. If not set, the default is **no**. The value of this environment variable can be overridden using the **-labelio** flag.

**MP\_STDINMODE**

Determines the input mode – how STDIN is managed for the parallel tasks. Valid values are:

- all** all tasks receive the same input data from STDIN.
- none** no tasks receive input data from STDIN; STDIN will be used by the home node only.
- n*** STDIN is only sent to the task identified (*n*).

If not set, the default is **all**. The value of this environment variable can be overridden using the **-stdinmode** flag.

**MP\_HOLD\_STDIN**

Determines whether or not sending of STDIN from the home node to the remote nodes is deferred until the message passing partition has been established. Valid values are **yes** or **no**. If not set, the default is **no**.

**MP\_STDOUTMODE**

Determines the output mode – how STDOUT is handled by the parallel tasks. Valid values are:

- unordered**  
all tasks write output data to STDOUT asynchronously.
- ordered**  
output data from each parallel task is written to its own buffer. Later, all buffers are flushed, in task order, to STDOUT.
- a task id**  
only the task indicated writes output data to STDOUT.

If not set, the default is **unordered**. The value of this environment variable can be overridden using the **-stdoutmode** flag.

The following environment variables are associated with the generation of diagnostic information.

**MP\_INFOLEVEL**

Determines the level of message reporting. Valid values are:

- 0 error
- 1 warning and error
- 2 informational, warning, and error
- 3 informational, warning, and error. Also reports diagnostic messages for use by the IBM Support Center.
- 4, 5, 6 Informational, warning, and error. Also reports high- and low-level diagnostic messages for use by the IBM Support Center.

If not set, the default is 1 (warning and error). The value of this environment variable can be overridden using the **-infolevel** or **-ilevel** flags.

**MP\_PMDLOG**

Determines whether or not diagnostic messages should be logged to a file in */tmp* on each of the remote nodes. Typically, this environment variable/command-line flag is only used under the direction of the IBM Support Center in resolving a PE for z/OS related problem. Valid values are **yes** or **no**. If not set, the default is **no**. The value of this environment variable can be overridden using the **-pmdlog** flag.

**MP\_PMDSUFFIX**

Determines a string to be appended to the normal tcp service. The normal tcp service specified in */etc/services* is named *pmv2*. By setting **MP\_PMDSUFFIX**, you can append a string to *pmv2*. If **MP\_PMDSUFFIX** is set to *abc*, for example, then the service requested in */etc/services* is *pmv2abc*. This permits testing of alternate versions of the Partition Manager Daemon. Typically, this environment variable is only used under the direction of the IBM Support Center in resolving a PE for z/OS related problem. Valid values are any string. This environment variable has no associated command-line flag. In addition this variable can be used to run *pmvs* from other releases in a multi release environment.

The following environment variables are associated with the Message Passing Interface.

**MP\_BUFFER\_MEM**

Changes the maximum size of memory used by the communication subsystem to buffer early arrivals. The default is 2.8 megabytes. The value of this environment variable can be overridden using the **-buffer\_mem** flag.

**MP\_CSS\_INTERRUPT**

Determines whether or not arriving message packets cause interrupts. This may provide better performance for certain applications. Valid values are **yes** and **no**. If not set, the default is **no**. The value of this environment variable can be overridden using the **-css\_interrupt** flag.

**MP\_EAGER\_LIMIT**

Changes the threshold value for message size, above which rendezvous protocol is used. Valid values are integers less than or equal to 65535. The value of this environment variable can be overridden using the **-eager\_limit** flag.

**MP\_INTRDELAY**

Allows user programs to tune the delay parameter without having to recompile existing applications. Valid values are integers greater than or equal to 0. The value of this environment variable can be overridden using the **-intrdelay** flag.

**MP\_IONODEFILE**

The name of a parallel I/O node file — a text file that lists the nodes that should be handling parallel I/O. This enables you to limit the number of nodes that participate in parallel I/O, guarantee that all I/O operations are performed on the same node, and so on. Valid values are any relative or full path name. If not specified, all nodes will participate in parallel I/O operations. The value of this environment variable can be overridden using the **-ionodefile** command-line flag.

**MP\_MAX\_TYPEDEPTH**

Changes the maximum depth of message buffer types. Valid values are positive integers. The value of this environment variable can be overridden using the **-max\_typedepth** flag.

**MP\_POLLING\_INTERVAL**

Specifies the interval (in microseconds) POE waits between polling for data in case a blocked thread or task is waiting and **MP\_WAIT\_MODE=poll**. The default is 180,000 and the maximum value used by poe is approximately 2000 seconds. The value of this environment variable can be overridden using the **-polling\_interval** flag.

**MP\_SHARED\_MEMORY**

Determines whether or not tasks running on the same node should use shared memory for message passing (instead of UDP). Valid values are **yes** or **no**. If not set the default is **no**. The value of this environment variable can be overridden using the **-shared\_memory** flag.

**MP\_SINGLE\_THREAD**

Avoids mutex lock overheads in a single threaded program. This is an optimization flag, with values of **no** and **yes**. The default value is **no**, which means that multiple user message passing threads are assumed.

**Note:** MPI-IO cannot be used if this is set to **YES**. Results are undefined if this is **YES**, with multiple message passing threads in use.

The value of this environment variable can be overridden using the **-single\_thread** flag.

**MP\_THREAD\_STACKSIZE**

Determines the additional stacksize allocated for user programs executing on an MPI service thread. If you allocate insufficient space, the program may encounter a SIGSEGV exception. See "Chapter 8. Programming considerations for user applications in POE" on page 63 for more details. The default is 0 (No addition to the minimum). The value of this environment variable can be overridden using the **-thread\_stacksize** flag.

**MP\_USE\_FLOW\_CONTROL**

Throttles sender before the number of outstanding eager send messages can overflow the early arrival buffer at a destination. Flow control insures that programs with weak synchronization and aggressive use of small messages will never overflow early arrival buffers. Most MPI programs may be run without flow control if that improves performance but without flow control certain programs may occasionally fail with an "out of

memory" error. Possible values are **yes** and **no**. The default value is **yes**. The value of this environment variable can be overridden using the **-use\_flow\_control** flag.

#### MP\_WAIT\_MODE

Determines how a thread or task behaves when it discovers that it is blocked, waiting for a message to arrive. Values are **poll**, **yield**, and **sleep**. The default mode is **sleep**. The value of this environment variable can be overridden using the **-wait\_mode** flag.

The following are miscellaneous environment variables:

#### MP\_EUIDEVELOP

Determines whether or not the Message Passing Interface performs more detailed checking during execution. This additional checking is intended for developing applications, and can significantly slow performance. Valid values are **yes** or **no**, **deb** (for "debug"), **nor** (for "normal"), and **min** (for "minimum"). If not set, the default is **no**. The value of this environment variable can be overridden using the **-euiddevelop** flag.

#### MP\_FENCE

Determines a *fence string* to be used for separating options you want parsed by POE from those you do not. Valid values are any string, and there is no default. Once set, you can then use the *fence string* followed by *additional options* on the **poe** command line. The *additional options* will not be parsed by POE. This environment variable has no associated command-line flag.

#### MP\_NOARGLIST

Determines whether or not POE ignores the argument list. Valid values are **yes** and **no**. If set to **yes**, POE will not attempt to remove POE command-line flags before passing the argument list to the user's program. This environment variable has no associated command-line flag.

#### MP\_PMLIGHTS

Indicates the number of lights displayed per row on the Program Marker Array. The value of this environment variable can be overridden using the **-pmlights** flag.

#### MP\_TMPDIR

The temporary directory to which the debug log file is written if **MP\_DEBUG\_LOG** is set. **pedb** also uses this as the location for some work files. The default value is */tmp*. The value of this environment variable can be overridden using the **-tmpdir** flag.

#### MP\_USRPORT

Indicates the port id used by the Partition Manager to connect to the Program Marker Array. By default, the Partition Manager connects to the Array using a socket assigned to port 9999. If you get an error message indicating that the port is in use, specify a different port. Standard TCP/IP practice suggests using ports greater than 5000 and less than 10000. The value of this environment variable can be overridden using the **-usrport** flag.

## Examples

1. Assume the **MP\_PGMMODEL** environment variable is set to **spmd**, and **MP\_PROCS** is set to 6. To load and execute the SPMD program *sample* on the six remote nodes of your partition, enter:

```
poe sample
```

## poe

2. Assume that you have an MPMD application consisting of two programs – *master* and *workers*. These programs are designed to run together and communicate via calls to message passing subroutines. The program *master* is designed to run on one processor node. The *workers* program is designed to run as separate tasks on any number of other nodes. The **MP\_PGMMODEL** environment variable is set to **mpmd**, and **MP\_PROCS** is set to 6. To individually load the six remote nodes with your MPMD application, enter:

```
poe
```

Once the partition is established, the **poe** command responds with the prompt:

```
0:host1_name>
```

To load the *master* program as task 0 on host1\_name, enter:

```
master
```

The **poe** command responds with a prompt for the next node to be loaded (e.g. *workers*). When you have loaded the last node of your partition, the **poe** command begins execution.

3. Assume that you want to run three SPMD programs – *setup*, *computation*, and *cleanup* – as job steps on the same partition of nodes. The **MP\_PGMMODEL** environment variable is set to **spmd**, and **MP\_NEWJOB** is set to **yes**. You enter:

```
poe
```

Once the partition is established, the **poe** command responds with the prompt:

```
Enter program name (or quit):
```

To load the program *setup*, enter:

```
setup
```

The program *setup* executes on all nodes of your partition. When execution completes, the **poe** command again prompts you for a program name. Enter the program names in turn. To release the partition, enter:

```
quit
```

4. To check the process status (using the non-parallel command **ps**) for all remote nodes in your partition, enter:

```
poe ps
```

## Files

host.list (Default host list file)

## Related Information

commands **mpcc**, **mpCC**, **pdbx**, **pedb**, **pmarray**

## poekill

### Purpose

**poekill** – terminates all remote tasks for a given program.

### Format

```
▶▶—poe—poekill—program name—┌poe options—▶▶
```

```
▶▶—poekill—program name—▶▶
```

**poekill** is a shell script that searches for the existence of running programs owned by the user, and terminates them via SIGTERM signals. Only programs are searched which are named **program name** or the name containing the term **program name**. If run under **poe**, **poekill** uses the standard POE mechanism for identifying the set of remote nodes (*host.list*).

If called directly **poekill** terminates all running programs on the local node. If such program exist and the communication infrastructure is complete, this will terminate the whole partition.

### Flags

Standard POE flags apply for **poe poekill**.

### Usage

**poekill** determines the user ID of the user that submitted the command. It then uses the ID to obtain a list of active processes, which is filtered by the *program name* argument into a scratch file in /tmp. The file is processed by an awk script that sends a SIGTERM signal to each process in the list, and echoes the action back to the user. The scratch file is then erased, and the script exits with code of 0.

If you do not provide a *program name*, an error message is printed and the script exits with a code of 1.

The *program name* can be a substring of the program name of the program to be killed.

**Note:** Termination initiated by **poe poekill** runs concurrently with the termination process within the partition of the parallel program. This may result in error messages issued by **kill**. These error messages may be ignored.

### Related Information

commands **poe**, **kill**

**poekill**

---

## Appendix B. POE environment variables and command-line flags

This appendix contains tables which summarize the environment variables and command-line flags discussed throughout this book. You can set these variables and flags to influence the execution of parallel programs, and the operation of certain tools. The command-line flags temporarily override their associated environment variable. The tables divide the environment variables and flags by function.

- Table 11 on page 284 summarizes the environment variables and flags for controlling the Partition Manager. These environment variables and flags enable you to specify such things as an input or output host list file, and the method of node allocation. For a complete description of the variables and flags summarized in this table, see “Chapter 4. Executing parallel programs” on page 15.
- Table 12 on page 285 summarizes the environment variables and flags for job specifications. These environment variables and flags determine whether or not the Partition Manager should maintain the partition for multiple job steps, whether commands should be read from a file or STDIN, and how the partition should be loaded. For a complete description of the variables and flags summarized in this table, see “Chapter 4. Executing parallel programs” on page 15.
- Table 13 on page 286 summarizes the environment variables and flags for determining how I/O from the parallel tasks should be handled. These environment variables and flags set the input and output modes, and determine whether or not output is labeled by task id. For a complete description of the variables and flags summarized in this table, see “Managing standard input, output, and error” on page 32.
- Table 14 on page 287 summarizes the environment variables and flags for collecting diagnostic information. These environment variables and flags enable you to generate diagnostic information that may be required by the IBM Support Center in resolving PE related problems.
- Table 15 on page 289 summarizes the environment variables and flags for the Message Passing Interface. These environment variables and flags allow you to change message and memory sizes, as well as other message passing information.
- Table 16 on page 290 summarizes some miscellaneous environment variables and flags.

You can use the POE command-line flags on the **poe**, **pdbx**, and **pedb** command. You can also use some of these flags on program names when individually loading nodes from STDIN or a POE commands file. The flags are:

- **-infolevel** or **-ilevel**
- **-euidevelop**

Table 11. POE environment variables and command-line flags for Partition Manager control

The environment variable and command-line flag(s):	Set:	Possible values:	Default:
MP_EUILIBPATH -euilibpath	The path to the message passing and communication subsystem.	Any path specifier	none
MP_HOSTFILE -hostfile -hfile	The name of a host list file for node allocation. "NULL" or "" (empty string) if no host list file should be used.	<ul style="list-style-type: none"> <li>• Any file specifier or</li> <li>• the word NULL or</li> <li>• an empty string (" ")</li> </ul>	<i>.host.list</i> in the current directory.
MP_PROCS -procs	The number of program tasks.	Any number from 1 to 512.	1
MP_PULSE -pulse	The interval (in seconds) at which POE checks the remote nodes to ensure that they are actively communicating with the home node.	An integer greater than or equal to 0.	600
MP_REMOTEDIR (no associated command-line flag)	Specifies the name of the current directory to be used on the remote nodes. By default, the current directory is the directory on the home node at the time POE is run.	Any valid directory specifier	None
MP_RESD -resd	Whether or not the Workload Manager should automatically allocate nodes. If set to 'yes', WLM is used for automatic task allocation. If set to 'no', Round Robin automatic task allocation is used.	<i>yes</i> <i>no</i>	no
MP_RMPOOL -rmpool	The name of the WLM scheduling environment that should be used for non-specific node allocation. The setting of this environment variable is ignored if MP_HOSTFILE is not "NULL" nor "" (empty string).	A WLM scheduling environment name.	None
MP_SAVEHOSTFILE -savehostfile	The name of an output host list file to be generated by the Partition Manager.	Any relative or full path name.	None
MP_TIMEOUT (no associated command-line flag)	The length of time that POE waits before abandoning an attempt to connect to the remote nodes.	Any number greater than 0.	150 seconds
MP_WLM_ENCLAVE -wlm_enclave	Whether or not the parallel job should run in a WLM Multi-System enclave.	<i>yes</i> <i>no</i>	no

Table 12. POE environment variables and command-line flags for job specification

The environment variable and command-line flag(s):	Set:	Possible values:	Default:
MP_CMDFILE -cmdfile	The name of a POE commands file used to load the nodes of your partition. If set, POE will read the commands file rather than STDIN.	Any file specifier.	None
MP_NEWJOB -newjob	Whether or not the Partition Manager maintains your partition for multiple job steps.	<i>yes</i> <i>no</i>	<i>no</i>
MP_PGMMODEL -pgmmodel	Determines the programming model you are using.	<i>smd</i> <i>mpmd</i>	<i>smd</i>

Table 13. POE environment variables and command-line flags for I/O control

The environment variable and command-line flag(s):	Set:	Possible values:	Default:
MP_LABELIO -labelio	Whether or not output from the parallel tasks is labeled by task id.	<i>yes</i> <i>no</i>	<i>no</i>
MP_STDINMODE -stdinmode	The input mode. This determines how input is managed for the parallel tasks.	<i>all</i> All tasks receive the same input data from STDIN.  <i>none</i> No tasks receive input data from STDIN; STDIN will be used by the home node only.  a task id STDIN is only sent to the task identified.	<i>all</i>
MP_HOLD_STDIN (no associated command-line flag)	Whether or not sending of STDIN from the home node to the remote nodes is deferred until the message passing partition has been established.	<i>yes</i> Any other value defaults to <i>no</i>	<i>no</i>
MP_STDOUTMODE -stdoutmode	The output mode. This determines how STDOUT is handled by the parallel tasks.	One of the following:  <i>unordered</i> all tasks write output data to STDOUT asynchronously.  <i>ordered</i> output data from each parallel task is written to its own buffer. Later, all buffers are flushed, in task order, to STDOUT.  a task id only the task indicated writes output data to STDOUT.	<i>unordered</i>

Table 14. POE environment variables and command-line flags for diagnostic information

The environment variable and command-line flag(s):	Set:	Possible values:	Default:
MP_INFOLEVEL  -infolevel -ilevel	The level of message reporting.	One of the following integers:  0 error 1 warning and error 2 informational, warning, and error 3 informational, warning, and error. Also reports high-level diagnostic messages for use by the IBM Support Center.  4, 5, 6 informational, warning, and error. Also reports high- and low-level diagnostic messages for use by the IBM Support Center.	1
MP_DEBUG_LOG  (no associated command-line flag)	The level of diagnostic messages written to <b>\$MP_TMPDIR/debug_log.pid.taskid</b> . Typically, this environment variable is only used under the direction of the IBM Support Center in resolving a PE-related problem. <b>Note:</b> MP_DEBUG_LOG is only valid for <b>pedb</b> .	0 - 4	0
MP_DEBUG_INITIAL_STOP  (no associated command-line flag)	The initial breakpoint in the application where <b>pdbx</b> or <b>pedb</b> will get control.	One of the following: <i>"filename":line_number</i> <i>function_name</i>	The first executable source line in the main routine.
MP_PMDLOG  -pmdlog	Whether or not diagnostic messages should be logged to a file in <i>/tmp</i> on each of the remote nodes. Typically, this environment variable/command-line flag is only used under the direction of the IBM Support Center in resolving a PE-related problem.	One of the following: <i>yes</i> <i>no</i>	<i>no</i>

Table 14. POE environment variables and command-line flags for diagnostic information (continued)

The environment variable and command-line flag(s):	Set:	Possible values:	Default:
<p>MP_PMDSUFFIX</p> <p>(no associated command-line flag)</p>	<p>A string to be appended to the normal tcp service. The normal tcp service specified in <i>/etc/services</i> is named <i>pmv2</i>. By setting <b>MP_PMDSUFFIX</b>, you can append a string to <i>pmv2</i>. If <b>MP_PMDSUFFIX</b> is set to <i>abc</i>, for example, then the service requested in <i>/etc/services</i> is <i>pmv2abc</i>. This permits testing of alternate versions of the Partition Manager Daemon. Typically, this environment variable is only used under the direction of the IBM Support Center in resolving a PE-related problem. In addition this variable can be used to run <b>pmds</b> of other releases in a multi-release environment. See 277.</p>	<p>any string</p>	<p>None</p>

Table 15. POE environment variables and command-line flags for Message Passing Interface (MPI)

The environment variable and command-line flag(s):	Set:	Possible values:	Default:																
MP_BUFFER_MEM -buffer_mem	To change the maximum size of memory used by the communication subsystem to buffer early arrivals.	An integer less than or equal to 64 MB.	2,800,000 bytes																
MP_CSS_INTERRUPT -css_interrupt	To determine whether arriving message packets cause interrupts.	One of the following: <i>yes</i> <i>no</i>	<i>no</i>																
MP_EAGER_LIMIT -eager_limit	To change the threshold value for message size, above which rendezvous protocol is used.  To ensure that at least 32 messages can be outstanding between any 2 tasks, <b>MP_EAGER_LIMIT</b> will be adjusted based on the number of tasks according to the following table (when <b>MP_EAGER_LIMIT</b> and <b>MP_BUFFER_MEM</b> have not been set by the user):  <table border="0"> <thead> <tr> <th>Number of Tasks</th> <th>MP_EAGER_LIMIT</th> </tr> <tr> <th>=====</th> <th>=====</th> </tr> </thead> <tbody> <tr> <td>1 - 16</td> <td>4096</td> </tr> <tr> <td>17 - 32</td> <td>2048</td> </tr> <tr> <td>33 - 64</td> <td>1024</td> </tr> <tr> <td>65 - 128</td> <td>512</td> </tr> <tr> <td>129 - 256</td> <td>256</td> </tr> <tr> <td>257 - 512</td> <td>128</td> </tr> </tbody> </table>	Number of Tasks	MP_EAGER_LIMIT	=====	=====	1 - 16	4096	17 - 32	2048	33 - 64	1024	65 - 128	512	129 - 256	256	257 - 512	128	An integer less than or equal to 65535.	128 byte to 4KB
Number of Tasks	MP_EAGER_LIMIT																		
=====	=====																		
1 - 16	4096																		
17 - 32	2048																		
33 - 64	1024																		
65 - 128	512																		
129 - 256	256																		
257 - 512	128																		
MP_INTRDELAY -intrdelay	To tune the delay parameter without recompiling existing applications.	An integer greater than or equal to 0.	35																
MP_IONODEFILE -ionodefile	To limit the number of nodes that participate in parallel I/O.	<ul style="list-style-type: none"> <li>• relative pathname</li> <li>• full pathname</li> </ul>	none																
MP_MAX_TYPEDEPTH -max_typedepth	To change the maximum depth of message buffer types.	An integer greater than or equal to 1.	5																
MP_POLLING_INTERVAL -polling_interval	Specifies the interval (in microseconds) POE waits between polling for data in case a blocked thread or task is waiting and <b>MP_WAIT_MODE=poll</b> .	A value between 0 (periodic polling is turned off) and 2,000,000,000 (2000 seconds)	180,000																
MP_SHARED_MEMORY -shared_memory	To determine whether tasks running on the same node use shared memory.	<i>yes</i>  <i>no</i>	<i>no</i>																
MP_SINGLE_THREAD -single_thread	To avoid mutex lock overheads in a program which is known to be single threaded.	<i>no</i> <i>yes</i>	<i>no</i>																

Table 15. POE environment variables and command-line flags for Message Passing Interface (MPI) (continued)

The environment variable and command-line flag(s):	Set:	Possible values:	Default:
MP_THREAD_STACKSIZE -thread_stacksize	To specify the additional stacksize allocated for user programs executing on an MPI service thread. If you allocate insufficient space, the program may encounter a SIGSEGV exception.	<i>nnnnn</i> or <i>nnnK</i> or <i>nnM</i> (where K=1024 bytes and M=1024*1024 bytes)	None
MP_USE_FLOW_CONTROL -use_flow_control	To limit the maximum number of outstanding messages posted by a sender.	<i>yes</i> <i>no</i>	<i>yes</i>
MP_WAIT_MODE -use_wait_mode	To specify how a thread or task behaves when it discovers it is blocked, waiting for a message to arrive.	<i>poll</i> <i>yield</i> <i>sleep</i>	<i>sleep</i>

Table 16. Other POE environment variables and command-line flags

The environment variable and command-line flag(s):	Set:	Possible values:	Default:
MP_EUIDEVELOP -euiddevelop	Whether or not the Message Passing Interface performs more detailed checking during execution. This additional checking is intended for developing applications, and can significantly slow performance. You can also start and stop parameter checking with <i>deb</i> (for “debug”) and <i>min</i> (for “minimum”).	<i>yes</i> (for “develop”) <i>no</i> (for “no checking”) <i>nor</i> (for “normal”) <i>deb</i> (for “debug”) <i>min</i> (for “minimum”)	<i>no</i>
MP_FENCE (no associated command-line flag)	A “fence” character string for separating arguments you want parsed by POE from those you do not.	Any string.	None
MP_NOARGLIST (no associated command-line flag)	Whether or not POE ignores the argument list. If set to <i>yes</i> , POE will not attempt to remove POE command-line flags before passing the argument list to the user’s program.	<i>yes</i> <i>no</i>	<i>no</i>
MP_PMLIGHTS -pmlights	The number of lights displayed (per row) on the Program Marker Array.	An integer greater than or equal to 0.	0
MP_TMPDIR -tmpdir	The directory to which diagnostic message files and temporary work files are written by the debugger.	Any path specifier	<i>/tmp</i>
MP_USRPORT -usrport	The port id used by the Partition Manager to connect to the Program Marker Array.	Any positive integer less than 32767. Standard TCP/IP practice suggests using ports greater than 5000 and less than 10000.	9999

---

## Appendix C. Command-line flags for Normal or Attach Mode

This appendix lists the **poe** command-line flags that **pdbx** and **pedb** use, indicating which ones are valid in normal and in attach debugging mode. When starting in attach mode, the debugger gives a message listing the invalid flags used, and then exits.

Table 17. Command-line flags for Normal or Attach Mode

Flag	Description	Normal Mode	Attach Mode
-procs	number of processors	yes	no
-tmpdir	diagnostic message directory	yes	yes
-hostfile	name of host list file	yes	no
-hfile	name of host list file	yes	no
-infolevel	message reporting level	yes	yes
-ilevel	message reporting level	yes	yes
-pmlights	number of LEDs	yes	no
-usrport	port for API-to-user programmable monitor	yes	no
-resd	directive to use Workload Manager	yes	no
-euidevelop	EUI develop mode	yes	no
-pmdlog	use pmd logfile	yes	yes
-savehostfile	list of hosts from resource manager	yes	no
-stdoutmode	STDOUT mode	yes	no
-stdinmode	STDIN mode	yes	no
-labelio	label output	yes	yes
-rmpool	Workload Manager scheduling environment	yes	no
-d	nesting depth of program blocks	yes	yes
-I (upper case i)	path to search for source files	yes	yes
-a	start in attach mode	N/A	yes
-wlm_enclave	usage of WLM Multi-system enclave	yes	no



---

## Appendix D. MPI safety

This appendix provides information on creating a *safe* MPI program. Much of the information presented here comes from *MPI: A Message-Passing Interface Standard, Version 1.1* available from the University of Tennessee.

---

### Safe MPI coding practices

#### What is a safe program?

This is a hard question to answer. Many people consider a program to be *safe* if no message buffering is required for the program to complete. In a program like this, you should be able to replace all standard sends with synchronous sends, and the program will still run correctly. This is considered to be a conservative programming style, which provides good portability because program completion doesn't depend on the amount of available buffer space.

On the flip side, there are many programmers that prefer more flexibility and use an *unsafe* style that relies, at least somewhat, on buffering. In such cases, the use of standard send operations can provide the best compromise between performance and robustness. Good MPI programs will supply sufficient buffering so that these programs will not result in deadlock. The buffered send mode can be used for programs that require more buffering, or in situations where you want more control. Since buffer overflow conditions are easier to diagnose than deadlock, this mode can also be used for debugging purposes.

Nonblocking message passing operations can be used to avoid the need for buffering outgoing messages. This prevents deadlock situations due to a lack of buffer space, and improves performance by allowing computation and communication to overlap. It also avoids the overhead that is associated with allocating buffers and copying messages into buffers.

#### Some general hints and tips

To ensure that you have a truly MPI-based application, you need to conform to a few basic rules of point-to-point communication. In this section, we will alert you to some of the things you need to pay attention to as you create your parallel program. Note that most of the information in this section was taken from *MPI: A Message Passing Interface Standard*, so you may want to refer to this document for more information (see "Related non-IBM publications" on page 307).

#### Order

With MPI, it is important to know that messages are *non-overtaking*; the order of sends must match the order of receives. Assume a sender sends two messages (Message 1 and Message 2) in succession, to the same destination, and both match the same receive. The receive operation will receive Message 1 before Message 2. Likewise, if a receiver posts two receives (Receive 1 and Receive 2), in succession, and both are looking for the same message, Receive 1 will receive the message before Receive 2. Adhering to this rule ensures that sends are always matched with receives.

If a process in your program has a single thread of execution, then the sends and receives that occur follow a natural order. However, if a process has multiple

threads, the various threads may not execute their relative send operations in any defined order. In this case, the messages can be received in any order.

Here is an example of using non-overtaking messages. Note that the message sent by the first send must be received by the first receive, and the message sent by the second send must be received by the second receive.

```
MPI_Comm_rank(comm,&rank);
if(rank == 0){
    MPI_Bsend(buf1,count,MPI_FLOAT,1,tag,comm);
    MPI_Bsend(buf2,count,MPI_FLOAT,1,tag,comm);
}
else{
    /* rank == 1 */
    MPI_Recv(buf1, count, MPI_FLOAT,0,MPI_ANY_TAG,comm,&status);
    MPI_Recv(buf2, count, MPI_FLOAT,0,tag,comm,&status);
}
```

## Progress

If two processes initiate two matching sends and receives, at least one of the operations (the send or the receive) will complete, regardless of other actions that occur in the system. The send operation will complete unless its matching receive has already been satisfied by another message, and has itself completed. Likewise, the receive will complete unless its matching send message is claimed by another matching receive that was posted at the same destination.

The following example shows two matching pairs that are intertwined in this manner. Here is what happens:

1. Both processes invoke their first calls.
2. *process 0*'s first send indicates buffered mode, which means it must complete, even if there is no matching receive. Since the first receive posted by *process 1* does not match, the send message gets copied into buffer space.
3. Next, *process 0* posts its second send operation, which matches *process 1*'s first receive, and both operations complete.
4. *process 1* then posts its second receive, which matches the buffered message, so both complete.

```
MPI_Comm_rank(comm,&rank);
if(rank == 0){
    MPI_Bsend(buf1,count,MPI_FLOAT,1,tag1,comm);
    MPI_Ssend(buf2,count,MPI_FLOAT,1,tag2,comm);
}
else{
    /* rank == 1 */
    MPI_Recv(buf1, count, MPI_FLOAT,0,tag2,comm,&status);
    MPI_Recv(buf2, count, MPI_FLOAT,0,tag1,comm,&status);
}
```

## Fairness

MPI does not guarantee *fairness* in the way communications are handled, so it is your responsibility to prevent starvation among the operations in your program.

So what might *unfairness* look like? An example might be a situation where a send with a matching receive on another process doesn't complete because another message, from a different process, overtakes the receive.

## Resource limitations

If a lack of resources prevents an MPI call from executing, errors may result. Pending send and receive operations consume a portion of your system resources. A good MPI program uses only a small amount of resources for each pending send and receive, but this buffer space is required for storing messages sent in either standard or buffered mode when no matching receive is available.

When a buffered send operation cannot complete because of a lack of buffer space, the resulting error could cause your program to terminate abnormally. On the other hand, a standard send operation that cannot complete because of a lack of buffer space will merely block and wait for buffer space to become available or for the matching receive to be posted. In some situations, this behavior is preferable because it avoids the error condition that is associated with buffer overflow.

Sometimes a lack of buffer space can lead to deadlock. The program in the example below will succeed even if no buffer space for data is available.

```
MPI_Comm_rank(comm,&rank);
if(rank == 0){
    MPI_Send(sendbuf,count,MPI_FLOAT,1,tag,comm);
    MPI_Recv(recvbuf,count,MPI_FLOAT,1,tag,comm,&status);
}
else{
    /* rank == 1 */
    MPI_Recv(recvbuf, count, MPI_FLOAT,0,tag,comm,&status);
    MPI_Send(sendbuf, count, MPI_FLOAT,0,tag,comm);
}
```

In this next example, neither process will send until other the process sends first. As a result, this program will always result in deadlock.

```
MPI_Comm_rank(comm,&rank);
if(rank == 0){
    MPI_Recv(recvbuf,count,MPI_FLOAT,1,tag,comm,&status);
    MPI_Send(sendbuf,count,MPI_FLOAT,1,tag,comm);
}
else{
    /* rank == 1 */
    MPI_Recv(recvbuf, count, MPI_FLOAT,0,tag,comm,&status);
    MPI_Send(sendbuf, count, MPI_FLOAT,0,tag,comm);
}
```

The example below shows how message exchange relies on buffer space. The message send by each process must be copied out before the send returns and the receive starts. Consequently, at least one of the two messages sent needs to be buffered in order for the program to complete. As a result, this program can execute successfully only if the communication system can buffer at least the words of data specified by *count*.

```
MPI_Comm_rank(comm,&rank);
if(rank == 0){
    MPI_Send(sendbuf,count,MPI_FLOAT,1,tag,comm);
    MPI_Recv(recvbuf,count,MPI_FLOAT,1,tag,comm,&status);
}
else{
    /* rank == 1 */
    MPI_Send(sendbuf, count, MPI_FLOAT,0,tag,comm);
    MPI_Recv(recvbuf, count, MPI_FLOAT,0,tag,comm,&status);
}
```

When standard send operations are used, deadlock can occur where both processes are blocked because buffer space is not available. This is also true for synchronous send operations. For buffered sends, if the required amount of buffer space is not available, the program won't complete either, and instead of deadlock, we'll have buffer overflow.

---

## Appendix E. Copying Parallel Environment Executables in a Security Environment

In a security environment where the BPX.DAEMON facility class is defined the programs

- pmd,
- poe (for V2R7 and following releases) and
- pdbx (for V2R7 and following releases)

must be defined as 'program controlled' to the security manager. In addition poe, pmd, pdbx and mpcc/mpCC compiled applications (including mcp, mcpgath and mcpscat) must have the 'share address space' attribute unset. The HFS extended attributes for both 'program controlled' and 'share address space' get lost during HFS copies. Therefore these extended attributes must be explicitly set or unset respectively. The system administrator's userid must be granted permission to the BPX.FILEATTR.PROGCTL RACF® profile in order to grant the necessary authority to set the 'program controlled' extended attribute bit. The extattr command from the UNIX System Services shell to label a program as being program controlled. For example:

```
extattr +p /mybin/pmd
```

In case this fails the administrator may have to perform the next two steps to get permission to the BPX.FILEATTR.PROGCTL RACF profile.

1. Create the BPX.FILEATTR.PROGCTL RACF profile and grant READ permission to the system programmer's id.
2. Perform a SETROPTS RACLIST(FACILITY) REFRESH.

To unset the 'share address space' extended attribute of a program the owner of the program can use the extattr command from the UNIX System Services shell. For example:

```
extattr -s /mybin/mcp
```

To check the extended attributes of a file, use the command:

```
ls -E <file>
```



---

## Appendix F. Migration and Multi-release Compatibility

---

### Parallel Environment Compatibility

The partition manager (poe), the partition manager daemons (pmd) and the MPI libraries (ppe.dll, libppe.a) of different Parallel Environment releases are incompatible. Therefore the same release of Parallel Environment must be used on all systems that participate in running a parallel program with poe. To enable running a parallel program on systems with different operating system releases back-level releases of Parallel Environment must be installed on the newer systems. This support is not part of the default installation of Parallel Environment. This appendix describes how to set up the Parallel Environment in a way that multiple versions of PE may coexist on the same system.

---

### Parallel Environment Releases

Since the Parallel Environment is shipped as part of z/OS UNIX System Services, it may not be obvious whether different systems run different releases of Parallel Environment. The following table shows the relation between Parallel Environment releases and OS/390 or z/OS releases.

*Table 18. Relation between OS releases, Parallel Environment releases and PTF's*

OS release	Parallel Environment release	Compatibility PTF
OS/390 Version 2 Release 4	Release 1	OW64380
OS/390 Version 2 Release 5	Release 1	OW64381
OS/390 Version 2 Release 6	Release 1	OW64382
OS/390 Version 2 Release 7	Release 2	OW64383
OS/390 Version 2 Release 8	Release 2	OW64384
OS/390 Version 2 Release 9	Release 3	n/a
z/OS Version 1 Release 1	Release 3	n/a

**Note:** All statements made in the following for OS/390 V2R9 also apply to z/OS V1R1.

Since publications never mention Parallel Environment release numbers we will henceforth only refer to OS/390 releases. With regard to the OS/390 Parallel Environment, OS/390 releases that share the same release of the Parallel Environment are considered equivalent.

---

### Principles of Multi-release Compatibility

The architecture of Parallel Environment requires that the poe, all pmds and the MPI libraries (ppe.dll or libppe.dll) are of the same release of Parallel Environment. Therefore the same Parallel Environment release must be installed on all systems that participate in the execution of a parallel program. This Parallel Environment release must be the oldest of all systems participating in the execution of the parallel program. Henceforth we will call this release the *partition release*. Note, the partition release may not be older than the Parallel Environment release used to build the parallel program. The general idea of a Multi-release Compatibility setup

is to install the partition release on all newer systems participating in the execution of a parallel program. This implies that on these newer systems a back-level release of Parallel Environment must be installed in addition to the release available with the standard OS/390 installation. Once common (back-level) releases of Parallel Environment are installed on a set of systems the user can control which release of Parallel Environment shall execute a parallel program by setting the environment variables PATH, MP\_PMD\_SUFFIX and MP\_EUILIBPATH appropriately. Note, a precondition to any compatibility solution for PE is that the user (and the application) adheres to the compatibility guidelines set forth by the Language Environment® and the C/C++ compilers.

---

## Installing a Back-level Release

To install a back-level release of Parallel Environment on a system the key components of Parallel Environment of that release must be copied to appropriate locations in the systems' HFSes and access to these components must be ensured. The key components of PE are

- poe,
- pmd and
- ppe.dll.

These installation steps must be performed by a system administrator (UNIX super user).

### Installing a Back-level poe

The releases of Parallel Environment shipped before OS/390 Version 2 Release 9 did not support the coexistence of multiple PE releases on a single system. To fix this problem the PTFs mentioned in Table 18 on page 299 must be installed first. Subsequently the (fixed) poe executable from the partition release must be copied to an appropriate location in the HFSes of the newer releases. As location we suggest the HFS directory */samples/pe/Rx/bin* where Rx denotes the OS/390 release of the partition release (e.g. R4 or R7). The back-level version of poe must have read and execution permission for the world. For example set

```
chmod a+rx /samples/pe/Rx/bin/poe
```

to grant read and execution permission of */samples/pe/Rx/bin/poe* to all users.

See "Appendix E. Copying Parallel Environment Executables in a Security Environment" on page 297 if the system is running in a security environment.

### Installing a Back-level pmd

The pmd executable of the partition release must be copied to an appropriate location in the HFSes of the newer releases. As location we suggest the HFS directory */samples/pe/Rx/bin* where Rx denotes the OS/390 release of the partition release (e.g. R4 or R7). The back-level version of pmd must have read and execution permission for the world. For example set

```
chmod a+rx /samples/pe/Rx/bin/pmd
```

to grant read and execution permission of */samples/pe/Rx/bin/pmd* to all users.

See "Appendix E. Copying Parallel Environment Executables in a Security Environment" on page 297 if the system is running in a security environment.

The pmd is started by the internet daemon inetd. Therefore a new service denoting the partition release of the pmd must be registered on all systems participating in

running the parallel program. Such services are defined in */etc/services* and */etc/inetd.conf*. For example the administrator may enter the following line to */etc/services*:

```
pmv2Rx    6128/tcp    #POE Partition Manager Daemon for OS/390 Version 2 Release x
```

The port number chosen (here 6128) must be the same on all systems and it may not conflict with existing or reserved port numbers. In */etc/inetd.conf* the following line might be added on all newer releases:

```
pmv2Rx stream tcp nowait OMVSKERN /samples/pe/Rx/bin/pmd pmd
```

and the following line might be added on systems where the partition release is installed by default:

```
pmv2Rx stream tcp nowait OMVSKERN /bin/pmd pmd
```

After changing these configuration files the *inetd* must be forced to read them. This can be done by restarting the *inetd* or by sending the *SIGHUP* signal to the *inetd*:

```
kill -HUP <pid of inetd>
```

## Installing a Back-level ppe.dll

The MPI library *ppe.dll* of the partition release must be copied to an appropriate location in the HFSes of the newer releases. As location we suggest the HFS directory */samples/pe/Rx/lib* where *Rx* denotes the OS/390 release of the partition release (e.g. R4 or R7). The back-level version of *ppe.dll* must have read and execution permission for the world. For example

```
chmod a+rx /samples/pe/Rx/lib/ppes.dll
```

---

## Running a Back-level Release of Parallel Environment

Once the partition release of Parallel Environment is installed on all systems in a mixed OS/390 release environment this release can be chosen to run a parallel program by the user. To do this the following environment variables must be set appropriately.

1. The *PATH* environment variable must be set to point to the **poe** executable of the partition release.
2. The *MP\_PMDSUFFIX* environment variable must be set to the suffix of the service calling the *pmd* of the partition release
3. The *MP\_EUILIBPATH* variable must be set to point to the *ppe.dll* of the partition release. **poe** will concatenate the *MP\_EUILIBPATH* and the *LIBPATH* and use the result to link to *ppe.dll*.

If the partition release has been installed as suggested above this may look as follows:

```
export PATH=/samples/pe/Rx/bin:$PATH
export MP_PMDSUFFIX=Rx
export MP_EUILIBPATH=/samples/pe/Rx/lib
```

Note that these environment variables need only to be set on the system used to start the parallel program. **poe** will send these settings to all systems participating in the execution of the parallel program.

---

## Multi-release Compatibility for Statically Linked Programs

If the parallel program is statically linked then the aforementioned set-up must be modified as follows: for statically linked parallel programs the partition release must always be equal to the Parallel Environment release used to build the parallel program.

---

## Servicing Back-level Parallel Environment Releases

Note SMP/E based service to PE can only be applied to the OS/390 release it was shipped with. Therefore applying service to components of a back-level PE installation means:

1. Apply SMP/E based service to the OS/390 release from which the back-level PE release was copied.
2. Copy the affected components from the HFS of the serviced back-level system to the HFS of the target system.

---

## Appendix G. Notices

This information was developed for products and services offered in the USA.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described 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  
North Castle Drive  
Armonk, NY 10504-1785  
USA

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation  
Licensing  
2-31 Roppongi 3-chome, Minato-ku  
Tokyo 106, Japan

**The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:**

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

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 Deutschland Entwicklung GmbH  
Information Development  
Department 3248  
Schönaicher Str. 220  
D-71032 Böblingen  
Germany

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

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

---

## Trademarks

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

IBM  
Intelligent Miner  
Language Environment  
OpenEdition  
OS/2  
OS/390  
Parallel Sysplex  
RACF  
S/390  
z/OS

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, and service names may be trademarks or service marks of others.

Some of the references and quotes herein are from *MPI: A Message-Passing Interface Standard, Version 1.1* by Message Passing Interface Forum, June 6, 1995. Permission to copy *MPI: A Message-Passing Interface Standard, Version 1.1* by Message Passing Interface Forum, is granted, provided the University of Tennessee copyright notice and the title of the document appear, and notice is given that copying is by permission of the University of Tennessee. © 1993, 1995 University of Tennessee, Knoxville, Tennessee.

---

## Accessing Licensed Books on the Web

z/OS licensed documentation in PDF format is available on the Internet at the IBM Resource Link Web site at:

[www.ibm.com/servers/resourceLink](http://www.ibm.com/servers/resourceLink)

Licensed books are available only to customers with a z/OS license. Access to these books requires an IBM Resource Link Web userid and password, and a key code. With your z/OS order you received a memo that includes this key code.

To obtain your IBM Resource Link Web userid and password log on to:

[www.ibm.com/servers/resourceLink](http://www.ibm.com/servers/resourceLink)

To access the licensed books:

1. Log on to Resource Link using your Resource Link user ID and password.
2. When prompted, enter the key code.
3. Click on **Library**.
4. Click on **z/OS**.
5. Navigate to the licensed documents.

---

## LookAt System for Online Message Lookup

LookAt is an online facility that allows you to look up explanations for z/OS messages and system abends.

Using LookAt to find information is faster than a conventional search because LookAt goes directly to the explanation.

LookAt can be accessed from the Internet or from a TSO command line.

You can use LookAt on the Internet at:

[www.ibm.com/servers/eserver/zseries/zos/bkserv/lookat/lookat.html](http://www.ibm.com/servers/eserver/zseries/zos/bkserv/lookat/lookat.html)

To use LookAt as a TSO command, LookAt must be installed on your host system. You can obtain the LookAt code for TSO from the LookAt Web site by clicking on the **News and Help** link or from the *z/OS Collection*.

To find a message explanation from a TSO command line, simply enter: **lookat message-id** as in the following:

```
lookat cbda100i
```

This results in direct access to the message explanation for message CBDA100I.

To find a message explanation from the LookAt Web site, simply enter the message ID. You can select the release if needed.

**Note:** Some messages have information in more than one book. For such messages, LookAt prompts you to choose which book to open.



---

# Bibliography

---

## Related publications

### Parallel Environment publications

- *z/OS UNIX System Services Parallel Environment: Operation and Use*, SA22-7810
- *z/OS UNIX System Services Parallel Environment: MPI Programming and Subroutine Reference*, SA22-7812

### Related z/OS publications

- *z/OS UNIX System Services Command Reference*, SA22-7802
- *z/OS UNIX System Services Messages and Codes*, SA22-7807
- *z/OS UNIX System Services Planning*, GA22-7800
- *z/OS UNIX System Services Programming Tools*, SA22-7805

### Related non-IBM publications

- Message Passing Interface Forum: *MPI: A Message-Passing Interface Standard, Version 1.1*, University of Tennessee, Knoxville, Tennessee, June 6, 1995
- Message Passing Interface Forum: *MPI-2: Extensions to the Message-Passing Interface, Version 2.0*, University of Tennessee, Knoxville, Tennessee, July 18, 1997
- Almasi, G., Gottlieb, A.: *Highly Parallel Computing*, Benjamin-Cummings Publishing Company, Inc., 1989
- Foster, I.: *Designing and Building Parallel Programs*, Addison Wesley, 1995
- Gropp, W., Lusk, E., Skjellum, A.: *Using MPI*, The MIT Press, 1994
- Bergmark, D., Pottle, M.: *Optimization and Parallelization of a Commodity Trade Model for the SP1*, Cornell Theory Center, Cornell University, June, 1994
- Phister, Gregory F.: *In Search of Clusters* (2nd ed.), Prentice Hall, 1998
- Snir, M., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J.: *MPI: The Complete Reference*, The MIT Press, 1996
- Spiegel, Murray R.: *Vector Analysis*, McGraw-Hill, 1959



---

## Glossary of terms and abbreviations

This glossary includes terms and definitions from:

- The *Dictionary of Computing*, New York: McGraw-Hill, 1994.
- The *American National Standard Dictionary for Information Systems*, ANSI X3.172-1990, copyright 1990 by the American National Standards Institute (ANSI). Copies can be purchased from the American National Standards Institute, 1430 Broadway, New York, New York 10018. Definitions are identified by the symbol (A) after the definition.
- The *ANSI/EIA Standard - 440A: Fiber Optic Terminology*, copyright 1989 by the Electronics Industries Association (EIA). Copies can be purchased from the Electronic Industries Association, 2001 Pennsylvania Avenue N.W., Washington, D.C. 20006. Definitions are identified by the symbol (E) after the definition.
- The *Information Technology Vocabulary* developed by Subcommittee 1, Joint Technical Committee 1, of the International Organization for Standardization and the International Electrotechnical Commission (ISO/IEC JTC1/SC1). Definitions of published parts of this vocabulary are identified by the symbol (I) after the definition; definitions taken from draft international standards, committee drafts, and working papers being developed by ISO/IEC JTC1/SC1 are identified by the symbol (T) after the definition, indicating that final agreement has not yet been reached among the participating National Bodies of SC1.

This glossary defines technical terms and abbreviations used in *z/OS UNIX System Services Parallel Environment* documentation. If you do not find the term you are looking for, refer to the index of the appropriate *z/OS UNIX System Services Parallel Environment* manual or view *IBM Dictionary of Computing*, available from: <http://www.ibm.com/ibm/terminology>.

This glossary includes terms and definitions from:

- *Vocabulary for Information Processing*, ANSI, copyright 1970 by American National Standards Institute (ANSI). It was prepared by Subcommittee X3K5 on Terminology and Glossary of the American National Standards Committee X3. Copies may be purchased from the American National Standards Institute, 11

West 42nd Street, New York, New York 10036, USA. Definitions are identified by the symbol (A) after the definition.

Other definitions in this glossary are taken from *IBM Dictionary of Computing*.

### A

**address.** A value, possibly a character or group of characters that identifies a register, a device, a particular part of storage, or some other data source or destination.

**API.** Application Programming Interface.

**application.** The use to which a data processing system is put; for example, to payroll application, an airline reservation application.

**argument.** A parameter passed between a calling program and a called program or subprogram.

**attribute.** A named property of an entity.

### B

**bandwidth.** The total available bit rate of a digital channel.

**blocking operation.** An operation which does not complete until the operation either succeeds or fails. For example, a blocking receive will not return until a message is received or until the channel is closed and no further messages can be received.

**breakpoint.** A place in a program, specified by a command or a condition, where the system halts execution and gives control to the workstation user or to a specified program.

**broadcast operation.** A communication operation in which one processor sends (or broadcasts) a message to all other processors.

**buffer.** A portion of storage used to hold input or output data temporarily.

### C

**C.** A general purpose programming language. It was formalized by ANSI standards committee for the C language in 1984 and by Uniforum in 1983.

**C++.** A general purpose programming language that is based on C, which includes extensions that support an object-oriented programming paradigm. Extensions include:

- strong typing
- data abstraction and encapsulation
- polymorphism through function overloading and templates
- class inheritance.

**client.** A function that requests services from a server, and makes them available to the user.

**cluster.** A group of processors interconnected through a high speed network that can be used for high performance computing. It typically provides excellent price/performance.

**collective communication.** A communication operation which involves more than two processes or tasks. Broadcasts, reductions, and the MPI\_Allreduce subroutine are all examples of collective communication operations. All tasks in a communicator must participate.

**communicator.** An MPI object that describes the communication context and an associated group of processes.

**compatibility mode.** A mode of processing, in which the IEAIPsxx and IEAICSxx parmlib members determine system resource management. See also *goal mode*.

**compile.** To translate a source program into an executable program.

**condition.** One of a set of specified values that a data item can assume.

**core dump.** A process by which the current state of a program is preserved in a file. Core dumps are usually associated with programs that have encountered an unexpected, system-detected fault, such as a Segmentation Fault, or severe user error. The current program state is needed for the programmer to diagnose and correct the problem.

**core file.** A file which preserves the state of a program, usually just before a program is terminated for an unexpected error. See also *core dump*.

## D

**data decomposition.** A method of breaking up (or decomposing) a program into smaller parts to exploit parallelism. One divides the program by dividing the data (usually arrays) into smaller parts and operating on each part independently.

**data parallelism.** Refers to situations where parallel tasks perform the same computation on different sets of data.

**dbx.** A symbolic command line debugger that is often provided with UNIX systems.

**debugger.** A debugger provides an environment in which you can manually control the execution of a program. It also provides the ability to display the program's data and operation.

**domain name.** The hierarchical identification of a host system (in a network), consisting of human-readable labels, separated by decimals.

## E

**environment variable.** 1. A variable that describes the operating environment of the process. Common environment variables describe the home directory, command search path, and the current time zone. 2. A variable that is included in the current software environment and is therefore available to any called program that requests it.

**event.** An occurrence of significance to a task; for example, the completion of an asynchronous operation such as an input/output operation.

**executable.** A program that has been link-edited and therefore can be run in a processor.

**execution.** To perform the actions specified by a program or a portion of a program.

**expression.** In programming languages, a language construct for computing a value from one or more operands.

## F

**fairness.** A policy in which tasks, threads, or processes must be allowed eventual access to a resource for which they are competing. For example, if multiple threads are simultaneously seeking a lock, then no set of circumstances can cause any thread to wait indefinitely for access to the lock.

**FDDI.** Fiber distributed data interface (100 Mbit/s fiber optic LAN).

**file system.** A collection of files and file management structures on a physical or logical mass storage device.

**filesset.** 1) An individually installable option or update. Options provide specific function while updates correct an error in, or enhance, a previously installed product. 2) One or more separately installable, logically grouped units in an installation package. See also *Licensed Program Product* and *package*.

**foreign host.** See *remote host*.

**functional decomposition.** A method of dividing the work in a program to exploit parallelism. One divides the program into independent pieces of functionality which are distributed to independent processors. This is in contrast to data decomposition which distributes the same work over different data to independent processors.

**functional parallelism.** Refers to situations where parallel tasks specialize in particular work.

## G

**global max.** The maximum value across all processors for a given variable. It is global in the sense that it is global to the available processors.

**global variable.** A variable defined in one portion of a computer program and used in at least one other portion of the computer program.

**goal mode.** A mode of processing where the active service policy determines system resource management. See also *compatibility mode*.

**GUI (Graphical User Interface).** A type of computer interface consisting of a visual metaphor of a real-world scene, often of a desktop. Within that scene are icons, representing actual objects, that the user can access and manipulate with a pointing device.

## H

**home node.** The node from which an application developer compiles and runs his program. The home node can be any workstation on the LAN.

**host.** A computer connected to a network, and providing an access method to that network. A host provides end-user services.

**host list file.** A file that contains a list of host names, and possibly other information, that was defined by the application which reads it.

**host name.** The name used to uniquely identify any computer on a network.

**hot spot.** A memory location or synchronization resource for which multiple processors compete excessively. This competition can cause a disproportionately large performance degradation when one processor that seeks the resource blocks, preventing many other processors from having it, thereby forcing them to become idle.

## I

**IBM Parallel Environment for z/OS.** A licensed program that provides an execution and development environment for parallel C or C++ programs. It supports the Parallel Intelligent Miner™ for z/OS.

**InfoExplorer.** A program that displays hypertext and allows navigation within it.

**Internet.** The collection of worldwide networks and gateways which function as a single, cooperative virtual network.

**Internet Protocol (IP).** A protocol that defines how information gets passed between systems in a network. It is used to route data from its source to its destination in an Internet environment.

**IP.** See *Internet Protocol*.

## K

**kernel.** The core portion of the UNIX operating system which controls the resources of the CPU and allocates them to the users. The kernel is memory-resident, is said to run in *kernel mode* (in other words, at higher execution priority level than *user mode*) and is protected from user tampering by the hardware.

## L

**latency.** The time interval between the instant at which an instruction control unit initiates a call for data transmission, and the instant at which the actual transfer of data (or receipt of data at the remote end) begins. Latency is related to the hardware characteristics of the system and to the different layers of software that are involved in initiating the task of packing and transmitting the data.

**local variable.** A variable that is defined and used only in one specified portion of a computer program.

**loop unrolling.** A program transformation which makes multiple copies of the body of a loop, placing the copies also within the body of the loop. The loop trip count and index are adjusted appropriately so the new loop computes the same values as the original. This transformation makes it possible for a compiler to take additional advantage of instruction pipelining, data cache effects, and software pipelining.

See also *optimization*.

## M

**menu.** A list of options displayed to the user by a data processing system, from which the user can select an action to be initiated.

**message catalog.** A file created using the Message Facility from a message source file that contains application error and other messages, which can later be translated into other languages without having to recompile the application source code.

**message passing.** Refers to the process by which parallel tasks explicitly exchange program data.

**MPMD (Multiple Program Multiple Data).** A parallel programming model in which different, but related, programs are run on different sets of data.

**MPI.** Message Passing Interface; a standardized API for implementing the message passing model.

## N

**network.** An interconnected group of nodes, lines, and terminals. A network provides the ability to transmit data to and receive data from other systems and users.

**node.** (1) In a network, the point where one or more functional units interconnect transmission lines. A computer location defined in a network. (2) In terms of a sysplex, or systems connected through a network, a single location or workstation in a network.

**node ID.** A string of unique characters that identifies the node on a network.

**nonblocking operation.** An operation, such as sending or receiving a message, which returns immediately whether or not the operation was completed. For example, a nonblocking receive will not wait until a message is sent, but a blocking receive will wait. A nonblocking receive will return a status value that indicates whether or not a message was received.

## O

**object code.** The result of translating a computer program to a relocatable, low-level form. Object code contains machine instructions, but symbol names (such as array, scalar, and procedure names), are not yet given a location in memory.

**optimization.** A not strictly accurate but widely used term for program performance improvement, especially for performance improvement done by a compiler or other program translation software. An optimizing compiler is one that performs extensive code transformations in order to obtain an executable that runs faster but gives the same answer as the original. Such code transformations, however, can make code debugging and performance analysis very difficult because complex code transformations obscure the correspondence between compiled and original source code.

**option flag.** Arguments or any other additional information that a user specifies with a program name. Also referred to as *parameters* or *command-line options*.

## P

**parallelism.** The degree to which parts of a program may be concurrently executed.

**parallelize.** To convert a serial program for parallel execution.

**Parallel Operating Environment (POE).** An execution environment that smooths the differences between serial and parallel execution. It lets you submit and manage parallel jobs. It is abbreviated and commonly known as POE.

**parameter.** (1) An item in a menu for which the operator specifies a value or for which the system provides a value when the menu is interpreted. (2) A name in a procedure that is used to refer to an argument that is passed to the procedure. (3) A particular piece of information that a system or application program needs to process a request.

**Partition Manager.** The component of the Parallel Operating Environment (POE) that allocates nodes, sets up the execution environment for remote tasks, and manages distribution or collection of standard input (STDIN), standard output (STDOUT), and standard error (STDERR).

**PE.** The Parallel Environment program product.

**performance monitor.** A utility which displays how effectively a system is being used by programs.

**POE.** See Parallel Operating Environment.

**point-to-point communication.** A communication operation which involves exactly two processes or tasks. One process initiates the communication through a *send* operation. The partner process issues a *receive* operation to accept the data being sent.

**procedure.** (1) In a programming language, a block, with or without formal parameters, whose execution is invoked by means of a procedure call. (2) A set of related control statements that cause one or more programs to be performed.

**process.** A program or command that is actually running the computer. It consists of a loaded version of the executable file, its data, its stack, and its kernel data structures that represent the process's state within a multitasking environment. The executable file contains the machine instructions (and any calls to shared objects) that will be executed by the hardware. A process can contain multiple threads of execution.

The process is created via a `fork()` system call and ends using an `exit()` system call. Between `fork` and `exit`, the process is known to the system by a unique process identifier (pid).

Each process has its own virtual memory space and cannot access another process's memory directly. Communication methods across processes include pipes, sockets, shared memory, and message passing.

**profiling.** The act of determining how much CPU time is used by each function or subroutine in a program. The histogram or table produced is called the execution profile.

**program marker array.** An X-Windows run time monitor tool provided with the Parallel Operating Environment used to provide immediate visual feedback on a program's execution.

## R

**reduction operation.** An operation, usually mathematical, which reduces a collection of data by one or more dimensions. For example, the arithmetic SUM operation is a reduction operation which reduces an array to a scalar value. Other reduction operations include MAXVAL and MINVAL.

**remote host.** Any host on a network except the one at which a particular operator is working.

## S

**shell script.** A sequence of commands that are to be executed by a shell interpreter such as C shell, korn shell, or Bourne shell. Script commands are stored in a file in the same form as if they were typed at a terminal.

**segmentation fault.** A system-detected error, usually caused by referencing an invalid memory address.

**server.** A functional unit that provides shared services to workstations over a network; for example, a file server, a print server, a mail server.

**source line.** A line of source code.

**source code.** The input to a compiler or assembler, written in a source language. Contrast with object code.

**SPMD (Single Program Multiple Data).** A parallel programming model in which different processors execute the same program on different sets of data.

**standard input (STDIN).** In the Parallel Operating Environment for z/OS system, the primary source of data entered into a command. Standard input comes from the keyboard unless redirection or piping is used, in which case standard input can be from a file or the output from another command.

**standard output (STDOUT).** In the Parallel Operating Environment for z/OS system, the primary destination of data produced by a command. Standard output goes to the display unless redirection or piping is used, in which case standard output can go to a file or to another command.

**subroutine.** (1) A sequence of instructions whose execution is invoked by a call. (2) A sequenced set of instructions or statements that may be used in one or more computer programs and at one or more points in a computer program. (3) A group of instructions that can be part of another routine or can be called by another program or routine.

**synchronization.** The action of forcing certain points in the execution sequences of two or more asynchronous procedures to coincide in time.

**system administrator.** (1) The person at a computer installation who designs, controls, and manages the use of the computer system. (2) The person who is responsible for setting up, modifying, and maintaining the Parallel Environment.

## T

**task.** A unit of computation analogous to a UNIX process.

## U

**User Datagram Protocol (UDP).** A communications protocol that offers limited service for messages exchange between applications in a network which uses the Internet Protocol (IP). It neither adds reliability, flow control nor error recovery to IP.

**user.** (1) A person who requires the services of a computing system. (2) Any person or any thing that may issue or receive commands and message to or from the information processing system.

**utility program.** A computer program in general support of computer processes; for example, a diagnostic program, a trace program, a sort program.

**utility routine.** A routine in general support of the processes of a computer; for example, an input routine.

## V

**variable.** (1) In programming languages, a named object that may take different values, one at a time. The values of a variable are usually restricted to one data type. (2) A quantity that can assume any of a given set of values. (3) A name used to represent a data item whose value can be changed while the program is running. (4) A name used to represent data whose value can be changed, while the program is running, by referring to the name of the variable.

**view.** (1) In an information resource directory, the combination of a variation name and revision number that is used as a component of an access name or of a descriptive name.

## W

**workload.** A group of work to be tracked, managed, and reported as a unit. Also, a group of service classes.

**workload management mode.** The mode in which workload management manages system resources on an MVS image. The mode can be either *compatibility mode*, or *goal mode*.

# Index

## Special Characters

- hostfile option 14
- infolevel option 11
- infolevel option, -ilevel option 14
- labelio option 8, 14
- pmdlog option 14
- procs option 13
- stdoutmode option 9, 14

## A

- access, to nodes 4
- address, IP 100
- administration, PE 3
- aliases
  - creating, removing, and listing with pdbx 121
- allocation of nodes, specific and automatic 18
- application
  - parallelizing the application 1
- Application Message Queues
  - window 170
- application programming interface (API)
  - mpc\_disableintr() 43
  - mpc\_enableintr() 44
  - mpc\_queryintr() 43
  - mpc\_queryintrdelay() 43
  - mpc\_setintrdelay(int val) 43
- area in pedb
  - break/trace area 139
  - global data area 139
  - local data area 139
  - stack data area 139
  - task area 139
  - threads area 139
- argument
  - command-line arguments for pedb 133
  - condition 111
  - ignoring argument list within POE 32
  - ignoring within POE 31
  - procedure 111
  - source\_line 111
  - string 68
- arithmetic operations 122
- asynchronous interrupts support 41
- Attach Dialog window 82
- Attach screen, pdbx debugger 100
- attaching the debugger
  - \_BPX\_PTRACE\_ATTACH 81
  - Attach Dialog window 82
  - Global data window 85
  - pedb main window 83
  - problems 87
- automatic node allocation 18, 21
- automatic node allocation, example 23

## B

- back-level release 300
- blocking read 109

- blocking send 116
- BPX.DAEMON 6
- breakpoint
  - blocking read 109
  - break/trace area in pedb 139
  - conditional breakpoint in pedb 148
  - conditional breakpoint window 149
  - conditions, in pedb 147
  - deleting 114
  - identifying tasks 149
  - redefining initial, automatic 124
  - removing 149
  - setting 109
  - setting, in pedb 134, 146
  - setting in pdbx 111
  - thread specific conditions 149
- buffer 35
  - pedb main window 139
- button
  - play control 152

## C

- call stacks 117
- cancelling a POE job 41
- CEEBINT 65
- CEEBXITA 65
- codes
  - debugging SPMD and MPMD codes with pedb 134
- collective communication 196
- collective communications 71
- Collective Communications Details
  - window 176
- command
  - extattr 297
- command-line flags
  - for Normal or Attach Mode 291
  - for POE 283
- command man
  - pdbx subcommands 94
- commandline flags, POE 22
- commands, PE 185
- common problems
  - attaching the debugger 87
  - bad output, attaching the debugger 88
  - can't compile a parallel program 74
  - can't connect with the remote host 75
  - can't execute a parallel program 76
  - can't start a parallel job 75
  - debugging a parallel program 77
  - no output 78
  - no output or bad output 77
- communication subsystem library 1
- compile flags
  - cpp 15
  - D\_THREAD\_SAFE 16
  - W c,dll,'LANG(EXTENDED)' 16
- compile problems 74

- compiler scripts 10
- compiling
  - C example with MPI function calls 10
- compiling parallel C and C++ programs
  - c89, c++ 15
  - c89, c++ complile flags 16
  - enabling the MPI-2 C++ bindings 15
  - mpcc, mpCC 15
- compiling with mpcc 11
- copy files to individual nodes 16
- copy utilities
  - mcp 44, 46
  - mcpqath 44, 46
  - mcpscat 44, 46

## D

- data decomposition 51, 52
- data mining 58
- dbx subcommands 214
  - help for subcommands 120
- debugger
  - \_BPX\_PTRACE\_ATTACH 81
  - Attach dialog window 82
  - attaching to POE job 81
  - Global data window 85
  - MP\_DEBUG\_INITIAL\_STOP 96
  - option flags for pdbx 97
  - pdbx, attach mode 99
  - pdbx, normal mode 96
  - pedb 127
  - pedb main window 83
  - starting pdbx 96
- debugging
  - pmarray 89
  - threaded programs 88
- debugging parallel programs
  - with pdbx 93
  - with pedb 127
- decomposition
  - data 51
  - functional 51, 58
- directories, changing on remote nodes 39
- dynamic libraries 16

## E

- Early Arrival Message Details
  - window 176
- environment
  - scheduling environment 5
  - security environment 6
  - setting up for debugging with pedb 127
- environment variables
  - \_BPX\_PTRACE\_ATTACH 81
  - \_BPX\_SHAREAS 6
  - LANG 73
  - MP\_DEBUG\_INITIAL\_STOP 96

- environment variables (*continued*)
  - MP\_EUIDEVELOP, error messages
    - when attaching the debugger 88
  - MP\_EUILIBPATH 300
  - MP\_HOSTFILE 14
  - MP\_HOSTFILE, problems starting a
    - parallel job 75
  - MP\_INFOLEVEL 14
  - MP\_INFOLEVEL, problems
    - connecting a remote host 76
  - MP\_LABELIO 14
  - MP\_LABELIO, error messages when
    - attaching the debugger 88
  - MP\_PMDLOG 14
  - MP\_PMD\_SUFFIX 300
  - MP\_PMLIGHTS 47
  - MP\_PROCS 13, 22
  - MP\_PROCS, problems starting a
    - parallel job 75
  - MP\_RESD, problems starting a
    - parallel job 75
  - MP\_RESD, problems starting a
    - parallel program 76
  - MP\_RMPOOL, problems starting a
    - parallel job 75
  - MP\_SAVEHOSTFILE 24
  - MP\_SNDBUF 69
  - MP\_STDOUTMODE 14
  - MP\_STDOUTMODE, output
    - problems 78
  - MP\_THREAD\_STACKSIZE 67
  - MP\_USRPORT 48
  - MP\_WLM\_ENCLAVE 24
  - NLSPATH 73
  - PAGER 99
  - PATH 136, 300
  - POE 283
  - reserved 71
- errors
  - logging to a file 74
  - messages, attaching the debugger 88
- events
  - changing 115
  - deleting 114
- executable
  - creating 15
  - invoking 25
  - POE additions 65
- executing parallel programs 15
- execution
  - automatically repeat execution within
    - pedb 151
  - controlled with pdbx debugger 110
  - interrupt a waiting process 151
  - stepping execution within pedb 150
  - tracing program execution within
    - pedb 153
- execution environment 17
- exit status 64
- exits, normal and abnormal 67
- expressions, specifying with pdbx 122
- F**
  - file descriptor numbers 67
  - Find window 180
  - fork limitation 68
  - function
    - context sensitive subcommands 94
- function (*continued*)
  - Parallel Utility Function 36
- functional decomposition 51, 58
- G**
  - Global data window 85
  - group
    - adding group in pedb 141
- H**
  - help
    - for dbx subcommands 120
    - for pdbx subcommands 120
  - HFS, shared 71
  - host list file
    - comment in pin-list 18
    - creating 23
    - description 4
    - examples 5, 22
    - pin entry 4
    - pool entry 4
- I**
  - I/O, MPI
    - shared HFS 71
  - I/O, standard 69
  - individual nodes, copying files 16
  - initialization, how implemented 61
  - interrupt mode control 43
  - IP address 100
- J**
  - job-step, progression and termination 65
  - job steps, reading from standard
    - input 28
- K**
  - killing a POE job 41
- L**
  - libraries, dynamic 16
  - light, displaying details 48
  - log file 12
  - logging errors to a file 74
  - loops, unrolling 52
    - example 52
- M**
  - manpages
    - pdbx subcommands 94
  - mcp 185
  - mcp\_gath 187
  - mcp\_scat 191
  - memory, shared 71
  - message
    - blocking receive 116
    - FOMO0249 72
  - message catalog considerations 71
  - message catalogs
    - errors 73
    - Japanese version 73
    - national language support 26
    - NLSPATH xiv, 73
  - Message Group Information
    - window 177
  - message passing
    - communication model 51
    - multiple windows in pedb 182
  - Message Passing Interface
    - linking to MPI 15
    - parallelizing the application 1
    - placing calls to MPI 1
  - Message Passing Interface Forum 307
  - message passing routine 181
  - message queue debugger 169
    - starting the message queue
      - debugger 170
    - using the message queue
      - debugger 169
  - messages
    - errors, attaching the debugger 88
    - finding 73
    - format 74
    - identifiers 74
    - labeling message output 37
    - level reported 11
    - PE message catalog components 74
    - PE message catalog errors 73
    - problem determination 73
  - migration xv, 299
  - modes
    - attach mode of pedb debugger 130
    - function modes of pedb 128
    - normal mode of pedb debugger 128
  - MP\_HOSTFILE 14
  - MP\_INFOLEVEL 14
  - MP\_LABELIO 14
  - MP\_PMDLOG 14
  - MP\_PROCS 13
  - MP\_STDOUTMODE 14
  - mpc\_disableintr() 43
  - mpc\_enableintr() 44
  - mpc\_marker 90
  - mpc\_queryintr() 43
  - mpc\_queryintrdelay() 43
  - mpc\_setintrdelay(int val) 43
  - mpcc 11, 194
  - mpCC
    - cpp flag 15
  - mpCC 194
  - MPI-2 C++ bindings
    - enabling 15
  - MPI\_Comm\_rank 53
  - MPI\_Comm\_size 53
  - MPI\_COMM\_WORLD 53, 60
  - MPI\_Finalize 53
  - MPI function calls
    - C example 10
  - MPI-I/O
    - shared HFS 71
  - MPI\_Init 53, 70
  - MPI\_PROD 61
  - MPI programs, safety 87
  - MPI\_Reduce 61
  - MPI\_Scan 61

MPI\_SUM 61  
 MPMD (Multiple Program Multiple Data) 1  
 MPMD model (Multiple Program Multiple Data) 96  
 MPMD program, invoking 26  
 Multi-release Compatibility xv, 299  
   back-level release 300  
   partition release 299

## N

national language support xiv, 3  
 network tuning 68  
 newjob mode 29  
 NLSPATH  
   message catalogs xiv  
 node allocation  
   automatic 21  
   specific 18, 20  
   specific and automatic 18  
 nodes, loading  
   from standard input 26  
   POE commands file 27  
 nonblocking operation 42  
 Notices 303

## O

operation, nonblocking 42  
 operations, arithmetic 122  
 operators  
   arithmetic, in pedb 147  
   bitwise 123  
   bitwise, in pedb 147  
   data access and size 123  
   data access and size, in pedb 148  
   miscellaneous 123  
   miscellaneous, in pedb 148  
   relational and logical 123  
   relational and logical, in pedb 147  
 options  
   -hostfile 14  
   -infilevel 11  
   -infilevel, -ilevel 14  
   -labelio 8, 14  
   -pmdlog 14  
   -procs 13  
   -procs 4 8  
   -stdoutmode 9, 14  
   most likely needed 13  
 output  
   displaying task output 48  
   labeling message output 37  
   no or bad output 77  
   no output at all 78  
   output problems attaching the debugger 88  
 output mode  
   ordered 36  
   single 36  
   unordered 35

## P

parallel C and C++ programs  
   compiling with mpcc and mpCC 15

parallel file copy utilities 44  
 parallel job  
   problems starting a 75  
   termination 67  
 Parallel Operating Environment  
   -hostfile option 14  
   -infilevel option 11  
   -infilevel option, -ilevel option 14  
   -labelio option 14  
   -pmdlog option 14  
   -procs option 13  
   -stdoutmode option 14  
   compiling parallel C and C++ programs 15  
   description 7  
   options, most likely needed 13  
   parallel application, development and execution 7  
   running, examples 7  
 Parallel Operating Environment (POE)  
   executing parallel programs 15  
 Parallel Operating Environment commands  
   mpcc 15  
 parallel programs  
   controlling program execution 30  
   debugging with pdbx 93  
   debugging with pedb 127  
   executing 15  
   monitoring execution using the Program Marker Array 45  
 parallel programs, creating techniques 51  
 parallel task, definition 93  
 Parallel Utility function  
   mcp\_marker 46  
   mcp\_nlights 46  
 Parallel Utility Function 36  
 parallelizing the application 1  
 partition  
   defining size 22  
   definition 93  
   establishing 26  
   loading partition within pdbx 101  
   loading the partition within pedb 135  
 Partition Manager 1, 25, 101  
   linking to Partition Manager 15  
 Partition Manager Daemon  
   problems connecting remote host 76  
 partition release 299  
 pdbx  
   relationship between home node and remote tasks 106  
   subset of commands 109  
 pdbx debugger 93  
   accessing help for dbx subcommands 120  
   accessing help for pdbx subcommands 120  
   attach mode 99  
   attach screen 99  
   command context 93  
   command-line arguments, considerations 101  
   controlling program execution 110

  pdbx debugger 93 (*continued*)  
     creating, removing, and listing aliases 121  
     deleting breakpoints 114  
     deleting events 114  
     deleting tracepoints 114  
     displaying source 119  
     displaying task states 102  
     displaying tasks 102  
     ending a debug session 124  
     events, checking status 115  
     exiting pdbx 125  
     hooking tasks 116  
     interrupting tasks 112  
     loading the partition 101  
     normal mode 96  
     overloaded symbols 124  
     reading subcommands from a command file 122  
     setting breakpoints 111  
     setting command context 106  
     setting tracepoints 112  
     specifying expressions 122  
     specifying variables on trace and stop subcommands 113  
     starting 96  
     unhooking tasks 116  
     variable 'task list' 104  
     viewing program call stacks 117  
     viewing program variables 117  
     viewing type of program variables 118  
 pdbx flags 93  
 pdbx prompt 110  
 pdbx subcommands 93, 94, 201  
   alias 121, 202  
   assign 204  
   attach 205  
   back 107, 206  
   case 207  
   catch 208  
   condition 209  
   cont 210  
   context insensitive subcommands 94  
   dbx 211  
   delete 114, 212  
   detach 124, 213  
   dhelp 120, 214  
   display memory 215  
   down 216  
   dump 217  
   file 218  
   func 219  
   goto 220  
   gotoi 221  
   group 103, 222  
   halt 224  
   help 225  
   help for subcommands 120  
   hook 116, 124, 226  
   ignore 227  
   list 119, 228  
   listi 230  
   load 101, 231  
   map 232  
   mutex 233  
   next 234

- pedb subcommands 93, 94, 201
  - (continued)
  - nexti 235
  - on 106, 107, 236
  - online information, command man 94
  - overview 93
  - print 117, 124, 238
  - quick reference listing 94
  - quit 124, 239
  - registers 240
  - return 241
  - search 242
  - set 243
  - sh 244
  - skip 245
  - source 246
  - status 115, 247
  - step 248
  - stepi 249
  - stop 111, 113, 250
  - task long 102
  - tasks 102, 252
  - thread 253
  - trace 112, 113, 124, 254
  - unalias 121, 256
  - unhook 116, 257
  - unset 258
  - up 259
  - use 260
  - whatis 118, 124, 261
  - where 117, 262
  - whereis 263
  - which 264
- PE administration 3
  - access 4
  - inetd.conf 3
  - partition daemon 3
- PE commands 185
  - mcp 185
  - mcpgath 187
  - mcpscat 191
  - mpcc 194
  - mpCC 194
  - pedb 196
  - pedb 264
  - pmarray 267
  - poe 269
  - poekill 280
- pedb.ad file
  - setting up 128, 183
- pedb debugger
  - add group window 141
  - Attach window 132
  - Attach window, buttons 132
  - break/trace area 139
  - breakpoints, setting 146
  - buffer 139
  - changing a variable's format 166
  - changing a variable's value 166
  - command-line arguments 133
  - compiling options 133
  - context, setting 139
  - control buttons 144
  - controlling program execution 144
  - controlling source code 178
  - customizing pedb resources 183
- pedb debugger (continued)
  - debugging multiple views 182
  - debugging programs using multiple views 182
  - displaying local variables within the program stack 157
  - displaying variable in more or less detail 164
  - editing current source file 179
  - examining program data 156
  - executables, SPMD and MPMD 137
  - execution 144
  - execution, stepping 150
  - execution controls 139
  - getting help 183
  - global data area 139
  - hiding a task's break/trace information 156
  - hiding a task's data information 156
  - hiding a task's stack information 156
  - identifying breakpoint task 149
  - increasing storage 128
  - interrupt waiting process 151
  - leaving pedb 183
  - load executables window 136
  - loading phase 135
  - loading the partition 135
  - local data area 139
  - locating breakpoint in source 156
  - Main window 134
  - main window after partition is loaded 138
  - mode, attach 130
  - mode, normal 128
  - multiple windows and views 182
  - option flags 130
  - pedb.ad file 128, 183
  - pedb main window
    - buttons 144
    - process identifier (PID) 132
    - program execution, tracing 153
    - program path 136
    - removing breakpoints 149
    - repeat execution 151
    - setting up environment 127
    - single step execution 150
    - specifying the array subrange 168
    - stack area 139
    - starting pedb 128
    - status codes 143
    - stop execution 151
    - storage requirements 128
    - task area 139
    - task groups, creating 140
    - task groups, deleting 140, 142
    - task number 132
    - thread specific breakpoints and tracepoints 149
    - threads 145
    - threads area 139
    - trace record 153
    - understanding data types 157
    - unhooking tasks 155
    - using pedb 127
    - viewing the contents of an array 167
  - X-server 127
- pedb main window 83
- pedb subcommands
  - workstation-name 128
  - xhost 127
    - workstation-name 128
    - z/OS-machine name 128
  - z/OS-machine name 128
- performance improvements, delay parameter 42
- pin entry 4, 18
- pin-list 18
- pin list, examples 22
- piping
  - example 69
- play control button, customizing 152
- pmarray 90, 267
- pmd daemon 63
- poe 269
- POE
  - hostfile option 14
  - infolevel option 11
  - infolevel option, -ilevel option 14
  - labelio option 14
  - pmdlog option 14
  - procs option 13
  - stdoutmode option 14
  - application partition exit 125
  - commands file, for loading nodes individually 27
  - commands file, reading job steps from 29
  - compiling parallel C and C++ programs 15
  - controlling program execution using POE 30
  - description of POE 7
  - executing non-parallel programs using POE 30
  - invoking executables in 25, 30
  - options, most likely needed 13
  - parallel application, development and execution 7
  - running, examples 7
  - setting up the execution environment 17
- POE command-line flags 283
  - a 291
  - buffer\_mem 272, 289
  - cmdfile 27, 29, 272, 285
  - css\_interrupt 272, 289
  - d 291
  - eager\_limit 272, 289
  - euiddevelop 31, 273, 279, 290, 291
  - euilibpath 284
  - hfile 284, 291
  - hostfile 284, 291
  - I (upper case i) 291
  - ilevel 272, 277, 287, 291
  - infolevel 272, 277, 287, 291
  - infolevel, -ilevel 38
  - intrdelay 272, 289
  - ionodefile 273, 289
  - labelio 37, 272, 286, 291
  - max\_typedepth 273, 289
  - newjob 28, 272, 285
  - pgmmodel 25, 272, 285
  - pmdlog 38, 272, 277, 291
  - pmlights 47, 268, 273, 290, 291

POE command-line flags 283  
*(continued)*

- polling\_interval 273, 289
- procs 22, 284, 291
- promptpw 273
- pulse 31, 41, 284
- resd 284, 291
- rmpool 284, 291
- savehostfile 24, 284, 291
- shared\_memory 273, 289
- single\_thread 273, 289
- stdinmode 33, 272, 286, 291
- stdoutmode 35, 36, 272, 286, 291
- thread\_stacksize 273, 290
- tmpdir 273, 290, 291
- use\_flow\_control 273, 290
- usrport 273, 290, 291
- wait\_mode 273, 290
- wlm\_enclave 24, 284, 291

generating diagnostic logs 38  
labeling task output 37  
maintaining partition for multiple job steps 28  
managing standard input 33  
managing standard output 35  
setting the message reporting level 37  
specifying a commands file 27, 29  
specifying additional error checking 31  
specifying programming model 25

POE commandline flags

- hfile 19
- hostfile 19, 20, 21
- resd 22
- rmpool 21

setting number of task processes 22

POE considerations

- environment overview 63
- exit status 64
- exits, parallel task 67
- file descriptor numbers 67
- fork limitations 68
- job-step function 65
- message catalog considerations 71
- network tuning, considerations 68
- other thread considerations 70

POE additions 65  
reserved environment variables 71  
root limitation 67  
shared HFS 71  
shell scripts 68  
standard I/O 69  
standard I/O, piping 69  
stdin, stdout or stderr, rewinding 68  
task initialization 67  
termination of a parallel job 67  
thread termination 70  
user authentication 63  
user program, passing string arguments 68

POE environment variables 283

- generating diagnostic logs 38
- ignoring argument list 32
- labeling task output 37
- maintaining partition for multiple job steps 28

POE environment variables 283  
*(continued)*

- making POE ignore arguments 31
- managing standard input 33
- managing standard output 35
- MP\_BUFFER\_MEM 277, 289
- MP\_CMDFILE 27, 29, 276, 285
- MP\_CSS\_INTERRUPT 41, 277, 289
- MP\_CSS\_INTERRUPT, overriding the setting of 44
- MP\_DBXPROMPTMOD 199
- MP\_DEBUG\_INITIAL\_STOP 124, 200, 266, 287
- MP\_DEBUG\_LOG 266, 287
- MP\_EAGER\_LIMIT 277, 289
- MP\_EUIDEVELOP 30, 31, 279, 290
- MP\_EUILIBPATH 274, 284
- MP\_FENCE 30, 279, 290
- MP\_HOLD\_STDIN 30, 33, 276, 286
- MP\_HOSTFILE 274, 284
- MP\_HOSTLIST 19, 20, 21
- MP\_INFOLEVEL 31, 38, 277, 287
- MP\_INTRDELAY 43, 278, 289
- MP\_IONODEFILE 278, 289
- MP\_LABELIO 31, 37, 276, 286
- MP\_MAX\_TPEDEPTH 278, 289
- MP\_NEWJOB 28, 276, 285
- MP\_NOARGLIST 30, 32, 279, 290
- MP\_PGMODEL 25, 276, 285
- MP\_PMDLOG 31, 38, 277, 287
- MP\_PMSUFFIX 277, 288
- MP\_PMLIGHTS 47, 268, 279, 290
- MP\_POLLING\_INTERVAL 278, 289
- MP\_PROCS 22, 275, 284
- MP\_PULSE 275, 284
- MP\_REMOTEDIR 31, 275, 284
- MP\_RESD 22, 275, 284
- MP\_RMPOOL 18, 21, 275, 284
- MP\_SAVEHOSTFILE 24, 275, 284
- MP\_SHARED\_MEMORY 278, 289
- MP\_SINGLE\_THREAD 278, 289
- MP\_SNDBUF 69
- MP\_STDINMODE 30, 33, 276, 286
- MP\_STDINMODE, scenarios 35
- MP\_STDOUTMODE 30, 35, 36, 276, 286
- MP\_THREAD\_STACKSIZE 67, 278, 290
- MP\_TIMEOUT 275, 284
- MP\_TMPDIR 266, 279, 290
- MP\_USE\_FLOW\_CONTROL 278, 290
- MP\_USRPORT 48, 268, 279, 290
- MP\_WAIT\_MODE 279, 290
- MP\_WLM\_ENCLAVE 24, 275, 284

sending STDIN from home to remote node 33  
setting number of task processes 22  
setting the delay parameter 43  
setting the message reporting level 37  
specifying a commands file 27, 29  
specifying additional error checking 31  
specifying programming model 25

POE options

- most likely needed 13

poekill 280

Point to Point Message Details

- window 175

pool entry 4, 19

pool-list 17

pool list, examples 22

problems

- attaching the debugger 87
- attaching the debugger, bad output 88
- debugging a parallel program 77
- program hangs 78

problems, common

- bad output, attaching the debugger 88
- can't compile a parallel program 74
- can't connect with the remote host 75
- can't execute a parallel program 76
- can't start a parallel job 75
- no output 78
- no output or bad output 77

process 30

process id, getting with ps command 81

process id number, pid 12

process pinning 17

- pin and pool list, summary 21
- pin entry 4, 18
- pin-list 18
- pool entry 4, 19
- pool-list 17
- Round-Robin mode 17, 19, 21
- WLM mode 17, 21

processor nodes, definition 93

program

- termination 70

Program Marker Array 45

- displaying details of light on 48
- displaying task output on 48
- mpc\_marker 89

Parallel Utility Functions for PMA 46

- pmarray, debugging 89
- setting the number of lights on 47
- starting 47
- using 89
- window 90

programs

- loading as job steps 28
- MPMD 26
- non-parallel, invoking 30
- problems starting 76
- SPMD 25

PTF for Parallel Environment 299

publications

- Message Passing Interface Forum 307
- non IBM 307
- related 307
- z/OS UNIX System Services 307
- z/OS UNIX System Services Parallel Environment 307

**R**

releasing nodes

- used short keys 29

remote nodes 1, 179

- remote nodes 1, 179 *(continued)*
  - changing directories 39
  - detecting failures 41
  - problems with connecting 75
- Resource Manager 101
- return codes 78
- root limitation 67
- Round-Robin mode (RR) 17, 19, 21
- running POE 7

## S

- safe coding practices 293, 294
  - fairness 294
  - order 293
  - resource limitations 295
  - safe program, described 293
- safety
  - MPI programs 87
- scheduling environment 5
- security environment 6
- Select Filters window 172
- Send/Receive Message Details
  - window 175
- serial program 1
- setting up environment with pedb 127
- shared memory 71
- shell execution 68
- shell script 2
- short keys
  - 3270 29
  - rlogin 29
- sigaction()
  - sa\_handler 65
  - sa\_sigaction 65
- signal
  - SIGTERM 78
- signal handlers 65
- signals 66
- sine series algorithm 58
- source, displaying 119
- source code
  - control 178
  - creating 1
  - emphasis 181
- specific node allocation 18, 20
- specific node allocation, example 22
- SPMD (Single Program Multiple Data) 1
- SPMD model (Single Program Multiple Data) 96
- SPMD program, invoking 25
- standard error (STDERR) 32
- standard input (STDIN) 32
- standard output (STDOUT) 32
- start-up problems 76
- static executable, creating 15
- STDIN 29
- stdin, stdout, stderr 68
- stdin and stdout, piping example 69
- stepping execution 150
- stopping a POE job 41
- subcommands 201
  - context insensitive 94
  - context sensitive 94
  - dbx 120, 214
  - pdbx 120, 201
  - reading from a file 122
- subroutine 144

- subset, pdbx commands 109
- symbols, overloaded 124
- synchronization in message passing
  - model 51
- system call
  - sigaction() 65

## T

- task list
  - variable in pdbx debugger 104
- Task Message Queue window 173
- task number, taskid 12
- task states
  - pdbx debugger 106
- tasks
  - changing the name of a task
    - group 105
  - deleting from a task group 104
  - displaying 102
  - displaying output 48
  - getting additional information 84
  - grouping 103
  - hooking and unhooking 116
  - interrupting with pdbx debugger 112
  - status information in pedb 143
  - task area in pedb 139
  - task groups, blocking send 140
  - task groups, creating 140
  - task groups, deleting 142
  - task groups, deleting in pedb 140
- thread
  - other considerations 70
  - termination 70
- threaded programs
  - debugging 88
- threads
  - conditional breakpoint and
    - tracepoint 149
  - displayed thread 146
  - interrupted thread 146
  - multiple threads in pedb 145
  - threads area in pedb 139
  - trace record 153
- threads and debugging 89
- Threads Viewer window 162
- tracepoints 111, 112
  - conditional tracepoint window 149
  - identifying tasks 149
  - setting conditional tracepoint 154
  - specifying conditions 154
  - thread specific conditions 149
  - trace record 153
- tracing
  - program execution 153
- trademarks 304

## U

- UDP Protocol 1
- unrolling loops 52
  - example 52
- user authentication 63
- User Datagram Protocol (UDP) 1
- user exits, CEEBXITA and CEEBINT 65

## V

- variable 2
  - global 114
  - local 114
- variables
  - viewing program variables 117

## W

- WLM mode 17, 21
- WLM multi-system enclaves 24
- Workload Manager (WLM)
  - multi-system enclaves 5

## X

- X-windows environment, setting up 46

---

## Readers' Comments — We'd Like to Hear from You

z/OS  
UNIX System Services  
Parallel Environment:  
Operation and Use

Publication No. SA22-7810-00

Overall, how satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall satisfaction	<input type="checkbox"/>				

How satisfied are you that the information in this book is:

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Accurate	<input type="checkbox"/>				
Complete	<input type="checkbox"/>				
Easy to find	<input type="checkbox"/>				
Easy to understand	<input type="checkbox"/>				
Well organized	<input type="checkbox"/>				
Applicable to your tasks	<input type="checkbox"/>				

Please tell us how we can improve this book:

Thank you for your responses. May we contact you?  Yes  No

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

\_\_\_\_\_  
Name

\_\_\_\_\_  
Address

\_\_\_\_\_  
Company or Organization

\_\_\_\_\_  
Phone No.



Fold and Tape

Please do not staple

Fold and Tape



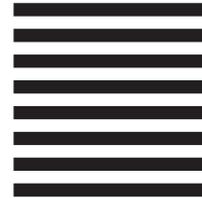
NO POSTAGE  
NECESSARY  
IF MAILED IN THE  
UNITED STATES

# BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Deutschland Entwicklung GmbH  
Information Development  
Department 3248  
Schönaicher Str. 220  
D-71032 Böblingen  
Germany



Fold and Tape

Please do not staple

Fold and Tape





Program Number: 5694-A01



Printed in the United States of America  
on recycled paper containing 10%  
recovered post-consumer fiber.

SA22-7810-00



Spine information:



z/OS

z/OS UNIX System Services PE: Operation and  
Use

SA22-7810-00