IBM PVMe for AIX
User's Guide and Subroutine Reference
Version 2, Release 1

Document Number GC23-3884-00

Note

Before using this information and the product it supports, be sure to read the general information in “Notices” on page viii.

| **First Edition, October 1995**

| This edition applies to Version 2, Release 1, Modification 0 of the IBM PVMe for AIX program product, and to all subsequent
| releases and modifications until otherwise indicated in new editions or technical newsletters.

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

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

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

FAX: (International Access Code)+1+914+432-9405

IBMLink (United States customers only): D58PUBS at KGNVMC
IBM Mail Exchange: USIB2NZL at IBMMAIL
Internet: d58pubs@vnet.ibm.com

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

Copyright International Business Machines Corporation 1993, 1994, 1995. All rights reserved. Note to U.S. Government Users — Documentation related to restricted rights — Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp..

Contents

Notices . . . viii
Trademarks and Service Marks . . . viii

About This Book . . . ix
Who Should Use This Book . . . ix
How to Use This Book . . . ix
Overview of Contents . . . ix
Highlighting . . . x
Related Publications . . . xi
IBM Publications . . . xi
Non-IBM Publications . . . xi
Ordering Additional Copies of This Book . . . xi

Chapter 1. Getting Started with IBM PVMe for AIX . . . 1
What is IBM PVMe for AIX? . . . 1
Differences Between PVM and IBM PVMe for AIX . . . 2
Parallel Programming Concepts . . . 4
What is Parallel Processing? . . . 4
Cooperation . . . 5
Synchronization . . . 5
Parallel Programming Models . . . 5
Understanding Latency and Bandwidth . . . 6

Chapter 2. Writing Parallel Programs . . . 7
Parallel Programming Overview . . . 7
Determining Computational Independence . . . 7
Decomposing a Serial Program . . . 8
Design and Performance Considerations . . . 10
Granularity and Load Balancing . . . 13
Tuning a Parallel Application . . . 14

Chapter 3. Product Installation . . . 17
Before You Begin . . . 17
Requirements . . . 17
Software Prerequisites . . . 17
Installing the IBM PVMe for AIX Software . . . 17
Installing IBM PVMe for AIX on a non-SP system . . . 17
Installing IBM PVMe for AIX on an SP system . . . 18
Customizing Your Installation . . . 19
Customizing the debugger and debugger2 Shell Scripts . . . 19

Chapter 4. Product Startup . . . 21
Before You Start . . . 21
Running an IBM PVMe for AIX Session . . . 21
Node Allocation . . . 25
Specifying Allocation Options . . . 25
Running Multiple Sessions . . . 26
Starting IBM PVMe for AIX from the Control Workstation . . . 27
Starting IBM PVMe for AIX from a Compute Node . . . 27
Including External Workstations in the IBM PVMe for AIX Machine . . . 27

© Copyright IBM Corp. 1993, 1994, 1995

Using the IBM PVMe for AIX Console 29
Running an IBM PVMe for AIX Batch Session 30
IBM PVMe for AIX Daemon Options 31
Syntax line for IBM PVMe for AIX daemon 32
Syntax line for IBM PVMe for AIX console 32
Using the -exec Option 32
Using the -ip Option 32
Using the -nokill Option 33
Using the -p Option 33
Using the -resd Option 33
Using the -r Option 34
Using the -R Option 34
Using the -share Option 34
Using the -trace Option 35
Syntax of the Hostlist File 35
Resolving Startup Problems 36
Verifying the IBM PVMe for AIX Program Product 37

Chapter 5. Using the IBM PVMe for AIX Program Product 41
Understanding IBM PVMe for AIX Features 41
Managing Tasks 42
Primitives for Managing Tasks 42
Examples for Managing Tasks 43
Enrolling a Process 45
Unenrolling a Process 46
Starting a Process 46
Searching for Executable Files 47
Terminating a Process 47
Managing Options 48
Querying for Information about Tasks 48
Managing I/O 48
Managing Hosts 49
Primitives for Managing Hosts 50
Starting and Stopping the Virtual Machine 50
Querying Information about the Virtual Machine 50
Examples for Managing Hosts 51
Understanding Data Communication 54
Primitives for Data Communication 54
Preparing a Message 56
Sending a Message 57
Receiving a Message 57
Checking the Arrival of a Message 59
Managing Multiple Buffers 59
Examples for Data Communication 60
Error Handling 62
Primitives for Error Handling 63
Automatic Printing of Error Messages 63
Examples for Managing Error Conditions 63
Signaling 65
Primitives for Signaling 65
Examples of Signaling 66
Group Management 67
Managing Task Groups with Primitives 68
Joining and Leaving Groups 68

Ensuring Reliable Group Operations . . . 69
Understanding the Broadcast Operation . . . 70
Understanding the Barrier Routine . . . 71
Understanding PVM_EXPORT . . . 71
Performing Global Operations . . . 71
Querying Information about Groups . . . 72
Examples for Managing Groups . . . 73
A Complete Example (FORTRAN) . . . 75

Chapter 6. Using IBM PVMe for AIX . . . 87
Compiling and Linking a Parallel Program on a POWERparallel System . . . 87
Compiling and Linking a Parallel Program on an External Workstation . . . 88
Running a Parallel Program . . . 89
Debugging a Parallel Program . . . 89
Finding Errors in Program Logic . . . 90

Chapter 7. IBM PVMe for AIX Reference: C and FORTRAN Subroutines . 93
Using the FORTRAN Interface . . . 93
Understanding Subroutine Format . . . 93
Syntax Diagrams . . . 93
pvm_addhosts *** CURRENTLY DISABLED *** . . . 95
pvm_archcode . . . 97
pvm_barrier . . . 99
pvm_bcast . . . 101
pvme_bcast . . . 103
pvm_bufinfo . . . 105
pvm_catchout . . . 107
pvm_config . . . 109
pvm_delhosts *** CURRENTLY DISABLED *** . . . 112
pvm_exit . . . 114
pvm_freebuf . . . 115
pvm_gather . . . 117
pvm_getinst . . . 120
pvm_getopt . . . 122
pvm_getrbuf . . . 124
pvm_getsbuf . . . 126
pvm_gsize . . . 128
pvm_gettid . . . 129
pvm_gettmask *** CURRENTLY DISABLED *** . . . 131
pvm_halt . . . 133
pvm_hostsync . . . 134
pvm_initsend . . . 136
pvm_joingroup . . . 138
pvm_kill . . . 140
pvm_lvgroup . . . 142
pvm_mcast . . . 144
pvm_mkbuf . . . 146
pvm_mstat . . . 148
pvm_mytid . . . 150
pvm_notify . . . 152
pvm_nrecv . . . 154
pvm_packf . . . 156
pvm_parent . . . 158
pvm_perror . . . 160

pvm_pkint, pvm_pkcplx, pvm_pkdcplx, pvm_pkfloat, ... 162
pvm_probe 165
pvm_precv 167
pvm_psend 170
pvm_pstat 172
pvm_recv 174
pvm_reduce 176
pvm_scatter 179
pvm_send 182
pvm_sendsig 184
pvm_serror 186
pvm_setopt 188
pvm_setrbuf 190
pvm_setsbuf 192
pvm_settmask *** CURRENTLY DISABLED *** 194
pvm_spawn 196
pvm_start_pvmd 200
pvm_tasks 202
pvm_tidtohost 205
pvm_trecv 206
pvm_unpackf 208
pvm_upkint, pvm_upkcplx, pvm_upkdcplx, pvm_upkfloat, ... 210

Chapter 8. Diagnosing Problems 213
Informational Messages 213
Error Messages 213

Appendix A. IBM PVMe for AIX Error Code Summary 223

Appendix B. IBM PVMe for AIX Environment Variables 225
IBM PVMe for AIX Environment Variables 225
PVMDPATH 225
PVMEPATH 226
PVMHFN 226
PVMe_CLH 226
PVMe_DATA_INTERRUPT 227
PVMe_MEMCPY 227
PVMe_BARRTYPE 227
N64KBUFF 228
SP_NAME 228
N256BUFF 229
PVM_EXPORT 229
PVMe_IP 229
PVMe_TRACEFILE 230
PVM_DEBUGGER 230
PVMe_IP_PORT 230

Appendix C. IBM PVMe for AIX Intrinsic Limits: Summary 233

Appendix D. IBM PVMe for AIX Quick Reference 235

Appendix E. IBM PVMe for AIX Console Commands 241

Appendix F. IBM PVMe for AIX Daemon Options 243

Appendix G. IBM PVMe for AIX Hostlist File Options 247

Appendix H. Enabling AFS 249

Glossary of Terms and Abbreviations 251

Index 255

Notices

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

IBM Corporation
IBM Director of Licensing
208 Harbor Drive
Stamford, Connecticut
United States 06904

Trademarks and Service Marks

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

AIX	LoadLeveler
AIXwindows	RISC System/6000
ES/9000	Scalable POWERparallel Systems
IBM	SP
IBM PVMe for AIX	

The following terms are trademarks of other companies as follows:

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

PVM (Parallel Virtual Machine) is a freeware package, available from Oak Ridge National Laboratory.

About This Book

This book describes the features and programming tasks for the IBM AIX PVMe program product. It also provides some general information for creating and porting applications.

Who Should Use This Book

This book is intended for experienced programmers who wish to write parallel applications with the IBM PVMe for AIX program product. Since the IBM PVMe for AIX program product is intended for applications that will run an IBM RISC System/6000 Scalable POWERparallel System (SP) system with the High Performance Switch feature, readers should be familiar with the SP. Readers should also be skilled in either FORTRAN or C programming and should be familiar with IBM AIX and UNIX commands. A basic understanding of distributed memory systems is also helpful.

While this book provides some general information for creating and porting parallel applications, it is not intended as a general guideline for parallel programming. If you require information that is outside the scope of this book, refer to “Related Publications” on page xi.

How to Use This Book

Overview of Contents

This book is divided into the following chapters:

- Chapter 1, “Getting Started with IBM PVMe for AIX” on page 1, provides an overview of parallel programming and describes the IBM PVMe for AIX program product.
- Chapter 2, “Writing Parallel Programs” on page 7, outlines a general approach to parallel programming.
- Chapter 3, “Product Installation” on page 17, describes the procedure for installing the IBM PVMe for AIX software.
- Chapter 4, “Product Startup” on page 21, contains information on configuring the IBM PVMe for AIX program at the system and user level.
- Chapter 5, “Using the IBM PVMe for AIX Program Product” on page 41, provides an overview of the IBM PVMe for AIX routines that are callable from your applications.
- Chapter 6, “Using IBM PVMe for AIX” on page 87, tells you how to compile and link parallel programs in a variety of environments. Debugging information is also included.
- Chapter 7, “IBM PVMe for AIX Reference: C and FORTRAN Subroutines” on page 93, provides information about each IBM PVMe for AIX routine, including a description, syntax rules, parameters, examples, and error conditions. The routines are arranged alphabetically throughout the chapter.

© Copyright IBM Corp. 1993, 1994, 1995

- Chapter 8, “Diagnosing Problems” on page 213 provides information on system messages that you may encounter during the operation of PVMe.
- Appendix A, “IBM PVMe for AIX Error Code Summary” on page 223, explains the IBM PVMe for AIX error codes.
- Appendix B, “IBM PVMe for AIX Environment Variables” on page 225, provides information about the IBM PVMe for AIX environment variables that may be useful.
- Appendix C, “IBM PVMe for AIX Intrinsic Limits: Summary” on page 233, discusses some limitations of the IBM PVMe for AIX program product.
- Appendix D, “IBM PVMe for AIX Quick Reference” on page 235, provides quick-access information for entering arguments for C and FORTRAN subroutines.
- Appendix E, “IBM PVMe for AIX Console Commands” on page 241, provides a table of the IBM PVMe for AIX console commands for quick reference.
- Appendix F, “IBM PVMe for AIX Daemon Options” on page 243, summarizes the IBM PVMe for AIX daemon options, in a table.
- Appendix G, “IBM PVMe for AIX Hostlist File Options” on page 247, describes the hostlist file options.
- Appendix H, “Enabling AFS” on page 249, describes how to run in an AFS environment.
- “Glossary of Terms and Abbreviations” on page 251 provides a list of terms used in this book.

Highlighting

This book uses the following highlighting conventions:

Convention	Usage
Bold	**Bold** words or characters represent system elements that you must use literally, such as commands, subcommands, flags, pathnames, directories, filenames, values, and selected menu options.
Italic	• *Italic* words indicate the first use of a term included in the glossary. • *Italic* words or characters represent variable values that you must supply. • *Italics* are used for book titles. • *Italics* are used for general emphasis.
`Monospace`	All of the following are displayed in `monospace` type: • Displayed information • Message text • Example text • Specified text typed by the user • Field names as displayed on the screen • Prompts from the system • References to example text

Related Publications

IBM Publications

- *IBM PVMe for AIX: Licensed Program Specification*, (GH23-0018-02)

Non-IBM Publications

For more information about parallel computing, you may want to consult some non-IBM publications.

- Almasi, G., Gottlieb, A. *Highly Parallel Computing*. Benjamin-Cummings Publishing Company, Inc., 1989.
- Hwang, K., Briggs, F.A. *Computer Architecture and Parallel Processing* McGraw-Hill, 1984.
- Perrot, R.H. *Parallel Programming* Addison-Wesley, 1987.
- Flynn, M.J. *Very High-Speed Computing Systems* Proc. IEEE, Vol. 29, 1966.
- Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., Sunderam, V. *PVM 3 User's Guide and Reference Manual*, Oak Ridge National Laboratory, 1993.

  This publication postscript file can be accessed and printed through one of the following Home Pages located on your Web browser:

  - **http://www.netlib.org/pvm3/ug.ps**
  - **ftp://netlib2.cs.utk.edu/pvm3/ug.ps**

Ordering Additional Copies of This Book

To order additional copies of this book, request order number SH23-0019-02.

Chapter 1. Getting Started with IBM PVMe for AIX

This chapter provides an overview of the IBM AIX PVMe program product. Included in this overview is a discussion of the differences between the freeware PVM package and the IBM PVMe for AIX program product. While this chapter also discusses some basic parallel programming concepts, it is not intended to serve as a general guide.

What is IBM PVMe for AIX?

The IBM PVMe for AIX program product is an implementation of PVM, a freeware API (application program interface) package. PVM, which stands for *Parallel Virtual Machine*, was originated by the Oak Ridge National Laboratory. It also now involves research at Emory University, University of Tennessee, Carnegie Mellon University and Pittsburgh SuperComputer center under DOE (Department of Energy), NSF (National Science Foundation) and the state of Tennessee research grants. It is designed to allow usage of a heterogeneous network of parallel and serial computers to act as a single computational resource. The IBM PVMe for AIX program product, on the other hand, is designed for use on a IBM RISC System/6000 Scalable POWERparallel Systems machine, with the High Performance Switch (HPS) feature. You can also use a RISC System/6000 workstation to initiate your job to run on a SP. Also, IBM PVMe for AIX supports clusters of RISC System/6000 workstations, however the SP will provide better performance.

The IBM PVMe for AIX program product consists of:

- A server process, called the IBM PVMe for AIX daemon
- Two libraries (for C, C++, and FORTRAN programs)

The IBM PVMe for AIX program product supports (as in the freeware PVM) FORTRAN, C, and C++ and it is expected to offer to PVM applications a dramatic improvement in terms of latency and bandwidth.

The daemon handles all process control, synchronization, host and group management. The IBM PVMe for AIX libraries provide all routines callable by your programs. These libraries both serve as the interface between your programs and the daemon, which allows your programs to submit process control requests to the daemon, group and host management and synchronization, and run data communication.

A special program (the console) is also available to monitor the behavior of an IBM PVMe for AIX session.

The IBM PVMe for AIX program product supports both Multiple Program, Multiple Data (MPMD) and Single Program, Multiple Data (SPMD) programming models. MPMD and SPMD are explained in “Parallel Programming Concepts” on page 4. New IBM PVMe for AIX tasks may be generated and others may leave at runtime, providing a high degree of flexibility.

This release of the IBM PVMe for AIX program product is object code compatible with version 3.3.7 of freeware PVM. This means that existing PVM applications require only a re-link step with the IBM PVMe for AIX libraries. For more

© Copyright IBM Corp. 1993, 1994, 1995

information, see the section entitled “Differences Between PVM and IBM PVMe for AIX” on page 2.

Differences Between PVM and IBM PVMe for AIX

The IBM PVMe for AIX program product is compatible with PVM, but its internal structure is completely new. Because the IBM PVMe for AIX program product runs on a homogeneous platform (the SP and the RISC System/6000), many operations are simpler. It can also take advantage of all the highly-tuned features of the operating system such as the special primitives for synchronization among tasks.

This section describes the characteristics that differentiate the IBM PVMe for AIX architecture from the freeware PVM. Most of these characteristics are transparent to the application developer but understanding the differences between the products can be helpful.

Protocols Supported

To perform data communication, the IBM PVMe for AIX program product interfaces with the Message Passing Client Interface (MPCI), the communication software that runs on the HPS adapters. The MPCI interfaces with a low level protocol, running in the user space (User Space protocol), which offers a low-latency and high-bandwidth communication path to user applications, running over the POWERparallel system's High Performance Switch. The MPCI also interfaces with the IP protocol. For optimal performance, IBM PVMe for AIX uses the User Space (US) protocol as its default communication path. However, IBM PVMe for AIX now lets you run parallel applications that exploit the IP component of the MPCI as well.

The US Protocol: The MPCI User Space (US) protocol allows user applications to take full advantage of the High Performance Switch, and should be used whenever communication is a critical issue (for instance, when running a parallel application in a production environment). However, this protocol cannot be used by more than one process per node at a given time (this means that only one IBM PVMe for AIX session can run on a set of nodes, and only one component of the parallel program can run on each node).

The IP Protocol: The MPCI IP protocol doesn't perform as well as the MPCI US protocol does, but it does support multiple processes per node. Although you are still restricted to one process per job per node, you can have multiple sessions, each with one process per node. As a result, the IP protocol can be used in development or test environments, where more attention is paid to the correctness of the parallel program than to its speed-up, and therefore, more users can work on the same nodes at a given time. Note that you may also run IBM PVMe for AIX over IP only, without interfacing to MPCI or the SP High Performance Switch.

In both cases data exchange always occurs between processes, without involving the pvmd3e daemon. See the “IBM PVMe for AIX Daemon.”

IBM PVMe for AIX Daemon

The IBM PVMe for AIX daemon (pvmd3e) runs on only one node instead of all the participating nodes, as it does in the freeware PVM. This allows simpler (and in many cases faster) task management.

Single Server Process

A single server process handles both the base functions (data communication and process management) as well as managing dynamic groups. The basic C and FORTRAN library of the IBM PVMe for AIX program product already contains the routines to work with process groups, while in the freeware PVM those routines are contained in an additional library. PVMe in IP mode also uses an “ipproc” daemon on each node.

Enhanced Memory Management

An enhanced memory management scheme of the IBM PVMe for AIX program product also allows better exploitation of the memory resources, which is a critical issue when running very large applications.

IBM PVMe for AIX Routine Not Included in Freeware PVM

IBM PVMe for AIX includes a routine called **pvme_bcast** which broadcasts a message stored in the active send buffer to all the members of a given group. For more information on this routine, refer to “pvme_bcast” on page 103.

PVM Routines Not Included in IBM PVMe for AIX

PVM supports the following routines that are currently not supported by IBM PVMe for AIX:

- pvm_recvf
- pvm_reg_hoster
- pvm_reg_rm
- pvm_reg_tasker
- pvm_setmwid

PVM Routines Included in PVMe but Disabled

The following PVM routines are included but are disabled:

- pvm_addhosts
- pvm_delhosts
- pvm_gettmask
- pvm_settmask

Unsupported Options for PVMe Routines

The following PVMe options are not supported:

- pvm_notify - PvmHostAdd, PvmHostDelete
- pvm_getopt - PvmFragSize
- pvm_setopt - PvmResvTids, PvmOutputTid, PvmOutputCode, PvmTraceTid, PvmTraceCode, PvmSelfOutputTid, PvmSelfOutputCode, PvmSelfTraceTid, and PvmSelfTraceCode
- pvm_spawn - PvmMppFront, PvmHostCompl

Comparing a send Operation

At the user level, the differences in the two packages impact the behavior of a process only when a **send** operation is attempted and the target process is no longer available. This can happen when the target has called the **pvm_exit** function or simply when it has terminated. In this case, an IBM PVMe for AIX process will keep trying to complete the operation, while a process that uses the freeware package PVM and that exploits the communication path passing through the daemons, will receive a **PvmNoComp** error code. On the other hand, if the operation is attempted when the PVM direct task-to-task communication link is exploited the two implementations behave identically. No check is performed on the status of the target process and the sender continues indefinitely in the **pvm_send** operation.

Comparing a Broadcast Operation

With the IBM PVMe for AIX program product, a broadcast operation can result in the same behavioral differences, mentioned above, depending on the chronology of the broadcast and of the **pvm_exit** operations. If the **pvm_exit** routine completes before the broadcast operation starts, the **pvm_bcast** operation takes into account the new number of instances and works as expected. Otherwise, the broadcast operation will never complete since it will keep on trying to send the message to the wrong number of instances. Also, when comparing send with PVM using direct task-to-task communication, the behavior of the IBM PVMe for AIX program product and of the public domain package is identical.

Checking the Status of a Target Process

If you suspect that a target process is no longer available, and wish to avoid possible problems, a sender process should explicitly ask for the availability of the target using the **pvm_pstat** function. In any case, whether a process calls the **pvm_exit** function or whether it is terminated through a call to the **pvm_kill** function, its change of status is immediately registered by the IBM PVMe for AIX daemon. With the IBM PVMe for AIX program product, it is good practice to update the daemon by having all processes (whose termination is not handled by other processes), explicitly call the **pvm_exit** function.

Parallel Programming Concepts

What is Parallel Processing?

Before approaching the task of creating a parallel program, it's important to first understand the difference between how a serial program and a parallel program is processed.

A program that executes in a serial environment runs on one node, a single computational unit, with private and/or shared hardware resources. This node executes the program in a sequential fashion, one logical part of the program at a time.

During parallel processing, two or more nodes each execute logical parts of a program simultaneously. Because of this, parallel processing allows you to create and run applications that can solve large problems quickly. As each part of the program executes, it must *cooperate* with the other parts.

The remainder of this section addresses some general considerations for creating and running applications in a parallel environment.

Cooperation

The term *cooperation* refers to the way the executing tasks of a parallel program work together to solve the problem at hand. Cooperation is related to both the *coupling* scheme of the nodes and the way they coordinate calculations and communication. In some systems, the nodes are *tightly coupled* and in others they are *loosely coupled*. In a tightly coupled system, the delay experienced when a message is sent from one node to another is short, and the data rate is high. In the loosely coupled system the opposite is true. In general, distributed memory systems, like the SP, are loosely coupled.

In addition to the coupling scheme of the nodes, two important components of cooperation are *communication* and *synchronization*.

Communication

A common criteria used to classify parallel processing systems is related to the way nodes *communicate* (share data). In the SP system, nodes communicate by explicitly exchanging messages as the tasks of a parallel program execute. This method is known as *message passing*.

The message passing model is typically used in distributed memory systems, where each node owns *private memory* and is linked by means of an interconnection network. In the SP, the High Performance Switch feature provides the interconnection network needed for high-speed exchange of messages.

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

The message passing model requires that each process operates exclusively in a private environment, which cannot be directly modified by other processes. In this situation, processes must exchange messages in order to interact with one another.

Synchronization

In order for nodes to cooperate effectively, they must be properly *synchronized*. Synchronization refers to the exact sequence in which nodes perform their various tasks, taking into account the logical dependence and the order of precedence among the nodes. The message passing model is somewhat self-synchronizing because sending and receiving messages involves implicit synchronization points. A message cannot be received if it has not already been sent. However, more specific synchronization can be achieved using function calls that create barriers and semaphores for controlling the sequence in which instructions are executed.

Parallel Programming Models

A programming model refers to the way a programmer maps out a particular problem. Each problem must be broken down into a set of tasks that can be solved concurrently. There are two types of parallel programming models that can be used in a distributed memory system, the SPMD (Single Program, Multiple Data), and MPMD (Multiple Program, Multiple Data) models.

SPMD Programming Model

In the SPMD model, a single program runs on more than one node. In other words, all nodes run the same program. In order to improve performance using the SPMD model, iterative tasks, like loop processing, are distributed in a controlled fashion among the different nodes, each executing on its own set of data. In this way, each loop iteration is independent from the results produced by another.

MPMD Programming Model

In the MPMD model, different, but related, programs run on different nodes (each with its own set of data) in order to solve a common problem and achieve a global result.

In order to dissect a problem, you must first understand the programming model you are using. Choosing the right model is crucial in improving your program's performance, that is, a speed-up that approaches its maximum potential. As a first approximation, the best performance, in terms of execution time, can be defined as the amount of time it takes to run the program serially on a single node, divided by the number of nodes running your job in parallel.

In short, a parallel programming model describes the method in which nodes are coordinated in executing their tasks and in sharing data when running a parallel program. This task can be accomplished by using specific hardware, but is commonly a function of the software.

Understanding Latency and Bandwidth

Two important characteristics of the hardware and software of your system that affect performance are *latency* and *bandwidth*.

Latency

Latency is defined as 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 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.

Bandwidth

Bandwidth is defined as the total available data rate of a digital channel. The bandwidth of a system is determined by the speed of its network, but it is also affected by the overhead of the control data added by the communication protocol.

Chapter 2. Writing Parallel Programs

This chapter outlines the tasks involved in translating a given problem into a parallel program. It especially focuses on analyzing and decomposing a serial program. It also provides some general hints and caveats for designing, fine tuning, and improving the performance of a parallel program.

Parallel Programming Overview

What is the advantage of writing parallel programs and running them on parallel architectures? The advantage is that a parallel program can require much less time to run when compared to running the same program serially. By breaking a program into its logical parts and running those parts simultaneously on multiple nodes, the time needed to run the entire program is much smaller. As a result, large, compute-intensive problems are solved quickly.

For the most part, the process of writing a parallel program can be summarized in three steps:

1. Select a parallel programming model.
2. Build the program according to the model chosen.
3. Tune the program for optimum performance.

The first step in creating a parallel program is to select a parallel programming model (see “Parallel Programming Models” on page 5). Once you have done this, the next step is to build your program based on the model you selected. However, you first need to understand the algorithm that you are using as well as the program structure and the data needed at various stages of execution.

Note: A mistake in selecting a model can limit improvement over a serial program. If necessary, restart from step 2 above, after selecting a new model.

There are no standard approaches to writing parallel programs. In most cases, you start with a serial version of the program and *parallelize* it; that is, tailor it for parallel execution. The first step is to determine which parts of your serial program can be run concurrently. To do this, you could profile your program to determine the *computational independence* of its logical parts.

Determining Computational Independence

A *task* is defined as a unit of computation, analogous to a UNIX** process. If one task does not modify data that is referenced by another task, it is said to be *computationally independent*. If computationally independent tasks are run in parallel, the result is the same as if they were run serially, but with reduced running time.

A program can be parallelized easily if its tasks are computationally independent, but it is not a requirement; you can use *barriers* and *messages* to ensure correct results. The idea is to serialize those sequences of instructions that are not computationally independent. Serialization can occur by passing messages between the tasks or the setting of a barrier, so one task waits for a condition to be true before continuing. *Synchronization* refers to the exact order in which the

© Copyright IBM Corp. 1993, 1994, 1995

nodes perform their various tasks. This is important in serializing tasks that are not computationally independent.

If your parallel program does not show some level of speed-up, either you did not use the right programming model or the problem is too sequential to benefit from parallel execution.

Decomposing a Serial Program

Parallel programs are usually created from serial programs that already exist. The first step in translating a serial program into a parallel program is to *decompose* it, by breaking it down into its logical sections. Each logical section is an autonomous part of the program that performs a well-defined task, and when two or more of these sections are computationally independent, they can run concurrently.

The following example demonstrates the concept of decomposition. Figure 1 represents a program that is composed of four logical sections. In a serial version of the program, they run sequentially on one node and the total time spent in execution is the sum of the time spent to run each section separately.

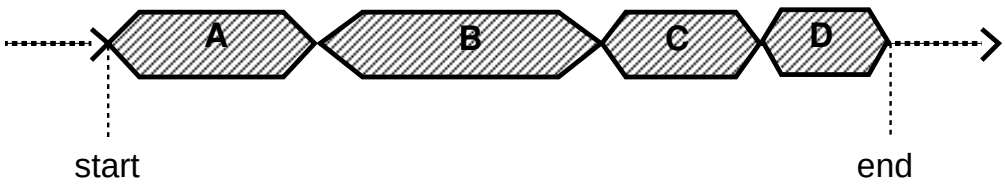


Figure 1. An Application Made Up of Four Logical Sections

However, if two sections, for instance B and C, are computationally independent, they can run concurrently on two nodes. If you were using a MPMD programming model, the application would look like Figure 2. Note that section C takes less than half the time that section B does to run, so one node will be idle until section B ends its run. Looking even closer, you can see that section B could be split into two independent sections, B1 and B2. Therefore, we can arrange our application to run as shown in Figure 3 on page 9. In this case, we have better *load balancing*, so that the amount of time spent to run each section concurrently is comparable.

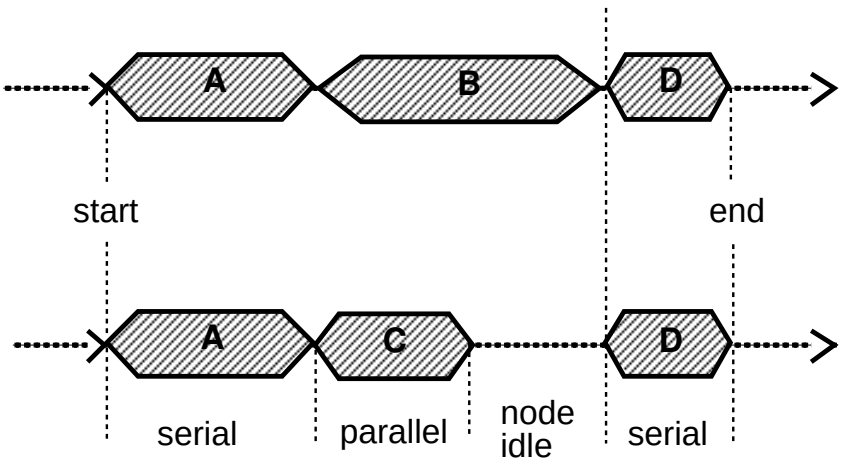


Figure 2. Program Decomposed into Two Parallel Tasks

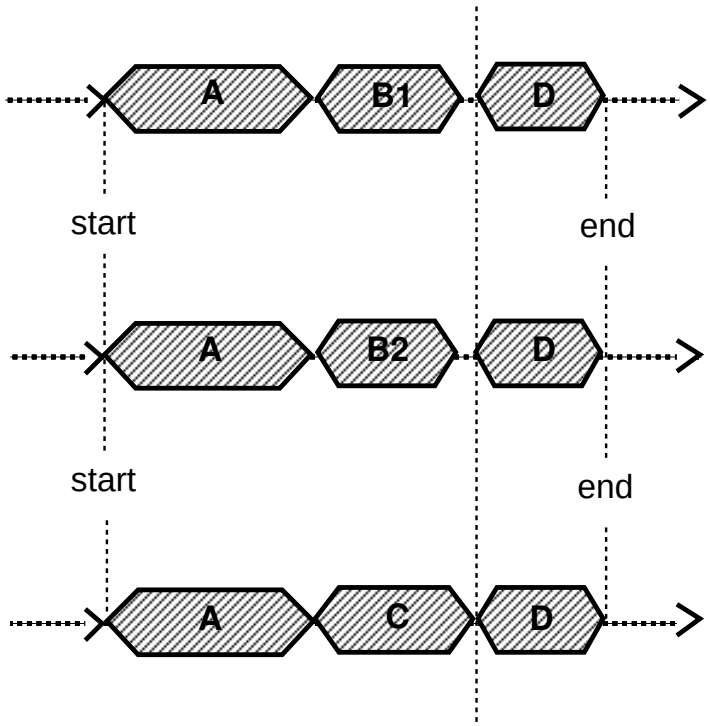


Figure 3. Better Load Balancing by Increasing the Number of Sections

Sometimes, dividing the application into even smaller parts does not improve performance. When the time that is required to schedule and synchronize the sections becomes comparable with the time spent in running, increasing the number of sections could actually lengthen execution.

Sections of your program can also be partially dependent on each other. Note that in Figure 4, section D is dependent only on the first part of section B (section B1). In this case, you can send a message when section B1 has completed to allow section D to start. Section B2 and D will then run concurrently. This example shows the best way to use messages to communicate the completion of an event. There is no time wasted in waiting for messages to arrive, the task that is ready to receive (section D) immediately finds the message available because it was already sent from section B1.

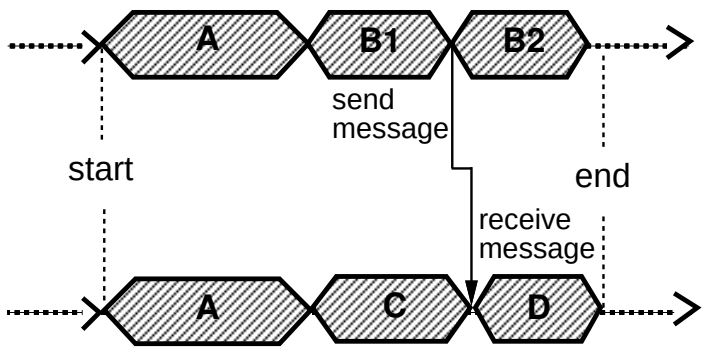


Figure 4. Partial Dependence Between Two Sections

Design and Performance Considerations

There are several factors that affect the performance of a parallel program during execution. Some are based on general considerations that are very similar to those you can apply to serial programming, while others are peculiar to the parallel environment and must be analyzed and treated carefully.

Sometimes you may need to change the way you partition and parallelize your program in order to improve execution. The main factors that influence the way you design your parallel program and your ability to control its performance are:

- Load balancing
- Communication overhead
- Granularity
- Data dependency
- The ratio of the parallel part of the program to the serial part.

Profiling a Serial Program

Before you parallelize a serial program, it is important to identify the parts of the program that take the most time to run. These are known as *hot spots*. Since improving performance involves reducing the processing time of the most compute-intensive parts of your program, parallelizing them is a good way to achieve this improvement.

Profiling a serial program then involves extracting information about the runtime spent by every significant part of the application, like subroutines and DO loops. This technique is also known as *hot-spot* analysis.

There are tools available to help you look for the hot spots in your program. In the AIX operating system, you can use the **prof** and **gprof** commands to determine total executions and the number of calls for each subroutine. These statistics are important in understanding how the program behaves and can influence the way you parallelize it.

Note: Hot-spot analysis is always recommended whether you know what the application does or not.

Considering Communication Overhead

Parallelizing a program creates a *parallel overhead* which can reduce performance. The factors that cause this overhead come from handling the parallel tasks (see the list on page 12) and communicating among nodes.

Amdahl's law concerns the relationship between the parallel speed-up and the serial portion of the program. The traditional form of Amdahl's law describes the total run time, T_1^s, of a serial program running on a single node as composed of two parts such that:

$$T_1^s = T_{<serial>}^s + T_{parallel}^s$$

T_{serial}^s = serial execution time of the non-parallelizable part of the application
$T_{parallel}^s$ = serial execution time of the parallelizable part of the application

Figure 5 illustrates this further.

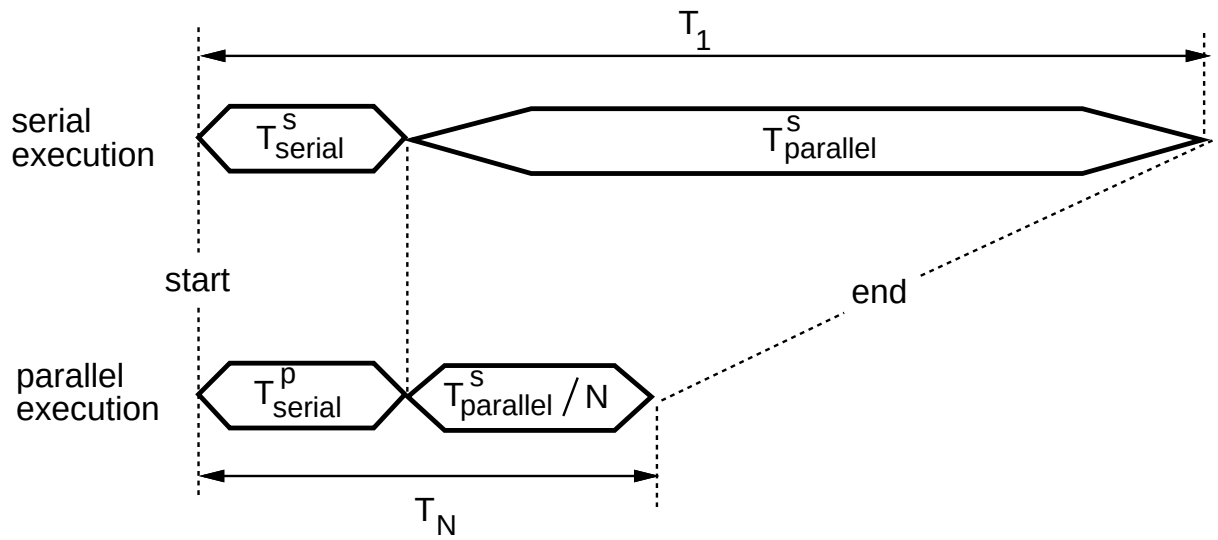


Figure 5. Serial and Parallel Execution Times

If you assume an ideal situation, where the parallel part can be parallelized by distributing equal shares among the nodes, you can expect the total execution time, T_N^p, for the parallel program running on N nodes to be:

$$T_N^p = T_{<serial>}^p + T_{<parallel>}^p = T_{<serial>}^s + \frac{T_{<parallel>}^s}{N}$$

The parallel speed-up, S_N, can be expressed as:

$$S_N = \frac{T_1^s}{T_N^p}$$

By denoting the parallelizable fraction of the program as *p*, the following relationships exist:

$$T^s_{\text{parallel}} = p\ T^s_1$$

$$T^s_{\text{serial}} = (1 - p)\ T^s_1$$

and:

$$S_N = \frac{T^s_1}{T^s_{\text{serial}} + \frac{T^s_{\text{parallel}}}{N}} = \frac{T^s_1}{(1 - p)\ T^s_1 + \frac{p\ T^s_1}{N}} = \frac{1}{(1 - p) + \frac{p}{N}}$$

The fundamental limits of the parallel speed-up can be computed by setting the parallel fraction, *p*, to 1. In other words, the application is completely parallelizable, while increasing the number of nodes, *N*, to infinite yields:

$$S < S_N\big|_{p=1} = N$$

$$S < S_N\big|_{N\to\infty} = \frac{1}{(1 - p)}$$

The expression for T^p_N is an approximation because it does not take into account the *parallel overhead*. You can express the parallel overhead as *OH* and calculate the speed-up like this:

$$S_N = \frac{1}{(1 - p) + \frac{p}{N} + OH}$$

The term *OH* describes several factors that affect the behavior of parallelized tasks, such as time spent in:

- Scheduling, initializing, and synchronizing parallel tasks
- Obtaining and freeing shared resources whose contention is handled by using locks and semaphores
- Waiting for other parallel tasks to be completed, if the work is unbalanced
- Communication among nodes

Communication overhead refers to the time spent in communicating among nodes, compared to the serial computing time of the parallelizable part of the application. Communication overhead is an important factor in evaluating the performance of a parallel program.

Communication between nodes is not instantaneous and the time needed to send a message depends on its size. You can split the execution time of the parallel part, $T^{p}_{<parallel>}$, into two terms:

$$T^{p}_{\text{parallel}} = TCOMP_{\text{parallel}} + TCOMM_{\text{parallel}} = \frac{T^{s}_{\text{parallel}}}{N} + TCOMM_{\text{parallel}}$$

$TCOMP_{\text{parallel}}$ = computation time of the parallel part of the application
$TCOMM_{\text{parallel}}$ = communication time of the parallel part of the application

To simplify, assume that the application is completely parallelizable, that is $p = 1$ and $T^{s}_{\text{serial}} = 0$. You can then write the following expression for the inverse of the parallel speed-up:

$$\frac{1}{S_N} = \frac{T^{p}_{\text{parallel}}}{T^{s}_{\text{parallel}}} = \frac{TCOMP_{\text{parallel}} + TCOMM_{\text{parallel}}}{T^{s}_{\text{parallel}}}$$

$$= \frac{\frac{T^{s}_{\text{parallel}}}{N} + TCOMM_{\text{parallel}}}{T^{s}_{\text{parallel}}} = \frac{1}{N} + \frac{TCOMM_{\text{parallel}}}{T^{s}_{\text{parallel}}}$$

Finally, we can consider the communication time as proportional to the number of nodes, and rewrite the inverse of the speed-up as:

$$\frac{1}{S_N} = \frac{1}{N} + N \times \frac{TCOMM_1}{T^{s}_{\text{parallel}}} = \frac{1}{N} + \frac{N}{r_1}$$

$r_1 = \frac{T^{s}_{\text{parallel}}}{TCOMM_1}$ = computation-to-communication ratio of the program

By using this expression, you can determine the number of nodes that allow the maximum speed-up, and find the optimum balance between increasing the level of parallelism and limiting the communication overhead.

Granularity and Load Balancing

The term *granularity* refers to the amount of computational work that is allocated to parallel tasks. It relates to the length of the parallel sections, in terms of the number of instructions that a parallel task executes.

When parallel tasks are made up of a sequence of small instruction streams, the granularity is said to be *fine*. On the other hand, when parallel tasks involve long computations, made up of a large number of instructions, the granularity is said to be *coarse*. Granularity is often used to refer to the quality of a parallel program's content.

Fine grained programs are typically less efficient than coarse grained programs because of the overhead involved in scheduling and synchronizing parallel tasks. In this case, you should take into account the time needed for setting up a parallel task compared to its execution time.

The advantage of fine grained parallelism, on the other hand, is that it allows you to more easily balance the computational load among nodes. This is because you can distribute small parallel tasks continuously, in turn, to each node, preventing a situation from arising where one or more nodes is idle. This situation occurs more frequently with coarse grained parallelism, when the parallel tasks are unbalanced.

Load balancing is also a crucial factor in obtaining good turnaround time. *Static load balancing* involves assigning parallel tasks to the available nodes in a predetermined and ordered way. With *dynamic load balancing*, you decide at execution which nodes should run the specified parallel task. Fine grained programs are well-suited to dynamic load balancing, but running a large number of small tasks introduces a considerable parallel overhead. Coarse grained programs involve lower overhead, but it is more difficult to balance tasks across the nodes.

Tuning a Parallel Application

This section provides suggestions on how to tune a parallel program and is oriented to systems that have a high *computation-to-communication ratio*. The computation-to-communication ratio refers to the number of operations that a node can perform in local memory per second, divided by the total number of words that the communication system can deliver per second. This definition assumes a very simple model; it ignores many relevant parameters like transmission latency, cache effects, and so on, and depends heavily on the application. Using this measure, both IBM RISC System/6000 *clusters* and the SP would be considered systems with very high computation-to-communication ratios.

Following are some suggestions for a programming environment in which the programmer has explicit control over communication. The time required to send a message can be roughly evaluated as the sum of the latency time, independent of the message length, and the transmission time, which is approximately proportional to the message length.

Other factors effecting performance that are influenced by the parallel architecture, include the time required to:

- Send a `0`-byte message (this is called latency)
- Generate a new task
- Terminate a task
- Synchronize between different tasks

These factors are more conveniently expressed in terms of the number of instructions that can be executed in the same period of time. As mentioned previously, the time required to send a message can be estimated as the sum of the latency and transmission times. Transmission time can be estimated as the message length divided by the available network bandwidth. These timings are dependent on the parallel architecture and hardware.

An application can be defined as coarse grained if the communication time is negligible when compared to the computation time, since most time is spent in computation. Otherwise, the application is said to be fine grained, since relatively less time is spent in computation and more time is spent in transmission.

It is worth noting that granularity depends on both the structure of the application and the hardware platform it is run on. An application, run on two different

hardware platforms, may appear as coarse grained on one, and fine grained on the other. This results from the efficiency of the parallel architecture in creating tasks, sending messages, synchronizing between tasks and terminating tasks.

Tuning the Parallelization

To tune a parallel program, you must start with a version of the parallel code that is already optimized for executing each sequential task.

The balance of the load among concurrent tasks should be as even as possible. In static load balancing, the work is assigned to each task beforehand, while dynamic load balancing can be implemented by a self-scheduling loop. Static load balancing is less flexible than dynamic load balancing, but it has lower overhead because it does not involve any communication or synchronization. Static load balancing is easier to program and can be used effectively unless a load imbalance forces you to use dynamic load balancing.

In order to reduce the overhead required to run concurrent tasks, keep the following rules in mind:

- To reduce the overhead required for originating and terminating tasks, originate tasks once and reuse them.
- Provided that you have more tasks than nodes, maximize the work each task performs by combining more tasks into one.

Tuning the Communication

The considerations for the computation-to-communication ratio, already discussed, suggest some simple rules for tuning the communication between tasks of a parallel program:

- Send as little data as possible. If part of the data to be sent can be computed locally, do not send it. The crossover point can be estimated as the computation-to-communication ratio. If computing new values takes less time than receiving them from another node, computing them locally will save time (unless you do not have enough local memory on your nodes).
- Gather messages together that go to the same task whenever possible and send only one message. This involves less latency time and, therefore, will take less time than sending two different messages. However, if both messages are long, the time required to copy one of them close to the other in virtual memory can be greater than the latency time. In this case, you will need to look more closely at this situation. **pvm_initsend** (**PvmDataInPlace**) can be used to reduce the amount of data copying required.

Optimizing the Use of System Functions

Most message passing systems offer more than the simple *send* and *receive* functions. These additional functions are often optimized and are convenient tools for the programmer. For example:

- To synchronize tasks, do not exchange messages. Instead, use the proper primitive if it is available (for example, IBM PVMe for AIX's *barrier*).
- To broadcast data, do not write a loop of *send*s. Instead, use the broadcast primitive if it is available (typically, it uses a binary tree for communication).

Chapter 3. Product Installation

This chapter discusses installation and configuration requirements for the IBM PVMe for AIX program product and outlines the required steps.

Before You Begin

Before installing IBM PVMe for AIX, be sure that the root id has the proper kerberos authentication.

Requirements

The following are requirements for the IBM PVMe for AIX program product:

- The IBM SP system, configured with the High Performance Switch feature.
- AIX level 4.1.3 (or later modifications).
- The Parallel System Support Programs (5765-529), included with the SP system, configured and installed (IBM PVMe for AIX uses the *Resource Manager* and a *low-level communication protocol* which is part of this software).

Software Prerequisites

This release of IBM PVMe for AIX requires that AIX 4.1.3 or later modifications be installed on your system. The following software from the IBM Parallel System Support Programs (5765-529), included with the SP system, must also be installed on your system:

- MPCI Protocol
- Resource Manager

Installing the IBM PVMe for AIX Software

The IBM PVMe for AIX program product is installed using the SMIT system management facility. This section contains a description of the installation procedures.

Installing IBM PVMe for AIX on a non-SP system

This section describes how to install IBM PVMe for AIX on a single node, control workstation, or external workstation.

1. Login as superuser and enter `smit` by typing:

   ```
   smit install
   ```

2. Perform the following sequence of menu selections:

   a. Standard Installation & Maintenance

   b. Software Installation & Maintenance

   c. Install/Update Software

   d. Install Software with Updates

© Copyright IBM Corp. 1993, 1994, 1995

3. Specify the device from which you are installing the IBM PVMe for AIX program.
4. Proceed to perform the actual installation.

   Note: You can leave the default values for all the other fields of the last menu.

During the installation process, status or error messages are displayed on the screen.

Installing IBM PVMe for AIX on an SP system

This section describes how to install IBM PVMe for AIX on an SP system. You can install it on a single node or on the entire system.

To begin installing IBM PVMe for AIX on all nodes of the SP system:

1. Login as superuser on the control workstation.
2. Execute the **dsh** command as follows:

   ```
   dsh -a installp -a -d <dev_name> pvme
   ```

   where **dev_name** is the device name or path name of the install image file. from which you are installing IBM PVMe for AIX. It must be available on each node (in other words, it can be on an NFS mounted filesystem, installed on the control workstation).

   Note: To install IBM PVMe for AIX on a subset of nodes, use the *-w* option of the **dsh** command.

To begin installing IBM PVMe for AIX on a single node of the SP system:

1. Login as superuser on the control workstation.
2. Execute the **installp** command as follows:

   ```
   installp -a -d <dev_name> pvme
   ```

   where **dev_name** is the device name or path name of the install image file. from which you are installing IBM PVMe for AIX. It must be available on each node (in other words, it can be on an NFS mounted filesystem, installed on the control workstation).

After installing the IBM PVMe for AIX package from the installation media, the appropriate directory structure is created under **/usr/lpp/pvme**.

Next, some of the system files get updated, on all the nodes, so that a new kernel extension will get loaded and configured whenever a node is started. This means that if one node (or the whole system) is switched off and then on again, the kernel extension will be loaded automatically (because the proper lines have been added to the files that are run when the system is booted).

For example, the following lines are added to **rc.net**:

```
###PVMe
###PVMe
/usr/lpp/pvme/etc/addkex /usr/lpp/pvme/etc/pvm3e.kex
###PVMe
```

Some symbolic links are created so you can link applications without specifying the full path to the IBM PVMe for AIX installation directory.

Note: You can cancel these updates by using the **smit** command to reject the installation.

Customizing Your Installation

Once the IBM PVMe for AIX program product has been successfully installed and configured, a few additional steps may be performed in order to customize the installation.

Note: If you use AFS, see Appendix H, “Enabling AFS” on page 249.

The **pvm6kcnf** file, located under the /**usr**/**lpp**/**pvme** directory, contains a few parameters that may be changed to suit your particular needs. This file, by default, contains these lines:

```
MAXTASKS=128
MAXNGRP=8
RESET_ON_START=FALSE
KILL_ON_EPOCH=TRUE
DEFPOOLID=1
```

The previous lines specify that:

- the system-wide maximum number of tasks is 128 (MAXTASKS=128)
- the system-wide maximum number of groups is 8 (MAXNGRP=8)
- the default for `RESET_ON_START` is false (override by the **-R** option).
- the default for `KILL_ON_EPOCH` is true (override by the **-nokill** option).
- the default pool id from which the Resource Manager will be asked to allocate nodes is 1 (DEFPOOLID=1).

MAXNGRP sets the maximum number of distinct groups supported by the IBM PVMe for AIX program product. MAXTASKS sets the maximum number of tasks supported by the IBM PVMe for AIX program product.

At startup, the IBM PVMe for AIX daemon (**pvmd3e**) looks for the **pvm6kcnf** file, first in your home directory and then, if not found there, under /**usr**/**lpp**/**pvme**. By editing the main copy of this file or by placing an updated copy under the home directory, you can override the default for any values on a system wide or user basis.

Customizing the debugger and debugger2 Shell Scripts

There are two *shell* scripts called **debugger** and **debugger2**. These scripts are run by the daemon whenever a new process must be started under the debugger. These shell scripts open a new window on the screen, run **dbx** on it, and then run the user process.

Starting with the templates provided with the IBM PVMe for AIX package, available under the /**usr**/**lpp**/**pvme**/**bin** directory, you can create your own copies of the files and then customize them. This allows you to do customization on a per-user basis.

The **debugger** and **debugger2** scripts can be in any of three locations, in this order of precedence:

1. If the daemon **bx** option (see “Syntax of the Hostlist File” on page 35) is set, it will be used as the pathname for the **debugger** script.
2. If the environment variable **PVM_DEBUGGER** is set, it will be used as the pathname for the **debugger** script.
3. If **PVMEPATH** is set or the **ep** daemon option is set, the daemon will look for **debugger** in that directory.
4. Otherwise, the daemon will assume the **debugger** script is in **$HOME/pvm3/bin/RS6K**.

Chapter 4. Product Startup

This chapter contains information about the configuration and startup procedures of the IBM PVMe for AIX program product, at both user and system level. The most important differences between the IBM PVMe for AIX program product and freeware PVM are also discussed.

The IBM PVMe for AIX program product interfaces both with the Resource Manager and with the LoadLeveler program products, so that on the SP system, you can easily run an IBM PVMe for AIX session either in *interactive* or *batch* mode. This allows the Resource Manager to allocate the computational resources.

IBM PVMe for AIX libraries rely on the Message Passing Client Interface (MPCI) for communication between processes over the High Performance Switch. There are two implementations of the MPCI; one running over the IP protocol, and one running over the User Space (US) protocol. US is a lower level protocol that runs in the user space, and has been specifically designed and tuned for the High Performance Switch and adapters.

IBM PVMe for AIX programs can exploit either of these protocols, the only difference being the MPCI libraries with which your program is linked. See "Compiling and Linking a Parallel Program on a POWERparallel System" on page 87 for more information on how to compile, link, and run your parallel program.

Before You Start

For both interactive and batch sessions, you must create the proper **.rhosts** file in your home directory before starting the IBM PVMe for AIX program product. Doing this allows you to run the **rsh** command from remote nodes without having to type your password. The **.rhosts** file must have read/write permission for the owner only. For each remote host, the **.rhosts** file can contain a line, using the following syntax, for running the **rsh** command on the local host:

<hostname>.<domainname> <username>

Any alternative syntax described in the AIX manual page for rhosts can also be used.

Note: AFS may be used if the system administrator has enabled PVMe for AFS.

Running an IBM PVMe for AIX Session

Running an IBM PVMe for AIX session requires three steps:

1. Start IBM PVMe for AIX daemon.
2. Run user program or programs.
3. Shut down IBM PVMe for AIX daemon.

The are three different ways to carry out these steps:

1. **Daemon runs only for the lifetime of a job**

© Copyright IBM Corp. 1993, 1994, 1995

Run the daemon, giving it the pathname of the program to be run. The name of the daemon executable is **pvmd3e**. The daemon will start, run the program, and shut down when the program terminates. This method combines all three steps in one command. This method is useful for running IBM PVMe for AIX programs in batch mode.

```
pvmd3e [opts] -exec "<absolute program pathname>
<optional program arguments>"
```

Note: See "IBM PVMe for AIX Daemon Options" on page 31.

For example:

```
pvmd3e 8 -trace -exec "/u/joe/pvm3/bin/RS6K/myprog -xyz"
```

2. **Daemon Runs in the Background**

   a. Start the daemon running in the background. The daemon will run until it's stopped, so a series of jobs may be run. If the nodes that the daemon obtains from Resource Manager are allocated in User Space mode or dedicated CPU mode, other jobs may not use those nodes until the daemon is shut down.

      ```
      pvmd3e  [opts]  &
      ```

   b. Run the program from the command line on one of the nodes the daemon has obtained from Resource Manager. Any number of programs can be run, one after another, before the daemon is shut down. Only one program (and its spawned tasks) may run simultaneously in the same session.

      To run more than one job at a time, see "Running Multiple Sessions" on page 26.

      ```
      <user program name> <user program arguments...>
      ```

   c. Shut down the daemon by starting the **pvme** console without options and issue the **halt** command.

      ```
      % pvme
      pvmd already running
      pvme> halt
      pvmd exiting
      %
      ```

      The daemon can also be halted with:

      ```
      kill -term <daemon pid>
      ```

      It should never be halted with:

      ```
      kill -9 <daemon pid>
      ```

      or

      ```
      kill -kill <daemon pid>
      ```

For example:

```
% pvmd3e 4 &
[1]     18256
% PVMD:
      Trying to get 4 nodes.
PVMD: Using k10n01 with dx=/usr/lpp/pvme/bin/pvmd3e,
      ep=/u/johndoe/pvm3/bin/RS6K.
PVMD: Using k10n03 with dx=/usr/lpp/pvme/bin/pvmd3e,
      ep=/u/johndoe/pvm3/bin/RS6K.
PVMD: Using k10n05 with dx=/usr/lpp/pvme/bin/pvmd3e,
      ep=/u/johndoe/pvm3/bin/RS6K.
PVMD: Using k10n07 with dx=/usr/lpp/pvme/bin/pvmd3e,
      ep=/u/johndoe/pvm3/bin/RS6K.
PVMD:
      Ready for <4> hosts.
% myprog -xyz 4
      This is output from myprog.
% PVMD:
      New epoch 1.

% anotherprog
      This is output from another program.
% PVMD:
      New epoch 2.

%pvme
pvmd already running
pvme> halt
pvmd exiting
%
```

3. **Running from the Console**

   a. Start the console, giving it appropriate command line options. The console will start the daemon in the background with the given options.

      ```
      pvme [opts...]
      ```

   b. When the console is running, it will give the user a command prompt which the user can enter a number of commands. For descriptions of these commands, see “Using the IBM PVMe for AIX Console” on page 29 or respond to “pvme>” with **help**. The **spawn** command can be used to start a user program from the console.

      ```
      pvme> spawn  [opts]  <user program name> <user program arguments...>
      ```

   c. Use the **halt** command to shut down the PVMe daemon and console.

      ```
      pvme> halt
      ```

For example:

```
% pvme 4
PVMD:
      Trying to get 4 nodes.
PVMD: Using k10n01 with dx=/usr/lpp/pvme/bin/pvmd3e,
      ep=/u/johndoe/pvm3/bin/RS6K.
PVMD: Using k10n03 with dx=/usr/lpp/pvme/bin/pvmd3e,
      ep=/u/johndoe/pvm3/bin/RS6K.
PVMD: Using k10n05 with dx=/usr/lpp/pvme/bin/pvmd3e,
      ep=/u/johndoe/pvm3/bin/RS6K.
PVMD: Using k10n07 with dx=/usr/lpp/pvme/bin/pvmd3e,
      ep=/u/johndoe/pvm3/bin/RS6K.
PVMD:
      Ready for <4> hosts.
pvme> spawn -> testout
  1 successful
  t10020000
  pvme>  10030001 @k10n05: This is output from task 10030001.
   10040002 @k10n07: This is output from task 10040002.
   10020000 @k10n03: This is output from task 10020000.
   10010003 @k10n01: This is output from task 10010003.
PVMD:
     New epoch 1.

pvme>
```

Only one instance of the console may be run on each node. While the console is running, the daemon is also running in the background, so you also have the option of starting programs from the command prompt in a different window.

Entering the **quit** command at the console prompt will shut down the console, leaving the daemon running in the background.

```
pvme> quit

pvmd still running.
%
```

Conversely, with the daemon running the in background, starting the console with no options will attach or reattach the console to the daemon and leave the user at the console prompt.

```
% pvme
pvmd already running
pvme>
```

Note: The daemon executable **pvmd3e** and the console executable **pvme** are installed in `/usr/lpp/pvme/bin`, but the install process creates symbolic links to them from the `/usr/bin` directory, so it's necessary only to have `/usr/bin` in your **PATH** environment variable to have easy access to both.

Node Allocation

The IBM PVMe for AIX daemon options determine how many and which nodes will be allocated for the session, and how they will be allocated.

Determining Which Nodes will be Allocated

- **One Node**

  Without arguments, the commands **pvme** and **pvmd3e** start the IBM PVMe for AIX program product only on the local host. Any tasks that are spawned without specifying a different host will be created on the local host.

- **Number of Nodes**

  To allocate a certain number of nodes, supply **pvme** or **pvmd3e** with an integer argument, the number of nodes required. The daemon will send a request to the Resource Manager to obtain the required number of nodes. The Resource Manager in turn, assigns the nodes to the daemon (if they are available).

  Nodes will be allocated from a **node pool**. The default **pool id** for node allocation is **1**. If you want nodes to be allocated from another **pool id**, you must do either of the following:

  – Use the **-p** option (See “Using the -p Option” on page 33).

  – Specify the stanza DEFPOOLID in the **pvm6kcnf** file. See “Customizing Your Installation” on page 19.

- **Specific Nodes**

  To allocate specific nodes or workstations, specify the name of a configuration file as an argument. It will be referred to as the **hostlist** file, although that filename is not mandatory. In the **hostlist** file, you must specify the name of the hosts that will be used for running the parallel application. The statement syntax for the hostlist file is described in “Syntax of the Hostlist File” on page 35. If the hosts listed in your **hostlist** file are available, the Resource Manager will allocate them for your job. If any of the nodes are unavailable, your session will not be able to start. For this reason, it is preferable to use the form specifying the number of required nodes.

Specifying Allocation Options

- **Communication Subsystem Sharing**

  By default, IBM PVMe for AIX uses the User Space communications protocol for message passing, which supports the greatest performance, but requires exclusive access to the High Speed Switch adapter on the nodes on which the IBM PVMe for AIX program is running. IBM PVMe for AIX can also use the IP protocol, which doesn't provide the performance of User Space, but allows multiple programs to communicate from the same node.

  To allocate nodes in IP mode, use the **-ip** option. See “Using the -ip Option” on page 32.

- **CPU Sharing**

  By default, IBM PVMe for AIX requests nodes from the Resource Manager to which IBM PVMe for AIX will have exclusive access. To allocate nodes that may be used by other jobs, use the **-share cpu** option. See “Using the -share Option” on page 34.

Running Multiple Sessions

IBM PVMe for AIX supports only one running job per session, and one queued job. If a job is running and another job is started, the new job will be blocked until the first job completes. If a third job is started before the first job has completed, it will fail to connect to the daemon.

To run multiple simultaneous jobs, it's necessary to create a different session for each job. Each session creates a data file that IBM PVMe for AIX programs need to contact the daemon, so to separate simultaneous jobs, each session needs to use a different data file. The default name of this file is **.pvmdname**. To change the name of the data file, set the **PVMHFN** environment variable to the name of the data file you wish to use for that session. The variable must be set in the shell where you run the daemon, as well as in the shell where you start the application program. Also, to properly start multiple sessions after the first one, the **-R** option must never be supplied to the daemon or console.

If the jobs you wish to run simultaneously will be using the same nodes, the nodes must be allocated for shared CPU usage, and the jobs must not use the User Space communications protocol. See “Specifying Allocation Options” on page 25.

For example:

```
[315] pvmd3e -ip -share cpu 4 &
[316] PVMD:
        Trying to get 4 nodes.
PVMD: Using k10n01 with dx=/usr/lpp/pvme/bin/pvmd3e,
      ep=/u/johndoe/pvm3/bin/RS6K.
PVMD: Using k10n03 with dx=/usr/lpp/pvme/bin/pvmd3e,
      ep=/u/johndoe/pvm3/bin/RS6K.
PVMD: Using k10n05 with dx=/usr/lpp/pvme/bin/pvmd3e,
      ep=/u/johndoe/pvm3/bin/RS6K.
PVMD: Using k10n07 with dx=/usr/lpp/pvme/bin/pvmd3e,
      ep=/u/johndoe/pvm3/bin/RS6K.
PVMD:
      Ready for <4> hosts.
[317]
[318] prog1 &
[319] Program 1 running.

[320]
[321] export PVMHFN=pvmdname.2
[322] pvmd3e -ip -share cpu 4 &
[323] PVMD:
      Trying to get 4 nodes.
```

```
PVMD: Using k10n01 with dx=/usr/lpp/pvme/bin/pvmd3e,
      ep=/u/johndoe/pvm3/bin/RS6K.
PVMD: Using k10n03 with dx=/usr/lpp/pvme/bin/pvmd3e,
      ep=/u/johndoe/pvm3/bin/RS6K.
PVMD: Using k10n05 with dx=/usr/lpp/pvme/bin/pvmd3e,
      ep=/u/johndoe/pvm3/bin/RS6K.
PVMD: Using k10n07 with dx=/usr/lpp/pvme/bin/pvmd3e,
      ep=/u/johndoe/pvm3/bin/RS6K.
PVMD:
      Ready for <4> hosts.
[324]
[325] prog2
Program 2 running
```

Starting IBM PVMe for AIX from the Control Workstation

An IBM PVMe for AIX daemon can be started from the control workstation through the daemon directly, or by invoking the IBM PVMe for AIX console.

If you are running the daemon, specifying the number of nodes, the Resource Manager will not include the control workstation in the list of hosts. As a result, you must run your program via the **spawn** command from the IBM PVMe for AIX console, or via a **rsh** or **rexec** on one of the allocated nodes. You may also run your program by logging into one of the allocated nodes and starting the program from the command line.

Since the daemon, by default, does not check for the presence of the local host in the *hostlist* file, you can also use the *hostlist* syntax to run an IBM PVMe for AIX session over a set of SP nodes. Then, your application can be started on the SP nodes by first invoking the IBM PVMe for AIX console, if not already running, and then issuing the **spawn** command, or by executing a **rsh** or **rexec** system command.

Starting IBM PVMe for AIX from a Compute Node

An IBM PVMe for AIX daemon can also be started from an SP node directly.

Select a node on which to run the daemon, and log into that node via **rlogin** or **telnet**. Start the daemon or the console. If you're using the Resource Manager to select nodes, the node on which the daemon is now running may not be one of the nodes allocated for the session. In that case, you'll need to log into any of the nodes that has been allocated. Once the daemon is started and you are logged into one of the allocated nodes, you can start an IBM PVMe for AIX program by opening a console on that node and using the **spawn** command, or by directly entering it from the command line.

Including External Workstations in the IBM PVMe for AIX Machine

You can run an IBM PVMe for AIX interactive session on a mixed set of hosts, including SP nodes and external RISC System/6000 workstations. To do this, you must also have the IBM PVMe for AIX program product installed on your external workstations, and you must provide different executables for running on the SP nodes and on the other workstations (see “Compiling and Linking a Parallel Program on a POWERparallel System” on page 87 and “Compiling and Linking a Parallel Program on an External Workstation” on page 88).

To set up the virtual machine (to start the IBM PVMe for AIX daemon on a mixed set of hosts), you need to run the daemon from the **SP system** with the *hostlist* parameter. The *hostlist* must contain a special statement (**#rm_nodes**) for the SP nodes that will be allocated by the Resource Manager, and one normal statement for each external host.

The **#rm_nodes** statement allows you

- to request for anonymous nodes from a *pool*.

  ```
  #rm_nodes= <num_nodes>
  ```

  Example:

  ```
  # PVMe configuration
  #
  # Set default option (to be used for SP nodes)
  #
  * ep=/u/pvmuser/pvm3/bin/RS6K_SP
  #
  # Request the Resource Manager to allocate 5 generic nodes:
  # (The following line is NOT a comment)
  #
  #rm_nodes=5
  #
  # The following hosts are external workstations
  rios10   ep=/u/pvmuser/pvm3/bin/RS6K
  rios11   ep=/u/pvmuser/pvm3/bin/RS6K
  rios12   ep=/u/pvmuser/pvm3/bin/RS6K
  ```

- to request specific nodes from the SP system.

  ```
  #rm_nodes=<name1>, <name2>, ...
  ```

  Example:

  ```
  # PVMe configuration
  #
  # Set default option (to be used for SP nodes)
  #
  * ep=/u/pvmuser/pvm3/bin/RS6K_SP
  #
  # Request the Resource Manager to allocate 4 specific nodes:
  #
  #rm_nodes=spn1, spn2, spn3, spn4
  #
  # The following hosts are external workstations
  rios10   ep=/u/pvmuser/pvm3/bin/RS6K
  rios11   ep=/u/pvmuser/pvm3/bin/RS6K
  rios12   ep=/u/pvmuser/pvm3/bin/RS6K
  ```

In both cases, you can run the IBM PVMe for AIX daemon from any of the SP nodes or from the control workstation. Workstations other than the control workstation may also be used as long as their names in the hostlist can be resolved without adding domain information.

Using the IBM PVMe for AIX Console

To start the IBM PVMe for AIX console, use the **pvme** command. In the IBM PVMe for AIX program product, a single console per node is supported. The console allows interactive starting and killing of the IBM PVMe for AIX processes and accepts the following commands from the standard input:

Command	Description
add	Disabled
alias	Allows you to define aliases for console commands.
conf	Shows the actual virtual machine configuration, that is, a line for each host containing the *hostname*, the **pvmd** task ID, host architecture (**RS6K**), the maximum fragment size and a relative speed rating.
delete	Disabled
echo	Prints the supplied argument(s).
halt	Halts all IBM PVMe for AIX processes including consoles and then shuts IBM PVMe for AIX down.
help or ?	Shows a list of the available commands.
help [command]	Displays an explanation of the command and its syntax.
jobs	lists the jobs that are running.
id	Prints the console task identifier.
kill	Terminates a process. The syntax is **kill** *tid*.
mstat	Shows the status of specified hosts.
ps [-a]	Lists all processes currently on the virtual machine, their locations, their task IDs and their parent's task ID.
pstat	Shows the status of a single IBM PVMe for AIX process.
quit	Exits the console leaving daemons and IBM PVMe for AIX jobs running.
reset	Kills all IBM PVMe for AIX processes except consoles and resets all the internal IBM PVMe for AIX tables. It forces a new epoch and leaves the daemon in an idle state.
setenv	Allows you to set an environment variable.
sig	Sends a signal to an IBM PVMe for AIX process.
spawn	Starts one or more IBM PVMe for AIX applications.
unalias	Undefine a command alias.
version	Prints the version of **libpvm** being used.

Further information on the console commands can be obtained by entering the online help command:

```
help [command]
```

Running an IBM PVMe for AIX Batch Session

When a parallel job is started in batch mode, LoadLeveler requests the Resource Manager for the number of nodes specified. The shell script that submits a parallel job to LoadLeveler must contain additional information, with respect to a shell script for a serial job, namely the minimum and maximum number of nodes desired. If the specified number of nodes are available, the IBM PVMe for AIX daemon is started, which in turn starts the IBM PVMe for AIX application through its **-exec** option. The absolute path of the application executable must be specified. For more information, see "IBM PVMe for AIX Daemon Options" on page 31.

LoadLeveler passes the environment variables that are specified to only **its** *parallel master* (IBM PVMe for AIX daemon). As a result, all environment variables must be explicitly passed to all of LoadLeveler *parallel workers* (IBM PVMe for AIX application tasks). The IBM PVMe for AIX application tasks can be either:

- the master task, which is specified via the **-exec** option
- the PVMe worker tasks which are spawned by the PVMe master task.

IBM PVMe for AIX exports:

- the environment variable **HOME** to all the IBM PVMe for AIX tasks (both master and workers)
- the variables indicated in **PVM_EXPORT** to all the IBM PVMe for AIX worker tasks
- the variable **PVMHFN** to the PVMe master task.

Note that **PVMHFN** is only needed by the IBM PVMe for AIX master task and *not* by the IBM PVMe for AIX worker tasks. Therefore **PVM_EXPORT** can be used to export environment variables to all the IBM PVMe for AIX worker tasks.

```
export MYVAR1=<myval1> MYVAR2=<myval2> ...
export PVM_EXPORT=MYVAR1:MYVAR2: ...
```

where *MYVAR1* and *MYVAR2* refer to the environment variables to be exported, and *val1* and *val2* refer to their corresponding values. To export these to the PVMe master task (the one specified via the "-exec" option), these variables must be passed as follows:

```
SETVARS="export MYVAR1=<val1> MYVAR2=<val2> ..."
/usr/lpp/pvme/bin/pvmd3e -exec "$SETVARS; <absolute path of program>"
```

Since **PVM_EXPORT** is only used by the daemon, it is not made part of "`$SETVARS`."

Below is an example of a script file that can be used to submit a parallel job to LoadLeveler. It can be customized by you. Note that you can use the same script to run an interactive IBM PVMe for AIX session. In this case, however, since the number of nodes is not specified as a daemon argument, the default of a single node IBM PVMe for AIX session is assumed.

```
#!/bin/ksh
#
# The following section contains statements for LoadLeveler only; it
# will be ignored if the script is executed directly by the user.
#
# @ initialdir = /u/johndoe/pvme/loadl
# @ output = $(Executable).$(Cluster).$(Process).out
# @ error = $(Executable).$(Cluster).$(Process).err
```

```
#
# Set the PVMe environment variable PVMHFN for the PVMe daemon and
# the PVMe master task. This will allow concurrent multiple PVMe
# sessions for a user.
# @ environment = PVMHFN=pvmdname.$(cluster).$(process)
#
# @ class = PVMe1
# @ job_type = parallel
# @ min_processors = 2
# @ max_processors = 2
# @ requirements = (Arch == "R6000") && (OpSys == "AIX41") && (Adapter == "hps_user")
# @ notify_user = johndoe@rios4.ecsec.romesc.ibm.com
# @ account_no = system
# @ shell = /bin/ksh
# @ notification = complete
# @ checkpoint = no
# @ restart = no
# @ queue
#
# The following section will be used both when the script is
# submitted and when it is executed directly.
#
echo "**** Starting ****"
# Set up environment variables for the PVMe daemon, ex.
# PVMEPATH, if it is different from $HOME/pvm3/bin/RS6K.
export PVMEPATH=/u/johndoe/pvme/samples
#
# If there are other environment variables to be passed to the
# PVMe tasks (both the master task and the worker tasks), then the
# following lines must be uncommented with the actual variable
# names and values substituted. Please refer to explanation above.
# SETVARS="export MYVAR1=<myval1> MUVAR2=<myval2> ..."
# ${SETVARS}
# export PVM_EXPORT=MYVAR1:MYVAR2: ...
#
# Start the daemon with the -exec option, specifying the absolute
# path name of the application executable.
# This is for the sample application provided with PVMe.
/usr/lpp/pvme/bin/pvmd3e -exec "$PVMEPATH/hostbenc<<%%%
100
10
3
%%%"
#
# If $SETVAR has been defined according to the comments given above,
# then invoke the PVMe daemon by first commenting out the above five
# lines, and uncommenting the following lines.
# /usr/lpp/pvme/bin/pvmd3e -exec "$SETVARS; $PVMEPATH/hostbenc<<%%%
# 100
# 10
# 3
# %%%"
#
echo "**** Ending ****"
```

IBM PVMe for AIX Daemon Options

The IBM PVMe for AIX daemon accepts several options that are explained on the following pages. Before reading about these options, note the following:

- Some of the daemon options can be used with the PVMe console.
- The -c option is no longer supported by PVMe.
- The -exec option must be used in batch mode.

Syntax line for IBM PVMe for AIX daemon

pvmd3e [numnodes [-p poolid] | hostlist [-resd no]] [-ip] [-share cpu] [-trace]

[-nokill] [-R]

[-exec “<absolute path of program> <optional program parameters>”]

pvmd3e hostlist -r

Note: The **-exec** option must be specified last when used with the IBM PVMe for AIX daemon.

Syntax line for IBM PVMe for AIX console

pvme [numnodes [-p poolid] | hostlist [-resd no]] [-ip] [-share cpu] [-trace] [-nokill] [-R]

Using the -exec Option

The **-exec** option is primarily for use in batch mode. It allows you to specify the task to be started on the local host. This option can be used for both batch and interactive jobs and is required for batch jobs via LoadLeveler.

The syntax (example) is:

pvmd3e <PVMe options> -exec “<absolute path of program> <optional program parameters>”

When you specify this option, the daemon, after initializing the environment, starts the specified task on the local host and terminates just after the first epoch.

If the command is contained within a shell script, and the application needs to read from the standard input, use the following:

```
pvmd3e [other options] -exec "myprog"<<%%
param1
param2
%%"
```

Note: The **-exec** option will not work from the control workstation. When the **-exec** option is used, it must be the last option given for the pvmd3e command.

Using the -ip Option

The **-ip** option is used in interactive mode and it allows you to specify that the task to be started will use the IP protocol rather than the User Space protocol. The US protocol is only usable by one process at a time, which means that only one parallel job, running over US, can run on a given set of nodes at a time. Using the IP protocol allows multiple jobs to run at the same time. As a result, multiple parallel jobs can run on the same set of nodes at a given time, but each job must request shared use of the CPU.

The syntax (example) is:

```
pvmd3e -ip -share cpu < numnodes > &
```

Note that the protocol to be used is selected at compile time and can be modified with the environment variable **LIBPATH**. See “Compiling and Linking a Parallel Program on a POWERparallel System” on page 87. The **-ip** option does not select the protocol to be used, it only selects the mode in which nodes are allocated.

Using the -nokill Option

The **-nokill** option is for interactive mode and is used to inform the IBM PVMe for AIX daemon to not kill the processes that continue after you have called the **pvm_exit** routine.

When all the tasks leave the IBM PVMe for AIX environment cleanly, the IBM PVMe for AIX daemon resets the content of the task table, event table, groups table, etc.. In IBM PVMe for AIX, this is known as starting a new epoch. After a new epoch is started, the IBM PVMe for AIX daemon has no memory of the previous run. The default action by the IBM PVMe for AIX daemon is to kill all worker tasks that are still running at the end of each epoch.

There may be situations where your application would want to continue running for a while after it has called **pvm_exit**. To enable the worker tasks to run after all of them have called **pvm_exit**, you need to start the IBM PVMe daemon with the **-nokill** option or set the **KILL_ON_EPOCH** environment variable to “FALSE” in your or the system's pvm6kcnf file.

The syntax (example) is:

```
pvmd3e -nokill < numnodes > [other options] &
```

This ensures that when you leave PVMe, the process that wants to continue on will not be killed.

Using the -p Option

The **-p** option is for use in interactive mode and it is used to select the *pool_id* where the Resource Manager allocates nodes for the parallel job execution. If the **-p** option is not specified, nodes are allocated from the *pool_id* **1**. If another *pool_id* must be used, supply the daemon with the **-p** option, followed by the integer identifying the pool.

The syntax (example) is:

```
pvmd3e 5 -p 2 [other options] &
```

Note: You can also specify the default *pool_id* with the DEFPOOLID stanza in the **pvm6kcnf** file.

Using the -resd Option

The **-resd** option can be used to activate and deactivate Resource Manager control over node usage and allocation. The default value is *yes* (Resource Manager control enabled), but you can override this by specifying *-resd no* at daemon startup.

The syntax (example) is:

```
pvmd3e mylist -resd no [other options] &
```

Note: Do **not** use this option if you are running IBM PVMe for AIX on an SP machine; use it only if you are running IBM PVMe for AIX on workstations

over IP. If -resd no is used on an SP, PVMe will not make use of the switch.

Using the -r Option

The **-r** option is for interactive mode and is used to reset all the user's IBM PVMe for AIX sessions running on the nodes listed in the **hostlist** file. The pvmd3e daemon terminates after doing the reset, so no other options are allowed.

The syntax (example) is:

```
pvmd3e hostlist -r
```

where *hostlist* is the file which contains the list of nodes used by the IBM PVMe for AIX program product. The structure of this file is described in "Running an IBM PVMe for AIX Session" on page 21.

If nodes were allocated by the Resource Manager, you must use the **pvmnodelist** file, created by the daemon when the previous session was started (see "Resolving Startup Problems" on page 36).

Note: The daemon must be started in the foreground when this option is used.

Using the -R Option

The **-R** option is for use in batch mode. It causes all the user's IBM PVMe for AIX sessions running on the nodes, listed in the hostlist file, (if any) to be reset, and a new one to start. The result would be the same if you ran the daemon with the **-r** option and then started the IBM PVMe for AIX session (specifying the <*number of nodes*> or the *hostlist* file).

The syntax (example) is:

```
pvmd3e -R < numnodes > [other options] &
```

Using the -share Option

This option is used in interactive mode. By default, IBM PVMe for AIX reserves an SP node for a user as a dedicated resource. This is done via the Resource Manager. As a result, other processes that request access to that node, via the Resource Manager, are denied. The **-share** option lets you override this default.

The syntax (example) is:

```
pvmd3e -share <argument> [other options] &
```

Where:

<**argument**> Is **cpu**. This is the only argument that the **-share** option currently supports. It allows an IBM PVMe for AIX user to request an SP node such that any other process, that is *not* running User Space over the SP's High-Performance Switch, can also use this node.

Other processes, including another IBM PVMe for AIX, that want to run on the same node must request shared access from the Resource Manager.

Using the -trace Option

This option is for use in interactive batch mode and it lets you enable tracing globally in an IBM PVMe for AIX session, without inserting statements in the application source code. The **-trace** option causes the IBM PVMe for AIX daemon to set the trace mask that will be assigned to the user tasks and to all their children. Tracing is enabled on all the events. This means that each user task will produce a trace record for each event.

The syntax (example) is:

```
pvmd3e -trace < numnodes > [other options] &
```

Syntax of the Hostlist File

The **hostlist** file contains a line for each host that participates in the virtual machine, including the *hostname* and possibly one or more of the **hostlist** options. Note that you must designate each host by it's name; you cannot use IP addresses.

The **hostlist** options are:

- **ep=** <absolute path of the directory for the executables>
- **dx=** <full pathname for the IBM PVMe for AIX daemon executable>
- **lo=** <user login name on the remote machine>
- **bx=** <pathname for the debug script>
- **wd=** <working directory where spawned tasks will be executed>

The meaning of the **hostlist** options is explained below. The default values are as follows:

- If the **ep=** option is not set, the **$HOME**/**pvm3**/**bin**/**RS6K** directory is searched for your executables.
- If the **dx=** option is not set, the full pathname for the daemon executable is assumed to be /**usr**/**lpp**/**pvme**/**bin**/**pvmd3e**.
- If the **lo=** option is not set, the login name is assumed to be the same as the user starting the daemon.
- If the **bx=** option is not set, the debug scripts, **debugger** and **debugger2**, are expected to be in the same directory as your executables (**ep**, **PVMEPATH**, or **$HOME**/**pvm3**/**bin**/**RS6K**).
- If the **wd=** option is not set, the working directory is **ep** (**ep**'s default is **$HOME**/**pvm3**/**bin**/**RS6K**).

You can set any of the above options for a series of hosts by placing these options on a single line with an asterisk (*) for the *hostname* field. The defaults are in effect for all subsequent hosts in the **hostlist** file unless they are overridden by another line that sets options for a specific host.

```
# Lines starting with "#" are comment lines
# PVMe configuration
#
spn09       dx=/u/pvmuser1/bin/pvmd3e
# Set a default option
* dx=/u/pvmuser/bin/pvmd3e
```

```
#
spn10
spn11       ep=/u/pvmuser1/bin
spn12       ep=/u/pvmuser2/bin
# spn13
#
```

Note: Freeware PVM supports the following two **so** options in its **hostlist** file:

so=pw
so=ms

Neither the **pw** or **ms** options are supported by IBM PVMe for AIX.

Using a hostlist to include external workstations in a PVMe virtual machine is described in "Including External Workstations in the IBM PVMe for AIX Machine" on page 27.

To start the master daemon on the first host listed in the configuration file, enter the following command:

```
pvmd3e hostlist &
```

The master daemon performs some operations of initialization on all the nodes. When it has completed the operations, your programs may be started on any of the nodes listed in the **hostlist** file.

Note: The IBM PVMe for AIX daemon does not check for the presence of the local host in the *hostlist* file unless you set the *PVMe_CLH* environment variable to *yes*. This means that the node where the daemon is run is not required to be listed in the *hostlist* file. See "PVMe_CLH" on page 226.

The following example uses the configuration file of the previous example:

```
user@spn10> pvmd3e hostlist &
pvmd3e ready for 3 hosts...
```

Resolving Startup Problems

If you are experiencing any startup problems, first ensure the following:

- MPCI is installed on all of the SP nodes.
- The Resource Manager is up and running.

If, after several startup attempts, an error message continues to appear, it might mean that an IBM PVMe for AIX session was left hanging on one or more of the machines defined in the **hostlist** file. In this case, you should run the daemon with the **-r** option. See "IBM PVMe for AIX Daemon Options" on page 31 for more information.

The pvmd3e **-r** and **-R** options will only clean up tasks belonging to the user invoking pvmd3e with this option. They can not distinguish a task as an orphan, so kill all pvme tasks belonging to the user including any that are part of another PVMe session. The options should be used with care. If there are orphan PVMe tasks belonging to other users, it will be necessary for the system administrator to use the **pvmd6kgh** command to find and kill these tasks. The **pvmd6kgh** command is described in a PVMe for AIX manpage.

When you start the **pvmd3e** daemon, the following message may appear:

```
PVMD: Another PVM is starting right now on <hostname>...
Please, wait a few seconds and retry
```

At startup time, the IBM PVMe for AIX daemon creates the **.pvmdname** file in the default directory of the executables, in your **$HOME** directory, or in any alternative directory specified by you for the executables. This file contains the name of the local host, the daemon's socket address and some other data that your programs must know to contact the running daemon. In fact, an IBM PVMe for AIX process attempts to open this file first, in the path supplied by you to the executables (if any), then in your **$HOME** directory, then in the default path (**$HOME**/**pvm3**/**bin**/**RS6K**). If an old copy of this file is present in the directory where you are running your parallel program (and it is different from one of the three listed above) your program will fail, unless you explicitly remove the old file.

The **.pvmdname** file will cause problems if:

- you are trying to run multiple IBM PVMe for AIX sessions, on the same set of nodes or on a disjoint set of nodes, at the same time **and**
- your filesystem is mounted using a network filesystem on all the nodes **and**
- the path to the IBM PVMe for AIX program product executable is the same for the two sessions.

In the scenario above, all programs will try to connect to the same daemon, whose address is written in the more recent copy of the **.pvmdname** file.

To avoid problems, set the environment variable **PVMHFN** to a filename of your choice. Example:

```
export PVMHFN=pvm_session1
```

This file will be created in the same three directories listed above for the **.pvmdname** file.

Note: If you use the **PVMHFN** variable, you must define it both in the shell where you run the daemon and in the shell where you start the program (if they are not the same), and the two definitions must contain the same filename.

For more information on environment variables, refer to Appendix B.

Note: PVMe uses error logging. If you have problems, try running the **errpt** command (aix command) to check the error log.

Verifying the IBM PVMe for AIX Program Product

There are two ways that you can verify that pvme is working correctly. The first way is by using hostlist and the second way is by using the job manager. Steps 1 through 8 describe how to verify pvme using hostlist. Steps 1, 2 and 9 through 11 describe how to verify pvme using the job manager.

1. As a non-root user, create the standard execution directory for code **/u/johndoe/pvm3/bin/RS6K** and copy source files from **/usr/lpp/pvme/sample** to the execution directory.
2. Log into a node which has a switch adapter card in it and run **make** to compile the sample code. (You must be on a node to have links established to the MPCI library).

3. Create a hostlist file in the execution directory that lists all hosts that will be used for verification using hostlist technique.

```
aquila-1:/u/johndoe/pvm3/bin/RS6K>cat hostlist
aquila-7.lrsm.upenn.edu
aquila-5.lrsm.upenn.edu
aquila-3.lrsm.upenn.edu
aquila-1.lrsm.upenn.edu
```

4. Start up the **pvmd3e** daemon and the **pvme** console using the **hostlist**.

```
aquila-1:/u/johndoe/pvm3/bin/RS6K>pvme hostlist
PVMD: assuming default control workstation name
PVMD: trying to get 4 nodes
PVMD: Using aquila-7 with dx=/usr/lpp/pvme/bin/pvmd3e, ep=/u/johndoe/
      pvm3/bin/RS6K
PVMD: Using aquila-5 with dx=/usr/lpp/pvme/bin/pvmd3e, ep=/u/johndoe/
      pvm3/bin/RS6K
PVMD: Using aquila-3 with dx=/usr/lpp/pvme/bin/pvmd3e, ep=/u/johndoe/
      pvm3/bin/RS6K
PVMD: Using aquila-1 with dx=/usr/lpp/pvme/bin/pvmd3e, ep=/u/johndoe/
      pvm3/bin/RS6K
WARNING: localhost(aquila-1.lrsm.upenn.edu) not in the list...
PVMD: Ready for 4 hosts...
```

5. Verify the configuration while in the console then quit the console. The **pvmd3e** daemon will continue to run in the background.

```
pvme> conf
4 hosts, 1 data format
                    HOST     DTID     ARCH   SPEED
                aquila-7  fff0001     RS6K    1000
                aquila-5  fff0002     RS6K    1000
                aquila-3  fff0003     RS6K    1000
                aquila-1  fff0004     RS6K    1000
pvme> ps -a
                    HOST        A.OUT      TID     PTID FLAG
pvme> quit

pvmd still running.

aquila-1:/u/johndoe/pvm3/bin/RS6K>ps -aef|grep pvm
 johndoe 15341 1 0 09:32:45  pts/1 0:00 /usr/lpp/pvme/bin/pvmd3e/hostlist
```

6. Run the **hostbenc** program answering the prompts.

```
aquila-1:/u/johndoe/pvm3/bin/RS6K>hostbenc
 HOW MANY WORDS (MAXIMUM IS 2000000)?
1000
 HOW MANY TIMES (MAX IS 2000) ?
100
 MODE (IF DIRECT ENTER 3)?
1
task 10010001 on aquila-7:  268697600 1000 100 1
 TIME:   15903.8305282592773  MICROSECONDS
 BANDWIDTH  0.251511733779697377  MB/s
 BANDWIDTH PUT 62.9539060324262749  MB/s
 BANDWIDTH SND 25.0968077152454754  MB/s
 TIME:   422.179698944091797  MICROSECONDS
 .......
 .......  MANY LINES OF SIMILAR OUTPUT
 .......
 TIME:   432.431697845458984  MICROSECONDS
 BANDWIDTH  9.30903923298188474  MB/s
 BANDWIDTH PUT 239.674513680594316  MB/s
 BANDWIDTH SND 34.1347221925566728  MB/s
aquila-1:/u/johndoe/pvm3/bin/RS6K>New epoch 1
```

7. Restart the IBM PVMe for AIX console and stop the **pvmd3e** daemon.

```
aquila-1:/u/johndoe/pvm3/bin/RS6K>pvme
pvmd already running
pvm> halt
pvmd exiting
aquila-1:/u/johndoe/pvm3/bin/RS6K>ps -aef|grep pvm
```

8. Start **pvme** using the job manager (jm_status -P will tell you that the job manager pools have been set up and the job manager has been started.)

```
aquila-1:/u/johndoe/pvm3/bin/RS6K>pvme 4
PVMD: assuming default control workstation name
PVMD: trying to get 4 nodes
PVMD: using aquila-9 with dx=/usr/lpp/pvme/bin/pvmd3e, ep=/u/johndoe/
      pvm3/bin/RS6K
PVMD: using aquila-11 with dx=/usr/lpp/pvme/bin/pvmd3e, ep=/u/johndoe/
      pvm3/bin/RS6K
PVMD: using aquila-13 with dx=/usr/lpp/pvme/bin/pvmd3e, ep=/u/johndoe/
      pvm3/bin/RS6K
PVMD: using aquila-15 with dx=/usr/lpp/pvme/bin/pvmd3e, ep=/u/johndoe/
      pvm3/bin/RS6K
PVMD: Ready for 4 hosts...
pvme>
pvme> conf
4 hosts, 1 data format
                     HOST     DTID     ARCH   SPEED
                 aquila-9  fff0001     RS6K    1000
                aquila-11  fff0002     RS6K    1000
                aquila-13  fff0003     RS6K    1000
                aquila-15  fff0004     RS6K    1000
pvme> quit

pvmd still running.
```

9. Login to one of the nodes allocated to pvme by the job manager and run the program

```
aquila-1:/u/johndoe/pvm3/bin/RS6K>rsh aquila-9
*******************************************************************************
*
*  Welcome to AIX Version 3.2!
*
*  Please see the README file in /usr/lpp/bos for information pertinent
*  to this release of the AIX Operating System.
*
*******************************************************************************
aquila-9:/u/johndoe>cd pvm3/bin/RS6K
aquila-9:/u/johndoe/pvm3/bin/RS6K>hostbenc
 HOW MANY WORDS (MAXIMUM IS 2000000)?
1000
 HOW MANY TIMES (MAX IS 2000) ?
100
 MODE (IF DIRECT ENTER 3)?
1
task 10020002 on aquila-11:  268500993 1000 100 1
 TIME:   6680.96542358398438  MICROSECONDS
 BANDWIDTH  0.598715864593590830  MB/s
 BANDWIDTH PUT 63.5500604456121323  MB/s
 BANDWIDTH SND 26.2143999338150024  MB/s
......
...... MANY LINES OF SIMILAR OUTPUT
......
 TIME:   310.420989990234375  MICROSECONDS
 BANDWIDTH  12.8857265419674345  MB/s
 BANDWIDTH PUT 244.922860695497832  MB/s
 BANDWIDTH SND 34.4148019643930283  MB/s
aquila-9:/u/johndoe/pvm3/bin/RS6K>New epoch 1
```

10. Return and kill the daemon via IBM PVMe for AIX console

```
aquila-9:/u/johndoe/pvm3/bin/RS6K>exit
Connection closed.
aquila-1:/u/johndoe/pvm3/bin/RS6K>pvme
pvmd already running
pvme> halt
pvmd exiting
aquila-1:/u/johndoe/pvm3/bin/RS6K>
```

Chapter 5. Using the IBM PVMe for AIX Program Product

This chapter is organized according to the kinds of tasks the IBM PVMe for AIX routines perform and provides the following information:

- An overview of the IBM PVMe for AIX features and of the routines that are callable by user applications. Its purpose is to explain how the IBM PVMe for AIX software can be used to write parallel programs, and to provide the kind of advice that allows even relatively inexperienced programmers to write parallel programs successfully.
- Coding examples have been included in both FORTRAN and C. However, for a detailed description of each routine, see Chapter 7, "IBM PVMe for AIX Reference: C and FORTRAN Subroutines" on page 93.
- Commands and instructions needed to compile, link and run a parallel program.
- Some hints to ease the difficult task of debugging a parallel program.

Understanding IBM PVMe for AIX Features

The IBM PVMe for AIX program product is an extremely flexible tool that imposes neither a particular programming model nor dependencies or coupling between the running processes. Although best suited for the master-worker programming model, it may be used for the SPMD programming model as well. Either of the following may be implemented:

- *Data parallelism*, where all the programs perform the same computation on different data sets.
- *Functional parallelism*, where processes are almost independent of each other and specialized to particular tasks.

The IBM PVMe for AIX program product includes the following features:

- Two user libraries for FORTRAN and C programs, that contain all the basic routines for:
  - Managing processes
  - Managing hosts
  - Managing data communication
  - Synchronizing processes
  - Requesting environment and process information
  - Managing error conditions
  - Managing groups and performing global operations
  - Signaling

  Note: All C library routines are prefixed by *pvm_*, while all FORTRAN routines are prefixed by *pvmf*. For the most part, this document refers to routines in the C form.
- The IBM PVMe for AIX daemon is also included to support running programs. The IBM PVMe for AIX daemon looks like a supervisor and a coordinator for your IBM PVMe for AIX processes. It is also able to start or terminate processes, to change the virtual machine configuration, and to fulfill requests coming from processes. It keeps a database that contains all the important information about the processes. For example:

© Copyright IBM Corp. 1993, 1994, 1995

  - The hosts where they are running
  - The synchronization events
  - The groups configuration
- User parallel programs contain calls to the IBM PVMe for AIX libraries, so that they can contact the daemon to submit requests and communicate with other IBM PVMe for AIX processes. The programmer must provide all coordination among tasks and explicitly invoke message passing primitives whenever a node needs to access a variable that was previously updated by another node.

Note: In the following sections, the term *IBM PVMe for AIX process* is used often. An *IBM PVMe for AIX process* behaves just like any other UNIX process, but it has been successfully registered to the IBM PVMe for AIX daemon and can take IBM PVMe for AIX-related actions.

Managing Tasks

The primitives for task management allow you to:

- Enroll a UNIX process to IBM PVMe for AIX
- Unenroll a process from IBM PVMe for AIX
- Start one or more IBM PVMe for AIX processes
- Kill an IBM PVMe for AIX process
- Get information about IBM PVMe for AIX processes

Primitives for Managing Tasks

The following table lists the IBM PVMe for AIX task management routines for both C and FORTRAN:

Table 1. IBM PVMe for AIX Task Management Routines

C	FORTRAN
int rc = pvm_catchout (FILE *filed)	call pvmfcatchout (*onoff*, *rc*, filename)
int val = pvm_getopt (*int opt*)	call pvmfgetopt (*opt*, *val*)
int rc = pvm_exit ()	call pvmfexit (*rc*)
int rc = pvm_kill (*int tid*)	call pvmfkill (*tid*, *rc*)
int tid = pvm_mytid ()	call pvmfmytid (*tid*)
int tid = pvm_parent ()	call pvmfparent (tid)
int oldval = pvm_setopt (*int opt*, *int val*)	call pvmfsetopt (*opt*, *val*, *oldval*)
int nt = pvm_spawn (*char *task*, *char **argv*, *int flag*, *char *where*, *int ntask*, *int *tids*)	call pvmfspawn (*task*, *flag*, *where*, *ntask*, *tids*, *nt*)
int status = pvm_pstat (*int tid*)	call pvmfpstat (*tid*, *status*)
int rc = pvm_tasks (*int where*, *int *ntasks*, *struct pvmtaskinfo **taskp*)	call pvmftasks (*where*, *ntasks*, *tid*, *ptid*, *htid*, *flag*, *name*, *rc*)

Examples for Managing Tasks

FORTRAN Example

The following example shows how the master-worker model can be implemented. The master program must start copies of a different executable, the worker program.

```
      program master

      include 'fpvm3.h'
      integer MAXP
      parameter (MAXP=31)
      integer nprocs, procs(MAXP)
      integer rc, i

c Enroll to PVMe
c
      call pvmfmytid (mytid)

c Spawn "nprocs" copies of program "node"
c
      write (6,*) 'Enter number of node processes'
      read  (5,*) nprocs
      call pvmfspawn ('node', PVMDEFAULT, '*', nprocs, procs, rc)

      if (rc.ne.nprocs) then
c   Something failed
           if (rc.lt.0) then
              call pvmfperror ('Master, startup phase', rc)
           else
              write (6,*) 'Only ', rc, ' processes have been started'
              do i=1,nprocs
                 if (procs(i).lt.0) then
                    write (6,*) 'Error ', procs(i),
     $                    ' while starting process n. ', i
                 endif
              enddo
            endif
      endif

c Get complete information about processes. Each time we
c invoke pvmftasks, we get information about one process,
c cyclically.
c
      call pvmftasks (0, nprocs, tid(i), ptid(i),
     $                dtid(i), flg(i), aouts(i), rc)
      write (6,*), 'TID     PTID      DTID     FLAG      NAME'
      write (6,*), tid(i), ptid(i), dtid(i), flg(i), aouts(i)
      do i=2, nprocs
        write (6,*), tid(i), ptid(i), dtid(i), flg(i), aouts(i)
        call pvmftasks (0, nprocs, tid(i), ptid(i),
     $                  dtid(i), flg(i), aouts(i), rc)
      enddo

c  Send data to node programs, then gather final results.
c
   ..........
   ..........
c
c  Terminate all children processes before exiting

      do i=1,nprocs
```

```
        call pvmfkill(procs(i),rc)
      enddo
      call pvmfexit(rc)

      stop
      end
```

C Example

The following example shows a SPMD parallel program. In the SPMD programming model, the same program runs on all the nodes. However, the process started manually by you or by a script file starts all the other processes.

```
#include <stdio.h>
#include "pvm3.h"

main(argc, argv)
int argc;
char **argv;
{

  int tasks[32], ntasks;
  int parent,me;
  int flag,i,rc;
  char where[32];

  me=pvm_mytid();
  if ((parent = pvm_parent())<0) {
     if (parent == PvmNoParent) {

       /* I'm the first, then I start the children */

       printf("Enter n. of new processes:\n");
       scanf("%d",&ntasks);

       printf("Enter flag:\n");
       scanf("%d",&flag);
       if ((flag & PvmTaskHost) || (flag & PvmTaskArch)) {
          printf("Enter where:\n",i);
          scanf("%s",where);
       }

       /* spawn "ntasks" copies of the same executable */
       /* supplying the same arguments I received from the user */

       rc=pvm_spawn(argv[0],&argv[1],flag,where,ntasks, tasks);

       if (rc<0)  pvm_perror("Node 0: spawning processes");
       else if (rc<ntasks) {
          printf ("only %d processes have been started\n",rc);
          for (i=0;i<ntasks;i++) {
             /* If something failed, print an error message;  */
             /* otherwise terminate the successfully spawned  */
             /* tasks to clean the environment before exiting */
             if (tasks[i]<0)
               printf("error %d starting process %d\n",tasks[i],i);
             else
               pvm_kill(tasks[i]);
          }
          pvm_exit(-1);
       }

       pvm_initsend(0);
```

```
        pvm_pkint(&ntasks,1,1);
        pvm_pkint(tasks,ntasks,1);

        /* Send initial data to all the other processes */

        if (rc =pvm_mcast(tasks, ntasks, 777) <0)
           pvm_perror ("Error in first mcast\n");

      } else
           /* If pvm_parent returns a negative number not equal */
           /* to PvmNoParent, it means that an error  occurred  */
           printf ("Startup Error\n");

    } else {
       /* If pvm_parent returns a positive number, it means */
       /* that I'm not the first process; I receive initial */
       /* data from node 0.                                 */

       pvm_recv (parent,777);
       pvm_upkint(&ntasks,1,1);
       pvm_upkint(tasks,ntasks,1);
    }

    /*  Program body. Each node executes the    */
    /*  same operations on different data sets

         ..............
         ..............                         */


    /* Before exiting, each process leaves PVM3 */

    pvm_exit();
    exit(0);

}
```

Enrolling a Process

Before a user program calls any other IBM PVMe for AIX routine (with the exception of the **pvm_start_pvmd** routine) it must enroll with the IBM PVMe for AIX daemon. This task is accomplished by calling the **pvm_mytid** routine. It returns the *task identifier* of the process within the IBM PVMe for AIX environment.

The *task identifier* is a positive integer assigned by the daemon, and is unique for each process. All IBM PVMe for AIX processes have an identifier (called *tid*), in order to be properly addressed by the daemon and by the other IBM PVMe for AIX processes.

The IBM PVMe for AIX daemon maintains a complete database of the active processes so that at any time it knows:

- How many processes are running
- Their identifiers
- The hosts where they are running
- The host addresses
- How to communicate with each process

When a process calls the **pvm_mytid** routine on its first call to the IBM PVMe for AIX library (apart from the **pvm_start_pvmd** routine), the daemon updates its

database and adds a new entry for the process and sets its status to active (**TRUE**).

Note: The **pvm_mytid** routine can be invoked multiple times by the same process. In all subsequent calls it returns the *tid* of the calling process. The status of a task may be checked with the **pvm_pstat** routine.

Unenrolling a Process

When a process no longer needs the IBM PVMe for AIX services, or is ready to exit, it should detach the IBM PVMe for AIX program product by calling the **pvm_exit** routine. It continues to exist as a UNIX process, but is no longer active inside of the IBM PVMe for AIX program product. The daemon changes the process status to inactive.

When all processes have left the IBM PVMe for AIX program product without errors, the IBM PVMe for AIX database is reset and the daemon loses memory of the previous parallel job executions. In IBM PVMe for AIX terms, a new *epoch* is started. Normally any process which has done **pvm_exit** but is still running, will be killed at the end of an epoch. To allow a process to live beyond the epoch which spawned it, the session must be started with the **-nokill** option. If any processes exit without calling a **pvm_exit** routine, or after an error occurs, a new epoch may be forced by typing **reset** at the IBM PVMe for AIX console during an interactive session.

Processes may enroll and unenroll the IBM PVMe for AIX program product multiple times during a single run, but they should expect to receive a new *tid* at any time within the same epoch.

Note: If IBM PVMe for AIX tasks do not synchronize themselves before making a call to **pvm_exit**, a situation may arise where the application hangs. To avoid this, it is suggested that you precede the **pvm_exit** call with **pvm_barrier**. Note that in order to use **pvm_barrier**, all the tasks must enroll in the same group via the **pvm_joingroup** routine. To verify if the group is complete, **pvm_barrier** needs to know the number of tasks. This can be broadcast with **pvm_mcast**.

Starting a Process

An IBM PVMe for AIX process may be started in the following ways:

- By you, either from the command line (interactive session) or by a script file (batch session). See “Running an IBM PVMe for AIX Session” on page 21 for more information.
- By another IBM PVMe for AIX process.

The IBM PVMe for AIX program product provides the **pvm_spawn** routine for starting new processes. You must supply the *filename* of the executable and the number of new instances to be started. In addition to these two parameters, the routine allows you to specify one or more flags indicating:

- Whether the new instances will be started under the debugger (**dbx**) or not. If this is the case, two *shell* scripts called **debugger** and **debugger2** are started. See “Customizing the debugger and debugger2 Shell Scripts” on page 19 for more information.

- Whether you require the new processes to be started on a particular host or on a host with a particular architecture. If you require a particular host, the proper *hostname* must be supplied. Requesting a host with a particular architecture has been retained only for compatibility with the freeware PVM. In fact, IBM PVMe for AIX supports only one architecture, RS6K. RS6K is the name of the architecture RISC System/6000 within the IBM PVMe for AIX program product, and it is the string to be used when the *architecture* parameter must be supplied to an IBM PVMe for AIX routine.

When a process calls the **pvm_spawn** routine, a request is sent to the IBM PVMe for AIX daemon. If a *hostname* is not specified, the daemon chooses a host for the new instance. Then, it runs an **rcmd** (remote command) on the remote host and creates a new entry in the processes database. At this point, the process status is still set to **FALSE**, and will be set to **TRUE** only when the daemon receives all the needed data about the new process, that is, when the new process enrolls with the IBM PVMe for AIX environment.

Searching for Executable Files

The IBM PVMe for AIX daemon searches the **$HOME/pvm3/bin/RS6K** directory by default, for the executable files or for the script files, unless you specified an alternate directory at daemon startup time or with **PVMEPATH**. However, the executable files must already exist on the remote host, either physically or by a network file system.

The C routine also allows you to pass arguments to the new processes. To do this, you must supply an array of pointers to the argument strings.

If no errors occurred, the **pvm_spawn** routine returns:

1. The number of processes that have actually been started.
2. The array supplied as the last argument will be filled with the *tids* of the new processes.

Note: The number of new processes that have been started should be equal to the number of new instances required by you.

If some tasks could not be started, the corresponding error conditions are placed in the proper position within the array above. In this case, the number of processes that have been started would be less than the number of new instances required by you.

Terminating a Process

The **pvm_kill** routine kills an IBM PVMe for AIX process. The process *tid* must be supplied. This routine should not be used by a process to kill itself. Instead, the sequence **pvm_exit** and **exit** (or a **STOP**, in FORTRAN) should be used.

Typically the **pvm_kill** routine is called by the master program (when the master-worker model has been adopted) before leaving, to kill all worker processes.

Managing Options

Two auxiliary routines, **pvm_setopt** and **pvm_getopt**, allow you to set and query a number of options for an IBM PVMe for AIX process. These options can be useful for changing the debugging level, selectively monitoring calls to the IBM PVMe for AIX routines, deciding which task is responsible for collecting output and error lines or trace data from other tasks, enabling automatic printing of error messages, etc. A detailed list of the supported options and their meanings is provided in "pvm_getopt" on page 122.

Querying for Information about Tasks

You can ask the system for information about the processes that are running at any time. The **pvm_tasks** routine allows you to:

- Access the daemon's processes database
- Obtain the number of running processes and a set of data including:
  - Process name
  - Task identifier
  - Host identifier
  - Parent identifier
  - Status

This is data for either all processes running in the virtual machine, or all processes running on a given host, or a single process.

Task information is made available by the C routine as an array of structures whose definition can be found in **pvm3.h**. The FORTRAN routine returns data at each call for a single process, by cycling through all tasks.

The **pvm_pstat** routine is a simpler routine that returns the status of a given process. The **pvm_pstat** routine is useful because the IBM PVMe for AIX daemon does not inform the other tasks when a task leaves or exits.

The **pvm_parent** routine is a utility that allows you to obtain the identifier of the parent process; that is, the identifier of the process that started the calling one using a **pvm_spawn** routine. If the process was not started through that mechanism, a special error code (**PvmNoParent**) is returned. This routine can also be used to determine if a particular process was started directly by you or by another IBM PVMe for AIX process. A process started with the console **spawn** command will have a parent identifier.

Managing I/O

Like any other UNIX process, an IBM PVMe for AIX process may access storage devices for read/write operations. Note, however, that processes started by the IBM PVMe for AIX daemon search the **ep** directory (or **$PVMEPATH** or **$HOME/pvm3/bin/RS6K**) for data files. This means that in order to ensure that calls like **open**, **read**, **write**, and **close** complete successfully, you must either locate data files in your **ep** directory (or **$PVMEPATH**, or **$HOME/pvm3/bin/RS6K**) or specify the full pathname of the data files.

Notes:

1. Only processes started manually by you in an interactive session are able to receive data from **stdin**, while processes started by the IBM PVMe for AIX daemon cannot.
2. **stdout** and **stderr** (for processes started by the IBM PVMe for AIX program product), are redirected, by default, to **stdout** of the daemon, so that they appear on the screen in the same window where you run the daemon. However, if you decide to spawn processes from the IBM PVMe for AIX console, you have an option to redirect the **stdout** of your processes on an alternative device or file. It can be useful in case you need to keep the **stdout** of your jobs to be examined later.

   Note: Type **help spawn** from the console prompt to get more information on spawning processes.

   If an IBM PVMe for AIX process calls the **pvm_catchout** routine before spawning child processes, each output and error line from the children will be sent, in the form of messages, with a special message type, to the parent process. The parent process will put the content of those messages into a file or discard them if the specified file is */dev/null* (you need to specify a file descriptor when invoking the routine). When running over MPCI, you must set the **PVMe_IP** environment variable to *yes* before starting the daemon if **pvm_catchout** is to be used.

Managing Hosts

The primitives for task management allow you to:

- Start/stop the IBM PVMe for AIX virtual machine
- Dynamically change the virtual machine configuration
- Query information about the virtual machine configuration

Primitives for Managing Hosts

The following table shows the IBM PVMe for AIX host management routines:

Table 2. IBM PVMe for AIX Host Management Routines	
C	**FORTRAN**
int code = pvm_archcode (*char *arch*)	call pvmfarchcode (*arch*, *code*)
int rc = pvm_config (*int *nhost*, *int *narch*, *struct pvmhostinfo **hosts*)	call pvmfconfig (*nhost*, *narch*, *tid*, *name*, *arch*, *mtu*, *speed*, *rc*)
int rc = pvm_halt ()	call pvmfhalt (*rc*)
int host = pvm_hostsync (*char *host*, *struct *clock*, *struct *delta*)	call pvmfhostsync (*host*, *clksec*, *clkusec*, *deltasec*, *deltausec*)
int status = pvm_mstat (*char *host*)	call pvmfmstat (*host*, *status*)
int dtid = pvm_start_pvmd (*int args*, *char **argvs*, *int block*)	call pvmfstartpvmd (*block*, *rc*)
int dtid = pvm_tidtohost (*int tid*)	call pvmftidtohost (*tid*, *dtid*)
Note: The **pvm_addhosts** and **pvm_delhosts** subroutines are currently disabled.	

Starting and Stopping the Virtual Machine

Your programs are allowed to start up the IBM PVMe for AIX environment, that is, to start the daemon on a set of hosts, by calling the **pvm_start_pvmd** routine. Though not strictly required, but recommended, the IBM PVMe for AIX environment should be up and running before your applications are started.

The first two parameters contain the number of arguments and an array of the arguments that will be passed to the daemon. The possible arguments to the daemon are discussed in detail in Chapter 4, “Product Startup” on page 21. If no arguments are supplied, the daemon is started on the local workstation, and the virtual machine is composed of a single host. If the first argument in the arguments list is the configuration file (see Chapter 4, “Product Startup” on page 21), then the IBM PVMe for AIX daemon is started on the set of hosts listed there. The last parameter indicates whether the routine must wait until startup completes.

The FORTRAN subroutine **pvmfstartpvmd** receives only the last parameter, and is not able to pass arguments to the daemon.

The virtual machine can be completely stopped by entering the **pvm_halt** routine. Once **pvm_halt** has been invoked, it kills your programs running on all the hosts of the virtual machine and the daemon.

Querying Information about the Virtual Machine

Information about the virtual machine and the status of the involved hosts is available at any time to your applications.

The **pvm_config** routine returns the following information:

- The number of hosts
- The number of architectures

  Note: This data is redundant, because IBM PVMe for AIX runs in an homogeneous environment, but it is compliant with the freeware PVM.

- The host name
- The host identifier
- The host architecture
- The relative CPU speed

  Note: The last data is the same for all the hosts, because IBM PVMe for AIX runs in a homogeneous environment.

The FORTRAN routine returns data for a single host at each call, and cycles among all the hosts.

To obtain the status of a particular host, the **pvm_mstat** routine can be invoked. It can be used to check whether a given host has already been added to the virtual machine or whether a given host is still properly working or has failed.

The **pvm_tidtohost** routine is an auxiliary routine that returns the identifier of the host where a given process is running. It simply masks the supplied task identifier, but does not check whether the host or the task exist.

Examples for Managing Hosts

FORTRAN Example

The following example is extracted from the host program of a *master-worker* fashioned application. The master program starts both the IBM PVMe for AIX environment on a set of hosts and the worker programs.

```
      program master

      include 'fpvm3.h'
      integer MAXP
      parameter (MAXP=31)
      integer nprocs, procs(MAXP+1)
      integer availproc, info, rc, i, dtid
      character*32 hosts(MAXP)

      open (15, file="/tmp/hosts.names", form="formatted")
      read (15,*) availproc
      do i=1, availproc
         read (15, '(A32)') hosts(i)
      enddo

      write (6,*) 'Enter number of nodes [1-31]'
  10  read  (5,*) nprocs
      if (nprocs.gt.availproc) then
          write (6,*) 'Value too large: try again'
          go to 10
      endif

c Start PVMe daemon on the local node
c
      call pvmfstartpvmd (1,info)
      if (info.ne.0) then
        write (6,*) 'Unable to start the daemon'
        stop
      endif

c Enroll to PVMe
c
      call pvmfmytid (procs(1))
```

```
c Add "nprocs" hosts to the virtual machine, so that I'll be able to run
c <nprocs> nodes.
c
c     call pvmftidtohost(mytid,dtid)
      j=1
      i=0
      do while (i.lt.nprocs)
         call pvmfaddhost(hosts(j), rc)
         if (rc.lt.-256) then
           i=i+1
         endif
         j=j+1
      enddo

c Start "nprocs" instances of program "node".
c
      call pvmfspawn ('node', PVMDEFAULT, '*', nprocs, procs(2), rc)

      if (rc.ne.nprocs) then
         if (rc.ge.0) then
         write (6,*) 'Only ', rc, ' processes have been started'
         do i=1,nprocs
           if (procs(i).lt.0) then
              write (6,*) 'Error ', procs(i),
     $             ' while starting process n. ', i
            endif
         enddo
        else
           call pvmfperror ('Startup phase',rc)
           call pvmfexit(rc)
        endif
      endif

c
c     .... main computation
c
c

      do i=1,nprocs
          call pvmfkill(procs(i),rc)
      enddo

      print *, ' End of program '
      print *, 'PVMe daemon will be killed'
      call pvmfhalt(rc)
      stop
      end
```

C Example

```
#include <stdio.h>
#include "pvm3.h"

#define HOST "pvmhost"
#define NODE "pvmnode"
#define MAXHOST 16
char *allhosts = { "spn01",
                   "spn02",
                    .......
                   "spn15",
                   "spn16" };

main(argc, argv)
int argc;
```

```
char **argv;
{

  int tasks[MAXHOST], ntasks;
  int nhosts, narchs;
  int me, i, rc;
  struct pvmhostinfo *hostl;

  me=pvm_mytid();

  /* Get information about the available hosts  */

  pvm_config (&nhosts, &narchs, &hostl);
  printf ("%d hosts are currently running\n",nhosts);
  printf ("  NAME          ARCH          HTID      \n");

  for (i=0; i<nhosts; i++)  {
    printf ("%s  %s      %d   \n", hostl[i].hi_name,
                                   hostl[i].hi_arch, hostl[i].hi_tid);
  }

  ntasks=nhosts-1;

  /* Spawn as many copies of node programs as     */
  /* many nodes are available. One node */
  /* (where I am running) is already busy         */

  rc=pvm_spawn(NODE, NULL, PvmTaskDebug, NULL, ntasks, tasks);
  if (rc<0)
     pvm_perror("Master: error starting nodes");
  else if (rc<ntasks) {
     ..........
     ..........
  } else  {
    printf ("Startup of nodes was successful\n");
  }

  /* Send initial data to node programs */

  pvm_initsend(0);
  pvm_pkint(&ntasks,1,1);
  pvm_pkint(tasks,ntasks,1);

  if (rc =pvm_mcast(tasks, ntasks, 777) <0)
      pvm_perror ("Error in first mcast\n");


  /* Send more data to node programs, and wait for results */
  /*
     .........                                              */


   /* Terminate all nodes before leaving  */

   for (i=0; i<ntasks; i++) {
      pvm_kill(tasks[i]);
   }
   pvm_exit();
   exit(0);

}
```

Understanding Data Communication

Within the IBM PVMe for AIX program product, a task can communicate with any other task. There are certain limits to the message size, but there is no limit to the number of messages that can be exchanged. The IBM PVMe for AIX program product assumes that each host is able to communicate with all the other hosts using TCP/IP and by using the low-level communication protocol on the High Performance Switch.

Primitives for Data Communication

The primitives for data exchange allow you to:

- Prepare a message
- Send a message to one or more IBM PVMe for AIX processes
- Probe for message arrival
- Receive a message
- Query for information about the last message received

Follow these steps to send a message:

1. Initialize the sending buffer
2. Pack the data
3. Send the message

Follow these steps to receive a message:

1. Receive the message
2. Unpack the data

The following table shows the IBM PVMe for AIX data communication routines:

Table 3 (Page 1 of 2). IBM PVMe for AIX Data Communication Routines

C	FORTRAN
int rc = pvm_bufinfo (*int bufid*, *int *len*, *int *msgtype*, *int* tid*)	call pvmfbufinfo (*bufid*, *len*, *msgtype*, *tid*, *rc*)
int rc = pvm_freebuf (*int bufid*)	call pvmffreebuf (*bufid*, *rc*)
int bufid = pvm_getrbuf ()	call pvmfgetrbuf (*bufid*)
int bufid = pvm_getsbuf ()	call pvmfgetsbuf (*bufid*)
int bufid = pvm_initsend (*int encoding*)	call pvmfinitsend (*encoding*, *bufid*)
int rc = pvm_mcast (*int *tids*, *int ntids*, *int msgtype*)	call pvmfmcast (*ntids*, *tids*, *msgtype*, *rc*)
int bufid = pvm_mkbuf (*int encoding*)	call pvmfmkbuf (*encoding*, *bufid*)
int bufid = pvm_nrecv (*int tid*, *int msgtype*)	call pvmfnrecv (*tid*, *msgtype*, *bufid*)
int rc = pvm_packf (*const char *fmt*, ...)	

Table 3 (Page 2 of 2). IBM PVMe for AIX Data Communication Routines

C	FORTRAN
int rc = pvm_pkbyte (*char *frombyte*, *int nitem*, *int stride*) int rc = pvm_pkcplx (*float *fromcplx*, *int nitem*, *int stride*) int rc = pvm_pkdcplx (*double *fromdcplx*, *int nitem*, *int stride*) int rc = pvm_pkdouble (*double *fromdouble*, *int nitem*, *int stride*) int rc = pvm_pkfloat (*float *fromfloat*, *int nitem*, *int stride*) int rc = pvm_pkint (*int *fromint*, *int nitem*, *int stride*) int rc = pvm_pklong (*long *fromlong*, *int nitem*, *int stride*) int rc = pvm_pkshort (*short *fromshort*, *int nitem*, *int stride*) int rc = pvm_pkstr (*char *fromstring*)	call pvmfpack (*what*, *from*, *nitem*, *stride*, *rc*)
int rc = pvm_psend (*int tid*, *int msgtype*, *char *buf*, *int len*, *int datatype*)	call pvmfpsend (*tid*, *msgtype*, *buf*, *len*, *datatype*, *rc*)
int bufid = pvm_probe (*int tid*, *int msgtype*)	call pvmfprobe (*tid*, *msgtype*, *bufid*)
int rc = pvm_precv (*int tid*, *int msgtype*, *char *buf*, *int len*, *int datatype*, *int atid*,*int amsgtype, int alen*)	call pvmfprecv (*tid*, *msgtype*, *buf len*, *datatype*, *atid*, *amsgtype*, *alen*, *rc*)
int bufid = pvm_recv (*int tid*, *int msgtype*)	call pvmfrecv (*tid*, *msgtype*, *bufid*)
int rc = pvm_send (*int tid*, *int msgtype*)	call pvmfsend (*tid*, *msgtype*, *rc*)
int oldbuf = pvm_setrbuf (*int bufid*)	call pvmfsetrbuf (*bufid*, *oldbuf*)
int oldbuf = pvm_setsbuf (*int bufid*)	call pvmfsetsbuf (*bufid*, *oldbuf*)
int bufid = pvm_trecv (*int tid*, *int msgtype*, *struct timeval *timeout*)	call pvmftrecv (*tid*, *msgtype*, *sec*, *usec*, *bufid*)
int rc = pvm_unpackf (*const char *fmt*, ...)	
int rc = pvm_upkbyte (*char *tobyte*, *int nitem*, *int stride*) int rc = pvm_upkcplx (*float *tocplx*, *int nitem*, *int stride*) int rc = pvm_upkdcplx (*double *todcplx*, *int nitem*, *int stride*) int rc = pvm_upkdouble (*double *todouble*, *int nitem*, *int stride*) int rc = pvm_upkfloat (*float *tofloat*, *int nitem*, *int stride*) int rc = pvm_upkint (*int *toint*, *int nitem*, *int stride*) int rc = pvm_upklong (*long *tolong*, *int nitem*, *int stride*) int rc = pvm_upkshort (*short *toshort*, *int nitem*, *int stride*) int rc = pvm_upkstr (*char *tostring*)	call pvmfunpack (*what*, *to*, *nitem*, *stride*, *rc*)

Note: Since the underlying communications software used by IBM PVMe for AIX uses the signals SIGIO, SIGPIPE, and SIGALRM, it is suggested that you avoid using these signals in your application programs. You should also keep this in mind when using a debugger. For example, you can ignore SIGIO, SIGPIPE, and SIGALRM signals when using **dbx** to debug an IBM PVMe for AIX program by using the following subcommands inside the debugger:

```
ignore io
ignore pipe
ignore alrm
```

You can also add these subcommands to the **.dbxinit** file.

Preparing a Message

The IBM PVMe for AIX program product uses a sending buffer to store data that must be sent to another process. Before sending a message, it must be packed into the sending buffer by the proper IBM PVMe for AIX routines. Multiple buffers can be managed within the same application, but only one of them is active at any time. However, most applications do not need to manage more than one send buffer. If you do not need to manage multiple buffers you are provided with transparent default send/receive buffers.

Unless the active buffer has just been created, you are required to initialize it with the **pvm_initsend** routine. This routine receives only one parameter (*encoding*), which can assume three different values. Two of them (**PvmDataRaw** and **PvmDataDefault**) were used in freeware PVM to enable/disable data conversion to/from the XDR format. Both of them are supported in the IBM PVMe for AIX program product. Because of the homogeneous environment, they actually produce the same effect. The last value (**PvmDataInPlace**) indicates that data must not be copied into the send buffer during the packing operation. Only the pointer to the data and the size will be written into the buffer, while data will be copied directly out of your memory only when a send is performed. This option allows you to avoid one data copy, with a substantial improvement in performance. A further advantage is that the same data can be sent multiple times, with modifications intervening between the sends, without having to pack it each time.

Using a Packing Routine

Once the active buffer has been cleared, the message to be sent must be composed by packing data within the buffer. C programmers are provided with a number of packing routines, such as: **pvm_pkbyte**, **pvm_pkint**, and **pvm_pkfloat**. There is one routine for each data type (such as, integer, character, and float). FORTRAN programmers are provided with a single routine called **pvmfpack**, where the first parameter indicates the data type to be packed. These routines copy data into the buffer, and a parameter (*stride*) can be used to indicate that noncontiguous data must be packed.

The **pvm_packf** routine, only available in C, allows you to pack data into the sending buffer following a **printf**-like syntax.

Multiple pack operations can be performed to compose a message. The result of each operation is the append of a new piece of data to the message. The buffer pointer is incremented accordingly.

Note: The maximum amount of data that can be copied with a single call to a **pvm_pk***xxx* routine is 32 megabytes, and the maximum length for a single message is 32 megabytes.

When more pack operations have been run to compose a single message, the actual message length can be greater than the sum of the messages' components. This is because of alignments and padding of strings.

Note: A **pvm_pkbyte** of a single byte implies that 4 bytes of the buffer will be used.

Sending a Message

Once you have finished copying data into the buffer, the message is ready to be sent using the **pvm_send** or the **pvm_mcast** routine. The **pvm_send** routine requires the task identifier of the recipient process as arguments and the message type that is used to identify the message. It can be used to send the message to a single process. The **pvm_mcast** routine is used to send a message to a set of processes, so that the number of recipient processes and an array containing their task identifiers must be supplied together with the message type. Both routines return a `0` if successful, or a negative number if unsuccessful.

When sending a message, the *type* parameter is recommended for identifying each message. It is not an error to use a particular message type more than once, although it is advisable to characterize each message with its own message type. The valid range for message types is 0.. 2147483519, (0x7fffff7f), inclusive.

When a send routine is invoked, the contents of the sending buffer are delivered to the recipient process. The sending buffer maintains data until a **pvm_initsend** routine is called, so that the same message can be sent to more than one process without running the pack operations again. Data can also be appended to a previously composed message. If a pack operation is performed after a **pvm_send** or a **pvm_mcast** routine (on the same buffer), without calling the **pvm_initsend** routine, the new data is appended to the previous message.

If you have to send homogeneous and contiguous data, you can use the **pvm_psend** routine, which packs data and sends messages within a single call. You must specify the:

- Starting address
- Number of items
- Type of data to be sent
- Target process identifier
- Message type.

Data will be packed into the current send buffer.

pvm_send, **pvm_psend** and **pvm_mcast** are blocking asynchronous routines; they return quickly, whether the recipient task performs a receive or not.

Broadcasting to all members of a given group is discussed in “Understanding the Broadcast Operation” on page 70.

Receiving a Message

There are four routines that can be used to receive messages:

- **pvm_recv** is a blocking routine, that blocks until the proper message arrives (it returns immediately if the proper message has already arrived).
- **pvm_nrecv** is a nonblocking routine, that returns immediately whether the proper message arrived or not. It can be called multiple times for the same message, while it allows you to perform some other useful work in the meantime.

- **pvm_trecv** is a blocking routine with a timeout: it blocks until the proper message arrives. However, it returns after a user-specified timeout, whether the message arrived or not. The return code either identifies the buffer where the message has been received, or, if the message was not received, contains zero.
- **pvm_precv** is a blocking routine, like **pvm_recv**, which blocks until the proper message arrives. It then unpacks the message content into a user-specified area. **pvm_precv** can be used when incoming data items are homogeneous and contiguous. When you use this routine, skip the unpacking step described below.

All of these routines let you specify one message type and one sender identifier. The type of message being received and the task identifier of the sender process must match. A Wildcard (-1) can be used either for one or both of the parameters. The buffer identifier where the message has been received is returned and data unpacking must take place when that buffer is active.

The receiving routines described above look through a queue of previously received messages searching for the appropriate message. You are guaranteed to receive messages according to their arrival sequence. Among all the messages matching the request, the one that arrived first is returned.

Unpacking Data

Once a message has been received, data must be unpacked using the **pvm_upk***xxx* routines. Data must be unpacked from the receiving buffer in exactly the same order it was packed into the sending buffer, or data corruption occurs. Alternatively, the **pvm_unpackf** routine can be used to unpack data from the receiving buffer following a **scanf**-like syntax, if data was packed using the **pvm_packf** routine by the sender.

Notes:

1. Data remains in the active receiving buffer until a new receiving routine is invoked.
2. Data is lost if a **pvm_recv** routine is called before data of the previous message has been unpacked and saved in the program data space.

Whenever a message is delivered to a given task, that task receives a signal and stores the incoming message in a memory buffer without processing it. The message content is only made available to the program when the task invokes a receive routine. It is possible that the space available for messages could be filled. This could happen if there was a collection of many messages on a single process, or if many very large messages were collected by a single process at one time. In that case, the failing process prints an error message. For more information, see Appendix C, “IBM PVMe for AIX Intrinsic Limits: Summary” on page 233.

After a receive routine has finished, the **pvm_bufinfo** routine allows you to obtain information about the message last received. That information could include the message length, message type, and the identifier of the sender process. That kind of information may be useful when an application receives messages, regardless of their message type or sender *tid* (for example, calling **pvm_recv(-1, -1)**), without an established order, because the impending action depends on the message type of the incoming message and on the *tid* of the sender process.

Checking the Arrival of a Message

The **pvm_probe** routine can be used to check to see if a given message has arrived. If it has arrived, it returns a valid buffer identifier but does not receive the message. If the message has not been received, an invalid buffer identifier (`0`) is returned.

Note: A **pvm_recv** or a **pvm_nrecv** routine must be invoked to receive the message.

This routine requires the same parameters as **pvm_recv** and it returns a buffer identifier. Although the message has not been received, that buffer contains information about the message, and can be used as an input parameter to call the **pvm_bufinfo** routine.

Managing Multiple Buffers

Multiple send and receive buffers can be managed within the IBM PVMe for AIX program product. Most parallel applications do not require this sophisticated feature, and you can simply use the default buffers.

New send buffers can be created using the **pvm_mkbuf** routine, which requires a single parameter (*encoding*) and returns the buffer identifier. When no more buffers are available a special code (**PvmNoMem**) is returned.

This routine receives only one parameter (*encoding*), which can assume three different values, two of them (**PvmDataRaw** and **PvmDataDefault**) were used in freeware PVM to enable/disable data conversion to/from the XDR format. Both of them are supported in the IBM PVMe for AIX program product and because of the homogeneous environment, they actually produce the same effect.

The last value (**PvmDataInPlace**) indicates that data must not be copied into the send buffer during the packing operation. Only the pointer to the data and the size are written into the buffer, while data is copied directly out of your memory only when a send is requested. This option allows you to avoid one data copy, with a substantial improvement in performance. Moreover, the same data can be sent multiple times, with modifications between the sends, without having to pack it each time.

When multiple send buffers are available you can switch between them. The **pvm_setsbuf** routine sets the supplied send buffer to active, and returns the identifier of the previous one. Any valid send buffer identifier returned by a **pvm_mkbuf**, **pvm_setsbuf** or **pvm_initsend** routine, can be supplied, as well as the identifier of a receive buffer. In this case, data is copied from the receive buffer to a new send buffer and is ready to be sent.

If the **pvm_setsbuf** routine receives `0` as an argument, the previous active buffer is saved, and no buffer becomes active. In this situation, another buffer must be explicitly set to active (by calling the same routine) or it will be automatically created with the next call to the **pvm_initsend** routine.

When a send buffer is no longer needed, it can be freed by calling the **pvm_freebuf** routine.

Similar arguments apply for the receive buffers:

- Multiple receive buffers can be managed, but only one of them is active at any time.
- You can switch between the receive buffers by invoking the **pvm_setrbuf** routine.
- The **pvm_setrbuf** routine can be called with an argument equal to `0` to save the previous active receive buffer. This can be done without installing a new buffer.
- The **pvm_freebuf** routine allows you to free buffers that were saved before and are no longer needed.
- At any time, the identifier of the active receive buffer (**pvm_getrbuf**) can be obtained.

Receive buffers cannot be created by a **pvm_mkbuf** routine. They are created automatically by the IBM PVMe for AIX program product package when a receive operation is attempted (and there is no active receive buffer). Receive buffers must be freed only if they were saved with a call to the **pvm_setrbuf** routine. However, send buffer identifiers cannot be supplied as arguments to a **pvm_setrbuf** routine.

Examples for Data Communication

FORTRAN Example

The following section shows the use of routines for data communication. Only the basic routines are used, and only the default send and receive buffers are exploited.

```
      program master

      include 'fpvm3.h'
      integer MAXP, MAXDATA
      parameter (MAXP=31, MAXDATA=8*1024)
      integer nprocs, procs(MAXP+1)
      integer info, rc, i
      integer len, type, who, rcvbuf
      integer ndata, start, end, block
      real*4 data(8192), outdata(8192), totdata(40)

      write (6,*) 'Enter number of nodes [1-31]'
      read  (5,*) nprocs
      write (6,*) 'Enter ndata [1-', MAXDATA, ']'
      read  (5,*) nprocs

c Enroll to PVMe
c
      call pvmfmytid (procs(1))

c Start "nprocs" instances of program "node".
c
      call pvmfspawn ('f3bis', PVMDEFAULT, '*', nprocs, procs(2), rc)
      write (6,*)  rc, ' processes have been successfully started'

      if (rc.ne.nprocs) then
        ....
        .....
      endif
```

```
c Send initial data to programs "node".  To send the same data
c to more processes the multicast routine is convenient.
c
      call pvmfinitsend(PVMDEFAULT, rc)
      call pvmfpack(INTEGER4, nprocs, 1, 1, rc)
      call pvmfpack(INTEGER4, procs, nprocs+1, 1, rc)
      call pvmfmcast (nprocs, procs(2), 10, rc)
      if (rc.lt.0) then
         call pvmfperror ('Error in multicast', rc)
      endif

      block=ndata/nprocs
      start=1
      end=block
      do i=2, nprocs+1
        call pvmfinitsend(PVMDEFAULT, rc)
        call pvmfpack(INTEGER4, i, 1, 1, rc)
        call pvmfpack(INTEGER4, start, 1, 1, rc)
        call pvmfpack(INTEGER4, end, 1, 1, rc)
        call pvmfpack(REAL4, data(start), end-start+1, 1, rc)
        call pvmfsend (procs(i), 20, rc)
        if (rc.lt.0) then
          call pvmfperror ('Error in send', rc)
        endif
        start=start+block
        end=end+block
        if (i.eq.nprocs) then
          end=ndata
        endif
      enddo
c
c Wait for results from node programs
c
c Receive any incoming message with message type 30, regardless
c of the sender tid
c
      do i=1,nprocs
        call pvmfrecv (-1, 30, rcvbuf)
        if (rcvbuf.gt.0) then
          call pvmfbufinfo (rcvbuf, len, type, who,rc)
          call pvmfunpack (INTEGER4, start, 1, 1, rc)
          call pvmfunpack (INTEGER4, end, 1, 1, rc)
          call pvmfunpack (REAL4, outdata(start), end-start+1, 1, rc)
        else
          call pvmfperror ('Error in recv', rc)
        endif
      enddo

c Receive ordered data from all node programs
c
      do i=2, nprocs+1
        if (rcvbuf.gt.0) then
          call pvmfrecv (procs(i), 40, rcvbuf)
          call pvmfunpack (REAL4, totdata(i), 1, 1, rc)
        else
          call pvmfperror ('Error in recv', rc)
        endif
      enddo

c Before leaving
      print *, ' End of program '
      do i=2,nprocs+1
         call pvmfkill(procs(i),rc)
      enddo
```

```
      call pvmfexit(rc)
      stop
      end
```

C Example

```
#include 'pvm3.h'
#define MAXDATA 16

main () {

int mytid, bufs[12];
int de, ds;
float array[MAXDATA];

if ((mytid = pvm_mytid())<0) {
  pvm_perror ("Main, error during enrollment");
  exit (-1);
}

  ............

/* pack initial data to be sent to all nodes */

bufs[0] = pvm_initsend (PvmDataDefault);
if (bufs[0] < 0)  {
   pvm_perror ("Error in initsend");
   goto errorlabel;
}
pvm_pkint (&nproc, 1, 1);
pvm_pkint (allproc, nproc, 1);
pvm_mcast (allproc, nproc, 100);

while (!(pvm_nrecv(-1,-1)))
   continue;

  /* while nodes compute something else, unpack data and  */
  /* print on the screen                                  */

  for (i=0; i<nproc; i++)
     old = pvm_setrbuf (bufs[i]);
     if (old)
         pvm_freebuf (old);
     pvm_upkint (&ds, 1, 1);
     pvm_upkint (&de, 1, 1);
     pvm_upkfloat (&array[dx], de-ds+1, 1);
  }
  pvm_freebuf (bufs[nproc-1]);

  print_screen (array);

}
```

Error Handling

Error conditions depend on the particular error that occurs. A set of useful **#define**s in the **pvm3.h** file (**fpvm3.h** for FORTRAN programs) links the value of each error code to a symbolic string. If your applications include the **pvm3.h** file (or **fpvm3.h**) such strings may be used in place of the corresponding numerical

values. All the error conditions are listed in Appendix A, “IBM PVMe for AIX Error Code Summary” on page 223.

Primitives for Error Handling

All the IBM PVMe for AIX routines return an integer variable. The value of the variable may be tested to check to see if the call was successful. You must test the return code of each routine, to see if the call was successful, and if not, you must take proper action. The return code is always a non-negative value if the routine was successful. All the routines, (with the exception of **pvm_spawn**, discussed below) return a negative value if they were unsuccessful.

Table 4. IBM PVMe for AIX Error Handling Routines

C	FORTRAN
int rc = pvm_perror (*char *msg*)	call pvmfperror (*msg*, *rc*)
int oldset = pvm_serror (*int set*)	call pvmfserror (*set*, *oldset*)

The routine cited above, **pvm_spawn**, can return a negative error code if invalid parameters were supplied. If valid parameters were supplied, it returns the actual number of spawned processes. If the routine was successful, this number is equal to the one requested by you. If it was less, only a limited number of **spawn** attempts were successful.

Automatic Printing of Error Messages

The two utility routines, **pvm_setopt** and **pvm_getopt** (see “Managing Options” on page 48) allow you to set the **PvmAutoErr** option and obtain its current value. You can set the **PvmAutoErr** option to either `0` or to `1`. This has the same effect as calling the **pvm_serror** routine (0 disables the automatic error messages printing, 1 enables it).

The **pvm_perror** routine allows you to print a line on **stderr** containing a message supplied by you, a colon (:), and the string corresponding to the value of the last error that occurred. If the message supplied by you contains a reference to the section of code being run, it is easy to recognize the cause of the error.

Note: **pvm_serror** has been removed from freeware PVM. For portability, use **pvm_setopt** instead.

Examples for Managing Error Conditions

FORTRAN Example

```
      program forthost
      integer MAXPROC
      parameter (MAXPROC=16)
      integer i, info, nproc, msgtype, mynum, inst(MAXPROC)
      ........
      ........
      character*64 errmsg

c Enroll to PVM3e
c
      call pvmfmytid (mynum)
      if(mynum.lt.0) then
```

```
         errmsg='Phase 1 enrolling the host'
         call pvmfperror (errmsg, rc)
         stop
       endif

c Initiate <nproc> instances of node program
c
       print *,'How many node programs (1-15)?'
       read *, nproc
c
       call pvmfspawn ("fortnode", PVMDEFAULT, '*',
      $                            nproc, inst, rc)
       if (rc.lt.0) then
         errmsg='Phase 2 initiating nodes'
         call pvmfperror (errmsg, rc)
       else if (rc.lt.nproc)
         do i=1, nproc
           if (inst(i).lt.0) then
             errmsg='Phase 2; spawning error was:'
             call pvmfperror (errmsg, rc)
            else
             call pvmfkill (procs(i), rc)
            endif
         enddo
         call pvmfexit(rc)
         stop
        endif
 10    continue

       .........

       stop
       end
```

C Example

Automatic printing of error messages from the IBM PVMe for AIX routines is disabled by the default. If this happens, the **pvm_perror** routine can be used to identify errors. Automatic printing can be temporarily enabled or disabled by using the **pvm_setopt** routine.

```
#include "pvm3.h"

#define NNODES 16

main (argc,argv)
int argc;
char **argv;
{
  int mynum, type;
  int oldset, ntypes;
  int inst[NNODES];

  if ((mynum=pvm_mytid())<0) {
     pvm_perror ("Error enrolling");
     exit(-1);
  }

  if ((rc=pvm_spawn ("nodeprog", NULL, PvmTaskDefault,
                           NULL, NNODES, inst)) < 0) {
      pvm_perror ("Error spawning nodes");
      pvm_exit ();
      exit (-1);
   } else {
```

```
        printf ("%d nodes have been successfully started\n", rc);
    }

  /* Now enable automatic error prints */
  oldset = pvm_setopt (1);

  ......
  ......

  /* To reset the default condition, call pvm_serror again */
  rc = pvm_setopt (oldset);

  ......

}
```

Signaling

The IBM PVMe for AIX program product provides two mechanisms to send signals to other tasks.

Primitives for Signaling

The first routine is **pvm_notify**, which is a general notification mechanism. If a process (for example, the host program) calls the **pvm_notify** routine, and supplies a list of processes and a message tag, it receives a message with the supplied message type when one of the supplied processes runs a particular event (when it exits). The calling process is responsible for checking the arrival of that message.

The second routine is **pvm_sendsig**, which allows a task to send a UNIX signal to other tasks. The following table shows the IBM PVMe for AIX signaling routines:

Table 5. IBM PVMe for AIX Signaling Routines

C	FORTRAN
int rc = pvm_notify (*int about*, *int msgtype*, *int ntids*, *int *tids*)	call pvmfnotify (*about*, *msgtype*, *ntids*, *tids*, *rc*)
int rc = pvm_sendsig (*int tid*, *int signum*)	call pvmfsendsig (*tid*, *signum*, *rc*)

Notes:

1. Since the underlying communications software used by IBM PVMe for AIX uses the signals SIGIO, SIGPIPE, and SIGALRM, it is suggested that you avoid using these signals in your application programs. You should also keep this in mind when using a debugger. You can ignore SIGIO, SIGPIPE, and SIGALRM signals when using **dbx** to debug an IBM PVMe for AIX program by using the following subcommands inside the debugger:

   ```
   ignore io
   ignore pipe
   ignore alrm
   ```

   You can also add these subcommands to the .dbxinit file.

2. Since the IBM PVMe for AIX routines are neither re-entrant nor restartable, signals should be appropriately masked when calling them.

Examples of Signaling

FORTRAN Example

The following sections of code show how the **pvm_notify** routine is used. A special message is sent to the host program when the node programs exit.

```
      program master

      include 'fpvm3.h'
      integer MAXP
      parameter (MAXP=31)
      integer nprocs, procs(MAXP)
      integer rc, i, dead

      ......

      call pvmfmytid (mytid)
      call pvmfspawn ('node', PVMDEFAULT, '*', nprocs, procs, rc)
      if (rc.ne.nprocs) then
         ..........
      endif

c Call pvmfnotify to be informed in case one of
c the children processes exits.
c
      call pvmfnotify (PVMTASKEXIT, 777, nprocs, procs, rc)

      ........

c Use a non-blocking receive to check whether someone
c exited or not.
c
      call pvmfnrecv(-1, 777, rc)
      if (rc.gt.0) then
         call pvmfunpack (INTEGER4, dead, 1, 1, rc)
         print *, 'Process ', dead, 'exited'
      endif

      ........

      call pvmfexit(rc)
      stop
      end
```

C Example

The following two sections show how the **pvm_sendsig** routine is used. The host program sends a UNIX signal to a set of node programs, and each of them invokes a routine supplied by you (**mysig** in the example) when that signal arrives.

```
/****************/
/* HOST PROGRAM */
/****************/

#define MAXCHILDREN 16
#include <stdio.h>
#include <signal.h>
#include "pvm3.h"

main()
{
```

```
  int me, children[MAXCHILDREN], nchild;
  ........

  me = pvm_mytid ();

  ........
  ........

  /* send signal SIGUSR1 to all node processes     */
  /*                                               */
  /* nchild   = total number of node processes     */
  /* children = array containing the children tids */

  for (i=0;i<nchild;i++) {
    rc = pvm_sendsig (children[i],SIGUSR1);
    if (rc)
       pvm_perror ("Error sending signal SIGUSR1\n");
  }

  .........
  ........

  pvm_exit();
  exit(0);
}


/****************/
/* NODE PROGRAM */
/****************/

#include <stdio.h>
#include <signal.h>
#include "pvm3.h"

main()
{

  int me,nchild,parent,i;
  int mypid,children[12];

  (void)signal(SIGUSR1,mysig);

  me = pvm_mytid();
  parent = pvm_parent();

  .......
  .......

}
```

Group Management

IBM PVMe for AIX processes can be organized into task groups. Task groups allows you to structure complex applications in an organized fashion. They run specific operations on subsets of tasks only. *Broadcasts* and *barriers* are typical operations that involve only members of a given group. However, you can also perform *global* operations, (such as applying reduction operations, data scatter and gather, etc.) on only the members of a specific group. These collective routines eliminate the need for repetitive coding of the operations you use frequently.

Managing Task Groups with Primitives

The basic concepts of group management in IBM PVMe for AIX are:

- Group size and members can change dynamically.
- At any time, processes can belong to one group, to multiple groups, or to no group at all.
- Groups are not required to be disjoint.

Each group is identified with a string, containing the group name. All processes belonging to a group have an instance number within that group.

The following table shows the IBM PVMe for AIX group management routines:

Table 6. IBM PVMe for AIX Group Management Routines

C	FORTRAN
int rc = pvm_barrier (*char *group*, *int count*)	call pvmfbarrier (*group*, *count*, *rc*)
int rc = pvm_bcast (*char *group*, *int msgtype*)	call pvmfbcast (*group*, *msgtype*, *rc*)
int rc = pvme_bcast (*char *group*, *int msgtype*, *rootinst*)	call pvmfbcast (*group*, *msgtype, rootinst, rc*)
int rc = pvm_gather (*void *result*, *void *data*, *int count*, *int datatype*, *int msgtype*, *char *group*, *int target*)	call pvmfgather (*result*, *data*,*count*,*datatype*, *msgtype*, *group*,*target*, *rc*)
int inst = pvm_getinst (*char *group*, *int tid*)	call pvmfgetinst (*group*, *tid*, *inst*)
int tid = pvm_gettid (*char *group*, *int inst*)	call pvmfgettid (*group*, *inst*, *tid*)
int size = pvm_gsize (*char *group*)	call pvmfgsize (*group*, *size*)
int inst = pvm_joingroup (*char *group*)	call pvmfjoingroup (*group*, *inst*)
int rc = pvm_lvgroup (*char *group*)	call pvmflvgroup (*group*, *rc*)
int rc = pvm_reduce (*void *func(), void *data, int count*, *int datatype*, *int msgtype*, *char *group*, *int rootinst*)	call pvmfreduce (*func*, *data*,*count*, *datatype*, *msgtype*, *group*, *rootinst*, *rc*)
int rc = pvm_scatter (*void *result*, *void *data*, *int count*, *int datatype*, *int msgtype*, *char *group*, *int rootinst*)	call pvmfscatter (*result*, *data*,*count*, *datatype*, *msgtype, group, rootinst, rc*)

Joining and Leaving Groups

To become a member of a given group, tasks must call the **pvm_joingroup** routine. If the group does not exist, it is created, and the calling process is the first member of the group. If the group does exist, the calling process is added to the group. In both cases, the *instance* number of the process within the group is returned. Instance numbers are assigned by the daemon, and are consecutive integers starting from `0`.

The **pvm_lvgroup** routine allows a process to leave a group. When this occurs, used instances may not be consecutive, because there will be vacant entries. Processes can rejoin the same group later, but there is no guarantee they will receive the same instance number. In fact, the daemon attempts to reuse entries that have been freed by tasks that leave.

Ensuring Reliable Group Operations

Because of the dynamic nature of IBM PVMe for AIX groups, it is essential that a new group is stable before you use it in group operations, and remains stable for the duration of these operations. For example, if the following program fragment is run on more than a few tasks:

```
pvm_joingroup("mygroup");
printf("There are %d tasks in this group.\n",
      pvm_gsize("mygroup"));
```

the group size that is reported may differ from task to task. This is because tasks may join and leave the groups at any time. Group construction is not a collective operation. As a result, if the tasks are not perfectly synchronized, some tasks may not have joined the group by the time other tasks call **pvm_gsize**. This can create some elusive bugs in situations such as the following:

```
pvm_joingroup("mygroup");
x = ComputeData();
pvm_reduce( op, &x, ... );
pvm_lvgroup("mygroup");
```

If the **ComputeData** function is quick, some tasks may attempt **pvm_reduce** before other tasks have completed **pvm_joingroup**, and the reduction may produce incorrect results or may hang.

To guarantee a group's stability, the members should be synchronized after they join the group with a call to **pvm_barrier**. Since the group may be incomplete before the call to **pvm_barrier**, **pvm_barrier** could compute the wrong size for the group. You must supply an explicit value (count) with **pvm_barrier** to indicate the size of the group. If *all* the tasks do not know the correct size of the group, it will be necessary to distribute that number from a task that does. Avoid **pvm_bcast** for this since **pvm_bcast**, like any other group operation, requires a stable group. To guarantee that a group has been created and is stable, a sequence such as the following is recommended:

```
if (is_master)
{       /* spawn off the worker tasks */
        nworkers = pvm_spawn( ...,  tids);
        /* compute the total number of tasks; the number
           of workers + 1 for the master */
        ntasks = nworkers + 1;
        /* send ntasks to each of the workers */
        pvm_pkint(&ntasks, 1, 1);
        pvm_mcast(nworkers, tids, 1);
}
else
{       /* receive ntasks */
        pvm_recv(pvm_parent(), 1);
        pvm_upkint(&ntasks, 1, 1);
}
/* join the group */
pvm_joingroup("mygroup");
/* synchronize everyone in the group */
pvm_barrier("mygroup", ntasks);
```

After collective operations are complete, it is important that no task leaves the group before other tasks complete the collective operations. To ensure this, each task in the group should call **pvm_barrier** before leaving the group with **pvm_lvgroup** or **pvm_exit**.

Understanding the Broadcast Operation

Broadcast is a typical operation dealing with group members. The **pvm_bcast** routine allows you to send a message to all instances of a given group, without specifying the task identifiers of each recipient task. The sender process can either belong to the group or not. Each of the other tasks in the group should call any receive routine to receive the broadcast.

For example:

```
if (sender_tid == mytid)
{
   value = ComputeMe();
   pvm_initsend(PvmDataRaw);
   pvm_pkint(&value, 1, 1);
   pvm_bcast("mygroup", 5);
}
else
{
   pvm_recv(sender_tid, 5);
   pvm_upkint(&value, 1, 1);
   printf("Received value %d\n", value);
}
```

Note: Because the size of groups can change dynamically, the number of recipient processes might be different in different runs of the same program, according to the timing relationships among processes.

The **pvme_bcast** routine is an enhanced version of **pvm_bcast** (it's not available in freeware PVM). Its syntax requires one more additional parameter (Instance number of the sender process). The routine must be invoked by all processes belonging to the group. All the tasks participate in collective operations. On all the tasks, except the sender, the data will be placed in the current receive buffer. It is not necessary to receive a message just to unpack the data.

For example:

```
if (sender_instance_no == myinst)
{
   value = ComputeMe();
   pvm_initsend(PvmDataRaw);
   pvm_pkint(&value, 1, 1);
   pvme_bcast("mygroup", 5, sender_instance_no);
}
else
{
   pvme_bcast("mygroup", 5, sender_instance_no);
   pvm_upkint(&value, 1, 1);
   printf("Received value %d\n", value);
}
```

The **pvm_bcast** routine works as a loop of calls to **pvm_send**, and can be used to write simpler and more easily readable code. On the other hand, the modified syntax of **pvme_bcast** lets you deal with the broadcast operation as if it were a

collective action, like scatter, gather, and reduce, and allows the implementation of more efficient algorithms. You should use **pvme_bcast** when the number of group members is high.

Understanding the Barrier Routine

The **pvm_barrier** routine is the only routine provided by the IBM PVMe for AIX program product that causes an explicit synchronization among tasks. In the message-passing programming paradigm on distributed memory systems, most of the synchronization is done (or can be done) with the message exchange itself.

The **pvm_barrier** routine causes a set of processes belonging to the same group to block until a specified number of processes have called the same barrier. It is necessary that all processes belonging to the same group be involved in the synchronization unless the variable **PVMe_BARRTYPE** is set. See "PVMe_BARRTYPE" on page 227.

Understanding PVM_EXPORT

Normally, no environment variables are copied from the master task's environment to the workers', but if an environment variable is named in the PVM_EXPORT variable in the environment of the master task, those variables will be copied from the master task's environment and re-created in each of the workers' environment.

In this example, "printdisplay" is a PVMe program that just prints the **DISPLAY** environment variable.

```
% printdisplay
Master: "edison.kgn.ibm.com:0"
Worker1: ""
Worker2: ""
Worker3: ""
% export PVM_EXPORT=DISPLAY
% printdisplay
Master: "edison.kgn.ibm.com:0"
Worker1: "edison.kgn.ibm.com:0"
Worker2: "edison.kgn.ibm.com:0"
Worker3: "edison.kgn.ibm.com:0"
```

Performing Global Operations

The **pvm_scatter** routine distributes data from one task to all the other members of the same group. Given the starting address of data on the root instance, typically an array, different chunks of the array are sent to the group members in order. The first items of the array (specified by *count*) will be received by the first member, the next items (specified by *count*) by the second member, and so on.

The **pvm_gather** routine performs the opposite task. It collects data from all members of a group to a single member. This target member, after the call, owns the whole array. Chunks coming from different processes are stored in order on the target process, depending on the group instance number of the sender.

The **pvm_reduce** routine applies reduction operations to all the tasks in a group, and places the result in one task. Predefined functions are provided by IBM PVMe for AIX (search for minimum and maximum, global sum, global product) but user written functions can be used as well.

Querying Information about Groups

The following routines provide group status information:

- The **pvm_getinst** routine, which returns the instance number of a given process (identified by its *tid*) within a group.
- The **pvm_gettid** routine, which returns the *tid* of a process when a group name and the process instance number within that group are supplied.
- The **pvm_gsize** routine, which returns the number of members of a given group.

Examples for Managing Groups

FORTRAN Example

```
      program hgroups

      include 'fpvm3.h'
      integer MAXP, MAXSTEPS, MAXDATA
      parameter (MAXP=31, MAXSTEPS=5, MAXDATA=1024*8)
      integer n1, n2, nprocs, procs(MAXP)
      integer cl1inst0, cl2inst0, rc, i , flag
      real*4 xdata(MAXDATA), ydata(MAXDATA)
      integer nsteps

      .........

      call pvmfmytid (mytid)
      call pvmfspawn ('ngroups', PVMDEFAULT , '*', nprocs, procs, rc)
      ............

c Join group "all". In this example all processes (host and
c nodes) belong to the "all" group.  Node processes also
c belong to either "class1" or  "class2" group.
c
      call pvmfjoingroup ("all", mygid)

c First data are broadcast to all processes
c
      call pvmfinitsend (PVMDEFAULT, rc)
      call pvmfpack (INTEGER4, nprocs, 1, 1, rc)
      call pvmfpack (INTEGER4, procs, nprocs, 1, rc)
      call pvmfbcast ("all", 4, rc)

      call inputdata (xdata)
      call inputdata (ydata)
      n1 = nprocs/2
      n2 = nprocs-n1

c Barrier involves nprocs+1 processes, that is, all processes
c that belong or are going to join the "all" group.
c
      call pvmfbarrier ("all", nprocs+1, rc)

c Broadcast data to all members of a particular group
c
      call pvmfinitsend (PVMDEFAULT, rc)
      call pvmfpack (REAL4, xdata(1), ndata, 1, rc)
      call pvmfbcast ("class1", 5, rc)

      call pvmfinitsend (PVMDEFAULT, rc)
      call pvmfpack (REAL4, ydata, ndata, 1, rc)
      call pvmfbcast ("class2", 6, rc)

c Get the task identifier of the process with instance "0"
c within each group; at each iteration the host program
c exchange data only with the first members of all groups.
c
      call pvmfgettid ("class1", 0, cl1inst0)
      call pvmfgettid ("class2", 0, cl2inst0)

      nsteps=0
      do while (nsteps.lt. MAXSTEPS)

        call pvmfrecv (cl1inst0, -1, rc)
```

```
        call pvmfunpack (....)

        call pvmfrecv (cl2inst0, -1, rc)
        call pvmfunpack (....)

        call merge ()

        call pvmfinitsend (PVMDEFAULT, rc)
        call pvmfpack (.........)
        call pvmfsend (cl1inst0, nsteps, rc)

        call pvmfinitsend (PVMDEFAULT, rc)
        call pvmfpack (.........)
        call pvmfsend (cl2inst0, nsteps, rc)

        nsteps= nsteps+1
      enddo

      call pvmfexit(rc)

      stop
      end
```

C Example

```
#include <stdio.h>
#include <strings.h>
#include "pvm3.h"

#define NNODES 16

int main(int argc, char **argv)
{
  int mytid, mygid, tids[NNODES];
  int i, indx;

  mytid = pvm_mytid();

  /* join group "allgroup" and print my instance within the group */
  mygid = pvm_joingroup("allgroup");
  fprintf(stderr, "%s %d %d\n", "allgroup", mytid, mygid);

  /* I'm the first to join, then start the others tasks */
  if (mygid == 0) {
    pvm_spawn(argv[0], argv+1, 0, "", NNODES-1, tids);
  }

  /* synchronization among all processes */
  if (pvm_barrier("allgroup", NNODES) < 0) {
     pvm_lvgroup("allgroup");
     pvm_exit();
     exit (-1);
  }

  /* everyone broadcast its group id and task id */
  pvm_initsend(PvmDataDefault);
  pvm_pkint(&mygid, 1, 1);
  pvm_pkint(&mytid, 1, 1);
  pvm_bcast("allgroup", 50);

  /* receive all the gids and tids (except from myself) */
  tids[mygid] = mytid;
  for (i = 0; i < NNODES-1; i++) {
    pvm_recv(-1, 50);
    pvm_upkint(&indx, 1, 1);
```

```
    pvm_upkint(tids+indx, 1, 1);
  }

  /* check to make sure the gids and tids are correct */
  for (i = 0; i < NNODES; i++) {
      if (i != pvm_getinst("allgroup", tids[i])) {
        fprintf(stderr, "gid %d doesn't match\n", i);
      }
      if (tids[i] != pvm_gettid("allgroup", i)) {
       fprintf(stderr, "gid %d doesn't match\n", i);
      }
  }

  ..............

  pvm_barrier("allgroup", NNODES);
  /* leave the group */
  pvm_lvgroup("allgroup");
  pvm_exit();
  exit();
}
```

Note: If IBM PVMe for AIX tasks do not synchronize themselves before making a call to **pvm_exit**( ), a situation may arise where the application hangs. To avoid this, it is suggested that you precede the **pvm_exit**( ) call with the **pvm_barrier**( ) routine. Note that in order to use **pvm_barrier**( ), all the tasks must enroll in the same group via the **pvm_joingroup**( ) routine. To unenroll from a group, use the **pvm_lvgroup**( ) routine.

A Complete Example (FORTRAN)

The following example is the solution of the Laplace equation on a uniform rectangular lattice of unit spacing. The domain size, the maximum number of iterations and the relaxation rate (*omega*) are read from the input file (**lap.input**), as well as the initial distribution for the potential array.

This example first includes serial code, along with an example for the input file, then shows the parallel programs (according to the master-worker model).

Serial Code

Program **laplace.f**.

```
        program main
c       ************************************************************
c
c       Solution of LAPLACE eq.n in 2-D
c
c       ************************************************************
        parameter (kx=201,ky=201)
        integer init, iter, maxit
        real omega, olde, energy, oldelp, delpma
        real p(0:kx,0:ky)
        logical lbc(0:kx,0:ky)
        real *8 msecond, tstart, tend, tot
c
        open(15,file='lap.input',form='formatted')
        open(16,file='lap.output',form='formatted')
c
c
```

```
c Read parameters from input file
c
         call inputparam (nx, ny, init, omega, maxit)
c
c Initialize array p (potential) and lbc (boundary conditions)
c
         call potarr (init, nx, ny, lbc, p)
c
c Prepare prints for output
c
         write (16,500)
         write (16,600)
         write (16,700)
         write (16,600)
         write (16,500)
         write (16,600)
         write (16,707)
c
c Main loop
c
         do 300 iter = 1, maxit
         tstart = msecond()
           olde = energy
           energy = 0.
           oldelp = delpma
           delpma = 0.
c
c Lattice relaxation
c
           call relax (delpma, energy, iter, 1, nx, 1, ny, omega, lbc, p
           de = energy-olde
           if (iter.eq.1) then
              de = 0.
              oldelp = 0.
           endif
           tend = msecond()
           tot = tot + tend - tstart
           write (16,708) iter, delpma, energy, de
c
300      continue
         print *, 'TOTCPU=', tot/1.0D6

500      format('*************************************************')
600      format('*        |              |              |              *')
700      format('*  iter. |  Delta P     |   Energy     |    Delta E   *')
707      format('*    0   |   0.000000  |   0.000000  |    0.000000  *')
708      format('*',i5,'  |',f11.6,'    |',f11.6,'    | ',f11.6,'    *')
         stop
         end
c
c*********************************************************************
c
      subroutine inputparam (nx, ny, init, omega, maxit)
c
      integer nx, ny, init, maxit, info
      real*4  omega

      read  (15,*) nx
      write (6,*) 'nx = ', nx
      if ((nx.le.0).or.(nx.gt.200)) then
        write (6,*) 'Wrong input, nx out of [1-200]. Try again.'
        stop
      endif
```

```
      read  (15,*) ny
      write (6,*) 'ny = ', ny
      if ((ny.le.0).or.(ny.gt.200)) then
        write (6,*) 'Wrong input, ny out of [1-200]. Try again.'
        stop
      endif

      read  (15,*) init
      write (6,*) 'init = ', init
      if ((init.lt.1).or.(init.gt.35)) then
        write (6,*) 'Wrong input, init out of [1-35]. Try again.'
        stop
      endif

      read  (15,*) omega
      write (6,*) 'omega = ', omega
      if ((omega.lt.0.).or.(omega.gt.1.)) then
        write (6,*) 'Wrong input, omega out of [0.-1.]. Try again.'
        stop
      endif

      read  (15,*) maxit
      write (6,*) 'maxit = ', maxit
      if (maxit.le.0) then
        write (6,*) 'Wrong input, maxit must be positive. Try again.'
        stop
      endif
      return
      end
c
c*********************************************************************
c
         subroutine potarr (init, nx, ny, lbc, p)
c
         parameter (kx=201,ky=201)
         integer row, col, init, nx, ny
         integer pot, ndown, nleft
         real p(0:kx,0:ky)
         logical lbc(0:kx,0:ky)
c
c Initialize   p == init everywhere
c Initialize lbc == FALSE everywhere
c
100      do 150 row = 1, ny
             do 130 col = 1, nx
                p(row,col) = float(init)
                lbc(row,col) = .FALSE.
130           continue
150      continue
c
c lbc == TRUE on boundaries
c
         do 250 row = 1,ny
                lbc(row,1) = .true.
                lbc(row,nx) = .true.
250       continue
c
         do 350 col = 1, nx
                lbc(1,col) = .true.
                lbc(ny,col) = .true.
350      continue
c
c   p = initial condition (pot) where defined by the user
c lbc = TRUE where p has been set
```

```
c
50       read (15,*) pot, ndown, nleft
         if (pot.eq.0) return

         if ((pot.lt.0).or.(ndown.gt.200).or.(nleft.gt.200)) then
           write (6,*) 'pot=', pot, 'ndown=', ndown, 'nleft=', nleft
           write (6,*) 'Wrong input. Try again'
           stop
         endif
c
         p(1+ndown,1+nleft) = float(pot)
         lbc(1+ndown,1+nleft) = .true.
         goto 50
         end
c
c*********************************************************************
c
         subroutine relax (delpma, energy, iter, x1, x2,
     $                     y1, y2, omega, lbc, p)
c
         integer y1, y2, x1, x2, iter, row, col, nx, ny, kx, ky
         parameter (kx=201, ky=201)
         real omega, energy, delpma, de
         real a, b, c, d, deltap
         real p(0:kx,0:ky), pnew(0:kx,0:ky)
         logical lbc(0:kx,0:ky)
c
         nx = y2 - y1 + 1
         ny = x2 - x1 + 1
         do 100 col = x1, x2
           do 200 row = y1, y2
c
              if ((row.eq.y1).and.(col.eq.x1)) goto 200
              if (lbc(row,col).eqv..true.) then
                  pnew(row,col) = p(row,col)
                  goto 50
              endif
c
              pnew(row,col) = (1.0-omega) * p(row,col) + omega*0.25*(
     $        p(row+1,col) + p(row-1,col) + p(row,col+1) + p(row,col-1))
              if (pnew(row,col).lt.1.) then
                   pnew(row,col) = 1.
              endif
              deltap = abs(p(row,col) - pnew(row,col))/p(row,col)
              if (deltap.gt.delpma) delpma = deltap
c
50            if (col.eq.x1) then
                 energy = energy + (pnew(row,x1) - p(row-1,col)) ** 2
                 goto 200
              endif
c
              if (row.eq.y1) then
                 energy = energy + (pnew(y1,col) - p(row,col-1)) ** 2
                 goto 200
              endif
              energy = energy + (pnew(row,col) - p(row-1,col)) ** 2+
     $                     (pnew(row,col) - p(row,col-1)) ** 2
200         continue
100      continue
c
         energy = energy / (2. * float(nx * ny))
         do i = x1, x2
            do j = y1, y2
               p(j,i) = pnew(j,i)
```

```
           enddo
         enddo
c
         return
         end
```

Input file **lap.input**

```
200                     X size (between 1 and 200)
200                     Y size (between 1 and 200)
1                       Potential value
0.9                     relaxation parameter (between 0. and 1.)
100                     max. number of iterations
9 24 49                 potential value in (x,y), x, y
9 24 50                        "
9 24 51                        "
9 25 49                        "
9 25 50
9 25 51
  ........
  ........
  ........
9 50 76
9 51 74
9 51 75
9 51 76
0 0 0                   End of data
```

Parallel Code

Host program, **laphost.f** file

```
         program main
c        ************************************************************
c
c        Solution of LAPLACE eq.n in 2-D
c
c     ************************************************************
      implicit none
      include 'fpvm3.h'
      integer kx, ky, MAXNODES
      parameter (kx=201, ky=201, MAXNODES=16)
      integer nx, ny, init, iter, maxit, itype
      integer i, j, info, errflag, me, nnodes, slice
      real omega, de, olde, energy, eenergy
      real ddelpma, delpma, oldelp
      real p(0:kx,0:ky)
      logical lbc(0:kx,0:ky)
      real *8 msecond, tstart, tend, tot
      integer inst(MAXNODES), starts(MAXNODES), ends(MAXNODES)
c
      open (15, file='/tmp/lap.input',  form='formatted')
      open (16, file='/tmp/lap.output', form='formatted')
c
c
c Read parameters from input file
c
      call inputparam (nx, ny, init, omega, maxit)
c
c Read number of nodes
c
      print *, 'Enter number of nodes'
      read  *, nnodes
```

```
      if (nnodes.gt.MAXNODES) then
        print *, 'Too many nodes. Try again'
        stop
      endif
c
c Enroll to PVMe
c
      call pvmfmytid (me)
      if (me.lt.0) then
        call pvmfperror ('Host enrolling\0')
        stop
      endif
c
c Spawn <nnodes> instances of program "lapnode"
c
      call pvmfspawn ('lapnode', PVMDEFAULT, '*', nnodes, inst, info)
      if (info.ne.nnodes) then
        if (info.lt.0) then
          call pvmfperror ('Spawning nodes', info)
        else
          do i=1,nnodes
            if (inst(i).lt.0) then
              write (6,*) 'Error ', inst(i), 'for process n. ', i
            else
              call pvmfkill (inst(i),info)
            endif
          enddo
        endif
        call pvmfexit (info)
      endif
c
c Initialize array p (potential) and lbc (boundary conditions)
c
      errflag = 0
      call potarr (init, nx, ny, lbc, p, errflag)
      if (errflag.ne.0) then
c
c Before exiting, kill all children to clean the environment
c
         do i = 1, nnodes
            call pvmfkill (inst(i), info)
         enddo
         call pvmfexit(info)
      endif
c
c Prepare prints for output
c
      write (16,500)
      write (16,600)
      write (16,700)
      write (16,600)
      write (16,500)
      write (16,600)
      write (16,707)
c
c Receive a dummy message from each node. Just for synchronization
c
      do i=1,nnodes
        call pvmfrecv(-1, -1, info)
      enddo
c
c Send initial data
c
      call pvmfinitsend (PVMDEFAULT, info)
```

```
      call pvmfpack (INTEGER4, nnodes, 1, 1, info)
      call pvmfpack (INTEGER4, nx, 1, 1, info)
      call pvmfpack (INTEGER4, ny, 1, 1, info)
      call pvmfpack (INTEGER4, inst, nnodes, 1, info)
      call pvmfpack (INTEGER4, maxit, 1, 1, info)
      call pvmfpack (REAL4, omega, 1, 1, info)
      call pvmfmcast (nnodes, inst, 1, info)
c
c Divide the computational domain and send different subsets of
c data to different nodes
c
      slice = ny/nnodes
      starts(1) = 0
      do i = 2, nnodes
         starts(i) = ((i-1) * slice)
         ends(i-1) = ((i-1) * slice + 1)
      enddo
      ends(nnodes) = ny
c
      do i = 1, nnodes
        call pvmfinitsend (PVMDEFAULT, info)
        call pvmfpack (INTEGER4, i, 1, 1, info)
        call pvmfpack (INTEGER4, starts(i), 1, 1, info)
        call pvmfpack (INTEGER4, ends(i), 1, 1, info)
        do j = starts(i), ends(i)
          call pvmfpack (REAL4, p(0,j), (ny+1), 1, info)
          call pvmfpack (REAL4, lbc(0,j), (ny+1), 1, info)
        enddo
        call pvmfsend (inst(i), inst(i), info)
      enddo
c
      itype = 100
      do 300 iter = 1, maxit
      tstart = msecond()
         olde = energy
         oldelp = delpma
         energy = 0.0
         delpma = 0.0
c
c At each iteration receive partial results from nodes and
c combine them.
c
         do i = 1, nnodes
           call pvmfrecv(-1, itype, info)
           call pvmfunpack (REAL4, eenergy, 1, 1, info)
           call pvmfunpack (REAL4, ddelpma, 1, 1, info)
           energy = energy + eenergy
           if (ddelpma.gt.delpma) then
             delpma = ddelpma
           endif
         enddo
         itype = itype + 1

         energy = energy / (2. * float(nx * ny))
         de = energy - olde
         if (iter.eq.1) then
           de = 0.
           oldelp = 0.
         endif
         tend = msecond()
         tot = tot + tend - tstart
         write (16,708) iter, delpma, energy, de
c
300      continue
```

```
         print *, 'TOTCPU=', tot/1.0D6

500      format('*************************************************')
600      format('*        |             |              |             *')
700      format('*  iter. |  Delta P    |   Energy     |    Delta E  *')
707      format('*    0   |   0.000000  |   0.000000   |    0.000000 *')
708      format('*',i5,'  |',f11.6,'    |',f11.6,'    | ',f11.6,'   *')

         call pvmfexit (info)
1000     stop
         end
c
c**********************************************************************
c
      subroutine inputparam (nx, ny, init, omega, maxit)
c
      ..................................
      ........   see serial code  ..........
      ..................................
c
c**********************************************************************
c
         subroutine potarr (init, nx, ny, lbc, p, errflag)
c
         integer kx,,ky
         parameter (kx=201, ky=201)
         integer row, col, init, nx, ny
         integer pot, ndown, nleft
         real p(0:kx, 0:ky)
         logical lbc(0:kx, 0:ky)
c
100      do 150 row = 1, ny
             do 130 col = 1, nx
                p(row,col) = float(init)
                lbc(row,col) = .FALSE.
130           continue
150      continue
c
         do 250 row = 1, ny
                lbc(row,1) = .true.
                lbc(row,nx) = .true.
250       continue
c
         do 350 col = 1, nx
                lbc(1,col) = .true.
                lbc(ny,col) = .true.
350      continue
c
50       read (15,*) pot, ndown, nleft
         if (pot.eq.0) then
            return
         endif

         if ((pot.lt.0).or.(ndown.gt.200).or.(nleft.gt.200)) then
           write (6,*) 'pot=', pot, 'ndown=', ndown, 'nleft=', nleft
           write (6,*) 'Wrong input. Try again'
           errflag = 1
           return
         endif
c
         p(1+ndown,1+nleft) = float(pot)
         lbc(1+ndown,1+nleft) = .true.
         goto 50
         end
```

Node program, **lapnode.f**

```
      program main
c     ************************************************************
c
c     Solution of LAPLACE eq.n in 2-D
c
c     ************************************************************
      implicit none
      include 'fpvm3.h'
      integer kx, ky, MAXNODES
      parameter (kx=201, ky=201, MAXNODES=8)
      integer nx, ny, i, j, init, iter, maxit
      real omega, olde, energy, oldelp, delpma
      real p(0:kx, 0:ky)
      logical lbc(0:kx, 0:ky)
      real *8 msecond, tstart, tend, tot
      integer nnodes, me, parent, inst(MAXNODES)
      integer myindex, mystart, myend, itype, info
c
c Enroll to PVMe, then send a dummy message for synchronization.
c
      call pvmfmytid (me)
      call pvmfparent (parent)
      call pvmfinitsend (PVMDEFAULT, info)
      call pvmfpack (INTEGER4, me, 1, 1, info)
      call pvmfsend (parent, 999, info)
c
c Receive initial data
c
      call pvmfrecv (parent, 1, info)
      call pvmfunpack (INTEGER4, nnodes, 1, 1, info)
      call pvmfunpack (INTEGER4, nx, 1, 1, info)
      call pvmfunpack (INTEGER4, ny, 1, 1, info)
      call pvmfunpack (INTEGER4, inst, nnodes, 1, info)
      call pvmfunpack (INTEGER4, maxit, 1, 1, info)
      call pvmfunpack (REAL4, omega, 1, 1, info)
c
c Receive my data subset
c
      call pvmfrecv (parent, me, info)
      call pvmfunpack (INTEGER4, myindex, 1, 1, info)
      call pvmfunpack (INTEGER4, mystart, 1, 1, info)
      call pvmfunpack (INTEGER4, myend, 1, 1, info)
      do j = mystart, myend
         call pvmfunpack (REAL4, p(0,j), (ny+1), 1, info)
         call pvmfunpack (REAL4, lbc(0,j), (ny+1), 1, info)
      enddo
c
      if (myend.eq.ny) then
         myend = ny+1
      endif
      itype = 100
      do 300 iter = 1,maxit
         energy = 0.0
         delpma = 0.0

         call relax (delpma, energy, iter, mystart+1, myend-1,
     $       1, nx, omega, lbc, p)
c
c Send partial result to the host program
c
         call pvmfinitsend (PVMDEFAULT, info)
         call pvmfpack (REAL4, energy, 1, 1, info)
```

```
         call pvmfpack (REAL4, delpma, 1, 1, info)
         call pvmfsend(parent, itype, info)
c
c Exchange data on boundaries with left and right neighbours
c
         if (myindex.ne.1) then
            call pvmfinitsend (PVMDEFAULT, info)
            call pvmfpack (REAL4, p(0,mystart+1), (nx+1), 1, info)
            call pvmfsend (inst(myindex-1), inst(myindex-1)+100, info)
         endif
         if (myindex.ne.nnodes) then
            call pvmfinitsend (PVMDEFAULT, info)
            call pvmfpack (REAL4, p(0,myend-1), (nx+1), 1, info)
            call pvmfsend (inst(myindex+1), inst(myindex+1)+200, info)
         endif
         if (myindex.ne.1) then
            call pvmfrecv(-1, me+200, info)
            call pvmfunpack (REAL4, p(0,mystart), (nx+1), 1, info)
         endif
         if (myindex.ne.nnodes) then
            call pvmfrecv(-1, me+100, info)
            call pvmfunpack (REAL4, p(0,myend), (nx+1), 1, info)
         endif

         itype=itype+1
300    enddo

       call pvmfexit (info)
       stop
       end
c
c**********************************************************************
c
         subroutine relax (delpma, energy, iter, x1, x2,
     $                     y1, y2, omega, lbc, p)
c
       integer kx, ky
       parameter (kx=201, ky=201)
       integer y1, y2, x1, x2, iter
       real omega, energy, delpma
       real a, b, c, d, pold, deltap
       real p(0:kx,0:ky), pnew(0:kx,0:ky)
       logical lbc(0:kx,0:ky)
c
       do 100 col = x1, x2
          do 200 row = y1, y2
c
             if ((row.eq.1).and.(col.eq.1)) goto 200
             b = p(row-1,col)
             d = p(row,col-1)
             if (lbc(row,col).eqv..true.) then
                 pnew(row,col) = p(row,col)
                 goto 50
             endif
c
             pold = p(row,col)
             a = p(row+1,col)
             c = p(row,col+1)
             pnew(row,col) = (1.0-omega) * pold + omega * 0.25 * (a+b+c+d)
             if (pnew(row,col).lt.1.) then
                pnew(row,col) = 1.
             endif
             deltap = abs(pold - pnew(row,col)) / pold
```

```
            if (deltap.gt.delpma) delpma = deltap
c
50          if (col.eq.1) then
               energy = energy + (pnew(row,1) - b) ** 2
               goto 200
            endif
c
            if (row.eq.1) then
               energy = energy + (pnew(1,col) - d) ** 2
               goto 200
            endif
            energy = energy + (pnew(row,col) - b) ** 2 +
     $           (pnew(row,col) - d) ** 2
200         continue
100      continue
c
         do i = x1, x2
           do j = y1, y2
             p(j,i) = pnew(j,i)
           enddo
         enddo
         return
         end
```

Chapter 6. Using IBM PVMe for AIX

This section discusses how to compile and link parallel programs in a variety of environments. In addition, you will learn how to run a parallel program as well as to debug it should you find errors.

Note: PVMe is not thread safe and should not be used with threaded user programs.

Compiling and Linking a Parallel Program on a POWERparallel System

The IBM PVMe for AIX libraries for C and FORTRAN programs were stored in the /**usr**/**lpp**/**pvme** directory during installation. Symbolic links were created so that you would not be required to specify the full pathname for the libraries. Since the MPCI is a shared library, at the beginning of your program, the **LIBPATH** environment variable may be used to select the path to the MPCI library you will use at run time.

If you're running your program over the MPCI US protocol:

```
export LIBPATH=/usr/lpp/ssp/css/libus:/lib:/usr/lib
```

If you're running your program over the MPCI IP protocol:

```
export LIBPATH=/usr/lpp/ssp/css/libip:/lib:/usr/lib
```

If `LIBPATH` is not specified, the MPCI version selected at link time will be used. **C** programs only need to be linked to **libpvm3.a**, while FORTRAN programs need to be linked to both **libpvm3.a** and to the **libfpvm3.a** FORTRAN library. For both C and FORTRAN programs, you must supply the linker with the following information to resolve symbols not defined in the IBM PVMe for AIX libraries:

```
-bI:/usr/lib/pvm3e.exp
```

The example below shows how to link the MPCI library, **libmpci.a**. In this example, the library is being linked with **libus**.

```
FLAGS = -O
FC = xlf
CLIB  = -lpvm3
FLIB  = -lfpvm3
LIBS  = $(CLIB) $(FLIB)
IMPORT= -bI:/usr/lib/pvm3e.exp
TARGETDIR = $(HOME)/pvm3/bin/RS6K

#To use US protocol
LFLAGS= -L/usr/lpp/ssp/css/libus -lmpci
#To use IP protocol
#LFLAGS= -L/usr/lpp/ssp/css/libip -lmpci
.f.o:
       xlf -c $(FLAGS) $*.f

all: hostpvm nodepvm

hostpvm: hostpvm.o
        $(FC) $(LFLAGS) hostpvm.f $(LIBS) $(IMPORT) -o hostpvm
        mv hostpvm $(TARGETDIR)
```

© Copyright IBM Corp. 1993, 1994, 1995

```
nodepvm : nodepvm.o
        $(FC) $(LFLAGS) nodepvm.f $(LIBS) $(IMPORT) -o nodepvm
        mv nodepvm $(TARGETDIR)
```

You can link the MPCI library statically by providing the linker with the explicit name of the library as well as the **-bnso** and **-bI:/lib/syscalls.exp** options.

The **-bnso** option will also statically link all the other dynamic libraries, potentially creating very large executables. An alternative is to create a non-shared version of the MPCI library, and then dynamically link to all the remaining shared libraries. A non-shared version of the MPCI library named **libmpci_temp.a**, can be created as follows:

```
cp /usr/lpp/ssp/css/libus/libmpci.a libmpci_temp.a
ar xv libmpci_temp.a mpci.shr.o
ld -r -bnso -o mpci.o mpci.shr.o
ar rv libmpci_temp.a mpci.o
ar dv libmpci_temp.a mpci.shr.o
rm mpci.shr.o mpci.o
```

Note: If you statically link your program to the MPCI library, you need different executables to be run over the MPCI IP and the MPCI User Space protocols.

Compiling and Linking a Parallel Program on an External Workstation

Creating an executable application is the same regardless of whether it will be run on a RISC System/6000 SP system or an external RISC System/6000 workstation. The only exception is the MPCI library that must be used. For external hosts, a dummy object file is provided that contains the MPCI symbols to be resolved by the linker. The name of this object file is **mpciglue.o**, and can be used as follows:

```
FLAGS = -O
FC = xlf
CLIB  = -lpvm3
FLIB  = -lfpvm3
LIBS  = $(CLIB) $(FLIB)
IMPORT= -bI:/usr/lib/pvm3e.exp
TARGETDIR = $(HOME)/pvm3/bin/RS6K

#$(LFLAGS) must preceed $(LIBS) in the link command
LFLAGS= /usr/lpp/pvme/lib/mpciglue.o
.f.o:
       xlf -c $(FLAGS) $*.f

all: hostpvm nodepvm

hostpvm: hostpvm.o
        $(FC) $(LFLAGS) hostpvm.f $(LIBS) $(IMPORT) -o hostpvm
        mv hostpvm $(TARGETDIR)

nodepvm : nodepvm.o
        $(FC) $(LFLAGS) nodepvm.f $(LIBS) $(IMPORT) -o nodepvm
        mv nodepvm $(TARGETDIR)
```

Note: You will have two distinct IBM PVMe for AIX executable applications; one to be run on the SP nodes, and one to be run on the external workstations. You can keep the two versions in two different directories (for example,

$HOME/**pvm3**/**bin**/**RS6K_EW** and **$HOME**/**pvm3**/**bin**/**RS6K_SP**) and use either the **ep** option in the **hostlist** file or the **PVMEPATH** variable to properly set the path to the binary files.

Running a Parallel Program

The IBM PVMe for AIX program product may run either in batch or interactive mode. The procedure for starting an IBM PVMe for AIX session was discussed in Chapter 4, “Product Startup” on page 21. Be sure to read this chapter carefully because it pertains to the startup step and how you may take advantage of the available tools during an interactive session to monitor the parallel program execution.

Your executable files can be located either in:

- The **$HOME**/**pvm3**/**bin**/**RS6K** directory, which is the default directory where the IBM PVMe for AIX program product searches for programs to be started
- An alternate directory, provided that you tell the IBM PVMe for AIX daemon the path to the directory

If the daemon is started on a set of nodes allocated by the Resource Manager, as is recommended, the environment variable **PVMEPATH** must be properly set and exported before the daemon is started. For example:

```
export PVMEPATH=$HOME/benchmark/gauss
```

If the daemon is started with a hostfile, the **ep=** option in the **hostlist** file must be set to the *pathname* of the chosen directory. For example:

```
hostA   <absolute path>
hostB   <absolute path>
hostC
```

A mechanism is available which allows environment variables to be passed to processes spawned by the IBM PVMe for AIX daemon. You have to select the variables that you want to be passed, and then concatenate them into a single variable, **PVM_EXPORT**, using the colon (:) as a separator. This is done in the shell which starts the master task. Example:

```
user@spn1> export INPUT_PATH=$HOME/data/input
user@spn1> export DISPLAY=xstat12:0
user@spn1> export PVM_EXPORT=DISPLAY:INPUT_PATH
user@spn1> master_task
```

Debugging a Parallel Program

Debugging a parallel program is a difficult task, not only because multiple processes are running at the same time, but also because errors might occur because of the interaction among processes. In addition, run-time conditions are not always the same for a parallel program, and they may be difficult to reproduce. Some errors appear only in small windows of time, or they may occur only under particular circumstances. Finally, the effects of an error may be noticed much later in the processing, when there is no trace of the actual failure.

Diagnostic print statements or calls to the **pvm_perror** routine allow you to detect error conditions from IBM PVMe for AIX routines. They are recommended during code development and debugging and can be removed when the program works as

expected. You may also find it helpful to enable automatic error messages with **setopt**.

Finding Errors in Program Logic

Errors are especially difficult to find if they are somehow related to the program logic. If that is the case, follow these steps:

1. Run the parallel program as a single process (if possible), just as if it were a serial program. By doing this, you can find logic errors easily. Then, the program may be run under a standard debugging tool (such as **dbx**), like any other serial program.
2. Run the parallel program in parallel on a single host (as a set of tasks all running on the same node). Provided the IBM PVMe for AIX session consists of only one node, you can run multiple processes on that node. To do this, request a single node from the Resource Manager. This is possible because when all processes run on the same host, shared memory segments are used to perform data exchange among processes, without involving the communication protocol.

   In this way, you can detect errors derived from bugs in the data exchange mechanisms. For example:

   - A process is blocked waiting for a message, with a given message type, and that message is never sent.
   - A process receives multiple messages with the same message type. This means that messages can be received in a different order from execution to execution, causing unpredictable results.
   - There may be errors due to the wrong use of the synchronization routines which can be detected in this phase.
3. Gradually increase the number of processes (and number of nodes) running the same program as before on two nodes, then on three and so on. This step is quite delicate, because problems related to the program synchronization can appear.

In both the second and third steps, standard debugging tools can still be successfully used, and the following two strategies may be followed:

1. The host program, or the program initiated by you, is started under a debugger (such as, **dbx**). Later on, the debugger may be attached to the node programs (or programs started by the IBM PVMe for AIX daemon), when they are running and waiting on a receive or a synchronization call. You need to obtain the *PID* (UNIX process identifier) for those processes and run the following command:

   ```
   dbx -a PID
   ```
2. The program started by you (possibly under **dbx**) invokes the **pvm_spawn** routine supplying the **PvmTaskDebug** option. For each new task that is started, two shell scripts, named **debugger** and **debugger2** are run. They open a window for each new task to be started. Each window will automatically start up **dbx**, loading the executable named in the **pvm_spawn** call.

The script files must be located in the same directory that was specified for the executables (**$HOME**/**pvm3**/**bin**/**RS6K**, or an alternate directory, if either the **PVMEPATH** environment variable or the **ep=** option have been set). Two templates for the script files are available under the /**usr**/**lpp**/**pvme**/**bin** directory,

and must be customized by you. For more information, see “Customizing the debugger and debugger2 Shell Scripts” on page 19.

Note: Since the underlying communications software used by IBM PVMe for AIX uses the signals SIGIO, SIGPIPE, and SIGALRM, it is suggested that you avoid using these signals in your application programs. You should keep this in mind when using a debugger. For example, you can ignore SIGIO, SIGPIPE, and SIGALRM signals when using **dbx** to debug an IBM PVMe for AIX program by using the following subcommands inside the debugger:

```
ignore io
ignore pipe
ignore alrm
```

You can also add these subcommands to the **.dbxinit** file.

Chapter 7. IBM PVMe for AIX Reference: C and FORTRAN Subroutines

This chapter contains an alphabetical listing of all the IBM PVMe for AIX subroutines. Each routine is described for both FORTRAN and C. The error conditions each routine may return are included as well as examples for both languages.

Using the FORTRAN Interface

The FORTRAN interface has been implemented as a set of subroutines that call the C routines. As a result, FORTRAN subroutines require an additional parameter where the return value of the C routines is set. This parameter is always the last argument passed to the subroutine.

The FORTRAN interface translates arguments, which are passed by reference in FORTRAN, to their values when needed by the C routines. It also takes into account the calling conventions that the FORTRAN compiler uses. For example, strings passed as arguments to a FORTRAN subroutine are not required to be explicitly null terminated by you, because the FORTRAN interface does it by itself.

Understanding Subroutine Format

Each routine may include the following sections as appropriate:

Description	A description of the major function of the routine.
Parameters	A list of variable or parameter definitions.
C or FORTRAN Syntax	A syntax diagram showing C or FORTRAN parameters and variables. **Note:** C or FORTRAN variable declarations can appear below the syntax diagrams.
C or FORTRAN Examples	Specific C AND FORTRAN examples.
Error Conditions	A list of error conditions.

Syntax Diagrams

Syntax diagrams are designed to provide information about how to call the routine (or subroutine) from your program. A syntax diagram can tell you:

- How many parameters you have to supply
- The data type of each parameter
- The data type of the return code

Notes:

1. FORTRAN is not case sensitive, therefore, you can enter the FORTRAN subroutines in either upper, lower, or mixed case. C is case sensitive, and the subroutines must be entered in lower case only.
2. In this book both C and FORTRAN syntax appear in all lower case.

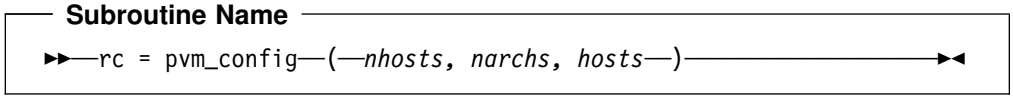
© Copyright IBM Corp. 1993, 1994, 1995

Reading Syntax Diagrams

- Read the diagrams from left-to-right, top-to-bottom, following the main path line. Double arrow heads (►►) mark the beginning of the syntax diagram.
- If a diagram is too long to fit on one line, the first line ends with a single arrowhead (►) and the second line begins with a single arrowhead (►).
- Facing arrowheads (►◄), mark the end of the syntax diagram.

A Sample Syntax Diagram

The following syntax diagram shows how a typical C or FORTRAN subroutine would look:




```
int *nhosts
int *ntasks
struct pvmhostinfo **hosts;
int rc;
```

Note: Variable declarations like those shown above may follow the C and FORTRAN syntax diagrams.

pvm_addhosts *** CURRENTLY DISABLED ***

Description

Note: A call to this routine will result in an error condition of PvmNotImpl.

The C routine allows you to add one or more hosts to the virtual machine configuration. The FORTRAN routine allows you to add a single host at each call. At any time your processes can call the **pvm_config** or the **pvm_mstat** routines to be informed about the virtual machine configuration.

When the full hostnames (for example, **hostA.domain**) are used, the routine strips the domain, so that only if the short aliases (for example, **hostA**) have been defined the routine is successful. The dotted decimal addresses (for example, 9.87.1.12) are not valid values to identify hosts.

The task identifiers for the daemon on the new hosts are returned, although no daemon processes are actually started on them. Such identifiers can be used in successive calls to the **pvm_spawn** and **pvm_tasks** routines and to any other routine where the identifier of the daemon on a given host is required.

The C routine also returns the number of hosts successfully added. If a failure occurred, this number is less than the value of the *nhosts* parameter. The error codes are stored in the proper elements of the *rcs* array. You can identify an error as a negative number, greater than -256, because the daemon *tids* are always negative numbers.

C Syntax

pvm_addhosts

```
►►—nh = pvm_addhosts—(—hosts, nhosts, rcs—)————————————►◄
```

```
char **hosts;
int nhosts;
int *rcs;
int nh;
```

FORTRAN Syntax

pvmfaddhost

```
►►—call pvmfaddhost—(—host, info—)——————————————————————►◄
```

```
character*64 host
integer info
```

Parameters

hosts
Array of pointers to strings where the names of the hosts to be added to the virtual machine are stored. The array must contain at least *nhosts* elements, and each string cannot be longer than 64 characters. *Hostnames* can be in mixed case.

host
Character string (no longer than 64 characters) containing the name of a single host to be added to the virtual machine. The *hostname* can be in mixed case.

nhosts
Number of hosts to be added to the virtual machine. The FORTRAN routine adds a single host with each call.

rcs
Points to an array of integers (at least *nhosts* long) where the return code for each host addition is returned. It contains the task identifier (*tid*) for the daemon on the remote host, if the addition was successful. It contains a negative value (but greater than -256) if it fails.

info
Return code for the host addition. It contains the task identifier (*tid*) for the daemon on the remote host, if the addition was successful. It contains a negative value (greater than -256) if it fails.

nh
The actual number of hosts added to the virtual machine. If all the additions were successful it is equal to the *nhosts* parameter.

C Example

```
char *newhosts[]={"hostA","hostB","hostC"};
int   rcs[3], nh, i;

if ((nh=pvm_addhosts(newhosts,3,rcs))!=3) {
   for (i=0; i<3; i++) {
       if (rcs[i]>-256)
          printf("Error starting host %d\n",i+1);
   }
}
```

FORTRAN Example

```
integer rc

call pvmfaddhost ('hostA', rc)
if (rc.lt.-256) then
   print *, 'Successfully added hostA, with tid',rc
endif
```

Error Conditions

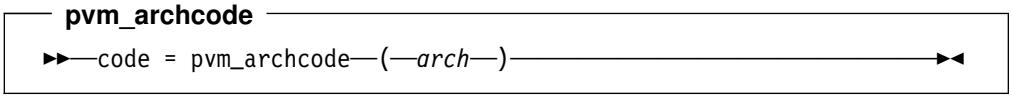
PvmSysErr	System error. The maximum number of hosts has been reached. No more sessions available on the host to be added. Process not enrolled.
PvmBadParam	*Hostname* longer than 64 characters.
PvmNoHost	Invalid *hostname*.
PvmDupHost	Host to be added is already running in the virtual machine.
PvmOutOfRes	Out of resources. Error starting the IBM PVMe for AIX program on the remote host.

pvm_archcode

Description

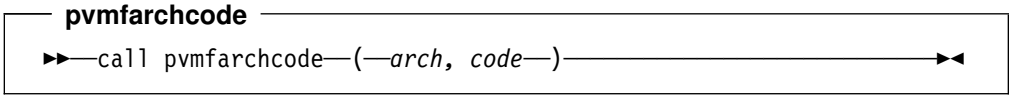
The **pvm_archcode** routine returns an integer associated with an architecture name. The code returned identifies machines with compatible binary data formats. In freeware PVM, which supports different architectures, this routine helps you to understand whether a data conversion to or from the XDR representation is really necessary when exchanging messages between processes running on different hosts. Since IBM PVMe for AIX only supports the RISC System/6000 architecture this routine is of little use; it has been implemented for compatibility reasons only. If an unknown string is supplied, a new entry will be created in the PVM architecture table, and it will be assigned a new integer code.

C Syntax




```
char *arch;
int code;
```

FORTRAN Syntax




```
character*4 arch
integer code
```

Parameters

arch
String describing a specific architecture (for example, *RS6K*).

code
Numerical code associated to the supplied architecture.

C Example

```
int code;

code = pvm_archcode ("RS6K");
```

FORTRAN Example

```
integer code

call pvmfarchcode ('RS6K', code)
```

Error Conditions

PvmSysErr System error.

pvm_barrier

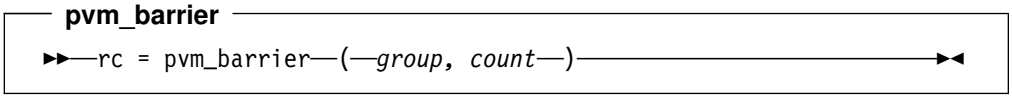
Description

The **pvm_barrier** routine blocks the calling processes until *count* processes of the same group call the routine with the same parameters. The **pvm_barrier** routine causes a set of processes belonging to the same group to block until a specified number of processes have called the same barrier. It is necessary that all processes belonging to the group be involved in the synchronization unless environment variable **PVMe_BARRTYPE=CENTRAL** is set.

If a `-1` is passed as *count* value, then the result of the **pvm_gsize** routine (the total number of members in the group), will be used.

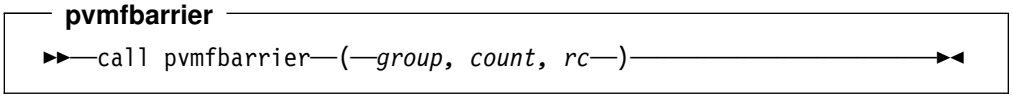
Note: If the group has not been synchronized since it was created, there is a chance that some tasks may not have enrolled. The result from **pvm_gsize** may not remain accurate for the duration of the barrier.

C Syntax




```
char *group;
int count;
int rc;
```

FORTRAN Syntax




```
character*32 group
integer count
integer rc
```

Parameters

group
Points to a string (no longer than 32 characters) containing the group name.

count
Number of processes participating with the barrier. Although not strictly required, the *count* parameter should contain the total number of members of the group. If *count* is `-1`, then the current group size is used.

rc
Return code. If the call was successful, it returns a `0` (zero). If the call was unsuccessful, it returns a negative value.

C Example

```
int   gtid, rc;

gtid = pvm_joingroup ("groupA");
 ....
rc = pvm_barrier ("groupA", -1);
```

FORTRAN Example

```
integer gtid, rc, nworkers
parameter (nworkers=12)

call pvmfjoingroup ("worker", rc)

if (rc.ge.0) then
   call pvmfbarrier ("workers", nworkers, rc)
endif
```

Error Conditions

PvmSysErr	System error. Process not enrolled.
PvmBadParam	*count* is less than -1.
PvmMismatch	Quorum mismatch with another process calling barrier. *count* is equal to 0.
PvmOverFlow	Group name longer than 32 characters.
PvmNullGroup	*group* is a NULL pointer.
PvmNotInGroup	*Group* does not exist. Task does not belong to *group*.

pvm_bcast

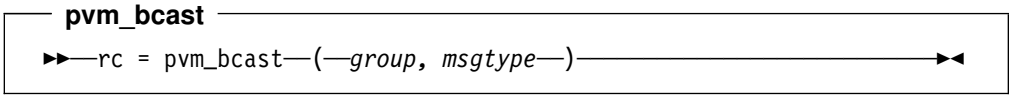
Description

The **pvm_bcast** routine broadcasts the message stored in the active send buffer to all the members of a given group. The sending process can either be a member of the group or not. In either case, the sending task will not receive the message.

pvm_bcast is a blocking asynchronous routine. It returns when the message has been delivered to all recipient tasks, even though the recipient tasks may not have called a receive routine (recipient tasks interrupt their processing to store incoming messages in a pre-receiving area). For large messages (greater than 2 Mb), **pvm_bcast** may block until the receiving task has started receiving the message.

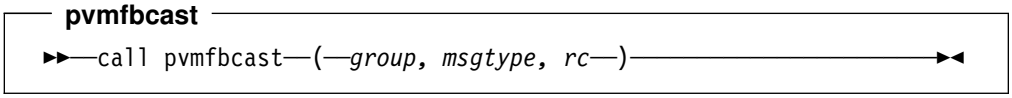
The **pvm_bcast** routine first determines the *tids* of the group members, then runs a multicast to these *tids*. If the group size is changed during the call, the change is not reflected in the broadcast.

C Syntax




```
char *group;
int msgtype;
int rc;
```

FORTRAN Syntax




```
character*32 group
integer msgtype
integer rc
```

Parameters

group
Points to a string (no longer than 32 characters) containing the group name.

msgtype
Integer tag supplied by you. It can take any positive value and is used to distinguish between different messages.

rc
Return code. If the call was successful, it returns a `0` (zero). If the call was unsuccessful, it returns a negative value.

C Example

```
int   gtid, rc;

pvm_initsend(PvmDataDefault);
 ...
gtid = pvm_joingroup ("all");
if ((rc=pvm_bcast("all",22))<0)
     printf("Error in bcast %d\n",rc);
```

FORTRAN Example

```
integer rc,msgtag
character*32 groupname

groupname='workers'
call pvmfbcast (groupname,msgtag,rc)
if (rc.eq.0) then
   print *, 'Message ', msgtag, 'broadcast to group ', groupname
endif
```

Error Conditions

PvmSysErr	System error. Process not enrolled.
PvmBadParam	*group* is a NULL pointer or *msgtype* is invalid.
PvmOverFlow	*group* is longer than 32 characters.

pvme_bcast

Description

The **pvme_bcast** routine broadcasts the message stored in the active send buffer to all the members of a given group. The sending process can either be a member of the group or not. If the sender task is a member of the group, every task must specify the sender's instance number within the given group. All the group members must invoke the routine. The semantics of this routine is similar to the syntax of the **pvm_reduce**, **pvm_scatter** and **pvm_gather** routines, and exploits a more efficient algorithm to distribute data when compared to **pvm_bcast**.

pvme_bcast is a blocking asynchronous routine. It returns when the message has been delivered to all recipient tasks, even though the recipient tasks may have not yet called the **pvme_bcast** routine. All the recipient tasks should also call **pvm_unpack** to unpack the data.

C Syntax

pvme_bcast

```
►►—rc = pvme_bcast—(—group, msgtype, rootinst—)————————►◄
```

```
char *group;
int msgtype;
int rootinst;
int rc;
```

FORTRAN Syntax

pvmefbcast

```
►►—call pvmefbcast—(—group, msgtype, rootinst, rc—)————————►◄
```

```
character*32 group
integer msgtype
integer rootinst
integer rc
```

Parameters

group
Points to a string (no longer than 32 characters) that contains the group name.

msgtype
Integer tag supplied by you. It can take any positive value and is used to distinguish between different messages.

rootinst
Instance number of the sender process. The instance number of the sender process must be `-1` if it is not a member of the group. Every task must use the same value for *rootinst*.

rc
Return code. If the call was successful, it returns a `0` (zero). If the call was unsuccessful, it returns a negative value.

C Example

```
int   ginst, rc;

ginst = 0;
pvm_initsend(PvmDataDefault);
 ...
gtid = pvm_joingroup ("all");
if ((rc=pvme_bcast("all", 22, ginst))<0)
     printf("Error in bcast %d\n",rc);
```

FORTRAN Example

```
integer rc, msgtag, root, parent
character*32 groupname

call pvmfparent (parent)
groupname='workers'
call pvmfgetinst (groupname, parent, root, rc)
call pvmefbcast (groupname, msgtag, root, rc)
if (rc.eq.0) then
   print *, 'Message ', msgtag, 'broadcast to group ', groupname
endif
```

Error Conditions

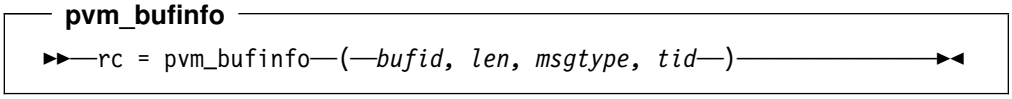
PvmSysErr	System error. Process not enrolled.
PvmBadParam	*group* is a NULL pointer. Invalid *msgtype*. *rootinst* -1.
PvmOverFlow	*group* is longer than 32 characters.

pvm_bufinfo

Description

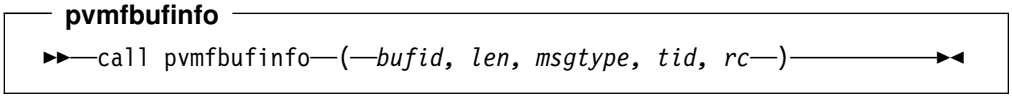
The **pvm_bufinfo** routine returns information about the message received last in the supplied *bufid*. It returns the message size, the message type and the identifier of the sending process. It is useful when the application receives any incoming message without following an established order. At any time, the action to be taken depends on the message type associated with the incoming message.

C Syntax




```
int bufid;
int *len;
int *msgtype;
int *tid;
int rc;
```

FORTRAN Syntax




```
integer bufid
integer len
integer msgtype
integer tid
integer rc
```

Parameters

bufid
Positive integer identifying a particular receive buffer.

len
Points to an integer where the total message length (in bytes) is returned. A NULL value may be supplied if you do not need this information.

msgtype
Points to an integer where the routine returns the message type of the last message received in the supplied *bufid*. A NULL value may be supplied if you do not need this information.

tid
Points to an integer where the *tid* of the sending process is returned. A NULL value may be supplied if you do not need this information.

rc
Return code. If the call was successful, it returns a `0` (zero). If the call was unsuccessful, it returns a negative value.

C Example

```
int type, who, len, rc;
int bufid;
  ...
bufid = pvm_recv (-1,-1);
rc = pvm_bufinfo(bufid, &len, &type, &who);
if (!rc)
   printf  ("Received message %d from %d; It is %d bytes long\n",
                          type, who, len);
    ....
```

FORTRAN Example

```
integer from, type, len, info
integer buffer
  ...
call pvmfrecv (-1, -1, buffer);
call pvmfbufinfo  (buffer, len, type, from, info)
if (info.eq.0) then
   print *, 'Message ', type, 'coming from ', from
   print *, 'has been received; it was', len 'bytes long'
endif
```

Error Conditions

PvmSysErr	System error. Process not enrolled.
PvmNoBuf	Invalid buffer identifier. No message in the supplied buffer. Buffer identifier is less than 1.
PvmBadParam	Invalid *msgtype*. *rootinst* -1.

pvm_catchout

Description

By default, the *stdout* and *stderr* of IBM PVMe for AIX tasks (that have been started by the **pvm_spawn** routine) are redirected to the *stdout* of the daemon. The **pvm_catchout** routine allows an IBM PVMe for AIX process to catch the output and error of the child tasks, (all the processes that will be spawned after a call to this routine).

stdout and *stderr* from the children are collected by the parent task, which tags each line so that each line of output appears as follows:

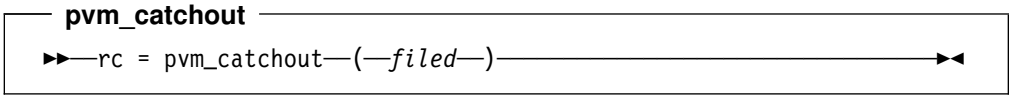
```
[txxxx] (text from child task)
```

where *txxxx* is the children's task identifier.

In C programs, you can specify a file descriptor as the target for the collected output and errors. If the file descriptor points to /**dev**/**null**, output and error messages from child tasks are discarded.

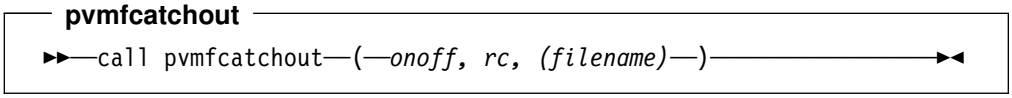
In FORTRAN, you can set output and error collection to *on* (1) or *off* (0). Setting the collection to *on* causes output and error from the child tasks to be logged to the *stdout* of the parent. Setting the collection to *off* causes output and error from the child tasks to be discarded. You can also provide a special value (-1) followed by a file name, so that output and error from the child tasks are printed in the specified file. For **pvm_catchout** to work properly the **PVMe_IP** environment variable must be set to *yes* before the daemon is started.

C Syntax




```
FILE *filed;
int rc;
```

FORTRAN Syntax




```
integer onoff
character(*) filename
integer rc
```

Parameters

filed
File descriptor where output and error from child tasks will be collected. A **NULL** pointer sets output and error collection to *off*.

onoff
Flag used to set the output and error collection from child tasks in FORTRAN programs to *on* and *off*. It can be any of the following values:

0	turn off
1	turn on
-1	turn on and print output and error in *filename*

filename
This parameter is only required if *onoff* is `-1`, and indicates the filename where output and error from the other tasks have to be printed.

rc
Return code. If the call was successful, it returns a `0` (zero). If the call was unsuccessful, it returns a negative value.

C Example

```
FILE *filed;
int rc;
  .....
  if ( (filed=fopen(filename, "w")) != NULL ) {
    pvm_catchout(filed);
  }
```

FORTRAN Example

```
      integer rc

      call pvmfcatchout (1, rc)
```

Error Conditions

PvmSysErr	System error. Process not enrolled.
PvmBadParam	Invalid file descriptor.

pvm_config

Description

The **pvm_config** routine returns information about the hosts participating with the virtual machine. The number of hosts and the number of different architectures are returned.

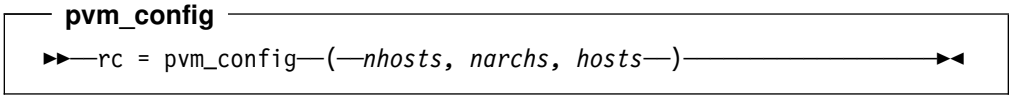
Note: The **narchs** parameter (for the number of different architectures) is always set to 1; it has been introduced only for compatibility with freeware PVM.

The C routine returns an array of structures, each of them containing information about a single host (memory allocation is done by the routine itself). When finished with the data, the user may **free** each **.hi_name** and **.hi_arch** in the array, and the array itself.

The FORTRAN routine returns data at each call for a single host, and cycles through all the hosts. If called *nhosts* times, it returns the data of all the hosts, one at each call.

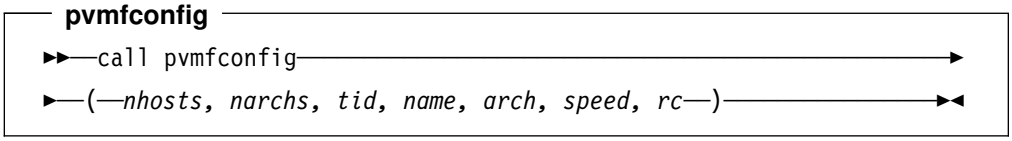
Note: Data returned from this routine is the same as that shown by the **conf** command from the IBM PVMe for AIX console.

C Syntax




```
int *nhosts;
int *narchs;
struct pvmhostinfo **hosts;
int rc;
```

FORTRAN Syntax




```
integer nhosts
integer narchs
integer tid
character(*) name
character(*) arch
integer speed
integer rc
```

Parameters

nhosts
Integer where the routine returns the number of active hosts.

narchs
Integer where the routine returns the number of different architectures in the virtual machine. It is always 1, because the IBM PVMe for AIX program runs on a homogeneous platform.

hosts
Address of an array of *pvmhostinfo* structures. The array will be *nhosts* long. Each structure contains data about a single host, and has the following shape:

```
struct pvmhostinfo {
   int   hi_tid                  /* host id            */
   char *hi_name                 /* host name          */
   char *hi_arch                 /* host architecture  */
   int   hi_speed                /* relative CPU speed */
} taskp
```

tid
Integer where the host identifier of a particular host is returned (see the *Description* on page 109).

name
Character string where the *hostname* is returned (see the *Description* on page 109).

arch
Character string where host architecture is returned (it will be always RS6K).

speed
Relative CPU speed across the virtual machine. Because all the hosts have the same architecture, it assumes the same value for all the hosts.

rc
Return code. If the call was successful, it returns a 0 (zero). If the call was unsuccessful, it returns a negative value.

C Example

```
int nhosts, narchs, rc;
struct pvmhostinfo *hosts;
  .....
if (!(rc = pvm_config (&nhosts, &narchs, &hosts))) {
  printf ("%d hosts are currently running\n", nhosts);
  printf ("   HTID    HNAME     HARCH     SPEED\n");
  for (i=0; i<nhosts; i++) {
    printf ("%8d %12s %12s %8d %8d\n", hosts->hi_tid, hosts->hi_name,
               hosts->hi_arch, hosts->hi_speed);

    hosts++;
  }
}
```

FORTRAN Example

```
      integer nhosts, narchs, rc, i
      integer htid(MAX), speeds(MAX)
      character*32 names(MAX), arches(MAX)
          .....
      i=1
      print *, '   HTID     NAME     ARCH      SPEED'
 66   call pvmfconfig (nhosts, narchs, htid(i), names(i), arches(i),
     $                 speed(i))
      print *, htid(i), name(i), arches(i), speed(i)
      i=i+1
      if (i.le.nhosts) goto 66

      print *, nhosts ,'are currently running'
```

Error Conditions

PvmSysErr System error. Process not enrolled.

pvm_delhosts *** CURRENTLY DISABLED ***

Description

Note: A call to this routine will result in an error condition of PvmNotImpl.

The C routine allows you to delete one or more hosts from the virtual machine. The FORTRAN routine allows you to delete a single host at each call. At any time, your processes can call the **pvm_config** or the **pvm_mstat** routines to get information about the virtual machine configuration.

When the full *hostnames* (for example, **hostA.domain**) are used, the routine strips the domain, so that the routine is only successful if the short aliases (for example, **hostA**) have been defined. The dotted addresses (for example, 9.87.1.12) are not valid values to identify hosts.

C Syntax

pvm_delhosts

```
►►—nh = pvm_delhosts—(—hosts, nhosts, rcs—)————————————►◄
```

```
char **hosts;
int nhosts;
int *rcs;
int nh;
```

FORTRAN Syntax

pvmfdelhost

```
►►—call pvmfdelhost—(—host, info—)——————————————————►◄
```

```
character*64 host
integer info
```

Parameters

hosts
Array of pointers to strings where the names of the hosts to be deleted from the virtual machine are stored. The array must contain at least *nhosts* elements, and each string cannot be longer than 64 character. *Hostnames* can be in mixed case.

host
Character string (no longer than 64 characters) containing the name of a single host to be deleted from the virtual machine. The *hostname* can be in mixed case.

nhosts
Number of hosts to be deleted from the virtual machine. The FORTRAN routine deletes a single host with each call.

rcs
Points to an array of integers (at least *nhosts* long) where the return code for each host deletion is returned. Each element of the array is 0 if the deletion of

the corresponding host was successful. It is a negative number if it was unsuccessful.

info
Return code for the host deletion. If the addition was successful, it is `0`. It is a negative value if it fails.

nh
The actual number of hosts deleted from the virtual machine. If all the deletions were successful it is equal to the *nhosts* parameter.

C Example

```
char *hosts[]={"hostA","hostB","hostC"};
int   rcs[3], nh, i;

if ((nh=pvm_delhosts(&hosts[1],2,rcs[1]))!=2) {
   for (i=1; i<3; i++) {
       if (rcs[i]<0)
          printf("Error deleting host %d\n",i+1);
   }
}
```

FORTRAN Example

```
integer rc
character*32 hostname

hostname='rs6000_01'
call pvmfdelhost (hostname, rc)
if (rc.eq.0) then
   print *, 'Successfully deleted ' hostname
endif
```

Error Conditions

PvmSysErr	System error. Process not enrolled.
PvmBadParam	*count* less than `0`. Attempt to delete the master host.
PvmNoHost	Invalid *hostname*.

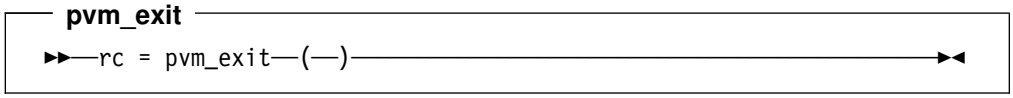
pvm_exit

Description

The **pvm_exit** routine informs the daemon that the process is leaving the IBM PVMe for AIX program environment. It should be called by all the IBM PVMe for AIX processes before they stop or exit.

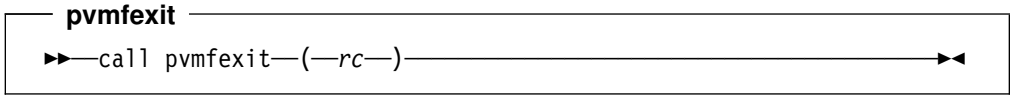
This routine does not kill the process, which can continue to run tasks just like any other serial UNIX process. Processes that leave the IBM PVMe for AIX program may enroll again. Entries in the processes database are reused. If a process calls the **pvm_exit** routine and then enrolls again, (and no other processes enrolled or exited in the meantime), it receives the same task identifier.

C Syntax




```
int rc;
```

FORTRAN Syntax




```
integer rc
```

Parameters

rc
Return code. If the call was successful, it returns a `0` (zero). If the call was unsuccessful, it returns a negative value.

C Example

```
pvm_exit();
exit();
```

FORTRAN Example

```
call pvmfexit(rc)
stop
```

Error Conditions

PvmSysErr System error. Process not enrolled.

pvm_freebuf

Description

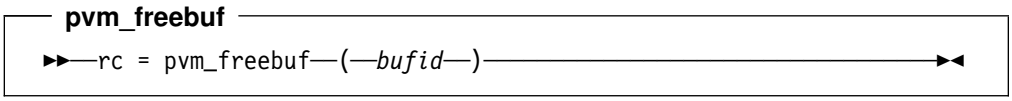
The **pvm_freebuf** routine unlocks a buffer and makes it available again to the IBM PVMe for AIX routines. The buffer identifier must be supplied.

This routine should be used to free:

- Send buffers created by the **pvm_mkbuf** routine, that are no longer needed
- Send buffers created by the **pvm_initsend** routine that are no longer needed
- Buffers that were saved with the **pvm_setsbuf** and **pvm_setrbuf** routines

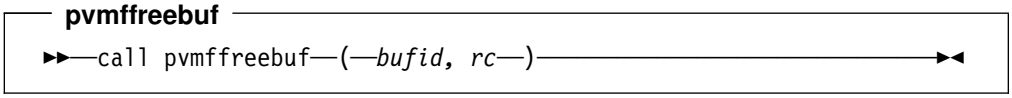
Most of your applications do not need to manage multiple buffers. If multiple buffers are not required, the default send and receive buffers are automatically created by the IBM PVMe for AIX routines.

C Syntax




```
int rc;
int bufid;
```

FORTRAN Syntax




```
integer rc
integer bufid
```

Parameters

bufid
: Positive integer identifying the buffer to be unlocked.

rc
: Return code. If the call was successful, it returns a `0` (zero). If the call was unsuccessful, it returns a negative value.

C Example

```
int newbuf, oldbuf;

newbuf = pvm_mkbuf(0);
oldbuf = pvm_setsbuf(newbuf);
  ...
  ...                       /* pack and send a message */
rc = pvm_freebuf(newbuf)
```

FORTRAN Example

```
integer rc, buff, oldbuf

call pvmfmkbuf(PVMDEFAULT,buff)
call pvmfsetsbuf(buff,oldbuf)
  ....
  ....                    ! pack and send a message
call pvmffreebuf(buff,rc)
```

Error Conditions

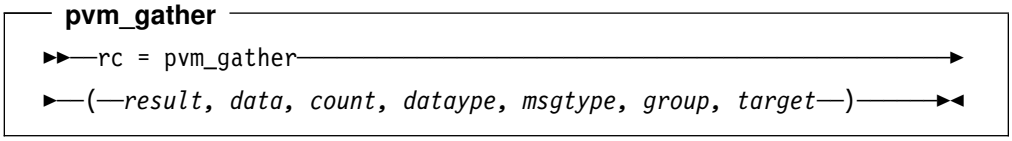
PvmSysErr	System error. Process not enrolled.
PvmBadParam	Invalid *bufid* is less than 0.
PvmNoSuchBuf	Buffer in non-exsistent or has already been de-allocated.

pvm_gather

Description

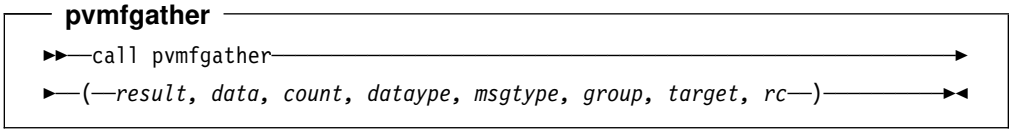
The **pvm_gather** routine gathers data from each member of a group to a single member of the same group. All group members must call the routine; each member sends *count* items of its *data* array of *datatype* data to the target process. The target process concatenates incoming messages in order, on the basis of the group instance number of the sender task. The end result is that data from all group members will be stored in a single array on the target process.

C Syntax




```
int rc;
void *result;
void *data;
int count;
int datatype;
int msgtype;
char *group;
int target;
```

FORTRAN Syntax




```
character(*) result
character(*) data
integer count
integer datatype
integer msgtype
character(*) group
integer target
integer rc
```

Parameters

result
: Pointer to an array of *datatype*, containing at least *count * group_size* elements. The data that comes from all group members will be stored in the array on the *target* process. This parameter is significant only on the *target* task.

data
: Pointer to an array of *datatype*, containing the *count* elements to be sent to the *target* task.

count
Number of items of the *data* array that each group member is going to send.

datatype
Integer that specifies the type of entries in the *data* and *result* arrays. It may be any of the following values (values in parenthesis are for use in FORTRAN programs):

PVM_BYTE (BYTE1)	1
PVM_SHORT (INTEGER2)	2
PVM_INT (INTEGER4)	3
PVM_FLOAT (REAL4)	4
PVM_CPLX (COMPLEX8)	5
PVM_DOUBLE (REAL8)	6
PVM_DCPLX (COMPLEX16)	7
PVM_LONG	8

msgtype
Positive integer used to identify the message.

group
Character string that identifies an existing group.

target
Instance number within the *group* of the process performing data collection. Every task must specify the same target.

rc
Return code. If the call was successful, it returns a `0` (zero). If the call was unsuccessful, it returns a negative value.

C Example

```
int msg;
double workarea[MAXDATA*MAXPROCS];
double myarea[MAXDATA];

info = pvm_gather (&workarea, &myarea, MAXDATA, PVM_DOUBLE,
                   msg, "all", 0);
```

FORTRAN Example

```
integer cols, ncols, MAXPROCS
parameter (MAXPROCS=16, ncols = 1024, cols = ncols*MAXPROCS)
    .....
integer info
integer matrix (rows, cols), data (rows, ncols)

call pvmfgather (matrix, data, rows*ncols, INTEGER4, 1000,
                 "base", 0, info)
```

Error Conditions

PvmSysErr	System error. Process not enrolled.
PvmBadParam	*data* is a NULL pointer. *result* is a NULL pointer on the *target* task. *count* less than zero. *msgtype* negative. *datatype* is not a supported type.

PvmOverFlow	Group name longer than 32 characters.
PvmNoMem	Not enough memory.
PvmNullGroup	*group* name is NULL.
PvmNoGroup	Group named *group* does not exist.
PvmNoInst	Calling process, or process identified with *target* instance number does not exist within the specified group.

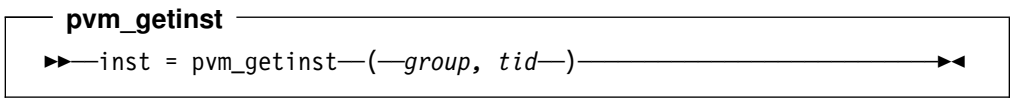
pvm_getinst

Description

The **pvm_getinst** routine queries for the instance number of a given process within a given group. The process identified by *tid* must already be a member of *group*.

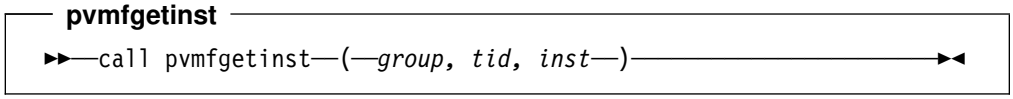
If the same process joins and leaves the same group multiple times, it is not guaranteed to receive the same instance number within the group. It receives the same instance number only if no other processes join or leave the same group in the meantime.

C Syntax




```
char *group;
int tid;
int inst;
```

FORTRAN Syntax




```
character*32 group
integer tid
integer inst
```

Parameters

group
String (no longer than 32 characters) identifying an existing group.

tid
Task identifier of a task belonging to the supplied group.

inst
Return code. Instance number (positive integer) of the *tid* process within the supplied group. A negative number indicates an error.

C Example

```
int pgtid, parent;

parent = pvm_parent ();
if ((pgtid = pvm_getinst ("all", parent))<0)
   printf ("Parent is not in group\n");
```

FORTRAN Example

```
integer inst, ltid

call pvmfgetinst ("workers", ltid, inst)
```

Error Conditions

PvmSysErr	System error. Process not enrolled.
PvmOverFlow	Group name longer than 32 characters.
PvmNullGroup	*group* name is NULL.
PvmNoGroup	Group named *group* does not exist.
PvmNotInGroup	Process identified by *tid* does not belong to the supplied group.
PvmBadParam	Invalid *tid*.

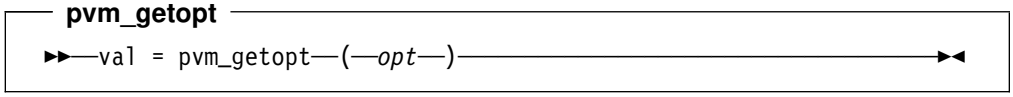
pvm_getopt

Description

The **pvm_getopt** routine allows you to obtain the current value of an option. The following options are available:

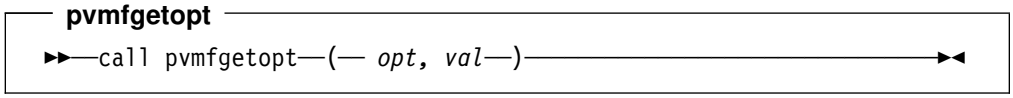
- **PvmRoute** is included only to maintain compatibility with freeware PVM, where it allows you to select direct communication links (TCP sockets) among tasks. The syntax is exactly the same, but since data exchange between tasks always exploits direct process-to-process links, this option has no effect with IBM PVMe for AIX.
- **PvmAutoErr** enables and disables automatic error printings whenever an error occurs in an IBM PVMe for AIX routine. The **pvm_serror** routine has the same effect.
- **PvmDebugMask** is included for compatibility with freeware PVM, where it is used to set debugging to *on* or *off*. When debugging is *on*, PVM logs detailed information about its operations on the daemon's *stderr stream*. *val* is the debugging level. This option has no effect with IBM PVMe for AIX. `0` redirects stdout to the IBM PVMe for AIX daemon. The **PvmTraceTid** to `0` to `0` redirects stdout to the stdout to `0` discards trace data. The default

C Syntax




```
int val;
int opt;
```

FORTRAN Syntax




```
integer val
integer opt
```

Parameters

opt

Integer that identifies the option. The following options can be set:

- **PvmRoute**
- **PvmDebugMask**
- **PvmAutoErr**

IBM PVMe for AIX does not currently support the following options:

- **PvmOutputTid**
- **PvmOutputCode**
- **PvmTraceTid**
- **PvmTraceCode**

- **PvmSelfOutputTid**
- **PvmSelfOutputCode**
- **PvmSelfTraceTid**
- **PvmSelfTraceCode**

val
Return code. If the call was successful, it returns a positive number that identifies the current value of the option. If the call was unsuccessful, it returns a negative value.

C Example

```
int val;

val = pvm_getopt(PvmAutoErr);
```

FORTRAN Example

```
integer info

call pvmfgetopt(PvmAutoErr, info)
```

Error Conditions

PvmSysErr	System error. Process not enrolled.
PvmBadParam	Invalid *opt*.

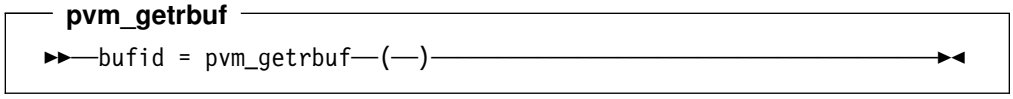
pvm_getrbuf

Description

The **pvm_getrbuf** routine returns the identifier of the current receive buffer. In the IBM PVMe for AIX program, only one receive buffer is active at any time.

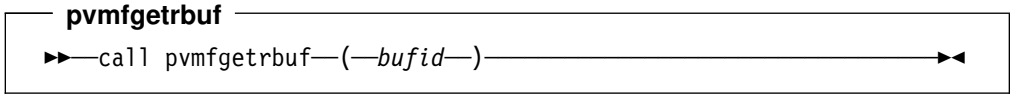
If there is no active buffer, the routine returns `0`. If a receive is attempted in this condition, the receiving routines set a new active buffer.

C Syntax




```
int bufid;
```

FORTRAN Syntax




```
integer bufid
```

Parameters

bufid
Return code. Buffer identifier of the current receive buffer.

C Example

```
int curbuf, buf_to_upk;

buf_to_upk = pvm_recv(..)
   ....

curbuf = pvm_getrbuf();
if (curbuf==buf_to_upk) {
   pvm_upkint(..)
    ...
}
```

FORTRAN Example

```
integer curbuf

call pvmfgetrbuf(curbuf)
if (curbuf.eq.0) then
   print *, 'No active receive buffer'
endif
```

Error Conditions

PvmSysErr	System error. Process not enrolled.

pvm_getsbuf

Description

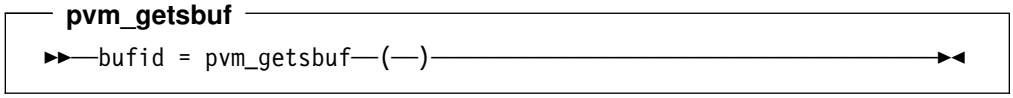
The **pvm_getsbuf** routine returns the identifier of the current send buffer.

In the IBM PVMe for AIX program only one send buffer is active at any time, although there can be multiple buffers in use. The active buffer is where the next **pvm_pk***xxx* routine packs data and the next **pvm_send** routine delivers to the recipient task.

The active buffer is the buffer returned by the last **pvm_initsend** routine, or the buffer that your program set by invoking a **pvm_setsbuf** routine.

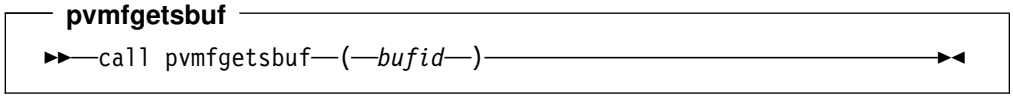
Note: If there is no active buffer, the routine returns `0`.

C Syntax




```
int bufid;
```

FORTRAN Syntax




```
integer bufid
```

Parameters

bufid
Return code. Buffer identifier of the current send buffer.

C Example

```
int curbuf;

   ....

curbuf = pvm_getsbuf();
if (curbuf==0) {
   printf ("No active send buffer\n");
    ...
}
```

FORTRAN Example

```
integer curbuf

call pvmfgetrbuf(curbuf)
print *, curbuf, 'is the current active buffer'
```

Error Conditions

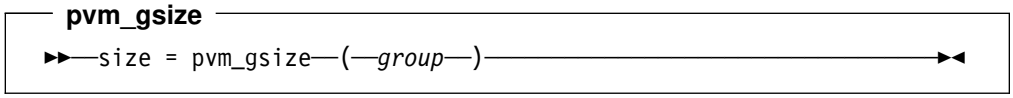
PvmSysErr	System error. Process not enrolled.

pvm_gsize

Description

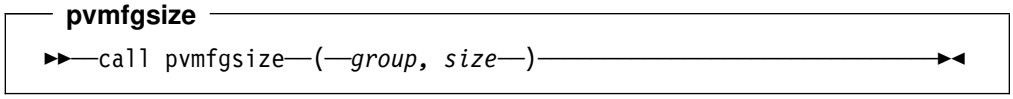
The **pvm_gsize** routine queries for the current number of members of the supplied group. Group size can change during the program execution.

C Syntax




```
char *group;
int size;
```

FORTRAN Syntax




```
character*32 group
integer size
```

Parameters

group
String (no longer than 32 characters) identifying an existing group.

size
Return code. Number of members of the supplied group. A negative number indicates an error.

C Example

```
int size, gtid;

gtid = pvm_joingroup ("workers");
  ....
size = pvm_gsize ("workers");
```

FORTRAN Example

```
integer size

call pvmfgsize ("workers", size)
```

Error Conditions

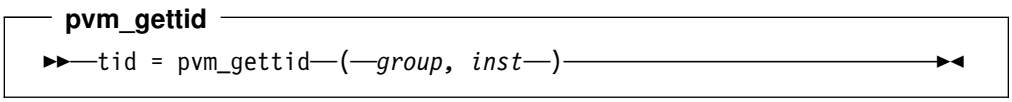
PvmSysErr	System error. Process not enrolled.
PvmOverFlow	Group name longer than 32 characters.
PvmNullGroup	*group* name is NULL.
PvmNoGroup	Group named *group* does not exist.

pvm_gettid

Description

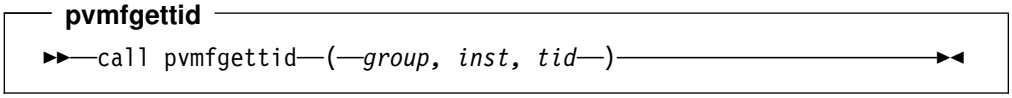
The **pvm_gettid** routine queries for the task identifier of the process that has the instance number *inst* within the group *group*.

C Syntax




```
char *group;
int inst;
int tid;
```

FORTRAN Syntax




```
character*32 group
integer inst
integer tid
```

Parameters

group
String (no longer than 32 characters) identifying an existing group.

inst
Instance number of a process within the group *group*.

tid
Return code. Task identifier (positive integer) of the process with the instance number *inst* within the supplied group. A negative number indicates an error.

C Example

```
int up_tid, my_gtid;

my_gtid = pvm_joingroup ("workers");
  ....
up_tid = pvm_gettid ("workers", my_gtid+1);
if (up_tid > 0)
   pvm_send (up_tid, msgtype);
```

FORTRAN Example

```
integer ltid, rtid, ..

call pvmfgettid ("workers", 0, tid)
```

Error Conditions

PvmSysErr	System error. Process not enrolled.
PvmOverFlow	Group name longer than 32 characters.
PvmNullGroup	*group* name is NULL.
PvmNoGroup	Group named *group* does not exist.
PvmNoInst	Process identified with *inst* instance number does not exist within the supplied group.
PvmBad Param	Invalid *inst*.

pvm_gettmask *** CURRENTLY DISABLED ***

Description

Note: A call to this routine will result in an error condition of PvmNotImpl.

The **pvm_gettmask** routine gets the current trace mask for the calling process or for its children. The trace mask is used to enable tracing of calls to the IBM PVMe for AIX routines. Each process that is started by the **pvm_spawn** routine inherits the trace mask from the parent. For processes started by other means, the trace mask is set to `0`. When calls are made to functions in **libpvm**, the parameters and results are sent in the form of messages to the trace sink of the task (the variable **PvmTraceTid** is set by calling **pvm_setopt**). By calling **pvm_settmask**, the trace mask can be modified for either the calling process or the child tasks that will be spawned. Setting the child trace mask has no effect on child tasks that are already running.

The trace mask can be kept in a temporary variable, declared as:

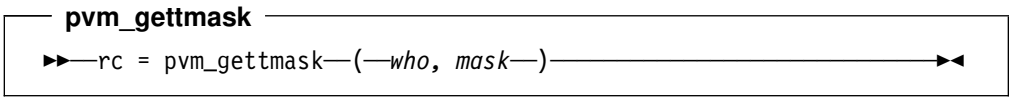
- Pvmtmask mymask;

The **Pvmtmask** datatype is defined in **pvmtev.h**.

Generating trace data - PVMe supports the generation of post-mortem tracefiles in SDDF format compatible with PVM3.3 and XPVM. To enable tracefile creation, use the **-trace** command line argument when starting up the pvmd3e daemon (See “Using the -trace Option” on page 35). By default, all trace information will be sent to to the file `/tmp/xpvm.trace.$USER` on the node on which the daemon is running, but if the environment variable **PVMe_TRACEFILE** is set before starting the daemon, its value will be used for the name of the tracefile.

XPVM is a freeware tool used for graphically displaying PVM3.3 tracefiles. A version of XPVM pre-compiled on the RISC System/6000 architecture can be downloaded via anonymous ftp from netlib2.cs.utk.edu in the directory `/pvm3/xpvm`.

C Syntax




```
int who;
Pvmtmask mask;
int rc;
```

Parameters

who

Specifies whether it is the calling process or the child task that requires the trace mask. It can be either of the following values:

PvmTaskSelf	The current task
PvmTaskChild	Child tasks to be spawned

mask
Trace mask which is owned by the caller and is filled in with the current trace vector.

rc
Return code. It returns `0` if the call was successful, a negative value if it was unsuccessful.

C Example

```
Pvmtmask mask;
int rc;

rc = pvm_gettmask (PvmTaskChild, mask );
```

Error Conditions

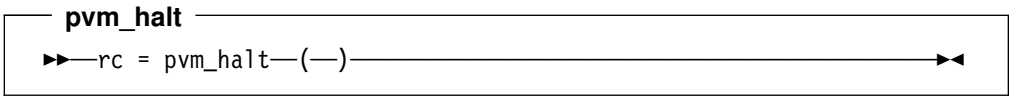
PvmSysErr	System error. Process not enrolled.
PvmBadParam	Invalid *who* flag.

pvm_halt

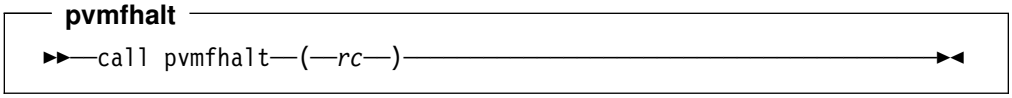
Description

The **pvm_halt** routine shuts the entire IBM PVMe for AIX system down including the remote tasks, the IBM PVMe for AIX daemon, and the local tasks.

C Syntax



FORTRAN Syntax



integer rc

Parameters

rc
Return code. If the call was successful, it returns a `0` (zero). If the call was unsuccessful, it returns a negative value.

C Example

```
int rc;

rc = pvm_halt ();
```

FORTRAN Example

```
integer info
call pvmfhalt (info)
```

Error Conditions

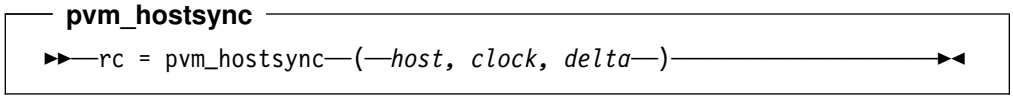
PvmSysErr System error. Process not enrolled.

pvm_hostsync

Description

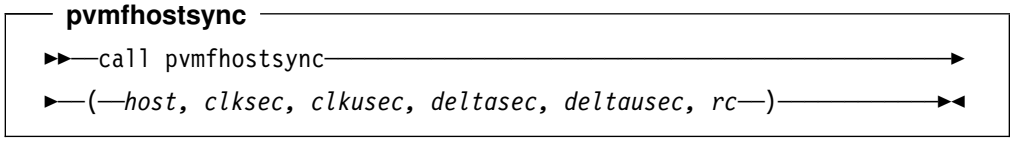
The **pvm_hostsync** routine returns the time-of-day of a remote host in the virtual machine, as well as the difference between the time-of-day on the local and remote host. This routine relies on the assumption that all the hosts are synchronized so that the time-of-day is the same on the remote and the local host, and the difference above is negligible. Note that this assumption will not hold when one or more workstations, external to the RISC System/6000 SP system, participate with the virtual machine. For RISC System/6000 SP systems, it is expected that all the nodes, as well as the control workstation, will be synchronized.

C Syntax




```
int host;
struct timeval *clock;
struct timeval *delta;
```

FORTRAN Syntax




```
integer host
integer clksec
integer clkusec
integer deltasec
integer deltausec
integer rc
```

Parameters

host
Identifier (TID) of the remote host.

clock
Pointer to a *timeval* struct (C programs only). If the pointer is not NULL, the *tv_sec* and *tv_usec* fields of the structure contain the time-of-day for the host.

clksec, clkusec
In FORTRAN programs, these two integers contain the time-of-day for the host.

delta
Pointer to a *timeval* struct (C programs only). If the pointer is not NULL, the *tv_sec* and *tv_usec* fields of the structure contain the difference between the time-of-day on the local and remote hosts.

deltasec, deltausec
In FORTRAN programs, these two integers contain the difference between the time-of-day on the local and remote hosts.

rc
Return code. It returns a `0` if the call was successful; a negative value if it was unsuccessful.

C Example

```
int rc, parent;
struct timeval tim;

parent = pvm_parent();
rc = pvm_hostsync (parent, &tim, NULL);
```

FORTRAN Example

```
integer*4 clsec, clusec, dsec, dusec, thishost, me

call pvmfmytid (me)
call pvmftidtohost (me, thishost)
call pvmfhostsync (thishost, clsec, clusec, dsec, dusec)
```

Error Conditions

PvmSysErr System error. Process not enrolled.

pvm_initsend

Description

The **pvm_initsend** routine clears the current send buffer (if any) and returns its *bufid*. If there is no active buffer, the routine prepares a new buffer, and returns its identifier.

If the **pvm_initsend** routine is not called after a **pvm_send** routine, the result of successive packing operations is the append of new data to the previous message. If a **pvm_send** routine is called after a **pvm_initsend** routine, but before any pack operation, a zero-length message is sent.

The *encoding* parameter can assume three different values: two of them (**PvmDataRaw** and **PvmDataDefault**) were used in freeware PVM to enable/disable data conversion to/from the XDR format. Both of them are supported in the IBM PVMe for AIX program and because of the homogeneous environment, they actually produce the same effect. The last value (**PvmDataInPlace**) indicates that data must not be copied into the send buffer during the packing operation. Only the pointer to the data and the size are written into the buffer, while data is copied directly out of your memory only when a send is performed. This option allows you to avoid one data copy, with a substantial improvement in performance. Also, the same data can be sent multiple times, with modifications intervening between the sends, without having to pack it each time.

C Syntax

pvm_initsend

```
►►—bufid = pvm_initsend—(—encoding—)———————————————►◄
```

```
int encoding;
int bufid;
```

FORTRAN Syntax

pvmfinitsend

```
►►—call pvmfinitsend—(—encoding, bufid—)———————————►◄
```

```
integer encoding
integer bufid
```

Parameters

encoding
Integer flag which can take any of the following values (values in parenthesis are to be used in FORTRAN programs):

PvmDataDefault (PVMDEFAULT)	0
PvmDataRaw (PVMRAW)	1
PvmDataInPlace (PVMINPLACE)	2

bufid
Return code. Positive integer identifying the current send buffer. If it fails, it is a negative number.

C Example

```
int bufid;

bufid = pvm_initsend (PvmDataDefault);
```

FORTRAN Example

```
integer sbuf

call pvmfinitsend (PVMDEFAULT,sbuf)
print *, sbuf, 'is the current active buffer'
```

Error Conditions

PvmSysErr	System error. Process not enrolled.
PvmBadParam	Invalid *encoding* value.

pvm_joingroup

Description

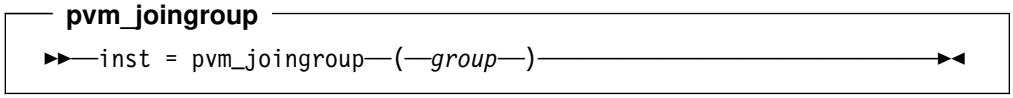
The **pvm_joingroup** routine enrolls the calling process to the supplied *group* and returns its instance number within the group. If the named *group* does not exist, the routine creates it.

Instance numbers start from `0` and increment by one for each new process that joins the supplied group. If a task leaves the group and joins it again later, it is not guaranteed that it will get the same instance number. This is because the IBM PVMe for AIX program reuses entries in the groups database when they are no longer needed.

Notes:

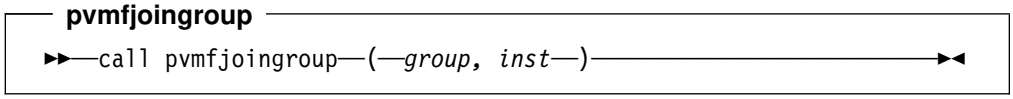
1. At any time, a process can belong to one group, multiple groups, or to no group.
2. The default maximum group size is 128. See "Customizing Your Installation" on page 19.
3. The default maximum number of groups is 8. See "Customizing Your Installation" on page 19.

C Syntax




```
char *group;
int inst;
```

FORTRAN Syntax




```
character*32 group
integer inst
```

Parameters

group
String (no longer than 32 characters) containing the group name.

inst
Return code. Instance number (non-negative integer) of the process within the group, if the call was successful. If the call was unsuccessful, it returns a negative number.

C Example

```
int gtid;

gtid = pvm_joingroup ("all");
```

FORTRAN Example

```
integer gid

call pvmfjoingroup ("workers", gid)
if (gid.ge.0) then
  print *, 'Successfully joined group, gid=',gid
endif
```

Error Conditions

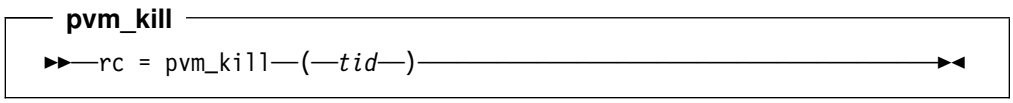
PvmSysErr	System error. Process not enrolled.
PvmOverFlow	*group* longer than 32 characters.
PvmNulGroup	*group* equal to NULL.
PvmOutOfRes	group size or number of groups exceeds maximum.
PvmDupGroup	The calling process is already member of *group*.

pvm_kill

Description

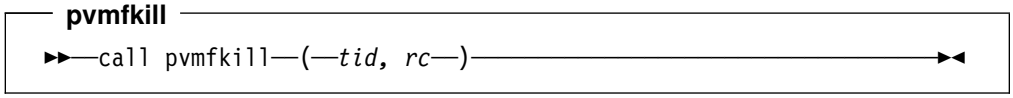
The **pvm_kill** routine kills the IBM PVMe for AIX process identified by *tid*. The **pvm_kill** routine is not to be used by a process to fill itself. To do that, call the **pvm_exit (pvmfexit)** routine followed by an **exit (STOP)**.

C Syntax




```
int tid;
int rc;
```

FORTRAN Syntax




```
integer tid
integer rc
```

Parameters

tid
Task identifier of the process to be killed.

rc
Return code. If the call was successful, it returns a `0` (zero). If the call was unsuccessful, it returns a negative value.

C Example

```
int insts[MAXPROC]
int rc,nproc=MAXPROC;

for (i=0; i<nproc; i++) {
   rc = pvm_kill (insts[i]);
}
```

FORTRAN Example

```
parameter (MAXPROC=32)
integer insts(MAXPROC), info

do i=1,MAXPROC
  call pvmfkill (insts(i), info)
enddo
```

Error Conditions

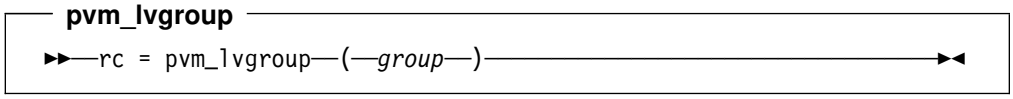
PvmSysErr	System error. Process not enrolled.
PvmBadParam	*tid* identifies the calling process. *tid* process does not exist.

pvm_lvgroup

Description

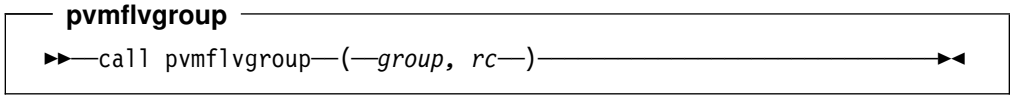
The **pvm_lvgroup** routine unenrolls the calling process from the supplied *group*. The process must already have called the **pvm_joingroup** routine to become a member of that group. Processes can join and leave groups at any time during the program execution, but they are not guaranteed to get the same instance number within each group if they leave and then enroll again. In fact, the IBM PVMe for AIX program attempts to reuse entries in the groups database and instance numbers when they are no longer needed.

C Syntax




```
char *group;
int rc
```

FORTRAN Syntax




```
character*32 group
integer rc
```

Parameters

group
String (no longer than 32 characters) containing the group name

rc
Return code. If the call was successful, it returns a `0` (zero). If the call was unsuccessful, it returns a negative value.

C Example

```
int gtid;

rc = pvm_lvgroup ("all");
```

FORTRAN Example

```
integer gid

call pvmflvgroup ("workers", gid)
if (gid.ne.0) then
  print *, 'Error leaving group: all'
endif
```

Error Conditions

PvmSysErr	System error. Process not enrolled.
PvmOverFlow	*group* longer than 32 characters.
PvmNulGroup	*group* equal to NULL.
PvmNotInGroup	The calling process is not a member of *group*. Group *group* does not exist.

pvm_mcast

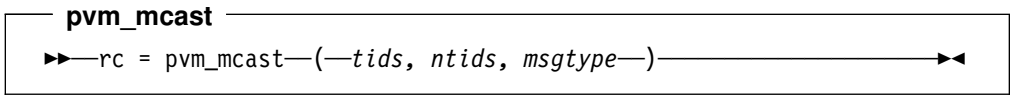
Description

The **pvm_mcast** routine sends a message to a set of IBM PVMe for AIX processes. The message must have already been packed into the sending buffer using the **pvm_pk***xxx* routines. The routine arguments are the number of recipient processes, a list of their identifiers, and the message type. The message type can be used on the receiving side to identify the message contents or to select the message from the receiving queue.

Note: Although this list of recipient processes can also include the sending one, the sender will not receive a message.

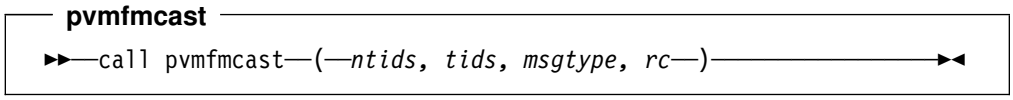
pvm_mcast is a blocking asynchronous routine. It returns when the message has been delivered to all recipient's tasks, even though the recipient tasks may not have called a receive routine (recipient tasks interrupt their processing to store incoming messages in a pre-receiving area). However, the recipient processes must still exist as IBM PVMe for AIX program processes. They can call any of the receive routines to receive the message. For large messages (greater than 2 Mb), **pvm_mcast** may block until the receiving task has started receiving the message.

C Syntax




```
int *tids;
int ntids;
int msgtype;
int rc;
```

FORTRAN Syntax




```
integer ntids
integer tids(*)
integer msgtype
integer rc
```

Parameters

tids
Array of integers containing the task identifiers of the processes the message is to be sent to.

ntids
Integers specifying the number of recipient tasks.

msgtype
Specifies the message type (>=0) of the message to be sent.

rc
If the call was successful, it returns a `0` (zero). If the call was unsuccessful, it returns a negative value.

C Example

```
int msgtag, tasks[NPROC];

pvm_initsend(0);
pvm_pkint (...);
pvm_pkfloat (...);
if (pvm_mcast(tasks,NPROC,msgtag)) {
   printf ("Error in mcasting message %d\n",msgtag);
}
```

FORTRAN Example

```
integer info, proc(MAXPROC), nproc
real*4 data(4096)

nproc=MAXPROC-2
call pvmfinitsend(0,info)
call pvmfpack(INTEGER4,nproc,1,1,info)
call pvmfpack(REAL4,data,ndata,1,info)
call pvmfmcast(nproc,proc(2),99,rc)
  if (rc.lt.0) then
    print *, 'Error mcasting message '
endif
```

Error Conditions

PvmSysErr	System error. Process not enrolled.
PvmBadParam	*type* negative. Invalid *ntids*
PvmNoBuf	No active send buffer.
PvmNoTask	Recipient process does not exist (not yet enrolled or already exited).

pvm_mkbuf

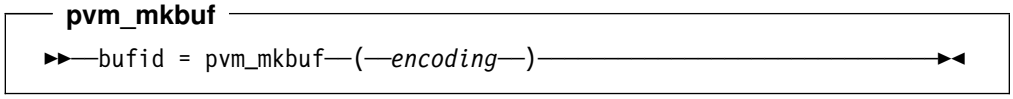
Description

The **pvm_mkbuf** routine returns the identifier of a new message buffer. It is required when you want to manage multiple message buffers in a single application. The *bufid* can be used later to switch between the sending buffers (**pvm_setsbuf**) and to unlock the buffer (**pvm_freebuf**) when it is no longer needed. When there are no more buffers available, **PvmNoMem** is returned.

Receive buffers are automatically created and freed by the receive routines. If you use **pvm_setrbuf**, you are not required to free them, unless they have been explicitly saved (see "pvm_setrbuf" on page 190 for more information).

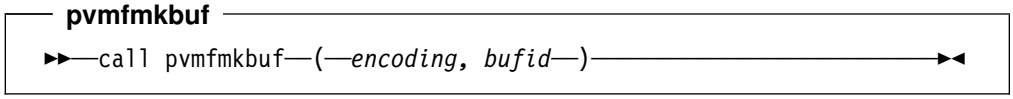
The *encoding* parameter can assume three different values: two of them (**PvmDataRaw** and **PvmDataDefault**) were used in freeware PVM to enable/disable data conversion to/from the XDR format. Both of them are supported in the IBM PVMe for AIX program and because of the homogeneous environment, they actually produce the same effect. The last value (**PvmDataInPlace**) indicates that data must not be copied into the send buffer during the packing operation. Only the pointers to the data and the size are written into the buffer, while data is copied directly out of your memory only when a send is performed. This option allows you to avoid one data copy, with a substantial improvement in performance. However, the same data can be sent multiple times, with modifications intervening between the sends, without having to pack it each time.

C Syntax




```
int encoding;
int bufid;
```

FORTRAN Syntax




```
integer encoding
integer bufid
```

Parameters

encoding

Integer flag which can take any of the following values (values in parenthesis are to be used in FORTRAN programs):

PvmDataDefault (PVMDEFAULT)	0
PvmDataRaw (PVMRAW)	1
PvmDataInPlace (PVMINPLACE)	2

bufid
Return code. A positive number identifying the new buffer if the call was successful, a negative value if it was unsuccessful.

C Example

```
int newbuf;

newbuf = pvm_mkbuf(PvmDataDefault);
oldbuf = pvm_setsbuf(newbuf);
  ...
```

FORTRAN Example

```
integer i,buff(N)

i=i+1
call pvmfmkbuf(PVMDEFAULT,buf(i))
call pvmfsetsbuf(buf(i),buf(i-1))
```

Error Conditions

PvmSysErr	System error. Process not enrolled.
PvmBadParam	Invalid *encoding* value.
PvmNoMem	No more buffers available.

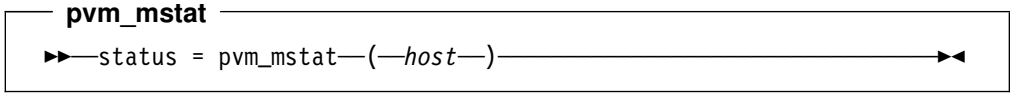
pvm_mstat

Description

The **pvm_mstat** routine returns the status of the supplied host. The first argument must be the name known by the IBM PVMe for AIX daemon. If the host is properly running the return code is `0`.

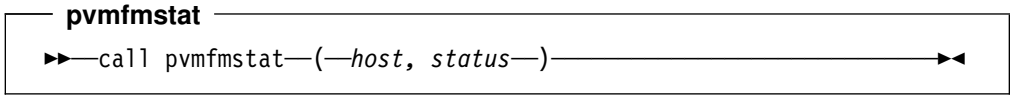
Note: When full names (for example, **hostA.domain**) are supplied to an IBM PVMe for AIX routine, the domain name is skipped, and the host is registered with only the short name (**hostA**). This implies that the short alias must always be used.

C Syntax




```
char *host;
int status;
```

FORTRAN Syntax




```
character*64 host
integer status
```

Parameters

host
Character string containing the *hostname*.

status
Integer where the status of the supplied host is returned. It returns a `0` if it is running, or a negative value if it is not running.

C Example

```
char *host= "RISC6"
int stat;
  .....
if (!(stat = pvm_mstat (host))
  printf ("Host %s is running\n", host);
```

FORTRAN Example

```
integer status;
  .....
call pvmfmstat ('HOST12',status)
if (status.eq.0) then
   print *, 'HOST12 is running'
endif
```

Error Conditions

PvmSysErr	System error. Process not enrolled.
PvmNoHost	Invalid *hostname*.

pvm_mytid

Description

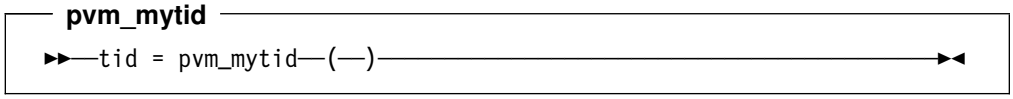
The **pvm_mytid** routine enrolls the calling process to the IBM PVMe for AIX program on its first call, and returns the task identifier. On all successive calls, it simply returns the task identifier of the calling process.

Note: This routine must be the first PVM call of any IBM PVMe for AIX program.

The *tid* is generated by the IBM PVMe for AIX daemon during the first call to the **pvm_mytid** routine, or, if the process was started by the IBM PVMe for AIX daemon, during the spawning of that process.

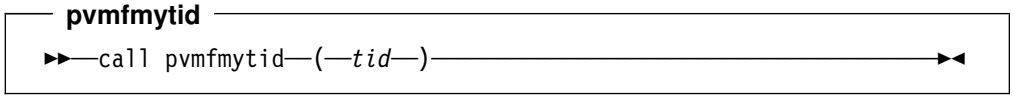
If a value less than `0` is returned, an error occurred. You should check to see if the IBM PVMe for AIX program has been started on the proper host node.

C Syntax




```
int tid;
```

FORTRAN Syntax




```
integer tid
```

Parameters

tid
Return code. If the call was successful, it returns the task identifier of the calling process. A value less than `0` indicates an error.

C Example

```
int mytid;

mytid = pvm_mytid();
```

FORTRAN Example

```
integer me

call pvmfmytid (me)
```

Error Conditions

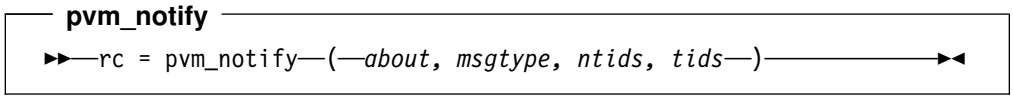
PvmSysErr	System error. The IBM PVMe for AIX program is not active on the host node. Process not enrolled.

pvm_notify

Description

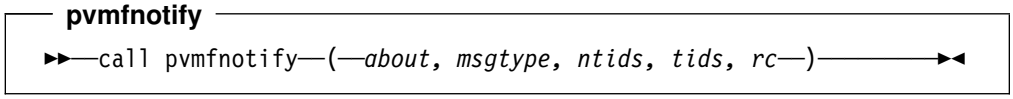
The **pvm_notify** routine informs the daemon that the calling process wants to be notified about certain events. The only supported event is **PvmTaskExit**. A list of task identifiers, their number and a particular message type must be supplied. The calling process will receive a message with the supplied message type when one of the supplied tasks exit. The calling task is responsible for receiving that message. The message contains the *tid* of the exiting task.

C Syntax




```
int about;
int msgtype;
int ntids;
int *tids;
int rc;
```

FORTRAN Syntax




```
integer about
integer msgtype
integer ntids
integer tids(*)
integer rc
```

Parameters

about
Integer flag that can take only the following value (the FORTRAN value is in parenthesis):

PvmTaskExit (PVMTASKEXIT) 1

The two other options supported by freeware PVM (**PvmHostAdd** and **PvmHostDelete**) are not supported here.

msgtype
Integer message type (`>= 0`) of the message that will be sent to the calling process.

ntids
Number of processes whose exit will cause a notification message to be sent to the calling process.

tids
Array of integers containing the task identifiers of the processes whose exit will cause a notification message to be sent to the calling process.

rc
Return code. If the call was successful, it returns a `0` (zero). If the call was unsuccessful, it returns a negative value.

C Example

```
int rc, nprocs;
int procs(MAXPRCS);
float data[MAXDATA];
   ....
rc = pvm_notify (PvmTaskExit, 777, nprocs, procs);
```

FORTRAN Example

```
integer info, nprocs
integer procs(MAXPROC)
   ...
call pvmfnotify(PVMTASKEXIT, 1000, nprocs, procs, info)
```

Error Conditions

PvmSysErr	System error. Process not enrolled.
PvmBadParam	*msgtype* less than `-1`. Invalid *about* argument. Invalid *tid*.

pvm_nrecv

Description

The **pvm_nrecv** routine receives a message with the supplied *msgtype* coming from the process with the supplied *tid*, and places it in the current receive buffer. A `-1` can be used as a wildcard both for the *tid* and the *msgtype* parameter.

The routine is nonblocking, that is, it returns immediately, whether the required message arrived or not. If the message arrived, the return code contains the identifier of the receive buffer, otherwise, it returns a `0`.

If the routine returns a positive number (the buffer identifier), data is available in the buffer and can be unpacked into your memory space by using the **pvm_upk***xxx* routines.

C Syntax

pvm_nrecv

```
►►—bufid = pvm_nrecv—(—tid, msgtype—)———————————————►◄
```

```
int tid;
int msgtype;
int bufid;
```

FORTRAN Syntax

pvmfnrecv

```
►►—call pvmfnrecv—(—tid, msgtype, bufid—)———————————►◄
```

```
integer tid
integer msgtype
integer bufid
```

Parameters

tid
: Integer task identifier of the sender process. A `-1` means that messages coming from any other IBM PVMe for AIX process can be accepted.

msgtype
: Integer message type (`>= 0 or -1`) of the message to be received. If a `-1` is supplied, the first message coming from the supplied *tid* process will be received (regardless of the message type).

bufid
: Return code. A positive integer that indicates the buffer where the proper message was received. It returns a `0` (zero) if the required message did not arrive, or a negative number if an error occurred.

C Example

```
int buft, rc, msg;
float data[MAXDATA];
   ....
while (!(pvm_nrecv(-1,-1))) {
   ...                  /* execute other statements */
}
pvm_upkint(&n,1,1);
pvm_upkfloat(&data,n,1);
```

FORTRAN Example

```
parameter (MAXDATA=1024)
integer buf, parent, MAXDATA, rc
real*8 data(MAXDATA)
   ...
call pvmfparent(parent)
call pvmfnrecv (parent, -1, buf)
if (buf.gt.0) then
  call pvmfunpack (REAL8, data, MAXDATA, 1, rc)
  ....
else
  ....                   ! do something else
```

Error Conditions

PvmSysErr	System error. Process not enrolled.
PvmBadParam	*msgtype* less than -1. Invalid *taskid*.

pvm_packf

Description

The **pvm_packf** routine packs data of various data types into the active send buffer, using a format like **printf**. The syntax of the format string is:

format init fmt*

init null | '%' '+'

fmt '%' count stride modifiers fchar

fchar 'c' | 'd' | 'f' | 'x' | 's'

count null | [0-9]+ | '*'

stride null | '.' ([0-9]+ | '*')

modifiers null | modifiers mchar

mchar 'h' | 'l' | 'u'

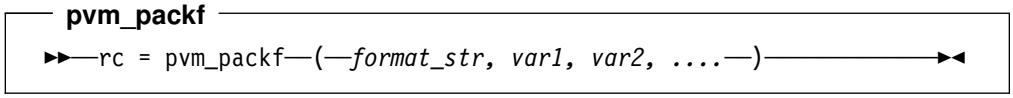
Where:

- fchar:
  - **+** Means **initsend** - must match an integer (encoding) in the parameter list.
  - **c** Pack/unpack bytes
  - **d** Pack/unpack integers
  - **f** Pack/unpack floats
  - **x** Pack/unpack complex float
  - **s** Pack/unpack strings
- Modifiers
  - **h** Short (int)
  - **l** Long (int, float, complex float)
  - **u** Unsigned (int)

Note: If a **count** or **stride** is specified, the matching argument should be a pointer to the given type. If either is not explicit, the value is assumed to be 1. For strings, **count** and **stride** are ignored and the matching argument should always be a **char***.

C Syntax

pvm_packf




```
const char *fmt;
int rc;
```

Parameters

fmt
Format expression like **printf** indicating data and data types to be packed.

rc
Return code. It returns a `0` (zero) if the call was successful, or a negative number if unsuccessful.

var1, var2, **...**
Variables that correspond to the format line which define how the variables are outputted.

C Example

```
#define SZ 4
double float LFLOAT_array[SZ];
int INT_array[SZ];
int count, rc;
char *str = "this array";

rc = pvm_packf ("%+ %d %*d %4lf %s", PvmDataRaw, count, SZ,
                INT_array, LFLOAT_array, str);
```

Error Conditions

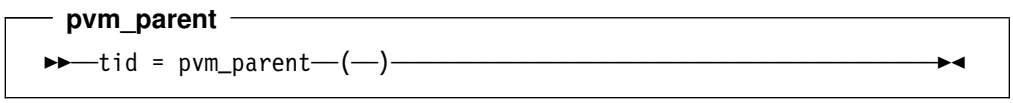
PvmSysErr	System Error. Process not enrolled.
PvmNoMem	Data exceeded available memory in the send buffer.
PvmBadParam	*count* less than `0` (zero). Invalid data type.
PvmNoBuf	No active send buffer.

pvm_parent

Description

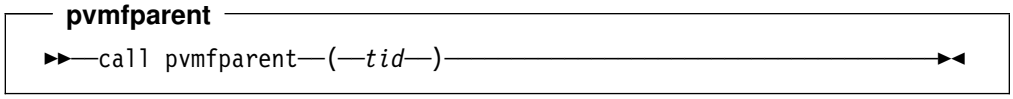
The **pvm_parent** routine can be used to determine the task identifier of the IBM PVMe for AIX process that spawned the calling one. If the **pvm_parent** caller was not started by a **pvm_spawn** routine, the return code is **PvmNoParent**.

C Syntax




```
int tid;
```

FORTRAN Syntax




```
integer tid
```

Parameters

tid
Return code. Integer where the routine returns the task identifier of the process that spawned the calling one. If the calling one was not started by another IBM PVMe for AIX process, then the *tid* will be set to **PvmNoParent**. As distinct from PVM, a normal *tid*, not **PvmNoParent** is returned if the calling process was started from the console.

C Example

```
int parent, me;

me = pvm_mytid
if ( (parent = pvm_parent()) == PvmNoParent)  {
                                          /* I'm the first: I have */
    nt = pvm_spawn ( .... );              /* to start the children */
}
```

FORTRAN Example

```
integer parent, info
  .....
call pvmfparent (parent)
call pvmfsend (parent, 99, info)
```

Error Conditions

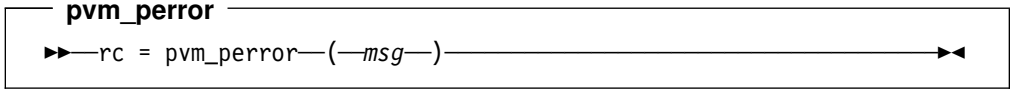
PvmSysErr	System Error. Process not enrolled.
PvmNoParent	The calling process was not started by a **pvm_spawn** routine.

pvm_perror

Description

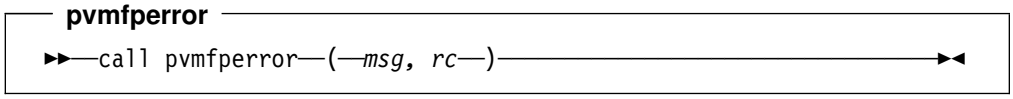
The **pvm_perror** routine prints a line on *stderr* containing a message supplied by you (if any), a colon (:) and an error string corresponding to the error code of the last error that occurred. The **pvm_perror** routine is intended for errors in the IBM PVMe for AIX library routines.

C Syntax




```
char *msg;
int rc;
```

FORTRAN Syntax




```
character msg(*)
integer rc
```

Parameters

msg
A null-terminated string.

rc
Return code. It returns a `0` (zero) if the call was successful, or a negative number if unsuccessful.

C Example

```
int dest, type;

  .....
if (pvm_send (dest,type))
    pvm_perror ("While sending back results to host:");
```

FORTRAN Example

```
integer me
  .....
call pvmfmytid (me)
if ( me.lt.0 ) then
    call pvmfperror ("error enrolling", info);
endif
```

Error Conditions

There are no error conditions.

pvm_pkint, pvm_pkcplx, pvm_pkdcplx, pvm_pkfloat, ...

Description

These routines pack arrays of various data types into the active send buffer. The arguments of the C routines are:

- A pointer to the array to be packed
- The total number of items to be packed
- The stride used to pack data

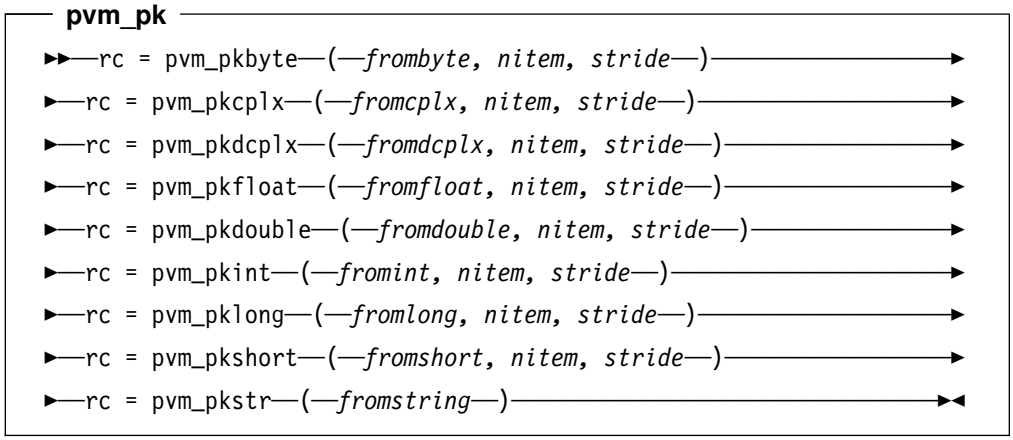
The only exception is the **pvm_pkstr** routine, that packs a character string and does not need either the *nitem* nor the *stride* parameters.

There is a single routine to pack data in FORTRAN programs. The first parameter is a flag identifying the type of data to be packed. If the data type is **STRING**, *nitem* must contain the number of characters in the string, and the *stride* is expected to be 1.

A single message may contain different data types. However, on the receiving side, to insure data integrity, data must be unpacked exactly in the same order it was packed.

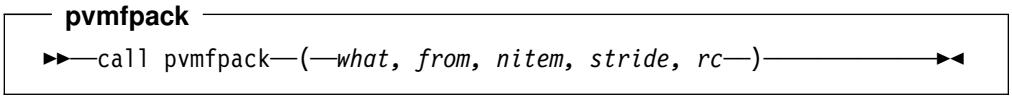
The maximum size for a single message is 32 megabytes. If more data is going to be packed into a buffer, the **PvmNoMem** error code is returned.

C Syntax




```
char *frombyte;
float *fromcplx;
double *fromdcplx;
float *fromfloat;
double *fromdouble;
int *fromint;
long *fromlong;
short *fromshort;
char *fromstring;
int nitem;
int stride;
int rc;
```

FORTRAN Syntax




```
integer what
character from (*)
integer nitem
integer stride
integer rc
```

Parameters

what
Integer flag to be supplied to the FORTRAN routine to specify the type of data to be copied. It may take any of the following values:

STRING	0
BYTE1	1
INTEGER2	2
INTEGER4	3
REAL4	4
COMPLEX8	5
REAL8	6
COMPLEX16	7

frombyte
fromcplx
fromdcplx
fromfloat
fromdouble
fromint
fromlong
fromshort
fromstring
from
Starting address of the memory region where data to be copied is stored. It must be a pointer to the proper data type in C or a variable name in FORTRAN.

nitem
Number of data items that are to be copied into the send buffer from the memory region starting at the specified address.

stride
The stride to be used when packing the items.

rc
Return code. It returns a `0` (zero) if the call was successful, or a negative number if unsuccessful.

C Example

```
double matrix[MAXROWS*MAXCOLS];
int nrows, ncols;

pvm_initsend(0);
pvm_pkint (&nrows,1,1);
pvm_pkint (&ncols,1,1);
pvm_pkdouble(matrix, nrows*ncols, 1);
```

FORTRAN Example

```
integer*4 MAXN, n, bufid, info
parameter (MAXN=128)
real*8 vec(MAXN)

call pvmfinitsend(0,bufid)
if (bufid.gt.0) then
   call pvmfpack(INTEGER4, n, 1, 1, info)
   call pvmfpack(REAL8,vec(1), n, 1, info)
   call pvmfsend(dest, msgtype, info)
endif
```

Error Conditions

PvmSysErr	System Error. Process not enrolled.
PvmNoMem	Data exceeded available memory in the send buffer.
PvmBadParam	*count* less than `0` (zero). Invalid data type.
PvmNoBuf	No active send buffer.

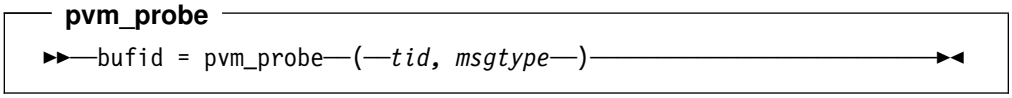
pvm_probe

Description

The **pvm_probe** routine checks to see if a message with the supplied *msgtype* coming from the supplied *tid* task has arrived and is in the receive queue for an IBM PVMe for AIX process. It returns without blocking whether this is the case or not. If the required message is found, the routine returns a buffer identifier that can be used in a **pvm_bufinfo** call to query information about the message. Otherwise, a `0` is returned. A negative code is returned if an error occurred.

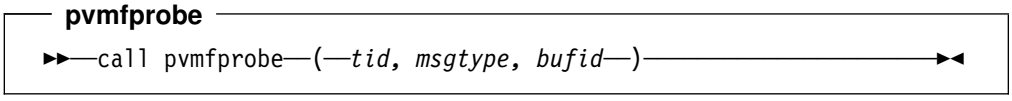
The **pvm_probe** routine may be invoked multiple times. Once it returns a valid *bufid*, a **pvm_recv** routine must be called before unpacking data.

C Syntax




```
int tid;
int msgtype;
int bufid;
```

FORTRAN Syntax




```
integer tid
integer msgtype
integer bufid
```

Parameters

tid
: Specifies the task identifier of the sender process. A `-1` means that any message with the supplied *msgtype* will satisfy the request.

msgtype
: Specifies the message type (`>=0`) to poll for. If a `-1` is supplied, then any message coming from the supplied *tid* will satisfy the request.

bufid
: Return code. It contains a buffer identifier that can be used in a **pvm_bufinfo** routine to query information about the message if the call was successful. It returns a `0` if the required message did not arrive. It returns a negative number if it fails.

C Example

```
int bufid;

if ((bufid = pvm_probe (-1,-1)) > 0) {
  rc = pvm_bufinfo(bufid, &len, &msg, &tid);
  printf ("Message %d from %d is going to be received\n", msg, tid);
  pvm_recv(tid, msg);
} else if ( bufid == 0 ) {
  printf ("No queued messages\n");
  .....
} else {
  printf ("An error occurred\n");
  ......
}
```

FORTRAN Example

```
integer msg, info, parent

call pvmfparent(parent)
call pvmfprobe (parent, 55, info)
if (info.gt.0) then
  print *, 'Message 55 is going to be received'
  call pvmfrecv(parent,55,info)
endif
```

Error Conditions

PvmSysErr	System error. Process not enrolled.
PvmBadParam	*type* less than -1. Invalid *taskid*.

pvm_precv

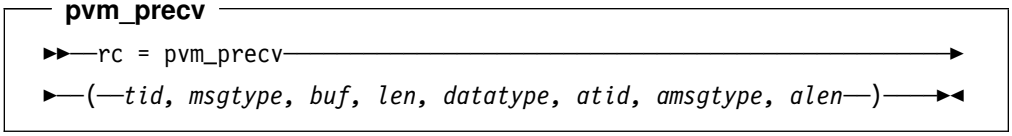
Description

The **pvm_precv** routine receives a message, with the specified *msgtype* coming from the specified *tid*, unpacks and copies the message content to a memory region, specified by the user.

This routine is blocking; it returns immediately if a message that matches the required *msgtype* and *tid* has already arrived. If is hasn't, **pvm_precv** waits until the proper message arrives. Wildcards (*-1*) can be used for both the *tid* and the *msgtype* parameter.

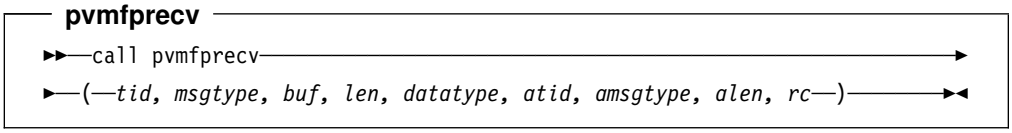
The **pvm_precv** routine can be used to receive messages sent by the **pvm_send**, **pvm_psend**, **pvm_mcast** or **pvm_bcast** routines.

C Syntax




```
int tid;
int msgtype;
char *buf;
int len;
int datatype;
int *atid;
int *amsgtype;
int *alen;
int rc;
```

FORTRAN Syntax




```
integer tid
integer msgtype
character buf(*)
integer len
integer datatype
integer atid
integer amsgtype
integer alen
integer rc
```

Parameters

tid
Specifies the task identifier of the sender process. A -1 means that any message with the supplied *msgtype* will satisfy the request.

msgtype
Specifies the message type (>=0) to poll for. If a -1 is supplied, then any message coming from the supplied *tid* will satisfy the request.

buf
Starting address of the memory region where data will be copied. It must be a pointer to the proper data type, and the allocated memory must be at least *len * size_of_datatype* long.

len
Number of data items to be copied (multiple of *size_of_datatype*).

datatype
Type of data to be copied. It may be any of the following values (values in parenthesis are for use in FORTRAN programs):

PVM_BYTE (BYTE1)	1
PVM_SHORT (INTEGER2)	2
PVM_INT (INTEGER4)	3
PVM_FLOAT (REAL4)	4
PVM_CPLX (COMPLEX8)	5
PVM_DOUBLE (REAL8)	6
PVM_DCPLX (COMPLEX16)	7
PVM_LONG	8

atid
The actual *tid* of the sender task.

amsgtype
The actual message type.

alen
The actual amount of copied data.

rc
Return code. 0 is successful. Negative values indicate that an error occurred.

C Example

```
#define MAXLEN 16*1024
  ...
int rc, from, type, len;
int work [MAXLEN];

rc = pvm_precv (-1, -1, work, MAXLEN, PVM_INT, &from, &type, &len);
if (rc >=0 ) {
  printf ("Received message %d from %d \n", type, from);
  printf ("%d items copied to the work array \n", len);
} else {
  printf ("An error occurred\n");
}
```

FORTRAN Example

```
integer rc, dummy, parent, len
real*8 array(ndata)

call pvmfparent(parent)
call pvmfprecv (parent, 10, array, ndata, REAL8, dummy, dummy, len, rc)
if (rc.ge.0) then
  print *, 'Message 10 received from parent;', len, 'items copied'
endif
```

Error Conditions

PvmSysErr System error. Process not enrolled.

PvmBadParam *msgtype* less than -1. Invalid *datatype* or *tid*. Invalid *len*.

PvmNoBuf Invalid *buf*.

pvm_psend

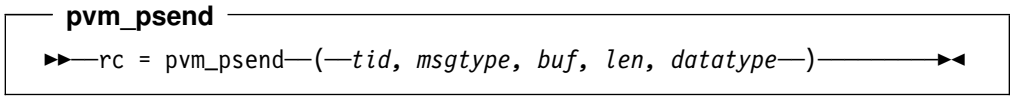
Description

The **pvm_psend** routine packs data (with the supplied *datatype*, starting from the supplied pointer) into a message, and sends the message to the task identified by *tid* with type *msgtype*.

pvm_psend is a blocking asynchronous routine; it returns as soon as the message has been successfully delivered, whether the recipient process receives it or not. For large messages (greater than 2 Mb), **pvm_psend** may block until the receiving task has started receiving the message.

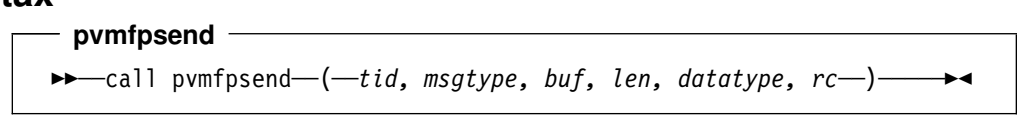
Messages sent by **pvm_psend** can be received by **pvm_recv**, **pvm_precv**, **pvm_trecv**, or **pvm_nrecv**. These routines are blocking; they return immediately if a message, that matches the *tid* and *msgtype* of the **pvm_psend** routine, has already arrived. If not, they wait until the proper message arrives.

C Syntax




```
int tid;
int msgtype;
char *buf;
int len;
int datatype;
int rc;
```

FORTRAN Syntax




```
integer tid
integer msgtype
character buf(*)
integer len
integer datatype
integer rc
```

Parameters

tid
Specifies the task identifier of the recipient process.

msgtype
Specifies the message type (>=0) of the message.

buf
Starting address of the memory region containing data to be copied. It must be a pointer to the proper data type.

len
Number of data items to be copied, not the number of bytes.

datatype
Type of data to be copied. It may be any of the following values (values in parenthesis are for use in FORTRAN programs):

PVM_BYTE (BYTE1)	1
PVM_SHORT (INTEGER2)	2
PVM_INT (INTEGER4)	3
PVM_FLOAT (REAL4)	4
PVM_CPLX (COMPLEX8)	5
PVM_DOUBLE (REAL8)	6
PVM_DCPLX (COMPLEX16)	7
PVM_LONG	8

rc
Return code. Negative values indicate that an error occurred.

C Example

```
int rc, ndata, msgtag, parent;
float data [ndata];

 ....
parent = pvm_parent();
rc = pvm_psend (parent, msgtag, data, ndata, PVM_FLOAT);
if (rc <0 ) {
  pvm_perror ("PSending array DATA to the parent process");
}
```

FORTRAN Example

```
integer rc, tasks[MAXTASKS]
real*8 array(ndata)

 ...
tag = 55
call pvmfpsend (tasks[1], tag, array, ndata, REAL8, rc)
if (rc.lt.0) then
  print *, 'Error in pvmfpsend routine'
endif
```

Error Conditions

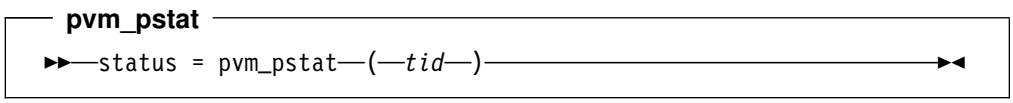
PvmSysErr	System error. Process not enrolled.
PvmBadParam	*msgtype* less than -1. Invalid *dataype* or *tid*. Invalid *len*.
PvmNoBuf	Invalid *buf*.

pvm_pstat

Description

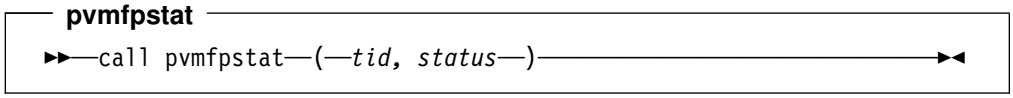
The **pvm_pstat** routine returns the status of the specified process. It returns **PvmNoTask** if the supplied taskid is invalid, or the task has already left the IBM PVMe for AIX program or exited. It returns a `0` if the process is properly running.

C Syntax




```
int tid;
int status;
```

FORTRAN Syntax




```
integer tid
integer status
```

Parameters

tid
Task identifier of the process that requires status.

status
Return code. It returns the process status, if the call was successful. It returns a negative value (other than `-1`) if an error occurred.

C Example

```
int tid;
  ....
pvm_initsend(0);
pvm_pkint (...);
if ((pvm_pstat(tid))==0) {
   pvm_send(tid,99);
} else ...
```

FORTRAN Example

```
integer parent, status

call pvmfparent(parent)
call pvmfpstat (parent,status)
if (status.eq.0) then
   call pvmfsend(parent, 1000)
endif
```

Error Conditions

PvmSysErr	System error. Process not enrolled.
PvmNoTask	No such process.

pvm_recv

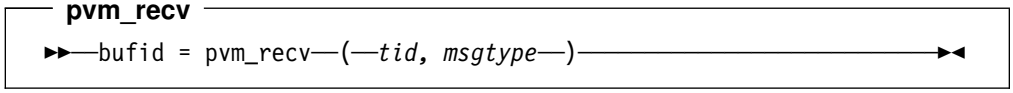
Description

The **pvm_recv** routine receives a message with the supplied *msgtype* coming from the process with the supplied *tid*, and places it in the current receive buffer. A -1 can be used as a wildcard both for the *tid* and the *msgtype* parameter.

The **pvm_recv** routine is blocking, that is, it waits until a message, matching the supplied *tid* and *msgtype*, arrives. If the message already arrived, **pvm_recv** returns immediately.

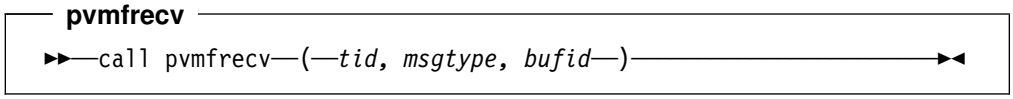
Once the **pvm_recv** routine returns, the data in the current receive buffer may be unpacked into your memory space using the **pvm_upk***xxx* routines.

C Syntax




```
int tid;
int msgtype;
int bufid;
```

FORTRAN Syntax




```
integer tid
integer msgtype
integer bufid
```

Parameters

tid
Integer task identifier of the sender process. A -1 means that messages coming from any other IBM PVMe for AIX process can be accepted.

msgtype
Integer message type (>= 0 or -1) of the message to be received. If a -1 is supplied, the first message coming from the supplied *tid* process is received, regardless of the message type.

bufid
Return code. If the call was successful, it returns a positive integer that indicates the buffer where the message was received. It returns a negative number if it was unsuccessful.

C Example

```
int buft, rc, msg;
float data[MAXDATA];
   ....
msg++;
buft = pvm_recv (-1,msg);
pvm_upkint(&n,1,1);
pvm_upkfloat(&data,n,1);
```

FORTRAN Example

```
parameter (MAXDATA=1024)
integer buf, parent, MAXDATA
real*8 data(MAXDATA)
   ...
call pvmfparent (parent)
call pvmfrecv (parent, -1, buf)
if (buf.gt.0) then
  call pvmfunpack (REAL8, data, MAXDATA, 1)
else
  print *, 'Error receiving from parent'
endif
```

Error Conditions

PvmSysErr	System error. Process not enrolled.
PvmBadParam	*msgtype* less than -1. Invalid *tid*.

pvm_reduce

Description

The **pvm_reduce** routine performs a global operation, such as max, min and sum, over all the members of a specified group. All group members call the routine, providing the local data, but the final result is computed and is available on the (user-specified) root member only. On the root member, the result of the global operation overwrites the local data.

IBM PVMe for AIX supplies the following predefined global functions:

- **PvmMin**
- **PvmMax**
- **PvmSum**
- **PvmProduct**

The **PvmMax** and **PvmMin** functions have been implemented for all data types. For complex datatypes, the maximum/minimum value is the pair with maximum/minimum modulus. The **PvmSum** and **PvmProduct** functions have been implemented for all datatypes except PVM_BYTE (BYTE1).

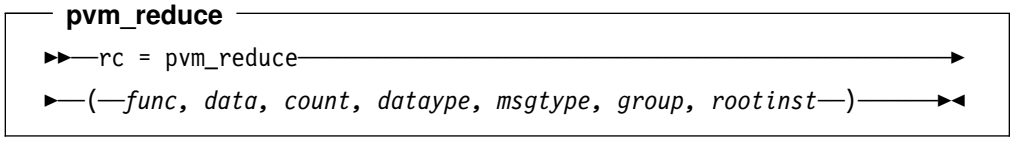
User-written functions that perform other tasks can be used in place of the predefined ones, providing that such functions conform to the following:

C void func (int *datatype, void *x, void *y, int *num, int *rc)

FORTRAN call func (datatype, x, y, num, rc)

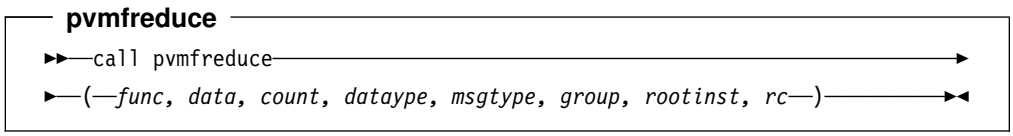
where *x* and *y* are arrays of the specified *datatype*, and *num* is the number of items in those arrays on which the function (*func*) must operate. x and y contain the data to be operated on. The results will be placed in x.

C Syntax




```
int rc;
void *func();
void *data;
int count;
int datatype;
int msgtype;
char *group;
int rootinst;
```

FORTRAN Syntax



```
character(*) data
integer count
integer datatype
integer msgtype
character(*) group
integer rootinst
integer rc
```

Parameters

func
Function that defines the global operation. Predefined values are **PvmMax**, **PvmMin**, **PvmSum**, and **PvmProduct**, but user-written functions can be supplied as well.

data
Pointer to an array of *datatype*, containing the number of elements (*count*) to be sent to the *rootinst* task, which is in charge of performing the global operation.

count
Number of items of the *data* array.

datatype
Integer that specifies the type of entries in the *data* and *result* arrays. It may be any of the following values (values in parenthesis are for use in FORTRAN programs):

PVM_BYTE (BYTE1)	1
PVM_SHORT (INTEGER2)	2
PVM_INT (INTEGER4)	3
PVM_FLOAT (REAL4)	4
PVM_CPLX (COMPLEX8)	5
PVM_DOUBLE (REAL8)	6
PVM_DCPLX (COMPLEX16)	7
PVM_LONG	8

msgtype
Positive integer that identifies the message.

group
Character string that identifies an existing group.

rootinst
Instance number within the group (*group*) of the root task performing the global operation.

rc
Return code. If the call was successful, it returns a `0` (zero). If the call was unsuccessful, it returns a negative value.

C Example

```
double data[MAXDATA];

info = pvm_reduce (PvmMax, &data, MAXDATA, PVM_DOUBLE,
                   100, "all", 0);
```

FORTRAN Example

```
integer ndata
parameter (ndata = 1024)
    .....
integer info
integer vector (ndata)

call pvmfreduce (PvmSum, vector, ndata, INTEGER4, 1000,
                 "ALLGROUP", 0, info)
```

Error Conditions

PvmSysErr	System error. Process not enrolled.
PvmBadParam	*data* is a NULL pointer. *result* is a NULL pointer on the *target* task. *count* less than zero. *msgtype* negative. *datatype* is not a supported type.
PvmOverFlow	Group name longer than 32 characters.
PvmNoMem	No more memory available
PvmNullGroup	*group* name is NULL.
PvmNoGroup	Group named *group* does not exist.
PvmNoInst	Calling process, or process identified with *rootinst* instance number, does not exist within the specified group.

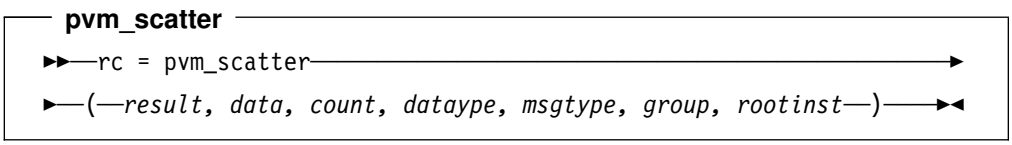
pvm_scatter

Description

The **pvm_scatter** routine distributes data from the specified root member of a group to all members of the same group, including the root member. Each member of the group must call **pvm_scatter**, and each receives a portion of the *data* array on the root member. Data is stored in the local array (*result*).

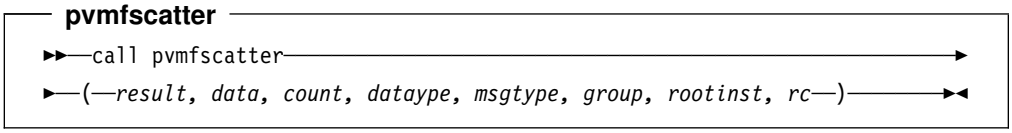
The root member sends the number of (*count*) contiguous items of the array (*data*) to each group member, in order, on the basis of the group instance numbers.

C Syntax




```
int rc;
void *result;
void *data;
int count;
int datatype;
int msgtype;
char *group;
int rootinst;
```

FORTRAN Syntax




```
character(*) result
character(*) data
integer count
integer datatype
integer msgtype
character(*) group
integer rootinst
integer rc
```

Parameters

result
: Pointer to an array of *datatype*, that contains, at the least, *count* elements. It is the array in which data, distributed by the root member, is stored.

data
: Pointer to an array of *datatype*, containing, at the least, the number of elements, specified by *count * group_size*, to be sent to the group members. This parameter is significant only on the root member.

count
Number of items of the *data* array that each group member will receive.

datatype
Integer that specifies the type of the entries in the *data* and *result* arrays. It may be any of the following values (values in parenthesis are for use in FORTRAN programs):

PVM_BYTE (BYTE1)	1
PVM_SHORT (INTEGER2)	2
PVM_INT (INTEGER4)	3
PVM_FLOAT (REAL4)	4
PVM_CPLX (COMPLEX8)	5
PVM_DOUBLE (REAL8)	6
PVM_DCPLX (COMPLEX16)	7
PVM_LONG	8

msgtype
Positive integer used that identifies the message.

group
Character string that identifies an existing group.

rootinst
Instance number within the group *group* of the process performing data distribution.

rc
Return code. If the call was successful, it returns a `0` (zero). If the call was unsuccessful, it returns a negative value.

C Example

```
int msgtype;
double array[MAXDATA*MAXPROCS];
double myarray[MAXDATA];

info = pvm_scatter (&myarray, &array, mydata, PVM_DOUBLE,
                    msgtype, "all", 0);
```

FORTRAN Example

```
integer cols, ncols, MAXPROCS
parameter (MAXPROCS=16, ncols = 1024, cols = ncols*MAXPROCS)
    .....
integer info
integer matrix (rows, cols), local (rows, ncols)

call pvmfscatter (local, matrix, rows*ncols, INTEGER4, 1000,
                  "ALLGROUP", 0, info)
```

Error Conditions

PvmSysErr	System error. Process not enrolled.
PvmBadParam	*data* is a NULL pointer on the *rootinst* task. *result* is a NULL pointer. *count* is non positive. *msgtype* is negative. *datatype* is not a supported type.

PvmOverFlow	Group name longer than 32 characters.
PvmNullGroup	*group* name is NULL.
PvmNoGroup	Group named *group* does not exist.
PvmNoInst	Calling process or process identified with *rootinst* instance number does not exist within the specified group.

pvm_send

Description

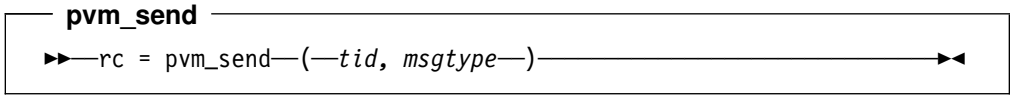
The **pvm_send** routine sends a message to another IBM PVMe for AIX process. The message must have already been packed into the sending buffer using the **pvm_pk***xxx* routines. The arguments of the routine are the task identifier of the recipient process and the message type that can be used on the receiving side. This is to identify the message content or to select the message from the receiving queue.

pvm_send is a blocking asynchronous routine. It returns when the message has been delivered to the recipient's task, even though the recipient task may not have called a receiving routine (it will interrupt its processing to store the incoming message in a pre-receiving area). However, the recipient process must still exist as an IBM PVMe for AIX process. It can either call the **pvm_recv**, **pvm_precv**, or the **pvm_nrecv** routines to receive the message.

Note: **pvm_send** does not clear the send buffer, so the buffer may be resent to multiple processes.

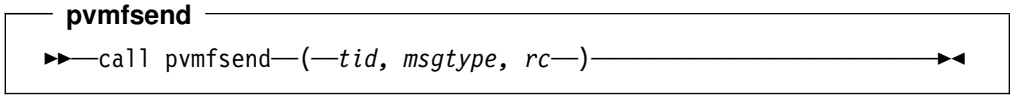
For large messages (greater than 2 Mb), **pvm_send** may block until the receiving task has started receiving the message.

C Syntax




```
int tid;
int msgtype;
int rc;
```

FORTRAN Syntax




```
integer tid
integer msgtype
integer rc
```

Parameters

tid
Task identifier (>0) of the process the message is to be sent to.

type
Specifies the message type (>=0) of the message to be sent.

rc
Return code. It returns a 0 (zero) if successful. It returns a negative number if an error occurred.

C Example

```
int tasks[NPROC];

pvm_initsend(0);
pvm_pkint (...);
pvm_pkfloat (...);
if (pvm_send(tasks[i],99)) {
   printf ("Error sending message to %d\n",task[i]);
}
```

FORTRAN Example

```
integer info
real*4 data(4096)

call pvmfinitsend(0,info)
call pvmfpack(REAL4,data,ndata,1,info)
do i=1,nproc
  call pvmfsend(proc(i),i,rc)
  if (rc.lt.0) then
    print *, 'Error sending to ', proc(i)
  endif
enddo
```

Error Conditions

PvmSysErr	System error. Process not enrolled.
PvmBadParam	*msgtype* negative. Invalid *tid*.
PvmNoBuf	No active send buffer.
PvmNoTask	Recipient process does not exist (not yet enrolled or already exited).

pvm_sendsig

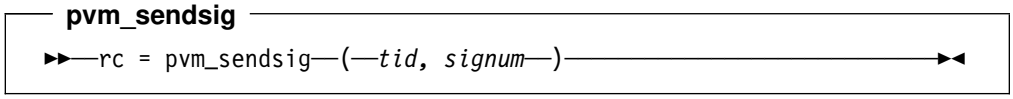
Description

The **pvm_sendsig** routine sends the UNIX signal number (*signum*) to another IBM PVMe for AIX process.

Note: This routine should only be used by programmers that are experienced with UNIX signal handling.

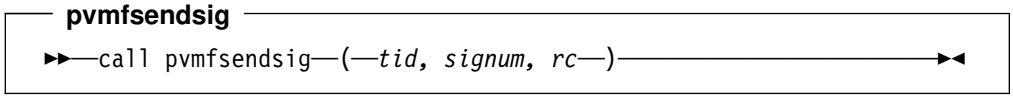
Since the underlying communications software used by IBM PVMe for AIX uses the signals SIGIO, SIGPIPE, and SIGALRM, you should avoid using these signals in your application programs.

C Syntax




```
int tid;
int signum;
int rc;
```

FORTRAN Syntax




```
integer tid
integer signum
integer rc
```

Parameters

tid
Task identifier (>0) of the process to which the signal is to be sent.

signum
Specifies the signal number of the message to be sent.

rc
Return code. It returns a 0 if successful. It returns a negative number if an error occurred.

C Example

```
int tasks[NPROC];

pvm_sendsig (tasks[0],SIGKILL);
```

FORTRAN Example

```
integer ptid, rc

call pvmfparent (ptid)
call pvmfsendsig (ptid, signum, rc)
```

Error Conditions

PvmSysErr	System error. Process not enrolled.
PvmBadParam	Invalid *signum*.

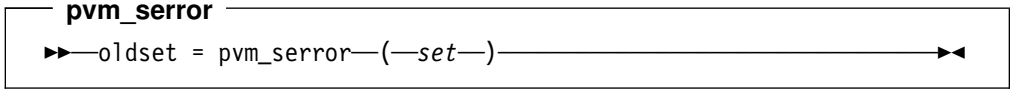
pvm_serror

Description

The **pvm_serror** routine enables or disables automatic error message printings on the *stderr*, whenever an error in an IBM PVMe for AIX library routine occurs. You must specify the new setting value, while the old value is returned if the call is successful.

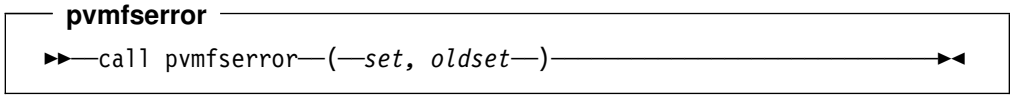
Note: **pvm_serror** has been removed from freeware PVM. For portability, use **pvm_setopt** instead.

C Syntax




```
int set;
int oldset;
```

FORTRAN Syntax




```
integer set
integer oldset
```

Parameters

set
Integer indicating the new setting value:

0 Automatic printing is turned off.
1 Automatic printing is turned on.

oldset
Return code. It contains the old setting value if the call was successful. It contains a negative number if it was unsuccessful.

C Example

```
int olds;

olds = pvm_serror (1);
```

FORTRAN Example

```
integer rc

call pvmfserror (0,rc);
```

Error Conditions

PvmBadParam Invalid set value.

pvm_setopt

Description

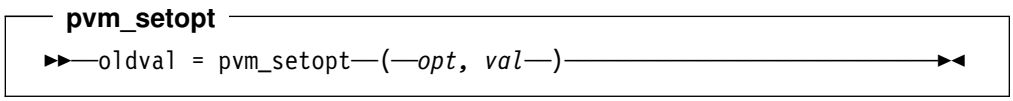
The **pvm_setopt** routine is a generic routine used to set options for IBM PVMe for AIX processes.

The **PvmRoute** option has been introduced only to maintain compatibility with freeware PVM, where it allowed you to select direct communication links (TCP sockets) among tasks. The syntax is exactly the same, but in the IBM PVMe for AIX program this option does not produce any effect, because in this implementation data exchange among tasks always exploits direct process-to-process links.

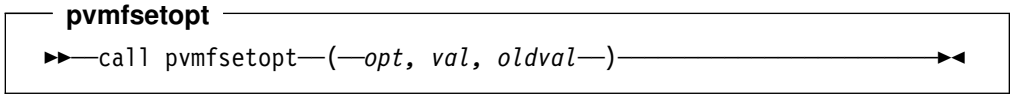
The **PvmAutoErr** option allows you to enable/disable automatic error printings whenever an error occurs in an IBM PVMe for AIX routine. The effect is the same as when the **pvm_serror** routine is called.

PvmDebugMask is included for compatibility with freeware PVM, where it is used to set debugging to *on* or *off*. When debugging is set to *on*, PVM logs detailed information about its operations on the daemon's *stderr stream*. *val* is the is debugging level. This option has no effect with IBM PVMe for AIX. Setting **PvmOutputTid** to `0` to `0` redirects stdout to the stdout to `0` discards trace data. The default setting Otherwise, it is set to `0`.

C Syntax



FORTRAN Syntax



Parameters

opt

Integer that identifies the option. The following options can be set:

- **PvmRoute**
- **PvmDebugMask**
- **PvmAutoErr**

IBM PVMe for AIX does not currently support the following parameters:

- **PvmOutputTid**
- **PvmOutputCode**
- **PvmTraceTid**
- **PvmTraceCode**
- **PvmSelfOutputTid**
- **PvmSelfOutputCode**
- **PvmSelfTraceTid**
- **PvmSelfTraceCode**

val
Value assigned to the specified option. Predefined values are:

PvmRoute	**PvmDontRoute**
	PvmAllowDirect
	PvmRouteDirect
PvmAutoErr	0
	1

oldval
Return code. If the call was successful, it returns a positive number that identifies the previous value of the option. It returns a negative value if it was unsuccessful.

C Example

```
int olds;

olds = pvm_setopt(PvmAutoErr, 1);
```

FORTRAN Example

```
integer info

call pvmfsetopt(PvmAutoErr, 0, info)
```

Error Conditions

PvmSysErr	System error. Process not enrolled.
PvmBadParam	Invalid *opt* or *set*.

pvm_setrbuf

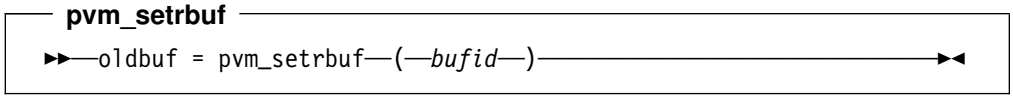
Description

The **pvm_setrbuf** routine sets the supplied *bufid* to the current receive buffer, and saves the previous one. The identifier of the old active buffer is returned, so that it can be used later to restore the old buffer. If `0` is returned, no receive buffer was active.

If the value of *bufid* is `0`, then the previous buffer is saved, and no other buffer becomes active. If a receive operation is attempted, the receive routine will set a new buffer to active.

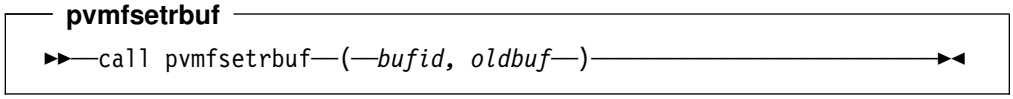
Only values returned by previous receive routines can be supplied. For example, the return value of a **pvm_mkbuf** routine is invalid.

C Syntax




```
int bufid;
int oldbuf;
```

FORTRAN Syntax




```
integer bufid
integer oldbuf
```

Parameters

bufid
Integer identifying the buffer.

oldbuf
Return code. It returns a positive number identifying the old receive buffer if the call was successful. It returns a negative value if it was unsuccessful.

C Example

```
int bufid, newbuf;

bufid = pvm_setrbuf(0);
 ...
newbuf = pvm_recv (...);
```

FORTRAN Example

```
integer oldbuf, newbuf

call pvmfsetrbuf(newbuf, oldbuf)
call pvmfunpack(..)
```

Error Conditions

PvmSysErr	System error. Process not enrolled.
PvmNoSuchBuf	*bufid* is not a valid receive buffer.
PvmBadParam	Invalid *bufid* is less than `0`.

pvm_setsbuf

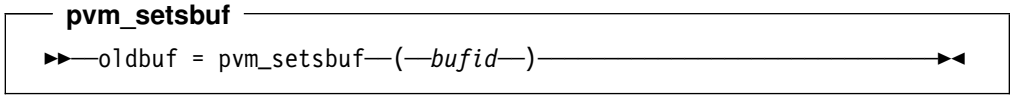
Description

The **pvm_setsbuf** routine sets the supplied *bufid* to the current send buffer, and saves the previous one. The identifier of the old active buffer is returned, so that it can be used later to restore the old buffer. If `0` is returned, no send buffer was active.

If the value of *bufid* is `0`, then the previous buffer is saved, and no other buffer becomes active. Then a new **pvm_setsbuf** or a new **pvm_initsend** routine must be called to set a new active buffer to send messages.

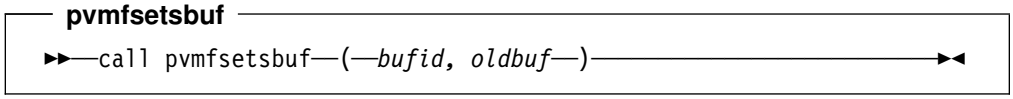
The *bufid* of a receive buffer is a valid parameter. The content of the receive buffer is copied into a new send buffer that is available for the next send operations.

C Syntax




```
int bufid;
int oldbuf;
```

FORTRAN Syntax




```
integer bufid
integer oldbuf
```

Parameters

bufid
Integer identifying the buffer.

oldbuf
Return code. It returns a positive number identifying the old receive buffer if the call was successful. It returns a negative value if it was unsuccessful.

C Example

```
int bufid, newbuf;

bufid = pvm_setsbuf (0);
 ...
newbuf = pvm_initsend (0);
```

FORTRAN Example

```
integer oldbuf, newbuf, rc

call pvmfmkbuf(0,newbuf)
call pvmfsetsbuf(newbuf, oldbuf)
call pvmfpack(REAL4,&data,NDATA,1,rc)
call pvmfsend(...)
call pvmfsetsbuf(oldbuf, rc)
```

Error Conditions

PvmSysErr	System error. Process not enrolled.
PvmNoSuchBuf	*bufid*. is not a valid send or receive buffer.
PvmBadParam	Invalid *bufid*. is less than 0.

pvm_settmask *** CURRENTLY DISABLED ***

Description

Note: A call to this routine will result in an error condition of PvmNotImpl.

The **pvm_settmask** routine sets the trace mask for the calling process, or for its children. The trace mask is used to enable tracing of calls to the IBM PVMe for AIX routines. Each process that is started by **pvm_spawn** inherits the trace mask from the parent. For processes started by other means, the trace mask is set to `0`. When calls are made to functions in **libpvm**, the parameters and results are sent, in the form of messages, to the trace sink of the task (the variable **PvmTraceTid** is set by calling **pvm_setopt**). By calling **pvm_settmask**, the trace mask can be modified, either for the calling process or for the children that will be spawned. Setting the child trace mask has no effect on children that are already running.

The trace mask can be kept in a temporary variable, declared as:

```
Pvmtmask mymask;
```

There are four macros defined to manipulate trace masks. Along with the functions, these allow a program to get the current trace mask, modify or save it, and put it back:

TEV_INIT_MASK(mask) initializes a mask to all bits cleared, and must be used to initialize a trace mask (unless it's done by setting it from **pvm_gettmask**).

TEV_SET_MASK(mask, kind) sets a bit in a mask. Bit values are defined in /usr/lpp/pvme/lib/pvmtev.h

TEV_UNSET_MASK(mask, kind) clears a single bit in a mask. Bit values are defined in <pvmtev.h>.

TEV_CHECK_MASK(mask, kind) checks to see if a bit is set and, if true, returns a non-zero value.

pvm_tmask datatypes and **TEV_*** macros are defined in **pvmtev.h**.

Bit value constraints are of the form

```
TEV_<routine><0|1>
```

The “0” event for a routine occurs at the start of the routine, and the “1” event occurs at the end. For example:

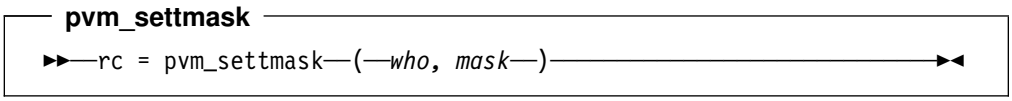
```
/* Trace the start of every call to pvm_bcast */
   TEV_SET_MASK(mask, TEV_BCAST0);
   pvm_settmask(PvmTaskSelf,Mask)
/* Trace the start and end of every call to pvm_recv */
   TEV_SET_MASK(mask, TEV_RECV0);
   TEV_SET_MASK(mask, TEV_RECV1);
```

Generating trace data - PVMe supports the generation of post-mortem tracefiles in SDDF format compatible with PVM3.3 and XPVM. To enable tracefile creation, use the **-trace** command line argument when starting up the pvmd3e daemon (See “Using the -trace Option” on page 35). By default, all trace information will be sent to to the file `/tmp/xpvm.trace.$USER` on the node on which the daemon is running,

but if the environment variable **PVMe_TRACEFILE** is set before starting the daemon, its value will be used for the name of the tracefile.

XPVM is a freeware tool used for graphically displaying PVM3.3 tracefiles. A version of XPVM pre-compiled on the RISC System/6000 architecture can be downloaded via anonymous ftp from netlib2.cs.utk.edu in the directory `/pvm3/xpvm`.

C Syntax




```
int who;
Pvmtmask mask;
int rc;
```

Parameters

who
Specifies whether the trace mask must be set for the calling process or for the children. It can be any of the following values:

PvmTaskSelf	The current task
PvmTaskChild	Children tasks to be spawned

mask
Trace mask used to set the current trace vector.

rc
Return code. It returns `0` if the call was successful, a negative value if it was unsuccessful.

C Example

```
Pvmtmask mask;
int rc;

TEV_INIT_MASK (mask);
TEV_SET_MASK (mask, TEV_FREEBUF0);
rc = pvm_settmask (PvmTaskChild, mask );
rc = pvm_spawn( "nodes", (char **)0, PvmTaskDefault, "", 1, (int *)0);
```

Error Conditions

PvmSysErr	System error. Process not enrolled.
PvmBadParam	Invalid *who* flag.

pvm_spawn

Description

The **pvm_spawn** routine starts *ntasks* copies of the *task* executable and enrolls them into the IBM PVMe for AIX environment. The executable files can be located either in the **$HOME**/**pvm3**/**bin**/**RS6K** directory, or in an alternate directory specified by you in the configuration file at startup time (see Chapter 4, “Product Startup” on page 21).

If the *flag* parameter does not contain the **PvmTaskHost** option, the IBM PVMe for AIX daemon chooses where each new process is to be started. Otherwise, you are required to supply a valid *hostname* in the *where* parameter. Then, the daemon runs a **rcmd** on the remote host and starts the new process.

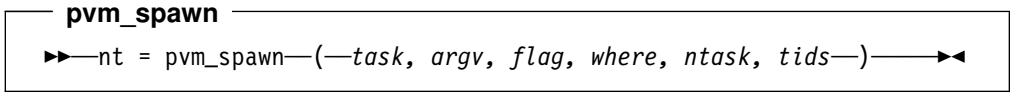
The return code of each spawn attempt is returned in the elements of the *tids* array. If it was successful, it is the task identifier of the new process. If it was not successful, it returns a negative number. The return code of the routine is the actual number of processes started, so that it will be equal to *ntasks* if all the attempts were successful.

The **PvmTaskDebug** option allows you to start the new tasks under the debugger. The IBM PVMe for AIX daemon starts the script files named **debugger** and **debugger2**. They open *ntasks* new windows on the screen, start **dbx** in them and run the new processes. The two script files (two templates are provided within the IBM PVMe for AIX package) must be located in the same directory of the executables, and can be properly customized by you (see “Customizing the debugger and debugger2 Shell Scripts” on page 19).

The C routine allows you to pass arguments to the spawned processes. *argv* can be an array of pointers (terminating with a NULL pointer) to strings where the arguments can be specified. If this facility is not needed, *argv* must be set to NULL.

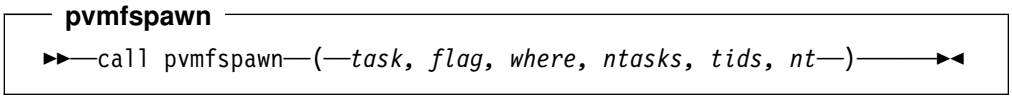
The IBM PVMe for AIX daemon by itself passes a set of arguments to the spawned processes, to provide them with all the necessary information. These kinds of arguments all start with the **pvm_** substring, so that you can easily distinguish them from the other arguments. However, arguments supplied by you always come first in the argument list.

C Syntax




```
char *task;
char **argv;
int flag;
char *where;
int ntasks;
int *tids;
int nt;
```

FORTRAN Syntax




```
charcter*32 task
integer flag
charcter*64 where
integer ntasks
integer tids(*)
integer nt
```

Parameters

task
Character string containing the executable *filename* of the process that is to be started. The maximum allowed length for *task* is 32 characters.

argv
Array of pointers to arguments to be passed to the new process. The array must be terminated with NULL. If the executable does not receive arguments, *argv* must be set to NULL.

flag
Integer flag which can take any of the following values or a sum of them (values in parenthesis are to be used in FORTRAN programs):

PvmTaskDefault (PVMDEFAULT)	0
PvmTaskHost (PVMHOST)	1
PvmTaskArch (PVMARCH)	2
PvmTaskDebug (PVMDEBUG)	4

PvmTaskDefault causes the IBM PVMe for AIX daemon to choose the host where the new process is to be started.

The **PvmTaskHost** option allows you to choose a particular host where the new process is to be started.

The **PvmTaskArch** option is supported for compatibility only with freeware PVM. It was used there to specify the architecture of the host where the new process was to be started. Because the IBM PVMe for AIX program only runs in a homogeneous environment, this flag is redundant.

The **PvmTaskDebug** routine allows you to start the new processes under **dbx**.

where
Character string indicating where the new process is to be started. If the *flag* parameter is set to **PvmTaskHost**, *where* must contain the *hostname*. Valid *hostnames* are only those of the hosts participating with the virtual machine (and cannot be longer than 64 characters). If the full *hostname* (for example, **hostA.domain**) is used, the routine removes the domain, so that only if the short alias (for example, **hostA**) has been defined, the routine is successful. The dotted addresses (for example, `9.87.1.12`) are not valid values for identifying hosts. If a flag is set to **PvmTaskArch** it must contain the host architecture. Only “RS6K” is valid with the **PvmTaskArch** flag.

ntask
Number of processes to be started.

tids
An array of integers returned by **pvm_spawn**. Each element of the array contains the task identifier of a new process, or, if that particular spawning attempt failed, it contains an error code.

nt
Return code. It contains the actual number of processes that have been started (that must be equal to *ntasks*) if the call was successful. If it is positive but less than *ntasks*, only a limited number of spawning processes were successful, and the proper element of the *tids* array contains the corresponding error code. If it is negative, an invalid parameter was supplied.

C Example

```
int procs[MAXPRC];
int flag;
char *args[32];

flag = PvmTaskHost;
for (j=0,i=2; i<argc,j<31;j++,i++)
  args[j]=strdup(argv[i]);
host = "HOST-A";

if ((nt = pvm_spawn ("child", args, flag, host, 1, procs)))
  for (i=0; i<n;i++) {
    if (procs[i]) <0 )
       printf ("error spawning task %d",i);
  }
```

FORTRAN Example

```
int procs(MAXPRC)

call pvmfspawn ('child', 0, '*', MAXPRC-1, procs(2), numt);
if (numt.lt.(MAXPRC-1)) then
   do i=2,MAXPRC
     if (procs(i).lt.0) then
       print *, 'Error initiating process node', i
     endif
   enddo
endif
```

Error Conditions

PvmSysErr	System error. Process not enrolled.
PvmBadParam	*task* is NULL. *ntasks* less than 1.
PvmNoHost	The *hostname* specified is NULL, an empty string or a host is not included in the virtual machine.
PvmNoFile	The executable file does not exist or cannot be found. The executable *filename* is an empty string.
PvmOverFlow	Process name too long.

PvmOutOfRes	An IBM PVMe for AIX process is already running on the supplied host. There are no available hosts.

pvm_start_pvmd

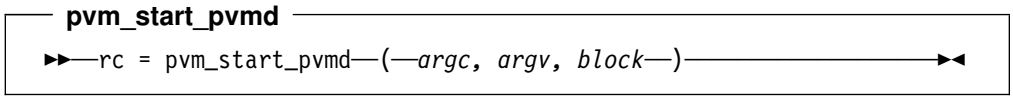
Description

The **pvm_start_pvmd** routine allows a process to start an IBM PVMe for AIX session, that is, to start the IBM PVMe for AIX daemon on a set of hosts. It receives the number of arguments (as parameters) that are to be passed to the daemon and an array of strings containing them. The **hostlist** file is the typical argument to give to the daemon. The FORTRAN routine is not allowed to pass arguments to the daemon.

The last parameter (*block*) specifies whether the routine must wait until startup completes or not. If *block* is `0`, you should not call an IBM PVMe for AIX routine immediately afterwards.

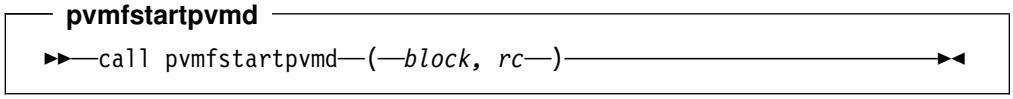
If an IBM PVMe for AIX session is already active, no new session is started, and no error message is returned.

C Syntax




```
int argc;
char **argv;
int block;
int rc;
```

FORTRAN Syntax




```
integer block
integer rc
```

Parameters

argc
Number of arguments that are passed to the daemon.

argv
Array of pointers to the arguments that are passed to the daemon. It must be at least *args* long.

block
Integer flag that can take the following values:

0	The routine returns immediately.
1	The routine blocks until daemon startup has been completed.

rc
Return code. It returns a `0` (zero) if the call was successful. It returns a negative value if it was unsuccessful.

C Example

```
if (pvm_start_pvmd (argc-1, &argv[1], 1))
  printf ("Error while starting the PVMe session\n");
```

FORTRAN Example

```
integer rc

call pvmfstartpvmd (1, rc)
if (rc.lt.0) then
   call pvmfperror ("Error starting a PVMe session")\n");
endif
```

Error Conditions

PvmSysErr	System error.
PvmBad Param	Invalid *block*.

pvm_tasks

Description

The **pvm_tasks** routine returns information about the IBM PVMe for AIX processes currently running. The first parameter allows you to specify the processes you are interested in. This could be:

- All processes running all on all the hosts
- All processes running on a specific host
- A single process with a supplied identifier

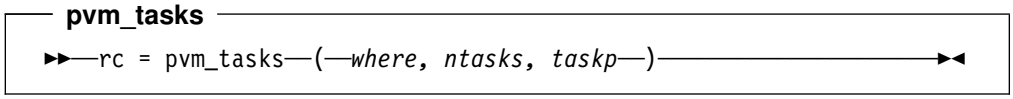
In the second argument (*ntasks*) the routine returns the number of processes you were interested in.

The C routine allocates memory space for a set of **pvmtaskinfo** structures (one for each process) and fills them with each process data. The address of the array is returned, and the allocated memory is available until the next call to the routine.

The FORTRAN routine returns data at each call for a single process, and cycles through all the tasks. If it is called *ntasks* times, the data of all the processes is returned, one at each call.

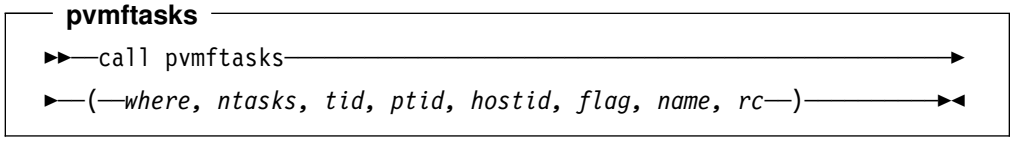
Data returned from this routine is the same as data shown by the **ps** command from the IBM PVMe for AIX console.

C Syntax




```
int where;
int *ntasks;
struct pvmtaskinfo **taskp;
int rc;
```

FORTRAN Syntax




```
integer where
integer ntasks
integer tid
integer ptid
integer hostid
integer flag
character*32 name
integer rc
```

Parameters

where
Integer indicating which tasks to return information about. It can take the following values:

0	Information about all IBM PVMe for AIX processes is returned.
host id	Information about the tasks running only on the host identified by the host identifier is returned.
task id	Information about the single task identified by *tid* is returned.

ntask
Integer where the routine returns the number of processes within the virtual machine (if the first of the above options was used), on the supplied host (if the second one was used), 1 (if the third option was used and the named process exists and is properly running).

taskp
Points to an array of **pvmtaskinfo** structures. The array will be *ntasks* long. Each structure contains data about a single task, and has the following shape:

```
struct pvmtaskinfo {
   int ti_tid                  /* process task id        */
   int ti_ptid                 /* id of the parent       */
   int ti_host                 /* id for the host        */
   int ti_flag                 /* process status         */
   char *ti_a_out              /* name of the executable */
   int ti_pid;                 /* task's AIX pid         */
} taskp
```

tid
Integer where the process identifier of a particular task is returned (see the *Description* on page 202).

ptid
Integer where the identifier of the parent of the task *tid* is returned (see the *Description* on page 202).

hostid
Since PVMe does not have a pvmd daemon on each host, this parameter returns a host identifier number rather than a pvmd's tid.

flag
Status is `0` (zero) if the process is properly running, or a negative number if it does not exist or did not yet enroll.

name
String where the name of the executable of *tid* is returned (for manually started processes too).

rc
Return code. It returns a `0` if the call was successful. It returns a negative value if it was unsuccessful.

C Example

```
int ntasks, rc;
struct pvmtaskinfo *tsk;
  .....
if (!(rc = pvm_tasks (0, &ntasks, &tsk))) {
  printf ("%d tasks are currently running\n", ntasks);
  printf ("    TID     PTID     HTID      NAME    FLAG\n");
  for (i=0; i<ntasks; i++) {
    printf ("%8d %8d %8d %12s %4d\n", tsk->ti_tid, tsk->ti_ptid,
                     tsk->ti_host, tsk->ti_a_out, tsk->ti_flag);
    tsk++;
  }
}
```

FORTRAN Example

```
      integer ntasks, rc, i
      integer tid(MAX), ptid(MAX), htid(MAX), flag(MAX)
      character*16 names(MAX)
          .....
      i=1
      print *, '    TID     PTID     HTID      NAME    FLAG'
66    call pvmftasks (0, ntasks, tid(i), ptid(i), htid(i), flag(i),
     $                 names(i), rc)
      print *, tid(i), ptid(i), htid(i), names(i), flag(i)
      i=i+1
      if (i.le.ntasks) goto 66

      print *, ntasks ,'are currently running'
```

Error Conditions

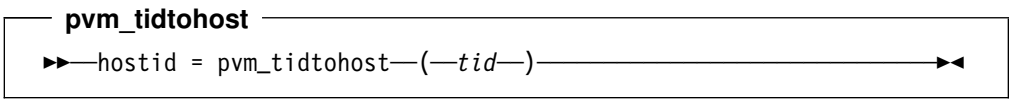
PvmSysErr	System error. Process not enrolled.
PvmBadParam	*ntask* is a NULL pointer.

pvm_tidtohost

Description

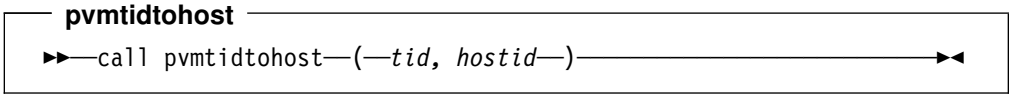
The **pvm_tidtohost** routine returns a host identifier for the host on which *tid* is running. This is not a *tid* of a pvmd since PVMe does not have a daemon on every node. This host id is usable as an argument to **pvm_tasks**.

C Syntax




```
int tid;
int hostid;
```

FORTRAN Syntax




```
integer tid
integer hostid
```

Parameters

tid
Task identifier of an IBM PVMe for AIX process.

hostid
Return code. Host identifier for the host the supplied process is running on.

C Example

```
int hostid, tid;

hostid = pvm_tidtohost (tid);
```

FORTRAN Example

```
integer hostid, tid
call pvmftidtohost (tid, hostid);
```

Error Conditions

PvmBadParam Invalid *tid*.

pvm_trecv

Description

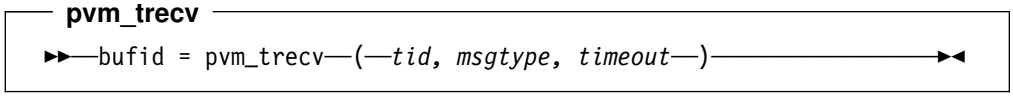
The **pvm_trecv** routine receives a message, with the specified *msgtype* coming from the process with the specified *tid*, and places it in the current receive buffer. A -1 can be used as a wildcard both for the *tid* and the *msgtype* parameters.

The **pvm_trecv** routine returns immediately if a message that matches the specified *tid* and *msgtype* has arrived. If it hasn't, it blocks and waits for the proper message. However, if it does not arrive within a user-specified *timeout*, the routine returns without a message.

Once the **pvm_trecv** routine returns with a rc > 0, the data in the current receive buffer may be unpacked into your memory space using the **pvm_upk***xxx* routines.

C Syntax

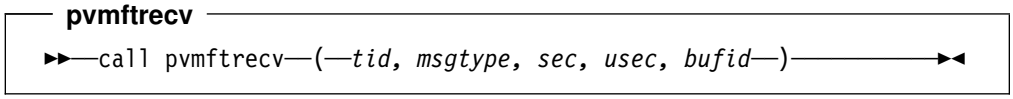
pvm_trecv




```
int tid;
int msgtype;
struct timeval *timeout;
int bufid;
```

FORTRAN Syntax

pvmftrecv




```
integer tid
integer msgtype
integer sec
integer usec
integer bufid
```

Parameters

tid
: Integer task identifier of the sender process. A -1 means that messages coming from any other IBM PVMe for AIX process can be accepted.

msgtype
: Integer message type (>= 0 or -1) of the message to be received. If a -1 is specified, the first message coming from the specified *tid* process is received, regardless of the message type.

timeout
: In C programs the **tv_sec** and **tv_usec** fields of the *timeout* structure specify how long the routine must wait for the message before returning to the caller program. If both are set to 0, the routine behaves like **pvm_nrecv**. Struct timeval is declared in <sys/time.h).

sec **and** *usec*
In FORTRAN programs, these parameters specify the number of seconds and microseconds the routine must wait for the message to arrive, before returning to the caller program. If both are set to `0`, the routine behaves like **pvm_nrecv**.

bufid
Return code. If the call was successful, it returns a positive integer that indicates the buffer where the message was received. If a matching message did not arrive in the given time span, it returns `0`. It returns a negative number if an error occurred.

C Example

```
int buft, rc, msg;
float data[MAXDATA];
struct timeval timeout;
   ....
timeout.tv_sec = 10;
timeout.tv_usec = 0;

buft = pvm_trecv (-1, 999, &timeout);
if (buft >0) {
   pvm_upkint(&n, 1, 1);
   pvm_upkfloat(&data, n, 1);
}
```

FORTRAN Example

```
integer MAXDATA
parameter (MAXDATA=1024)
integer buf, parent
real*8 data(MAXDATA)
   ...
call pvmfparent (parent)
call pvmftrecv (parent, -1, 0, 5000, buf)
if (buf.gt.0) then
  call pvmfunpack (REAL8, data, MAXDATA, 1)
else
  print *, 'Error receiving from parent'
endif
```

Error Conditions

PvmSysErr	System error. Process not enrolled.
PvmBadParam	*msgtype* less than `-1`. Invalid *tid*.

pvm_unpackf

Description

The **pvm_unpackf** routine unpacks data of various data types from the active receive buffer into arrays of proper data type, using a format like **printf**. The syntax is:

format	init fmt*
init	null | '%' '+'
fmt	'%' count stride modifiers fchar
fchar	'c' | 'd' | 'f' | 'x' | 's'
count	null | [0-9]+ | '*'
stride	null | '.' ([0-9]+ | '*')
modifiers	null | modifiers mchar
mchar	'h' | 'l' | 'u'

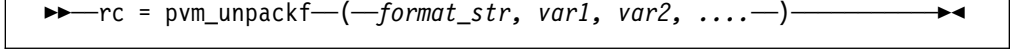
Where:

- Formats:
  - **+** Means **initsend** - must match an integer (*encoding*) in the parameter list.
  - **c** Pack/unpack bytes
  - **d** Pack/unpack integers
  - **f** Pack/unpack floats
  - **x** Pack/unpack complex float
  - **s** Pack/unpack strings
- Modifiers
  - **h** Short (int)
  - **l** Long (int, float, complex float)
  - **u** Unsigned (int)

Note: All variables are passed as addresses.

C Syntax

pvm_unpackf




```
const char *fmt;
int rc;
```

Parameters

fmt
A format expression like **printf** indicating data and data types that will be unpacked.

rc
Return code. It returns a `0` if the call was successful. It returns a negative number if it was unsuccessful.

var1, var2, **...**
Variables that correspond to the format line which define how the variables are outputted.

C Example

```
#define SZ  4
double float LFLOAT_array[SZ];
int INT_array[SZ];
int rc;

rc = pvm_unpackf ("%*d %4lf", SZ, INT_array, LFLOAT_array);
```

Error Conditions

PvmSysErr	System Error. Process not enrolled.
PvmNoData	Past end of buffer.
PvmBadParam	*encoding* invalid.
PvmNoBuf	No active receive buffer.

pvm_upkint, pvm_upkcplx, pvm_upkdcplx, pvm_upkfloat, ...

Description

The **pvm_upk** routines unpack arrays of various data types from the active receive buffer. The arguments of the C routines are:

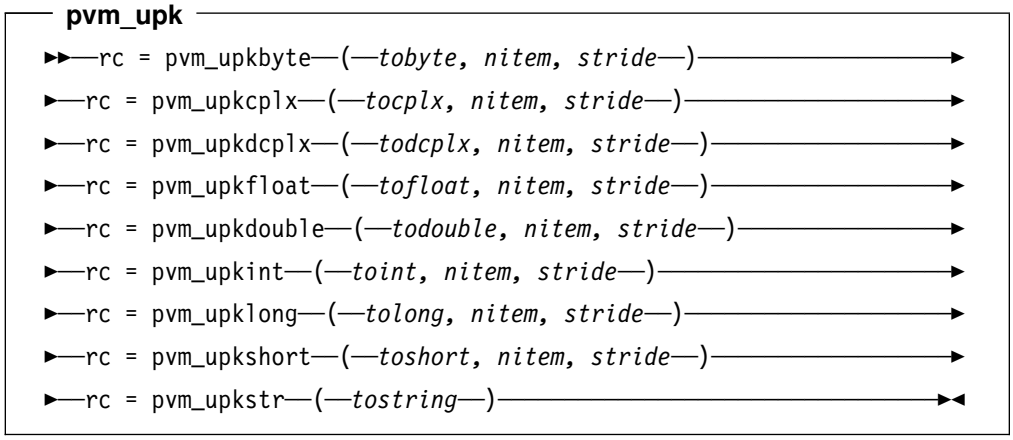
- A pointer to the array to be unpacked into
- The total number of items to be unpacked
- The stride to be used to unpack data

The only exception is the **pvm_upkstr** routine that unpacks a character string and does not need either the *nitem* or the *stride* parameters.

There is a single routine to unpack data in FORTRAN programs. The first parameter is expected to be a flag identifying the type of data to be unpacked. The *nitem* and *stride* parameters have the same meaning as the C routine. If data type is **STRING**, *nitem* must contain the number of characters in the string, and *stride* is expected to be 1.

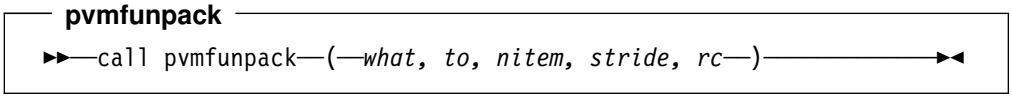
Note: A single message may contain different data types. Data must be unpacked exactly as it was packed to insure data integrity.

C Syntax




```
char *tobyte;
float *tocplx;
double *todcplx;
float *todfloat;
double *todouble;
int *toint;
long *tolong;
short *toshort;
char *tostring;
int nitem;
int stride;
int rc;
```

FORTRAN Syntax




```
integer what
character to (*)
integer nitem
integer stride
integer rc
```

Parameters

what
Integer flag supplied to the FORTRAN routine to specify the type of data to be copied. It may take any of the following values:

STRING	0
BYTE1	1
INTEGER2	2
INTEGER4	3
REAL4	4
COMPLEX8	5
REAL8	6
COMPLEX16	7

tobyte
todcplx
tofloat
todouble
toint
tolong
toshort
tostring
to
Starting address of the memory region where data is to be copied. It must point to the proper data type in C or a variable name in FORTRAN. The allocated memory region must allow for at least *nitem*stride* items of that data type.

nitem
Number of data items that are copied in the memory region starting from the specified address.

stride
The stride used when unpacking the items.

rc
Return code. It returns a `0` if the call was successful. It returns a negative number if it was unsuccessful.

C Example

```
double matrix[MAXROWS*MAXCOLS];
int nrows, ncols, nextmsg;

nextmsg=(k+1);
if (pvm_recv(-1,nextmsg) >=0 ) {
  pvm_upkint (&nrows,1,1);
  pvm_upkint (&ncols,1,1);
  pvm_upkdouble(matrix, nrows*ncols, 1);
}
```

FORTRAN Example

```
parameter (MAXN=128)
real*8 vec(MAXN)
integer*4 maxn, n, info

call pvmfrecv (-1,-1, info)
if (info.ge.0) then
   call pvmfunpack (INTEGER4, n, 1, 1, info)
   call pvmfunpack (REAL4, vec(1), n, 1, info)
endif
```

Error Conditions

PvmSysErr	System error. Process not enrolled.
PvmBadParam	*count* less than 0 (zero). Invalid data type.
PvmNoData	*count* greater than the actual message length.
PvmNoBuf	No active receive buffer.

Chapter 8. Diagnosing Problems

The following is a list of System messages (Informational and Error messages) that you may encounter during the operation of PVMe. Each list item provides the message number, message text, parameter(s), and the action you should take.

Informational Messages

INFO001 **PVMD: WARNING Task** *taskid (hex)* **on host** *hostname* **is still alive but will be killed.**

INFO002 **PVMD:** *nodenum* **nodes allocated from LL.**

INFO003 **PVMD: Assuming** *cwname* **as control workstation.**

INFO004 **PVMD: Trying to get** *nodenum* **nodes.**

INFO005 **[***taskid (hex)***]***hostname***:** *message***.**

INFO006 **Local task:** *message***.**

INFO007 **PVMD: Handling duplicated packet from** *hostname***.**

INFO008 **PVMD: Using** *hostname* **with dx=***dxpath***/pvmd3e, ep=***eppath***.**

INFO009 **PVMD: WARNING Option -exec was not specified.**

INFO010 **PVMD: Resending IP packet** *packetnum***...(***retries* **more attempts).**

INFO011 **PVMD: WARNING Received an invalid IP packet** *packetnum* **(***lastgoodpktnum***).**

INFO012 **PVMD: Starting** *command***.**

INFO013 **PVMD: Ready for** <*numhosts*> **hosts.**

INFO014 **PVMD: New epoch** *epochnum***.**

INFO015 **PVMD: Resource Manager pool id is** *poolid***.**

INFO016 **PVMD: Found option** *optioname***.**

INFO017 **Assuming ksh as a default.**

INFO018 **Usage pvmd3e [**<**options**>**] [**<**number**> | <**filename**>**]**

INFO019 **PVMD: WARNING Local host (***myhostname***) has been added to the session. Any task on it will run in IP mode.**

INFO020 **PVMD: WARNING Task** *taskid (hex)* **on host** *hostname* **is still alive and will not be killed.**

INFO501 **[***taskid***]: WARNING Collective routines are implemented for 'stable' groups only.**

INFO502 **[***taskid***]: WARNING** *variablename* **was not set, using** *default***.**

Error Messages

EMSG001 **PVMD: 2507-001 No other PVMe session must run on** *hostname* **when it is added to the configuration.**

Explanation: When adding a new host to the virtual machine configuration, only the first session is requested.

User Response: Stop the other PVMe sessions and/or reset the nodes.

EMSG002 **PVMD: 2507-002 Error in barrier** *barriername***: wrong counter.**

Explanation: Barrier quorum was different among the PVMe tasks.

User Response: Make barrier count the same across all tasks in the user code.

EMSG003 **PVMD: 2507-003 Group** *groupname* **not found.**

Explanation: No group with the given name exists.

User Response: Check user code for an incorrect group name.

© Copyright IBM Corp. 1993, 1994, 1995

EMSG004 **PVMD: 2507-004 All attempts to kill task** *taskid (hex)* **on host** *hostname* **have failed. Please check for orphaned processes.**

Explanation: The daemon is unable to kill the tasks indicated.

User Response: Check for (and kill) orphaned processes.

EMSG005 **PVMD: 2507-005 An error occurred during the cleanup routine. Please ensure that all tasks have exited.**

Explanation: The signal handler/cleanup routine could not be completed.

User Response: Check that all tasks have exited.

EMSG006 **PVMD: 2507-006 Error while releasing host** *hostname* **not in the host table.**

Explanation: The daemon was unable to determine a hostname, based on its index in the table.

User Response: Contact PVMe support

EMSG007 **PVMD: 2507-007 Invalid local AFS authentication.**

Explanation: The daemon was unable to get the AFS token.

User Response: Contact system administrator

EMSG008 **PVMD: 2507-008 Error initializing** *hostname***. Please check for one of the following causes: Kernel extension not loaded. PVMe not installed in** *dxdirectory***.**

Explanation: The daemon could not initialize the session on the indicated host.

User Response: Check to see if PVMe is installed on the host, and if the PVMe kernel extension is loaded on it.

EMSG009 **PVMD: 2507-009 LL reported error** *errnum* **during node allocation.**

Explanation: The LL API has returned an error from ll_get_hostlist().

User Response: Contact LL support

EMSG010 **PVMD: 2507-010 Unable to connect to Resource Manager.**

Explanation: The RM API has returned an error in jm_connect().

User Response: Check if RM is up and running.

EMSG011 **PVMD: 2507-011 Error** *errnum* **while requesting** *hostname* **from Resource Manager.**

Explanation: The RM API has returned an error in jm_allocate().

User Response: The next few messages give more information about the error cause.

EMSG012 **2507-012 Node is not available.**

User Response: Modify the hostlist file to ask for non-allocated nodes.

EMSG013 **2507-013 MPCI partition could not be initialized. Perhaps another process is running on that host.**

Explanation: MPCI partition could not be initialized.

User Response: Check for another process running on the host.

EMSG014 **2507-014 Invalid Pool identifier.**

Explanation: The pool id specified is invalid.

User Response: Use a valid pool id.

EMSG015 **2507-015 Invalid SubPool identifier.**

Explanation: The subpool id specified is invalid.

User Response: Use a valid subpool id.

EMSG016 **2507-016 Invalid (misspelled) name.**

Explanation: The hostname specified is invalid.

User Response: Check to see whether the hostname specified is valid.

EMSG017 **2507-017 There was a fault in the High Performance Switch.**

Explanation: The HPS has faulted.

User Response: Contact system administrator.

EMSG018 **2507-018 System Error.**

Explanation: A system error has occurred.

User Response: Contact System Administrator.

EMSG019 **PVMD: 2507-019 Cannot decode message, type** *msgtype***.**

Explanation: The message type cannot be decoded.

User Response: Please check for misaligned version between library daemon and console.

EMSG020 **PVMD: 2507-020 Option <-resd no> and a number of nodes as arguments are incompatible.**

Explanation: The two daemon options -resd no (bypassing the RM) and a number of nodes (which implied the use of an RM) are mutually exclusive.

User Response: If using -resd no, do not specify a number also.

EMSG021 **PVMD: 2507-021 Unable to allocate the nodes, quitting.**

Explanation: The nodes requested for a PVMe session could not be allocated.

User Response: Ask for a lesser number of nodes, or a different set of nodes. Also, check to see if the RM is up and running.

EMSG022 **PVMD: 2507-022 Error starting task.**

Explanation: The daemon was unable to start a PVMe task.

User Response: Check for misspelled executable/path, permission bits, etc..

EMSG023 **PVMD: 2507-023 Error getting tokens for** *cellname***.**

Explanation: The daemon was unable to get a token for the cell.

User Response: Contact System Administrator

EMSG024 **PVMD: 2507-024 Command line error:** *option***.**

Explanation: There is an error in the daemon command line.

User Response: Specify daemon options correctly.

EMSG025 **PVMD: 2507-025 Hostfile error:** *errorstring***.**

Explanation: There is an error in the hostfile format.

User Response: Specify the hostfile correctly.

EMSG026 **PVMD: 2507-026 Option will be ignored.**

Explanation: The option is not recognized by the daemon.

User Response: Specify the option correctly.

EMSG027 **PVMD: 2507-027 Line will be ignored.**

Explanation: The line in the hostfile is not in a format recognizable by the daemon.

User Response: Specify the hostfile line correctly.

EMSG028 **PVMD: 2507-028 Invalid host name.**

Explanation: While not using the RM, the console is being invoked on a host that is not specified in the hostfile.

User Response: Either change the host from which the console is being started, or add this host to the virtual machine.

EMSG029 **PVMD: 2507-029 Cannot open PVMe configuration file.**

Explanation: The daemon is unable to open the configuration file pvm6kcnf (either under $HOME or under /usr/lpp/pvme).

User Response: Check to see that the file exists in at least one of the two places specified above, and that it is readable.

EMSG030 **PVMD: 2507-030 Format error in configuration file:** *error***.**

Explanation: There configuration file has a format error.

User Response: Specify the configuration file stanzas correctly.

EMSG031 **PVMD: 2507-031 Cannot open** *hostfile***.**

Explanation: The daemon is unable to open the hostfile specified.

User Response: Check for the existence and read permission of the hostfile.

EMSG032 **PVMD: 2507-032 Local host (***hostname***) is not included in the hostfile.**

Explanation: Unless the environment variable, PVMe_CLH has been set, the local host must be included in the hostfile.

User Response: Either include the local host in the hostfile, or set PVMe_CLH.

EMSG033 **PVMD: 2507-033 SHELL environment variable is not set.**

Explanation: The SHELL environment variable is not set.

User Response: Set the SHELL environment variable.

EMSG034 **PVMD: 2507-034 Shell syntax for** *shellname* **is unknown.**

Explanation: The syntax for the given shell is not known.

User Response: Use either Korn shell, C shell or its derivatives.

EMSG035 **PVMD: 2507-035 No node is available to start a new process. Please allocate more nodes, or wait until the end of the epoch.**

Explanation: There are no more nodes to start tasks on.

User Response: Either include more nodes in the virtual machine or wait till the epoch is over (and some nodes in the virtual machine are available).

EMSG036 **PVMD: 2507-036 Host** *hostname* **is not part of the virtual machine.**

Explanation: The daemon is trying to do execute a remote command on an host which is not included in the virtual machine.

User Response: Check the program for a misspelled host name Contact PVMe support.

EMSG037 **PVMD: 2507-037 Memory allocation error in** *sourcefile***:** *sourceline***.**

Explanation: An error in memory allocation at the specified line.

User Response: Contact System Administrator (bad ulimit)

EMSG038 **PVMD: 2507-038 Socket** *socket_call* **failed in** *sourcefile***:** *sourceline***.**

Explanation: The specified socket call failed.

User Response: Contact PVMe support

EMSG039 **PVMD: 2507-039 Error sending** *message_type* **message to** *taskid***.**

Explanation: The daemon encountered an error in sending a message to the specified task.

User Response: Contact PVMe support.

EMSG040 **PVMD: 2507-040 Internal Error** *probd.error_code* **(***error_value***) in** *sourcefile***:** *sourceline***.**

Explanation: Internal error.

User Response: Contact PVMe support.

EMSG041 **PVMD: 2507-041 Unexpected output from** *outputsource* **in** *sourcefile***,** *sourceline***.**

Explanation: The daemon read an unexpected string from the ghost process.

User Response: Check for a misalignment between pvmd6kgh and pvmd3e.

EMSG042 **PVMD: 2507-042 Only one user-space process is allowed per SP node and a PVMe task is already running on** *spnodename***.**

Explanation: More than one user-space task was attempted on a node.

User Response: Either limit one user-space task per node, or use the IP mode.

EMSG043 **PVMD: 2507-043 A task is enrolling from host** *hostname***, which is not part of the PVMe virtual machine.**

Explanation: A process is enrolling from a host that is not part of the virtual machine.

User Response: Either change the host from which the task is being started, or add this host to the virtual machine.

EMSG044 **PVMD: 2507-044 More than one PVMe process is attempting to update the session database on host *hostname*. Please wait a few seconds and retry.**

Explanation: The daemon is unable to access the session registering database on the specified node because some other PVMe session (perhaps belonging to another user) is currently updating that database.

User Response: Wait a few moments and try again.

EMSG045 **PVMD: 2507-045 The number of PVMe sessions on *hostname* exceeds limit. Wait until one terminates or use daemon option -r to force a reset.**

Explanation: The number of PVMe sessions on a node has exceeded the maximum allowed.

User Response: Limit the number of sessions to the maximum allowed. Check for dead sessions (daemon killed abruptly). Use the indicated option if everything else fails.

EMSG046 **PVMD: 2507-046 The *systemcall* system call failed at *sourcefile*: *sourceline*.**

Explanation: Error in a system call.

User Response: Contact system Administrator or Contact PVMe support.

EMSG047 **PVMD: 2507-047 Too many (*illegal_value*) *resource_type*, max is: *limit*.**

Explanation: The number of a given resource has exceeded the corresponding maximum limit.

User Response: Limit the resource number to the maximum allowed.

EMSG048 **PVMD: 2507-048 Illegal value (*valuestr*) for *resourcename*.**

Explanation: An illegal value has been specified for a resource.

User Response: Specify a legal value for the resource.

EMSG049 **PVMD: 2507-049 Task *taskid* was built with an incompatible library.**

Explanation: There is an inconsistency in the task libraries?

User Response: Relink the tasks with compatible libraries.

EMSG050 **PVMD: 2507-050 Possible TCP/IP timeout in batch startup. Startup will be attempted *number* more times.**

Explanation: Very high network load can produce TCP/IP timeouts which can make an attempt to start a batch job fail.

User Response: If the message is sometimes seen and the retry works, TCP/IP network parameters may need tuning. If the retry attempts also fail there may be more serious network problems. Contact the system administrator.

EMSG501 **2507-501 Illegal value for *variablename*: *valuestring*.**

Explanation: Illegal value for the variable specified.

User Response: Specify a legal value.

EMSG502 **2507-502 PvmDataInPlace implemented only for stride one (pack/unpack).**

Explanation: The stride value is not 1.

User Response: Use a stride value of 1.

EMSG503 **2507-503 Unable to unpack** *unpacklen* **bytes from current buffer. (tag:** *tag* **src:** *sourcetid***, len:** *msglen***).**

Explanation: The number of bytes in the incoming message is less than the expected value.

User Response: Re-examine your program logic.

EMSG504 **2507-504 Cannot send to [***taskid, desttid***]. Not enrolled within timeout.**

Explanation: The destination task is not enrolled.

User Response: Contact PVMe support

EMSG505 **2507-505 Cannot send to [***taskid, desttid***]. Task is dead.**

Explanation: The destination task has called pvm_exit().

User Response: Re-examine your program logic.

EMSG506 **2507-506 The** *systemcall* **system call failed at** *sourcefile***:** *sourceline***.**

Explanation: System call error.

User Response: Contact System Administrator or Contact PVMe Support.

EMSG507 **2507-507 Received an unrecognized packet from daemon.**

Explanation: The task has received an unrecognized packet from the daemon.

User Response: Contact system Administrator or Contact PVMe Support.

EMSG508 **2507-508 PVMe daemon returned an error in** *pvmcall***:** *sourceline***.**

Explanation: Error returned by daemon.

User Response: Contact system Administrator or Contact PVMe Support.

EMSG509 **2507-509 Unable to kill PVMe reader** *processid***, please check for runaway processes.**

Explanation: The PVMe reader could not be killed.

User Response: Check for orphaned tasks and kill these.

EMSG510 **2507-510 Error** *errornum* **in the PVMe reader.**

Explanation: The PVMe reader has encountered an error.

User Response: Contact System Administrator or Contact PVMe Support.

EMSG511 **2507-511 Error in the MPCI communication layer (***mpci_routine***:** *errnum***) at** *sourcefile***:** *sourceline***.**

Explanation: The MPCI layer has encountered an error.

User Response: Contact System Administrator or Contact PVMe Support.

EMSG512 **2507-512 Memory allocation error in** *sourcefile***:** *sourceline***.**

Explanation: Error in memory allocation.

User Response: Contact PVMe support or Contact System Administrator (bad u limit).

EMSG513 **2507-513 Number of network interfaces on** *hostname* **exceeds limit (***limit***).**

Explanation: The number of network interfaces on a given hostname is greater than the maximum allowed.

User Response: Contact PVMe support.

EMSG514 **2507-514 Cannot connect to the PVMe daemon.**

Explanation: The task is unable to connect to the daemon.

User Response: Check to see if the daemon is running.

EMSG515 **2507-515 Cannot open file** *filename***, error was:** *errnum***.**

Explanation: The task is unable to open a file.

User Response: Check for the existence and permissions of the file.

EMSG516 **2507-516 File format error** *errorstr* **at** *sourcefile***:** *sourceline***.**

Explanation: Format error in file.

User Response: Use correct format.

EMSG517 **2507-517 String too long, only** *numchar* **characters will be used.**

Explanation: The string encountered is longer than the maximum.

User Response: Rename the executable so that it will fit.

EMSG518 **2507-518 Illegal (***nummsg***) number of messages for class** *classnum***.**

Explanation: For the given message class, the number of messages is larger than the maximum limit for that class.

User Response: Fix user code.

EMSG519 **2507-519 Too many (***mesgnum***) messages were defined, max. is** *maxmesgnum***.**

Explanation: The number of message types is greater than the maximum.

User Response: Fix user environment.

EMSG520 **2507-520 Space to allocate message slots (***space***) exceeds max.:** *spacemax***.**

Explanation: Space for message slots allocation is greater than the maximum.

User Response: Fix user environment.

EMSG521 **2507-521 The number of tasks in a** *cclname* **(***cclntasks***) is not equal to the group size (***gsize***).**

Explanation: The number of tasks in a collective communication routine does not match the group size.

User Response: Fix user code.

EMSG522 **2507-522 The IP reader has exited.**

Explanation: The IP reader has exited.

User Response: Contact System Administrator for installation problems.

EMSG523 **2507-523 The MPCI interface is not available, using only the IP path.**

Explanation: The MPCI interface is not available.

User Response: Contact system administrator.

EMSG524 **2507-524 Unable to start the** *"MPCI"|"IP"* **reader.**

Explanation: The task reader could not be started.

User Response: Contact system administrator/PVMe support for installation problems.

EMSG525 **2507-525 The number of tasks in a *cclname* must be greater than zero.**

Explanation: A collective communication routine has been called for zero number of tasks.

User Response: Check User Code (e.g. a broadcast to an empty group). Contact PVMe Support.

EMSG526 **2507-526 Warning: output from [*sourcetid*] cannot be written to file.**

Explanation: Unable to write a task's output to a file.

User Response: Check User Code (misspelled directories) or Contact System Administrator (permission bits).

EMSG527 **2507-527 Invalid operation (*opno*) requested.**

Explanation: An invalid operation has been requested.

User Response: Fix user code.

EMSG528 **2507-528 String format error in *sourcefile*: *sourceline*.**

Explanation: Error in string format.

User Response: Contact PVMe support.

EMSG529 **2507-529 Too many outstanding (class: *classnum*) messages.**

Explanation: There are too many outstanding messages in a class.

User Response: Fix user code and/or set the PVMe environment variables appropriately.

EMSG530 **2507-530 Internal error *errno* (*errno2*) in *sourcefile*: *sourceline*.**

Explanation: Internal error.

User Response: Contact PVMe support

EMSG531 **2507-531 Invalid number of *size* bytes buffers: *nbuffer* assuming *nbuffer2*.**

Explanation: The N64KBUFF (number of 64 Kb buffers allocated for the incoming PVMe messages) environment variable is set to an incorrect value (less than one or greater than maximum).

User Response: Correct Value.

EMSG532 **2507-532 *pvm routine name* depends on a running IP process. Please define environment variable PVMe_IP=yes and restart daemon.**

Explanation: Some arguments to pvm_setopt and the pvm_catchout depend on having a running IP process as part of the session.

User Response: Halt the pvmd3e daemon and set environment variable PVMe_IP=yes before starting another session.

EMSG533 **2507-533 Value parameters to pvm_packf cannot be packed with PvmDataInPlace encoding. Invalid data will be sent.**

Explanation: PvmDataInPlace encoding requires the address of the data being packed. If parameters are passed to pvm_packf by value, the address is not available.

User Response: Use PvmDataRaw or PvmDataDefault encoding or pass the data by reference.

EMSG534 **2507-534 The value** *user_val* **for PVMe_IP_PORT is invalid. It must be between** *min_val* **and** *max_val***. The default value of** *default_val* **is used.**

Explanation: The port numbers below 5000 are reserved and PVMe uses up to 4 ports starting from the default or PVMe_IP_PORT specified point so the maximum is less than 65535.

User Response: Use a value with range.

EMSG535 **2507-535 The PVM function** *function_name* **is not supported in the current release of PVMe.**

Explanation: Avoid these functions in programs to be run on PVMe.

Appendix A. IBM PVMe for AIX Error Code Summary

All IBM PVMe for AIX routines return an integer code. The value of the integer code can be tested by the user program to see if the call was successful and if not, to take the proper action. If the call was successful, the return code will always be a positive value. If unsuccessful, a negative value will always indicate an error condition. This makes it easy to check the return code to detect an error.

Note: Two routines behave differently. The **pvm_spawn** (and the corresponding FORTRAN subroutine **pvmfspawn**), and the **pvm_delhosts** routines return the actual number of tasks started, and hosts deleted.

For these routines, a possible error condition can be detected comparing the return code to the *count* argument (containing the number of new processes to be started, and of hosts to be added or deleted).

Error codes depend on the particular error that occurs. A set of useful *define*s in the **pvm3.h** file link the value of each error code to a symbolic string. If your C applications include the **pvm3.h** file, such strings may be used in place of the corresponding numerical values.

Analogous strings are defined as integer variables in the **fpvm3.h** file, where they are also assigned the proper values. Your FORTRAN programs should include this file, if the mnemonic strings are going to be used in place of the corresponding numerical values.

The following table summarizes the error codes and their meaning.

Table 7 (Page 1 of 2). IBM PVMe for AIX Error Code Summary

Error Code	Value	Meaning
PvmOk	0	No error
PvmBadParam	-2	Parameter outside range of legal values.
PvmMismatch	-3	Barrier count mismatch
PvmOverFlow	-4	Name too long
PvmNoData	-5	Read past end buffer
PvmNoHost	-6	No such host
PvmNoFile	-7	No such executable
PvmNoMem	-10	Cannot get memory
PvmBadMsg	-12	Cannot decode message
PvmSysErr	-14	Not connected to daemon or daemon not responding
PvmNoBuf	-15	No current buffer
PvmNoSuchBuf	-16	Bad buffer ID
PvmNullGroup	-17	Null group name is illegal
PvmDupGroup	-18	Already in group
PvmNoGroup	-19	No group with that name
PvmNotInGroup	-20	Not in group

© Copyright IBM Corp. 1993, 1994, 1995

Table 7 (Page 2 of 2). IBM PVMe for AIX Error Code Summary		
Error Code	**Value**	**Meaning**
PvmNoInst	-21	No such instance in group
PvmHostFail	-22	Host failed
PvmNoParent	-23	No parent task
PvmNotImpl	-24	Function not implemented
PvmDSysErr	-25	Daemon system error
PvmBadVersion	-26	Pvmd-pvmd protocol mismatch
PvmOutOfRes	-27	Out of resources
PvmDupHost	-28	Host already configured
PvmCantStart	-29	Failed to exec new worker task pvmd
PvmAlready	-30	Worker task pvmd already running
PvmNoTask	-31	Task does not exist
PvmNoEntry	-32	No such (group, instance)
PvmDupEntry	-33	(group, instance) already exists

Appendix B. IBM PVMe for AIX Environment Variables

Release 3.1.0 of IBM AIX PVMe provides environment variables with which you can change the default values of IBM PVMe for AIX parameters.

These variables do not conflict with each other, and can be set simultaneously. If you want the environment variables to be passed to processes spawned by the IBM PVMe for AIX daemon, you need to group them together for the PVM_EXPORT variable in the shell which starts the master task, and export it as follows:

```
export PVM_EXPORT=VAR1:VAR2
```

(assuming Bourne shell-like syntax)

```
setenv PVM_EXPORT VAR1:VAR2
```

(assuming C shell-like syntax)

For example, if DISPLAY, PVMe_DATA_INTERRUPT and PVMe_MEMCPY are set, to make these available to all tasks you should export them as follows:

```
export PVM_EXPORT=PVMe_DATA_INTERRUPT:PVMe_MEMCPY:DISPLAY
```

(assuming Bourne shell-like syntax)

```
setenv PVM_EXPORT DISPLAY,PVMe_DATA_INTERRUPT&colon;PVMe_MEMCPY
```

(assuming C shell-like syntax)

IBM PVMe for AIX Environment Variables

PVMDPATH

Action Sets the path to the directory that contains the IBM PVMe for AIX daemons.

Usage

- Bourne shell:

  ```
  export PVMDPATH=/usr/tools/pvme/bin
  ```

- C shell:

  ```
  setenv PVMDPATH /usr/tools/pvme/bin
  ```

  PVMDPATH should be set in the shell where the daemon is started.

Default /usr/lpp/pvme/bin

© Copyright IBM Corp. 1993, 1994, 1995

PVMEPATH

Action Sets the search path to the directory that contains the IBM PVMe for AIX application executables.

Usage

- Bourne shell:

  ```
  export PVMEPATH=$HOME/pvm3/bin/RS6K_SP
  ```

- C shell:

  ```
  setenv PVMEPATH /u/pvmuser/pvm3/bin/RS6K
  ```

  PVMEPATH should be set in the shell where the daemon is started.

Default $HOME/pvm3/bin/RS6K

PVMHFN

Action Defines the filename to which the daemon writes the information that IBM PVMe for AIX applications need in order to contact it. An example is the daemon's socket address.

This variable is needed if you are running multiple IBM PVMe for AIX sessions, and your **$HOME** file system is shared among the nodes. Different files will be used by different sessions, without conflicts.

The **$PVMHFN** file will be created under your **$HOME**, or **$PVMEPATH** directories.

Usage

- Bourne shell:

  ```
  export PVMHFN=session.test
  ```

- C shell:

  ```
  setenv PVMHFN session.test
  ```

  PVMHFN must be set in the shell where you run the daemon, as well as in the shell where you start the master task.

Default .pvmdname

PVMe_CLH

Action When running the IBM PVMe for AIX daemon with the *hostlist* parameter, the daemon does not check for the presence of the local host in the *hostlist* file. You can force this check by setting this environment variable to **yes** before starting the daemon.

Usage

- Bourne shell:

  ```
  export PVMe_CLH=yes
  ```

- C shell:

  ```
  setenv PVMeCLH yes
  ```

  PVMe_CLH should be set in the shell where the daemon is started.

Default no

PVMe_DATA_INTERRUPT

Action Turns on or off MPCI interrupts. With MPCI interrupts turned off, the interrupt handler does not notify the task about incoming messages. This may improve performance, but there may be other side effects if there are a large number of small messages (4K and smaller), or pending messages that are too numerous or too large. In this case, a deadlock may occur, so you should use this feature very carefully.

By default, **PVMe_DATA_INTERRUPT** is set to off. If you experience problems, you should set **PVMe_DATA_INTERRUPT** to on.

Usage Before starting the task which spawns all the other tasks, you should set and export this variable to all the tasks as follows:

- Bourne shell:

```
export PVMe_DATA_INTERRUPT=no or export PVMe_DATA_INTERRUPT=yes
export PVM_EXPORT=PVMe_DATA_INTERRUPT
```

- C shell:

```
setenv PVMe_DATA_INTERRUPT no or setenv PVMe_DATA_INTERRUPT yes
setenv PVM_EXPORT PVMe_DATA_INTERRUPT
```

Default no

PVMe_MEMCPY

Action Uses a fast memory copy routine which exploits the POWER2 architecture features. This feature can be used only if all the participating nodes have POWER2 architecture.

Usage Before starting the task which spawns all the other tasks, you should set and export this variable to all the tasks as follows:

- Bourne shell:

```
export PVMe_MEMCPY=power2
export PVM_EXPORT=PVMe_MEMCPY
```

- C shell:

```
setenv PVMe_MEMCPY power2
setenv PVM_EXPORT PVMe_MEMCPY
```

Default Not set.

PVMe_BARRTYPE

Action Uses a safe algorithm for barriers. IBM PVMe for AIX uses a new fast algorithm for the barriers. This algorithm assumes that all the instances of a group participate in a barrier. However, in case the count is not equal to the total number of instances, this variable can be set so that IBM PVMe for AIX uses the old algorithm which can handle such a scenario.

Usage Before starting the daemon, and the task which spawns all the other tasks, you should set and export this variable (this variable must be set for both the daemon and the tasks) as follows:

- Bourne shell:

```
export PVMe_BARRTYPE=CENTRAL
export PVM_EXPORT=PVMe_BARRTYPE
```

- C shell:

```
setenv PVMe_BARRTYPE CENTRAL
setenv PVM_EXPORT PVMe_BARRTYPE
```

N64KBUFF

Note: The environment variable NBUFNODE is replaced by N64KBUFF. The name has changed because there are now two environment variables related to the number of buffers and the new name makes the distinction clearer (the other name is N256BUFF). The value of N64KBUFF indicates, in the present version, the total number of buffers allocated to each node whereas the previous version, NBUFNODE, indicated the number of buffers per node so that the total number of buffers per node was the product of the number of nodes and the value of NBUFNODE.

Action Increases the number of 64K buffers posted in advance to receive messages. IBM PVMe for AIX posts a number of buffers to receive unsolicited messages. The number of buffers is, by default, equal to four times the number of nodes with a minimum of 16 buffers and a maximum of 512 buffers. In other words, for less than four nodes, 16 buffers are posted; for five nodes, 20 buffers are posted; for 6 nodes, 24 buffers are posted, etc. Although this is usually enough, some applications may still run out of buffers. This situation can lead to deadlocks or other pathological problems. To solve this problem, you can increase the number of buffers by setting the variable N64KBUFF. This variable controls the number of buffers that are posted, and can be any value between 1 and 512.

Usage Before starting the task which spawns all the other tasks, you should set and export this variable, in the shell of the master task, to all the tasks as follows:

- Bourne shell:

```
export N64KBUFF=<# of buffers>
export PVM_EXPORT=N64KBUFF
```

- C shell:

```
setenv N64KBUFF <# of buffers>
export PVM_EXPORT N64KBUFF
```

SP_NAME

Action Allows starting the IBM AIX PVMe daemon and console from any RISC System/6000 workstation. To allow any RISC System/6000 workstation to run the IBM PVMe for AIX daemon, as explained in "Starting IBM AIX PVMe from the Control Workstation", this variable must be set equal to the hostname of the control workstation.

Usage Before starting the daemon on a RISC System/6000 workstation, you should set this variable as follows:

- Bourne shell:

```
export SP_NAME=<hostname of the control workstation>
```

- C shell:

```
setenv SP_NAME <hostname of the control workstation>
```

SP_NAME should be set in the shell where the daemon is started.

N256BUFF

Action Increases the number of 256 byte buffers posted in advance to receive information related to groups management. By default, the number of these buffers is equal to the number of nodes. Although the default value is usually enough, some applications which perform a high number of group operations like **pvm_joingroup** or **pvm_lvgroup** may run out of buffers, which could lead to deadlocks or other pathological situations. If this happens, you will see an error message that instructs you to take corrective action. You can increase the number of buffers by setting the N256BUFF environment variable. The number can be changed from a minimum of one buffer to a maximum of twice the number of nodes.

Usage Before starting the task which spawns all the other tasks, you should set and export this variable, in the shell of the master task, to all the tasks as follows:

- Bourne-shell:

```
export N256BUFF=<# of buffers>
export PVM_EXPORT=N256BUFF
```

- c-shell:

```
setenv N256BUFF=<# of buffers>
setenv PVM_EXPORT=N256BUFF
```

PVM_EXPORT

Action Exports environment variables from the master task's environment to all the worker tasks' environments.

Usage While in the master task environment, select the variables you want to export and then concatenate them under a single variable (PVM_EXPORT) using a colon (:) as a separator.

- Bourne shell:

```
export PVM_EXPORT=DISPLAY:INPUT_PATH
```

- C shell:

```
setenv PVM_EXPORT DISPLAY:INPUT_PATH
```

PVM_EXPORT should be set in the shell where the master task is started.

PVMe_IP

Action The **PVMe_IP** environment variable causes a daemon to be started one each node in your session. This daemon manages the IP communication path that supports **pvm_catchout** and **pvm_setopt**. When running over MPCI, you must set the **PVMe_IP** environment variable to *yes* before starting the IBM PVMe for AIX daemon if **pvm_catchout** is to be used.

Usage

- Bourne shell:

  ```
  export PVMe_IP=yes
  ```

- C shell:

  ```
  setenv PVMe_IP yes
  ```

  PVMe_IP should be set in the shell where the daemon is started.

PVMe_TRACEFILE

Action If the IBM PVMe for AIX daemon **-trace** option is specified, by default, all trace data will be output to */tmp/xpvm_trace_$USER* of the node on which the daemon is run. However, you can use the **PVMe_TRACEFILE** environment variable to specify a different output file from trace data.

Usage

- Bourne shell:

  ```
  export PVMe_TRACEFILE=/tmp/aprogram_trace
  ```

- C shell:

  ```
  setenv PVMe_TRACEFILE /tmp/aprogram_trace
  ```

  PVMe_TRACEFILE should be set in the shell where the daemon is started.

PVM_DEBUGGER

Action If the **bx=** daemon option is not set, and **PVM_DEBUGGER** is set before starting the daemon, **PVM_DEBUGGER** will be used as the full pathname of the debugger script. See "Customizing the debugger and debugger2 Shell Scripts" on page 19.

Usage

- Bourne shell:

  ```
  export PVM_DEBUGGER=/u/johndoe/pvm3/bin/debugger
  ```

- C shell:

  ```
  setenv PVM_DEBUGGER /u/johndoe/pvm3/bin/debugger
  ```

PVM_DEBUGGER should be set in the shell where the daemon is started.

PVMe_IP_PORT

Action PVMe uses IP Ports 15631-15634 when running in MPCI/IP mode. If these ports are not available then the **PVMe_IP_PORT** variable can cause PVMe to use ports starting at any point between 5000 and 65532.

Usage Before starting the task which spawns all other tasks, you should set and export this variable, in the shell of the master task, to all of the tasks as follows:

- Bourne shell:

```
export PVMe_IP_PORT=7500
export PVM_EXPORT=PVMe_IP_PORT
```

- C shell:

```
setenv PVMe_IP_PORT=7500
setenv PVM_EXPORT PVMe_IP_PORT
```

Appendix C. IBM PVMe for AIX Intrinsic Limits: Summary

There are a few limitations in the present release of the IBM PVMe for AIX program product. Those that are of interest to you are shown in the following table.

Table 8. IBM PVMe for AIX: Intrinsic Limits Summary	
Maximum length of a host name	64 characters
Maximum length of a group name	32 characters
Maximum number of groups.	8
Maximum number of instances for a group.	128
Maximum number of tasks per host	1
Maximum number of hosts	128
Maximum size of a single message	32 megabytes
Maximum number of concurrent sessions on the same set of nodes	1 (US) 4 (IP)
pw option for the daemon	Not supported
Maximum number of consoles per host	1
Maximum length of filename.	32 characters

There are also limitations for the number of outstanding messages (messages which are ready for a receive operation). The maximum number of outstanding messages for the three message classes supported by the IBM PVMe for AIX program (the message size defines the class of the message) are provided in the following table:

Table 9. IBM PVMe for AIX: Limits (Number and Size of Pending Messages)	
Message Length	**Number of messages**
0 - 256 bytes	2048
257 - 4096 bytes	1024
4097 bytes - 32 megabytes	As many as fit in 128 megabytes

If the limit is exceeded for a message class, the application prints the following message:

```
Too many outstanding messages: <code>
```

where <code> can be **-2**, **-102** or **-202**, for the small, medium, or large messages class, as listed in the table above.

When this message appears, you should stop the application and reset IBM PVMe for AIX. If the daemon is running in the background, the quickest solution is to terminate the session by entering the following command:

```
kill -term <daemon pid>
```

Never use `kill -9 <daemon pid>` or `kill -kill <daemon pid>`.

© Copyright IBM Corp. 1993, 1994, 1995

A final limitation is that the SP system supports only one process per node when the High Performance Switch is used. This means that the number of nodes reserved for an IBM PVMe for AIX session must be at least equal to the total number of the IBM PVMe for AIX processes that will be concurrently active during that session.

Some routines, included with freeware PVM, are not available with the current version of IBM PVMe for AIX. There are also some options that are not supported. These are listed in the following tables:

Table 10. IBM PVMe for AIX: Routines Not Included in IBM PVMe for AIX

pvm_recvf
pvm_reg_hoster
pvm_reg_rm
pvm_reg_tasker
pvm_setmwid

Table 11. IBM PVMe for AIX: Routines Included in IBM PVMe for AIX, but Disabled

pvm_addhosts
pvm_delhosts
pvm_gettmask
pvm_settmask

Table 12. IBM PVMe for AIX: Options Not Supported by IBM PVMe for AIX

Routine name	Option
pvm_notify	PvmHostAdd PvmHostDelete
pvm_getopt **pvm_setopt**	PvmFragSize PvmResvTids PvmOutputTid PvmOutputCode PvmTraceTid PvmTraceCode PvmSelfOutputTid PvmSelfOutputCode PvmSelfTraceTid PvmSelfTraceCode
pvm_spawn	PvmMppFront PvmHostCompl

Appendix D. IBM PVMe for AIX Quick Reference

Use this appendix to quickly find out in what order you need to enter the arguments of a C or FORTRAN IBM PVMe for AIX subroutine. For more detailed information about the IBM PVMe for AIX routines, see Chapter 7, “IBM PVMe for AIX Reference: C and FORTRAN Subroutines” on page 93.

The following table lists the C subroutines:

C Subroutine	Argument Order	Flag Values
pvm_addhosts **DISABLED**	char **hosts, int nhosts, int* rcs	
pvm_archcode	char *arch	
pvm_barrier	char *group, int count	
pvm_bcast	char *group, int msgtype	
pvme_bcast	char *group, int msgtype, int rootinst	
pvm_bufinfo	int bufid, int *len, int *msgtype, int* tid	
pvm_catchout	FILE *filed	
pvm_config	int *nhost, int *narch, struct pvmhostinfo **hosts	
pvm_delhosts **DISABLED**	char **hosts, int nhosts, int *rcs	
pvm_exit	()	
pvm_freebuf	int bufid	
pvm_gather	void *result, void *data, int count, int datatype, int msgtype, char *group, int target)	
pvm_getinst	char *group, int tid	
pvm_getopt	int opt	**opt**: **PvmRoute** **PvmDebugMask** **PvmAutoErr**
pvm_getrbuf	()	
pvm_getsbuf	()	
pvm_gettid	char *group, int inst	
pvm_gettmask **DISABLED**	int who, Pvmtmask mask	
pvm_getsize	char *group	
pvm_halt	()	
pvm_hostsync	int host, struct timeval *clock, struct timeval *delta	
pvm_initsend	int encoding	**encoding**: **PvmDataDefault** 0 **PvmDataRaw** 1 **PvmDataInPlace** 2
pvm_joingroup	char *group	

© Copyright IBM Corp. 1993, 1994, 1995

C Subroutine	Argument Order	Flag Values
pvm_kill	int tid	
pvm_lvgroup	char *group	
pvm_mcast	int *tids, int ntids, int msgtype	
pvm_mkbuf	int encoding	**encoding**: **PvmDataDefault** 0 **PvmDataRaw** 1 **PvmDataInPlace** 2
pvm_mstat	char *host	
pvm_mytid	()	
pvm_notify	int about, int msgtype, int ntids, int *tids	**about**: **PvmTaskExit** 1
pvm_nrecv	int tid, int msgtype	
pvm_packf	const char *fmt, ...	
pvm_pkbyte	char *frombyte, int nitem, int stride	
pvm_pkcplx	float *fromcplx, int nitem, int stride	
pvm_pkdcplx	double *fromdcplx, int nitem, int stride	
pvm_pkdouble	double *fromdouble, int nitem, int stride	
pvm_pkfloat	float *fromfloat, int nitem, int stride	
pvm_pkint	int *fromint, int nitem, int stride	
pvm_pklong	long *fromlong, int nitem, int stride	
pvm_pkshort	short *fromshort, int nitem, int stride	
pvm_pkstr	char *fromstring	
pvm_parent	()	
pvm_perror	char *msg	
pvm_precv	int tid, int msgtype, char *buf, int len, int datatype, int *atid, int *amsgtype, int *alen	
pvm_probe	int tid, int msgtype	
pvm_psend	int tid, int msgtype, char *buf, int len, int datatype	
pvm_pstat	int tid	
pvm_recv	int tid, int msgtype	
pvm_reduce	void *func(), void *data, int count, int datatype, int msgtype, char *group, int rootinst	
pvm_scatter	void *result, void *data, int count, int datatype, int msgtype, char *group, int rootinst	
pvm_send	int tid, int msgtype	
pvm_sendsig	int tid, int signum	
pvm_serror	int set	

C Subroutine	Argument Order	Flag Values
pvm_setopt	int opt, int val	**opt**: **PvmRoute** **PvmDebugMask** **PvmAutoErr**
pvm_setrbuf	int bufid	
pvm_setsbuf	int bufid	
pvm_settmask **DISABLED**	int who, Pvmtmask mask	
pvm_spawn	char *task, char **argv, int flag, char *where, int ntask, int *tids	**flag**: **PvmTaskDefault** 0 **PvmTaskHost** 1 **PvmTaskArch** 2 **PvmTaskDebug** 4
pvm_start_pvmd	int argc, char **argv, int block	
pvm_tasks	int where, int *ntasks, struct pvmtaskinfo **taskp	
pvm_tidtohost	int tid	
pvm_trecv	int tid, int msgtype, struct timeval *timeout	
pvm_unpackf	const char *fmt, ...	
pvm_upkbyte	char *tobyte, int nitem, int stride	
pvm_upkcplx	float *tocplx, int nitem, int stride	
pvm_upkdcplx	double *todcplx, int nitem, int stride	
pvm_upkdouble	double *todouble, int nitem, int stride	
pvm_upkfloat	float *tofloat, int nitem, int stride	
pvm_upkint	int *toint, int nitem, int stride	
pvm_upklong	long *tolong, int nitem, int stride	
pvm_upkshort	short *toshort, int nitem, int stride	
pvm_upkstr	char *tostring	

The following table lists the FORTRAN subroutines:

FORTRAN Subroutine	Argument Order	Flag Values
pvmfaddhost **DISABLED**	host, rc	
pvmfarchcode	arch, cod	
pvmfbarrier	group, count, rc	
pvmfbcast	group, msgtype, rc	
pvmefbcast	group, msgtype, rootinst, rc	
pvmfbufinfo	bufid, len, msgtype, tid, rc	
pvmfcatchout	onoff, rc, (filename)	
pvmfconfig	nhost, narch, tid, name, arch, speed, rc	

FORTRAN Subroutine	Argument Order	Flag Values
pvmfdelhost **DISABLED**	host, rc	
pvmfexit	info	
pvmffreebuf	bufid, rc	
pvmfgather	result, data, count, datatype, msgtype, group, target, rc	
pvmfgetinst	group, tid, inst	
pvmfgetopt	opt, val	**opt**: **PVMROUTE** **PVMDEBUGMASK** **PVMAUTOERR**
pvmfgetrbuf	bufid	
pvmfgetsbuf	bufid	
pvmfgettid	group, inst, tid	
pvmfgetsize	group, size	
pvmfhalt	rc	
pvmfhostsync	host, clksec, clkusec, deltasec, deltausec)	
pvmfinitsend	encoding, bufid	**encoding**: **PVMDEFAULT** 0 **PVMRAW** 1 **PVMINPLACE** 2
pvmfjoingroup	group, inst	
pvmfkill	tid, rc	
pvmflvgroup	group, rc	
pvmfmcast	ntids, tids, msgtype, rc	
pvmfmkbuf	encoding, bufid	
pvmfmstat	hcost, status	
pvmfmytid	tid	
pvmfnotify	about, msgtype, ntids, tids, rc	**about**: **PVMTASKEXIT** 1
pvmfnrecv	tid, msgtype, bufid	
pvmfpack	what, from, nitem, stride, rc	**what**: **STRING** 0 **BYTE1** 1 **INTEGER2** 2 **INTEGER4** 3 **REAL4** 4 **COMPLEX8** 5 **REAL8** 6 **COMPLEX16** 7

FORTRAN Subroutine	Argument Order	Flag Values
pvmfparent	tid	
pvmfperror	msg, rc	
pvmfprecv	tid, msgtype, buf, len, datatype, atid, amsgtype, alen, bufid	
pvmfprobe	tid, msgtype, bufid	
pvmfpsend	tid, msgtype, buf, len, datatype, rc	
pvmfpstat	tid, status	
pvmfrecv	tid, msgtype, bufid	
pvmfreduce	func, data, count, datatype, msgtype, group, rootinst, rc	
pvmfscatter	result, data, count, datatype, msgtype, group, rootinst, rc	
pvmfsend	tid, msgtype, rc	
pvmfsendsig	tid, signum, rc	
pvmfserror	set, oldset	
pvmfsetopt	opt, val, oldval	**opt**: **PVMROUTE** **PVMDEBUGMASK** **PVMAUTOERR**
pvmfsetrbuf	bufid, oldbuf	
pvmfsetsbuf	bufid, oldbuf	
pvmfspawn	task, flag, where, ntask, tids, nt	**flag**: **PVMDEFAULT** 0 **PVMHOST** 1 **PVMARCH** 2 **PVMDEBUG** 4
pvmfstartpvmd	block, rc	
pvmftasks	where, ntasks, tid, ptid, htid, flag, name, rc	
pvmftidtohost	tid, dtid	
pvmftrecv	tid, msgtype, sec, usec, bufid	
pvmfunpack	what, to, nitem, stride, rc	

Appendix E. IBM PVMe for AIX Console Commands

The following table lists the IBM PVMe for AIX console commands for quick reference. For more information about using the IBM PVMe for AIX console, refer to “Using the IBM PVMe for AIX Console” on page 29.

Command	Description
add	*** Currently Disabled ***
alias	Allows you to define aliases for console commands.
conf	Shows the actual virtual machine configuration; a line for each host that contains the *hostname*, the **pvmd** task ID, host architecture (**RS6K**), the maximum fragment size and a relative speed rating.
delete	*** Currently Disabled ***
echo	Prints the specified argument(s).
halt	Halts all IBM PVMe for AIX processes, including consoles, and then shuts IBM PVMe for AIX down.
help or ?	Displays a list of the available commands.
help [command]	Displays an explanation of the command and its syntax.
id	Prints the console task identifier.
jobs	lists the jobs that are running.
kill	Terminates a process. The syntax is **kill** *tid*.
mstat	Shows the status of specified hosts.
ps [-a]	Lists all processes that are currently on the virtual machine, their locations, their task IDs and their parent's task ID.
pstat	Shows the status of a single IBM PVMe for AIX process.
quit	Exits the console, leaving daemons and IBM PVMe for AIX jobs running.
reset	Kills all IBM PVMe for AIX processes, except consoles, and resets all the internal IBM PVMe for AIX tables. It forces a new epoch and leaves the daemon in an idle state.
setenv	Allows you to set an environment variable.
sig	Sends a signal to an IBM PVMe for AIX process.
spawn	Starts one or more IBM PVMe for AIX applications.
unalias	Undefine a command alias.
version	Prints the version of **libpvm** being used (freeware PVM compatible level).

© Copyright IBM Corp. 1993, 1994, 1995

Appendix F. IBM PVMe for AIX Daemon Options

The following table lists the IBM PVMe for AIX daemon options.

Daemon option	Description

-exec	The **-exec** option is primarily for use in batch mode. It allows you to specify the task to be started on the local host. This option can be used for both batch and interactive jobs and is required for batch jobs via LoadLeveler.
	
	The syntax (example) is:
	
	`pvmd3e <PVMe options> -exec "<absolute path of program> <optional program parameters>"`
	
	When you specify this option, the daemon, after initializing the environment, starts the specified task on the local host and terminates just after the first epoch.
	
	If the command is contained within a shell script, and the application needs to read from the standard input, use the following:
	
	`pvmd3e -exec "myprog" <<%%`
	`param1`
	`param2`
	`%%"`
	
	Note: The **-exec** option will not work from the control workstation. When the **-exec** option is used, it must be the last option given for the pvmd3e command.

-ip	The **-ip** option is used in interactive mode and it allows you to specify that the task to be started will use the IP protocol rather than the User Space protocol. The US protocol is only usable by one process at a time, which means that only one parallel job, running over US, can run on a given set of nodes at a time. Using the IP protocol allows multiple jobs to run at the same time. As a result, multiple parallel jobs can run on the same set of nodes at a given time, but each job must request shared use of the CPU. The syntax (example) is: `pvmd3e -ip -share cpu < numnodes > &` Note that the protocol to be used is selected at compile time and can be modified with the environment variable **LIBPATH**. See "Compiling and Linking a Parallel Program on a POWERparallel System" on page 87. The **-ip** option does not select the protocol to be used.
-nokill	The **-nokill** option is for interactive mode and is used to inform the IBM PVMe for AIX daemon to not kill the processes that continue after you have called the **pvm_exit** routine. When all the tasks leave the IBM PVMe for AIX environment cleanly, the IBM PVMe for AIX daemon resets the content of the task table, event table, groups table, etc.. In IBM PVMe for AIX, this is known as starting a *new epoch*. After a new epoch is started, the IBM PVMe for AIX daemon has no memory of the previous run. The default action by the IBM PVMe for AIX daemon is to kill all worker tasks that are still running at the end of each epoch. There may be situations where your application would want to continue running for a while after it has called **pvm_exit**. To enable the worker tasks to run after all of them have called **pvm_exit**, you need to start the IBM PVMe daemon with the **-nokill** option or set the **KILL_ON_EPOCH** environment variable to "FALSE" in your or the system's pvm6kcnf file. The syntax (example) is: `pvmd3e -nokill < numnodes >` This ensures that when you leave PVMe, the process that wants to continue on will not be killed.

© Copyright IBM Corp. 1993, 1994, 1995

Daemon option	Description
-p	The **-p** option is for use in interactive mode and it is used to select the *pool_id* where the Resource Manager allocates nodes for the parallel job execution. If the **-p** option is not specified, nodes are allocated from the *pool_id* **1**. If another *pool_id* must be used, supply the daemon with the **-p** option, followed by the integer identifying the pool. The syntax (example) is: `pvme 5 -p 2` **Note:** You can also specify the default *pool_id* with the DEFPOOLID stanza in the **pvm6kcnf** file.
-resd	The **-resd** option can be used to activate and deactivate Resource Manager control over node usage and allocation. The default value is *yes* (Resource Manager control enabled), but you can override this by specifying *-resd no* at daemon startup. The syntax (example) is: `pvmd3e mylist -resd no &` **Note:** Do **not** use this option if you are running IBM PVMe for AIX on an SP machine; use it only if you are running IBM PVMe for AIX on workstations over IP. If -resd no is used on an SP, PVMe will not make use of the switch.
-r	The **-r** option is for batch mode and is used to reset all the IBM PVMe for AIX sessions running on the nodes listed in the **hostlist** file. The syntax (example) is: `pvmd3e hostlist -r` where *hostlist* is the file which contains the list of nodes used by the IBM PVMe for AIX program product. The structure of this file is described in “Running an IBM PVMe for AIX Session” on page 21. If nodes were allocated by the Resource Manager, you must use the **pvmnodelist** file, created by the daemon when the previous session was started (see “Resolving Startup Problems” on page 36). **Note:** The daemon must be started in the foreground when this option is used.
-R	The **-R** option is for use in batch mode. It causes all the user's IBM PVMe for AIX sessions running on the nodes, listed in the hostlist file (if any) to be reset, and a new one to start. The result would be the same if you ran the daemon with the **-r** option and then started the IBM PVMe for AIX session (specifying the <*number of nodes*> or the *hostlist* file). The syntax (example) is: `pvmd3e -R < numnodes >`
-share	This option is used in interactive mode. By default, IBM PVMe for AIX reserves an SP node for a user as a dedicated resource. This is done via the Resource Manager. As a result, other processes that request access to that node, via the Resource Manager, are denied. The **-share** option lets you override this default. The syntax (example) is: `pvmd3e -share <argument>` Where: <**argument**> Is **cpu**. This is the only argument that the **-share** option currently supports. It allows an IBM PVMe for AIX user to request an SP node such that any other process, that is *not* running User Space over the SP High-Performance Switch, can also use this node. Other processes, including another IBM PVMe for AIX, that want to run on the same node must request shared access from the Resource Manager.

Daemon option	Description
-trace	This option is for use in interactive batch mode and it lets you enable tracing globally in an IBM PVMe for AIX session, without inserting statements in the application source code. The **-trace** option causes the IBM PVMe for AIX daemon to set the trace mask that will be assigned to the user tasks and to all their children. Tracing is enabled on all the events. This means that each user task will produce a trace record for each event. The syntax (example) is: `pvmd3e -trace < numnodes >`

Appendix G. IBM PVMe for AIX Hostlist File Options

The **hostlist** file contains a line for each host that participates in the virtual machine, including the *hostname* and possibly one or more of the **hostlist** options. The following table lists the options available. For more information on these options, refer to “Syntax of the Hostlist File” on page 35.

Hostlist File option	Description
ep=	<absolute path of the directory for the executables>
dx=	<full pathname for the IBM PVMe for AIX daemon executable>
lo=	<user login name on the remote machine>
bx=	<pathname for the debug script>
wd=	<working directory where spawned tasks will be executed>

The meaning of the **hostlist** options is self-explanatory. The default values are as follows:

- If the **ep=** option is not set, the **$(HOME)/pvm3/bin/RS6K** directory is searched for your executables.
- If the **dx=** option is not set, the full pathname for the daemon executable is assumed to be /**usr**/**lpp**/**pvme**/**bin**/**pvmd3e**.
- If the **lo=** option is not set, the login name is assumed to be the same as the user starting the daemon.
- If the **bx=** option is not set, the debug scripts, **debugger** and **debugger2**, are expected to be in the same directory as your executables (**ep**, **$PVMEPATH**, or **$HOME/pvm3/bin/RS6K**).
- If the **wd=** option is not set, the working directory is **ep** (**ep**'s default is **$HOME/pvm3/bin/RS6K**).

© Copyright IBM Corp. 1993, 1994, 1995

Appendix H. Enabling AFS

You if want to use AFS, do the following:

1. Make sure **.rhosts** has the correct authorizations (same as for non-AFS use of PVMe).
2. Assuming that $HOME is afs mounted, issue the following command before attempting to run under AFS.

   ```
   fs sa $HOME system:anyuser rl
   ```

   Note: If the pvm6kcnf file provided by the system administrator doesn't contain the line `AFS_ENV=1`, then the daemon option -afs must be used or a private copy of pvm6kcnf must be created which contains this line.
3. Finally, the system administrator must modify the PVMe installation according to the directions in `/usr/lpp/pvme/afs/README.afs`

© Copyright IBM Corp. 1993, 1994, 1995

Glossary of Terms and Abbreviations

This section contains some of the terms that are commonly used in this book.

IBM is grateful to the American National Standards Institute (ANSI) for permission to reprint its definitions from the American National Standard *Vocabulary for Information Processing* (Copyright 1970 by American National Standards Institute, Incorporated), which was prepared by Subcommittee X3K5 on Terminology and Glossary of the American National Standards Committee X3. ANSI definitions are preceded by an asterisk (*).

Other definitions in this glossary are taken from *IBM Vocabulary for Data Processing, Telecommunications, and Office Systems* (GC20-1699) and *IBM DATABASE 2 Application Programming Guide for TSO Users* (SC26-4081).

© Copyright IBM Corp. 1993, 1994, 1995

A

AIX. Abbreviation for Advanced Interactive Executive, IBM's licensed version of the UNIX operating system. AIX is particularly suited to support technical computing applications, including high function graphics and floating point computations.

application program. A program written by or for a user that applies to the user's work.

B

bandwidth. Bandwidth is defined as the total available bit rate of a digital channel. The bandwidth of a system is determined by the speed of its network, but it is also affected by the overhead of the control data added by the communication protocol.

barrier routine. Imposes an explicit synchronization among tasks.

bit rate. The speed at which bits are transmitted, usually expressed in bits per second.

blocking routine. A routine that blocks until the proper message arrives. It returns immediately if the proper message has already arrived.

broadcast operation. This operation allows you to send a message to a set of processes, usually to all members of a process group, without specifying the task identifier of each recipient task.

buffer. An area of storage that is temporarily reserved for use in performing an input/output operation, into which data is read or from which data is written.

C

cluster. A group of nodes interconnected through a high-speed network that can be used for high performance computing. It provides excellent price/performance.

communication overhead. Communication overhead refers to the time spent in communicating among nodes, compared to the serial computing time of the parallelizable part of the application. Communication overhead is an important factor in evaluating the performance of a parallel program.

communication protocol. A formal description of the message format and of the rules two or more machines must follow to exchange those messages.

computation-to-communication ratio. The computation-to-communication ratio refers to the number of operations that a node can perform in local memory per second, divided by the total number of words that the communication system can deliver per second.

computational independence. Condition which guarantees the possibility for a valid parallelization. More tasks are said to be computationally independent of each other if data referenced by one of them is not referenced by another one and vice versa.

control data. Control information or control record.

D

daemon. A process, not associated with a particular user, that performs system-wide functions such as administration and control of networks, execution of time-dependent activities, line printer spooling, and so on.

data communication. Data exchange between processes. It can occur between processes running on the same host as well as between processes running on different hosts.

data parallelism. This is where all the programs perform the same computation on different data sets.

data type. (1) The structural characteristics, features and properties of data that may be directly specified by a programming language, (2) The mathematical properties and internal representation of data and functions.

decomposition. This is the starting point in arranging the program for parallel execution.

default. An alternative value, attribute, or option that is assumed when none has been specified.

distributed memory systems. Computing systems composed by multiple nodes each accessing only its own private memory. SP systems, as well as a cluster of workstations, belong to this class.

dynamic load balancing. Allows you decide at run time which node should run the specified parallel task.

E

enroll. In IBM PVMe for AIX terminology, it is the registration of a process to the IBM PVMe for AIX daemon.

F

fork-exec mechanism. Standard methods provided by UNIX to start a new process.

functional parallelism. This is where processes are almost independent of each other and specialized to particular tasks.

G

granularity. This refers to the amount of computational work that is allocated to parallel tasks. It relates to the length of the parallel sections, in terms of the number of instructions that a parallel task executes.

group management. Managing processes that are organized into task groups. It allows you to structure complex applications in an organized fashion.

H

hostlist file. A file that contains the list of nodes used by the IBM PVMe for AIX program.

hot spot. These are the parts of a program that take the most time to run.

I

instance number. Process identifier. In IBM PVMe for AIX it usually refers to the identifier of a task within a group and is assigned by the daemon when the task joins a group.

L

latency. Latency is defined as 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 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.

load balancing. This refers to assigning the proper amount of work to all processes cooperating to solve a given problem, in order to reduce the total runtime and avoid some processes being idle while waiting for the others to complete.

local host. The computer to which a user's terminal is directly connected.

low-level communication protocol. This terminology is used to refer to the software that provides the software communication support to the High Performance Switch.

M

master host. This is the host where the daemon is running.

menu. A display of a list of available functions for selection by the user.

message passing. Parallel programming paradigm, mainly used in distributed memory systems. It is based on the explicit exchange of messages to allow more processes to access the same variable.

MPMD. Multiple program, multiple data. Parallel programming model where different processes perform different computational work on different data, while cooperating to solve the same problem.

N

nonblocking routines. Routines that return when the message has been delivered to the reader of the recipient process, whether they run a receive or not.

P

parallel overhead. Parallelizing a program creates a parallel overhead which can reduce performance. Factors such as scheduling, initializing, and synchronizing parallel tasks can cause a parallel overhead.

parallel processing. A multiprocessor architecture which allows processes to be allocated to tightly coupled multiple nodes in a cooperative processing environment, allowing concurrent execution of tasks.

parallelize. Tailoring a program for parallel execution.

parameter. (1) A variable that is given a constant value for a specified application and that may denote the application. (2) 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. (3) A name in a procedure that is used to refer to an argument that is passed to the procedure. (4) A particular piece of information that a system or application program needs to process a request.

PID. UNIX process identifier.

primitive. Synonym for service primitive.

process control. Automatic control of a process, in which a computer system is used to regulate continuous operations or processes.

PVM. Parallel virtual machine.

R

reader. The reader is a server process. Each IBM PVMe for AIX task has a reader that simply waits for messages, receives them, and makes them available to the parent process.

S

server. (1) A function that provides services for users. A machine may run client and server processes at the same time. (2) A machine that provides resources to the network. It provides a network service, such as disk storage and file transfer, or a program that uses such a service.

shared memory segments. Memory regions that can be accessed by different processes.

shared memory systems. Computing systems composed of multiple nodes, all accessing a single shared memory (and, depending on the system, also a local private memory). IBM ES/9000 multiprocessors are examples from this class.

shell. The shell is the primary user interface for the UNIX operating system. It serves as command language interpreter, programming language, and allows foreground and background processing. There are three different implementations of the shell concept: Bourne, C, and Korn.

SMIT. System management interface toolkit. This is a set of menu driven utilities for AIX Version 3 that provide functions such as transaction login, shell script creation, automatic updates of object database, and so on.

speed-up. Ratio between the runtime of a serial program and the runtime of the parallel version of the same program. If the speed-up is closely related to the number of nodes involved in the parallel execution, then the value of the parallelization is increased.

SPMD. Single program, multiple data. Parallel programming model where different processes perform the same computational work on different subsets of data to solve the same problem.

static load balancing. Involves assigning parallel tasks to the available nodes in a predetermined and ordered way.

synchronization. This refers to the exact order in which the nodes perform their various tasks. This is important in serializing tasks that are not computationally independent.

T

task. A *task* is a unit of computation, analogous to a UNIX process.

task identifier. In IBM PVMe for AIX terminology, a task identifier is a positive integer assigned by the daemon, and is unique for each process. All IBM PVMe for AIX processes have an identifier (called *tid*), in order to be properly addressed by the daemon and by the other IBM PVMe for AIX processes.

TCP. Transmission Control Protocol, a stream communication protocol that includes error recovery and flow control.

TCP/IP. Transmission Control Protocol/Internet Protocol, a suite of protocols designed to allow communication between networks regardless of the technologies implemented in each network.

W

workstation. (1) A configuration of input/output equipment at which an operator works. (2) A terminal or microcomputer, usually one that is connected to a mainframe or to a network, at which a user can perform applications.

Index

Special Characters

-exec IBM PVMe for AIX daemon option 32, 243
  -exec 243
-ip IBM PVMe for AIX daemon option 32, 243
-ip option, to access the TB2 adapter via the IP 243
-nokill IBM PVMe for AIX daemon option 33, 243
-nokill option 243
-p IBM PVMe for AIX daemon option 33, 244
-p option, for selecting the pool_id 33, 243
-r IBM PVMe for AIX daemon option 34, 244
-R option IBM PVMe for AIX daemon 34
-r option, for resetting all the PVM sessions 244
-resd IBM PVMe for AIX daemon option 33, 244
-share IBM PVMe for AIX daemon option 34, 244
-share option, for sharing nodes 244
-trace IBM PVMe for AIX daemon option 35, 245

A

accessing storage devices for read/write operations 48
AIX, definition 252
alias console command 29, 241
Amdahl's law 11
application program, definition 252
automatic printing of error messages 63

B

bandwidth
  definition 252
  how it affects performance 6
barrier routine 71
barrier routine, definition 252
batch mode
  running an IBM PVMe for AIX session 29
bit rate, definition 252
blocking routine, definition 252
broadcast operation
  comparing 4
  definition 252
  understanding 70
buffer, definition 252
buffers, multiple 59
bx hostlist file option 35, 247

C

C example 74
C examples i
  data communication 62
  managing error conditions 64
  managing groups 74
C examples *(continued)*
  managing hosts 52
  managing tasks 44
  signaling 66
checking the status of a target process 4
cluster, definition 252
commands
  dbx 90
  rcmd 47
  used for interactive starting and killing of PVM processes 29
communication
  for sharing data 5
  overhead 10—13
  tuning 15
communication overhead
  definition 252
  performance 10
communication protocol, definition 252
communication subsystem sharing 25
comparing a broadcast operation 4
comparing a send operation 4
compiling and linking a parallel program 87, 88
components of cooperation 5
computation-to-communication ratio 14
computation-to-communication ratio, definition 252
computational independence i, 252
  determining 7
compute node, starting 27
concepts, parallel programming 4
conf console command 29, 241
configuration files 19
console commands 29, 241
  alias 29, 241
  conf 29, 241
  delete 29
  echo 29, 241
  halt 29, 241
  help 29, 241
  id 29, 241
  jobs 29, 241
  kill 29, 241
  mstat 29, 241
  ps 29, 241
  pstat 29, 241
  quit 29, 241
  reset 29, 241
  setenv 29, 241
  sig 29, 241
  spawn 29, 241
  unalias 29, 241
  version 29, 241

© Copyright IBM Corp. 1993, 1994, 1995

console commands, IBM PVMe for AIX 241
control data, definition 252
cooperation
components of 5
concepts
communication, for sharing data 5
message passing 5
synchronization 5
for task execution 5
coupling scheme of the node 5
CPU sharing 25
customizing AFS 249

D

daemon options, IBM PVMe for AIX 243
daemon, definition 252
daemon, location of 35, 225, 247
data communication
definition 252
examples 60
preparing a message 56
primitives used for 54
probe for message arrival 59
receiving a message 57
sending a message 57
understanding 54
data communication routines 54
data parallelism 41
definition 252
data type, definition 252
dbx command 90
dbx, running IBM PVMe for AIX program under 196
dbx, running IBM PVMe for AIX programs under 19
debugging a parallel program 89
finding errors in program logic 90
decomposing a serial program 8—9
decomposition, definition 252
default, definition 252
definitions
AIX 252
application program 252
bandwidth 252
barrier routine 252
bit rate 252
blocking routine 252
broadcast operation 252
buffer 252
cluster 252
communication overhead 252
communication protocol 252
computation-to-communication ratio 252
computational independence ratio 252
control data 252
daemon 252
data communication 252
definitions *(continued)*
data parallelism 252
data type 252
decomposition 252
default 252
distributed memory systems 252
dynamic load balancing 252
enroll 252
fork-exec mechanism 253
functional parallelism 253
granularity 253
group management 253
hostlist file 253
hot spot 253
instance number 253
latency 253
load balancing 253
local host 253
low-level communication protocol 253
master host 253
menu 253
message passing 253
MPMD 253
nonblocking routines 253
parallel overhead 253
parallel processing 253
parallelize 253
parameter 253
PID 253
primitive 253
process control 254
PVM 254
server 254
shared memory segments 254
shell 254
SMIT 254
speed-up 254
SPMD 254
static load balancing 254
synchronization 254
task 254
task identifier 254
TCP 254
TCP/IP 254
workstation 254
delete console command 29
description of syntax diagrams 93
design considerations 10
determining computational independence 7
differences between PVM and IBM PVMe for AIX 2
differences between PVM and the IBM PVMe for AIX program product 3
differences between tightly and loosely coupled systems 5
distributed memory systems, definition 252

dx hostlist file option 35, 247
dynamic load balancing 14
dynamic load balancing, definition 252

E

echo console command 29, 241
enroll, definition 252
enrolling a process 45
environment variables 30, 36, 37, 89, 225
  N256BUFF 229
  N64KBUFF 228
  PVM_DEBUGGER 230
  PVM_EXPORT 229
  PVMDPATH 225
  PVMe_BARRTYPE 227
  PVMe_CLH 226
  PVMe_DATA_INTERRUPT 227
  PVMe_IP 229
  PVMe_IP_PORT 230
  PVMe_MEMCPY 227
  PVMe_TRACEFILE 230
  PVMEPATH 226
  PVMHFN 226
  SP_NAME 228
ep hostlist file option 35, 247
error code summary 223, 224
error handling 62
error handling, primitives 63
  error handling routines 63
error messages, automatic printing 63
errors, finding in program logic 90
examples
  C 74
  data communication 60
  FORTRAN 73
  FORTRAN, complete 75
  managing error conditions 63
  managing hosts 51
  managing tasks 43
  signaling 66
executable files 47
executing tasks using cooperation 5
explanation, of subroutines 93

F

features and routines 41
finding errors in program logic 90
fork-exec mechanism, definition 253
format, explanation of subroutines 93
FORTRAN
  complete example 75
  interface usage 93
FORTRAN examples
  data communication 60
FORTRAN examples *(continued)*
  managing error conditions 63
  managing groups 73
  managing hosts 51
  managing tasks 43
  signaling 66
functional parallelism 41
  definition 253

G

getting started with IBM PVMe for AIX 1
global operations, performing 71
gprof command, using 10
granularity 13
granularity, definition 253
group management routines 68
group management, definition 253
groups
  examples 73
  information on status 72
  joining and leaving 68
  managing 67
  primitives 68

H

halt console command 29, 241
help console command 29, 241
High Performance Switch 5, 17, 21, 54
HOME 30
host management routines 50
hostlist file options 35, 247
  bx 247
  dx 247
  ep 247
  lo 247
  wd 247
hostlist file, definition 253
hosts
  example, managing hosts 51
  information 50
  managing 49, 67
  primitives, using 50
  querying information 50, 72
hot spot, definition 253
hot-spot analysis 10

I

I/O for IBM PVMe for AIX processes 48
IBM PVMe for AIX
  bandwidth 6
  configuration 21
  console, using 29
  customizing your installation 19

IBM PVMe for AIX *(continued)*
daemon, accepting options 31
determining computational independence 7
differences between PVM and IBM PVMe for AIX 2, 3
error code summary 223, 224
High Performance Switch feature 5
I/O 48
installation overview 17
intrinsic limits summary 233
latency 6
managing tasks
examples 51
managing error conditions 63
using primitives 42
options 35, 245
overview of features and routines 41
overview, getting started 1
parallel programming concepts 4
product startup 21—40
resolving startup problems 36
starting
testing 37
using
cooperation for executing tasks 5
the program product 41
using the program product 85
writing parallel programs 7
IBM PVMe for AIX console commands 241
IBM PVMe for AIX console syntax 32
IBM PVMe for AIX daemon options 31, 243
-exec 32
-ip 32, 243
-nokill 33, 243
-p 33, 244
-r 34, 244
-resd 33, 244
-share 34, 244
-trace 35, 245
IBM PVMe for AIX daemon syntax 32
IBM PVMe for AIX hostlist file options 35, 247
id console command 29, 241
information on tasks 48
installation
configuration files 19
customizing your installation 19
IBM PVMe for AIX software 17
requirements 17
instance number, definition 253
interconnection network 5

J

jobs console command 29, 241
joining groups 68

K

kerberos 17
kill console command 29, 241

L

latency, definition 253
latency, how it affects performance 6
leaving groups 68
linking and compiling a parallel program 87, 88
lo hostlist file option 35, 247
load balancing 13
load balancing, definition 253
LoadLeveler i, 21, 30
local host, definition 253
loosely coupled system 5
low-level communication protocol, definition 253

M

Manager 19
managing
error conditions 63
example, hosts 51
group examples 67, 73
hosts 49
multiple buffers 59
options 48
primitives 68
tasks 42
managing options 48
master host, definition 253
memory, private 5
menu, definition 253
message
definition 253
passing 5
preparing 56
probe for arrival 59
receiving 57
sending 57
MPMD, definition 253
mstat console command 29, 241
multiple buffers 59
multiple program, multiple data (MPMD) 1, 6

N

N256BUFF 229
N64KBUFF 228
network, interconnection 5
node allocation
number of nodes 25
one node 25
specific nodes 25

nodes 5
nodes, synchronization of 5
nonblocking routines, definition 253

O

optimizing the use of system functions 15
options, hostlist file 35, 247
options, IBM PVMe for AIX daemon 31

P

packing routine, using 56
parallel overhead 10
parallel overhead, definition 253
parallel processing, definition 253
parallel programming
  communication overhead 10—13
  compiling and linking 87
  compiling and linking on an external workstation 88
  concepts 4
  debugging a parallel program 89
  decomposing a serial program 8—9
  design considerations 10
  granularity, fine and coarse 13
  load balancing 13
  paradigms 5
  performance considerations 10
  running a parallel program 89
  tuning 14
  understanding the differences between serial and parallel programs 4
  writing parallel programs 7
parallelization, tuning 15
parallelize, definition 253
parameter, definition 253
performance considerations 10
PID, definition 253
preparing a message 56
primitive, definition 253
primitives
  data communication 54
  for error handling 63
  for joining and leaving groups 68
  for managing groups 68
  for managing tasks 42
  for signaling 65
  for the barrier routine 71
  for the broadcast operation 70
printing, error messages 63
private memory 5
probe for message arrival 59
process
  enrolling 45
  I/O 48
  starting 46, 47
process *(continued)*
  terminating 47
  unenrolling 46
process control, definition 254
prof command, using 10
profiling a serial program 10
programming models
  MPMD 1, 6
  SPMD 1, 6
ps console command 29, 241
pstat console command 29, 241
pvm_addhosts routine 95
pvm_archcode routine 97
pvm_barrier routine 99
pvm_bcast routine 101
pvm_bufinfo routine 105
pvm_catchout routine 107
pvm_config routine 109
PVM_DEBUGGER 230
pvm_delhosts routine 112
pvm_exit routine 114
PVM_EXPORT 30, 71, 229
pvm_freebuf routine 115
pvm_gather routine 117
pvm_getinst routine 120
pvm_getopt routine 122
pvm_getrbuf routine 124
pvm_getsbuf routine 126
pvm_gettid routine 129
pvm_gettmask routine 131
pvm_gsize routine 128
pvm_halt routine 133
pvm_hostsync routine 134
pvm_initsend routine 136
pvm_joingroup routine 138
pvm_kill routine 140
pvm_lvgroup routine 142
pvm_mcast routine 144
pvm_mkbuf routine 146
pvm_mstat routine 148
pvm_mytid routine 150
pvm_notify routine 152
pvm_nrecv routine 154
pvm_packf routine 156
pvm_parent routine 158
pvm_perror routine 160
pvm_pk routines 162
pvm_precv routine 167
pvm_probe routine 165
pvm_psend routine 170
pvm_pstat routine 172
pvm_recv routine 174
pvm_reduce routine 176
pvm_scatter routine 179
pvm_send routine 182

pvm_sendsig routine 184
pvm_serror routine 186
pvm_setopt routine 188
pvm_setrbuf routine 190
pvm_setsbuf routine 192
pvm_settmask routine 194
pvm_spawn routine 196
pvm_start_pvmd routine 200
pvm_tasks routine 202
pvm_tidtohost routine 205
pvm_trecv routine 206
pvm_unpackf routine 208
pvm_upk routines 210
PVM, definition 254
pvmd3e, daemon syntax 32
PVMDPATH 225
PVMe_BARRTYPE 227
pvme_bcast routine 103
PVMe_CLH 36, 226
PVMe_DATA_INTERRUPT 227
PVMe_IP 229
PVMe_IP_PORT 230
PVMe_MEMCPY 227
PVMe_TRACEFILE 230
pvme, console syntax 32
PVMEPATH 89, 226
PVMHFN 30, 37, 226

Q

quit console command 29, 241

R

rcmd command 47
reading syntax diagrams 93
receiving a message 57
reliable group operations, ensuring 69
requirements, installation 17
reset console command 29, 241
resetting an IBM PVMe for AIX session, using the -r options 31
resolving startup problems 36
Resource Manager 17, 30
running
  a parallel program 89
  an IBM PVMe for AIX batch session 29
running multiple sessions 26

S

searching for executable files 47
sending a message 57
serial program 4
  decomposing 8—9
  profiling 10
server, definition 254
session
  batch 29—31
  interactive 21—27
setenv console command 29, 241
shared memory segments, definition 254
shell, definition 254
sig console command 29, 241
signaling
  examples 66
  primitives 65
  routines 65
single program, multiple data (SPMD) 1, 6
SMIT 17
software, required for installation 17
SP_NAME 228
spawn console command 29, 241
specifying allocation options
  communication subsystem sharing 25
  CPU sharing 25
speed-up i, 8, 11, 12, 13
speed-up, definition 254
SPMD, definition 254
starting
  a process 46, 47
  and killing processes, using commands 29
  the virtual machine 50
starting an IBM PVMe for AIX session, using the -exec options 31
starting an IBM PVMe for AIX session, using the -ip options 31
starting an IBM PVMe for AIX session, using the -nokill option 31
starting an IBM PVMe for AIX session, using the -p options 31
starting an IBM PVMe for AIX session, using the -R option 31
starting an IBM PVMe for AIX session, using the -share options 31
starting an IBM PVMe for AIX session, using the -trace options 31
startup
  IBM PVMe for AIX batch session 29
  information 21
  resolving problems 36
  running an IBM PVMe for AIX batch session 29
  understanding IBM PVMe for AIX options 35, 245
  using the console 29
  Verifying the IBM PVMe for AIX program product 37
static load balancing 14
static load balancing, definition 254
status, of a target process 4
stopping the virtual machine 50
subroutines, C and FORTRAN
  pvm_addhosts 95

subroutines, C and FORTRAN *(continued)*
pvm_archcode 97
pvm_barrier 99
pvm_bcast 101
pvm_bufinfo 105
pvm_catchout 107
pvm_config 109
pvm_delhosts 112
pvm_exit 114
pvm_freebuf 115
pvm_gather 117
pvm_getinst 120
pvm_getopt 122
pvm_getrbuf 124
pvm_getsbuf 126
pvm_gettid 129
pvm_gettmask 131
pvm_gsize 128
pvm_halt 133
pvm_hostsync 134
pvm_initsend 136
pvm_joingroup 138
pvm_kill 140
pvm_lvgroup 142
pvm_mcast 144
pvm_mkbuf 146
pvm_mstat 148
pvm_mytid 150
pvm_notify 152
pvm_nrecv 154
pvm_packf 156
pvm_parent 158
pvm_perror 160
pvm_pk 162
pvm_precv 167
pvm_probe 165
pvm_psend 170
pvm_pstat 172
pvm_recv 174
pvm_reduce 176
pvm_scatter 179
pvm_send 182
pvm_sendsig 184
pvm_serror 186
pvm_setopt 188
pvm_setrbuf 190
pvm_setsbuf 192
pvm_settmask 194
pvm_spawn 196
pvm_start_pvmd 200
pvm_tasks 202
pvm_tidtohost 205
pvm_trecv 206
pvm_unpackf 208
pvm_upk 210
pvme_bcast 103
subroutines, C and FORTRAN *(continued)*
understanding format 93
summary, error code 223, 224
summary, intrinsic limits 233
Switch, High Performance i, 5, 21, 54
synchronization, definition 254
syntax diagrams, description of 93
syntax, IBM PVMe for AIX console 32
syntax, IBM PVMe for AIX daemon 32
system functions, optimizing the use of 15

T

target process, checking status 4
task identifier, definition 254
task management routines 42
task, definition 254
tasks
information 48
managing 42
using primitives to manage tasks 42
TCP, definition 254
TCP/IP, definition 254
terminating a process 47
testing the IBM PVMe for AIX program
using the hostbenc and nodebenc programs 37
using the include and makefile programs 37
tightly coupled system 5
tracing, enabling 230
tuning
a parallel application 14
communication 15
parallelization 15

U

unalias console command 29, 241
understanding IBM PVMe for AIX options 35, 245
unenrolling a process 46
unpacking data 58
using
a packing routine 56
the -exec option 32
the -ip option 32, 243
the -nokill option 33, 243
the -p option 33, 243
the -r option 34, 244
the -resd option 33
the -share option 34, 244
the -trace option 35, 244
the IBM PVMe for AIX console 29
the IBM PVMe for AIX program product 41

V

variables, environment 225
verify procedure 37
version console command 29, 241
virtual machine
  querying information 50
  starting 50
  stopping 50

W

wd hostlist file option 35, 247
what is parallel processing 4
workstation, definition 254
writing parallel programs
  communication overhead 10—13
  decomposing a serial program 8—9
  design and performance considerations 10
  determining computational independence 7
  granularity, fine and coarse 13
  load balancing 13
  optimizing the use of system functions 15
  overview 7
  parallelization 15
  profiling a serial program 10