IBM AIX PVMe User's Guide and Subroutine Reference Release 3.0

SH23-0019-02

IBM

IBM AIX PVMe User's Guide and Subroutine Reference Release 3.0

SH23-0019-02

Note

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

Third Edition, July 1994

This edition applies to Version 1, Release 3 of the IBM AIX PVMe 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 52QA, Mail Station 911
1 Enterprise Drive
Kingston, NY 12401-1099
United States of America

FAX: (International Access Code)+1+914+385-0662

IBMLink (United States customers only): KGNRCF at KGNVMC
IBM Mail Exchange: USIB27BQ at IBMMAIL
Internet: kgnrcf@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. All rights reserved. Note to U.S. Government Users &mdas Documentation related to restricted rights — Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp..

Contents

Notices . . . vii
Trademarks and Service Marks . . . vii

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

Chapter 1. Getting Started with IBM AIX PVMe . . . 1
What is IBM AIX PVMe? . . . 1
Differences Between PVM and IBM AIX PVMe . . . 2
Parallel Programming Concepts . . . 2
What is Parallel Processing? . . . 2
Cooperation . . . 3
Synchronization . . . 3
Parallel Programming Models . . . 3
Understanding Latency and Bandwidth . . . 4

Chapter 2. Writing Parallel Programs . . . 5
Parallel Programming Overview . . . 5
Determining Computational Independence . . . 5
Decomposing a Serial Program . . . 6
Design and Performance Considerations . . . 8
Granularity and Load Balancing . . . 11
Tuning a Parallel Application . . . 12

Chapter 3. Product Installation . . . 15
Requirements . . . 15
Before You Begin . . . 15
Installing the IBM AIX PVMe Software . . . 15
Creating Directory Structure . . . 16
Customizing Your Installation . . . 16
Customizing the debugger and debugger2 Shell Scripts . . . 17

Chapter 4. Product Startup . . . 19
Running an IBM AIX PVMe Interactive Session . . . 19
Understanding IBM AIX PVMe Options . . . 21
Running an IBM AIX PVMe Batch Session . . . 22
Using the IBM AIX PVMe Console . . . 26
Options for the IBM AIX PVMe Daemon . . . 27
Using the -q Option . . . 27
Using the -c Option . . . 28
Using the -r Option . . . 28
Using the -p Option . . . 29
Resolving Startup Problems . . . 29

© Copyright IBM Corp. 1993, 1994

Testing the IBM AIX PVMe Program Product 30

Chapter 5. Using the IBM AIX PVMe Program Product 31
Understanding IBM AIX PVMe Features 31
Managing Tasks 32
Primitives for Managing Tasks 32
Examples for Managing Tasks 33
Enrolling a Process 35
Unenrolling a Process 36
Starting a Process 36
Searching for Executable Files 37
Terminating a Process 37
Querying Information about Tasks 37
Accessing Storage Devices for Read/Write Operations 38
Managing Hosts 38
Primitives for Managing Hosts 39
Starting and Stopping the Virtual Machine 39
Adding/Deleting Hosts from the Virtual Machine 39
Querying Information about the Virtual Machine 40
Examples for Managing Hosts 41
Understanding Data Communication 44
Primitives for Data Communication 44
Preparing a Message 45
Sending a Message 46
Receiving a Message 47
Checking the Arrival of a Message 48
Managing Multiple Buffers 48
Examples for Data Communication 49
Error Handling 52
Primitives for Error Handling 52
Automatic Printing of Error Messages 53
Examples for Managing Error Conditions 53
Signaling 55
Primitives for Signaling 55
Examples of Signaling 55
Group Management 57
Managing Task Groups with Primitives 57
Joining and Leaving Groups 58
Understanding the Broadcast Operation 58
Understanding the Barrier Routine 58
Querying Information about Groups 58
Examples for Managing Groups 59
Understanding the Differences between the IBM AIX PVMe Program Product and PVM 61
Comparing a snd Operation 61
Comparing a Broadcast Operation 61
Checking the Status of a Target Process 62
Starting New Tasks 62
Compiling and Linking a Parallel Program 62
Running a Parallel Program 63
Debugging a Parallel Program 64
Finding Errors in Program Logic 64
A Complete Example (FORTRAN) 65

Chapter 6. IBM AIX PVMe Reference: C and FORTRAN Subroutines . . . 77
Using the FORTRAN Interface . . . 77
Understanding Subroutine Format . . . 77
Syntax Diagrams . . . 77
pvm_addhosts . . . 79
pvm_advise . . . 81
pvm_barrier . . . 83
pvm_bcast . . . 85
pvm_bufinfo . . . 87
pvm_config . . . 89
pvm_delhosts . . . 92
pvm_exit . . . 94
pvm_freebuf . . . 95
pvm_getinst . . . 97
pvm_getopt . . . 99
pvm_getrbuf . . . 101
pvm_getsbuf . . . 103
pvm_getsize . . . 105
pvm_gettid . . . 106
pvm_halt . . . 108
pvm_initsend . . . 109
pvm_joingroup . . . 111
pvm_kill . . . 113
pvm_lvgroup . . . 115
pvm_mcast . . . 117
pvm_mkbuf . . . 119
pvm_mstat . . . 121
pvm_mytid . . . 123
pvm_notify . . . 125
pvm_nrecv . . . 127
pvm_packf . . . 129
pvm_pkint, pvm_pkcplx, pvm_pkdcplx, pvm_pkfloat, ... 131
pvm_parent . . . 134
pvm_perror . . . 136
pvm_probe . . . 138
pvm_pstat . . . 140
pvm_recv . . . 142
pvm_send . . . 144
pvm_sendsig . . . 146
pvm_serror . . . 148
pvm_setopt . . . 150
pvm_setrbuf . . . 152
pvm_setsbuf . . . 154
pvm_spawn . . . 156
pvm_start_pvmd . . . 159
pvm_tasks . . . 161
pvm_tidtohost . . . 164
pvm_unpackf . . . 165
pvm_upkint, pvm_upkcplx, pvm_upkdcplx, pvm_upkfloat, ... 167

Appendix A. IBM AIX PVMe Error Code Summary . . . 171

Appendix B. IBM AIX PVMe Intrinsic Limits: Summary . . . 173

Appendix C. IBM AIX PVMe Quick Reference 175

Glossary of Terms and Abbreviations 179

Index 183

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, denoted by an asterisk (*) in this publication, 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	9076 SP2
IBM AIX PVMe	SP2

The following terms, denoted by a double asterisk (**), in this publication, are trademarks of other companies as follows:

UNIX	Novell, Inc.

*** PVM (Parallel Virtual Machine) is a public domain 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 AIX PVMe program product. Since the IBM AIX PVMe program product is only intended for applications that will run an IBM 9076 Scalable POWERparallel Systems* (SP2*) system with the High-Performance switch feature, readers should be familiar with the 9076 SP2. 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 x.

How to Use This Book

Overview of Contents

This book is divided into the following chapters:

- Chapter 1, “Getting Started with IBM AIX PVMe” on page 1, provides an overview of parallel programming and describes the IBM AIX PVMe program product.
- Chapter 2, “Writing Parallel Programs” on page 5, outlines a general approach to parallel programming.
- Chapter 3, “Product Installation” on page 15, describes the procedure for installing the IBM AIX PVMe software.
- Chapter 4, “Product Startup” on page 19, contains information on configuring the IBM AIX PVMe program at the system and user level.
- Chapter 5, “Using the IBM AIX PVMe Program Product” on page 31, provides an overview of the IBM AIX PVMe routines that are callable from your applications.
- Chapter 6, “IBM AIX PVMe Reference: C and FORTRAN Subroutines” on page 77, provides information about each IBM AIX PVMe routine, including a description, syntax rules, parameters, examples, and error conditions. The routines are arranged alphabetically throughout the chapter.
- Appendix A, “IBM AIX PVMe Error Code Summary” on page 171, explains the IBM AIX PVMe error codes.
- Appendix B, “IBM AIX PVMe Intrinsic Limits: Summary” on page 173, discusses some limitations of the IBM AIX PVMe program product.

© Copyright IBM Corp. 1993, 1994

- Appendix C, “IBM AIX PVMe Quick Reference” on page 175, provides a table of IBM AIX PVMe routines for easy reference.
- “Glossary of Terms and Abbreviations” on page 179 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 AIX PVMe: 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.

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 AIX PVMe

This chapter provides an overview of the IBM* AIX* PVMe* program product. Included in this overview is a discussion of the differences between the public domain PVM package and the IBM AIX PVMe 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 AIX PVMe?

The IBM AIX PVMe program product is an implementation of PVM***, a public domain API (application program interface) package. PVM, which stands for *Parallel Virtual Machine*, was developed by the Oak Ridge National Laboratory and was originally designed to allow usage of a heterogeneous network of parallel and serial computers to act as a single computational resource. The IBM AIX PVMe program product, on the other hand, was designed for use on a homogenous platform, specifically, the 9076 Scalable POWERparallel Systems with the High-Performance Switch feature.

The IBM AIX PVMe program product consists of:

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

The IBM AIX PVMe program product supports (as in the public domain PVM) both FORTRAN 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 AIX PVMe 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 a IBM AIX PVMe session.

The IBM AIX PVMe 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 2. New IBM AIX PVMe tasks may be generated and others may leave at runtime, providing a high degree of flexibility.

This release of the IBM AIX PVMe program product is object code compatible with version 3.2.6 of the public domain PVM. This means that existing PVM applications require only a relink step with the IBM AIX PVMe libraries.

© Copyright IBM Corp. 1993, 1994

Differences Between PVM and IBM AIX PVMe

The IBM AIX PVMe program product is compatible with PVM, but its internal structure is completely new. Because the IBM AIX PVMe program product runs on a homogeneous platform (the 9076 SP2), 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.

Although they are transparent to the application developer, the following characteristics differentiate the IBM AIX PVMe architecture and the public domain PVM:

- The IBM AIX PVMe program product does not interface directly with TCP/IP to perform data communication between processors. Instead, it interfaces with the Communication Subsystem (CSS), communication software that runs on the HPS adapters. Data exchange always occurs between processes, without involving the daemon.
- The IBM AIX PVMe daemon runs on only one processor node instead of all the participating processors, as it does in the public domain PVM. This allows simpler (and in many cases faster) task management.
- 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 AIX PVMe program product already contains the routines to work with process groups, while in the public domain PVM those routines are contained in an additional library.
- An enhanced memory management scheme of the IBM AIX PVMe program product also allows better exploitation of the memory resources, which is a critical issue when running very large applications.

For further information on the differences between the IBM AIX PVMe program product usage and the public domain PVM, see “Understanding the Differences between the IBM AIX PVMe Program Product and PVM” on page 61.

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 are processed.

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

During parallel processing, two or more processor 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 processor nodes and the way they coordinate calculations and communication. In some systems, the processor 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 processor 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 9076 SP2, are loosely coupled.

In addition to the coupling scheme of the processors 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 processor nodes *communicate* (share data). In the 9076 SP2 system, processor 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 processor owns *private memory* and is linked by means of an interconnection network. In the 9076 SP2, 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 processor 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 processor nodes to cooperate effectively, they must be properly *synchronized*. Synchronization refers to the exact sequence in which processor nodes perform their various tasks, taking into account the logical dependence and the order of precedence among the processor 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 processor node. In other words, all processor 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 processor 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 processor 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 processor node, divided by the number of processor nodes running your job in parallel.

In short, a parallel programming model describes the method in which processor 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 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.

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 processor 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 3). 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 provide little 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

processors 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 processor node and the total time spent in execution is the sum of the time spent to run each section separately.

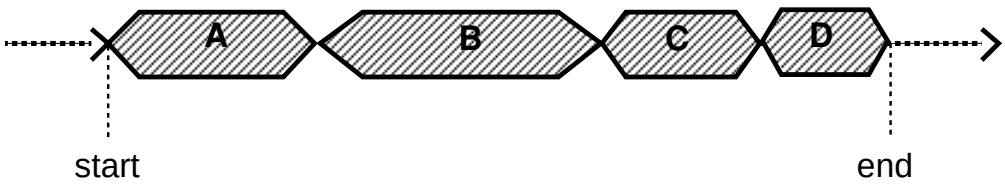


Figure 1. An Application Made Up of Four Logical Sections

However, if two sections, for instance B and C, are computationally independent, they can run concurrently on two processor 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 processor 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 7. In this case, we have better *load balancing*, so that the amount of time spent to run each section concurrently is comparable.

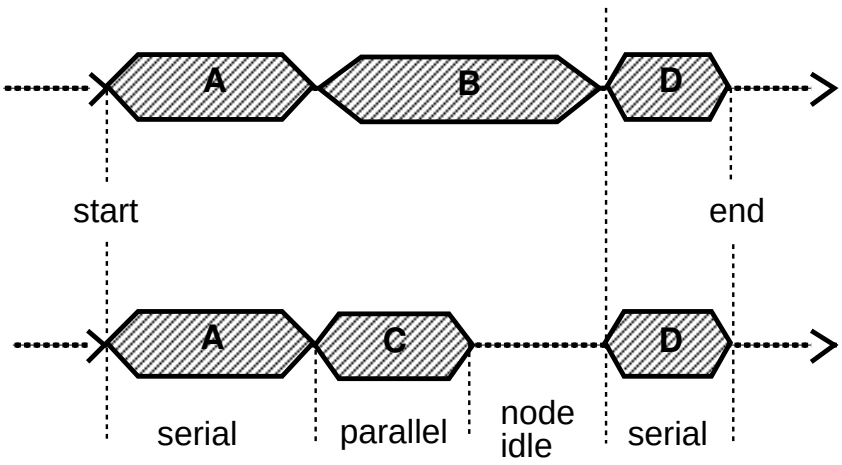


Figure 2. Program Decomposed into Two Parallel Tasks

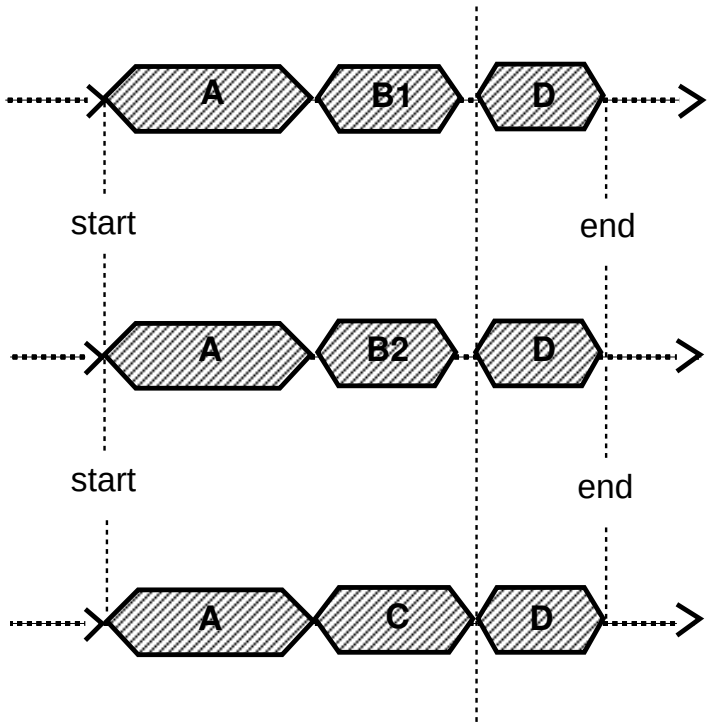


Figure 3. Better Load Balancing by Increasing the Number of Sections

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.

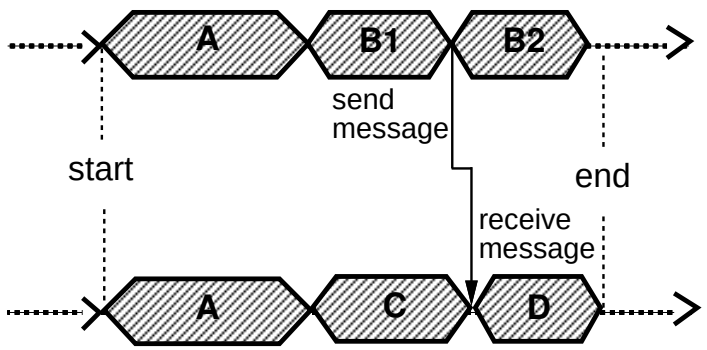


Figure 4. Partial Dependence Between Two Sections

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 10) and communicating among processor 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 processor node as composed of two parts such that:

$$T_1^s = T_{\text{serial}}^s + T_{\text{parallel}}^s$$

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.

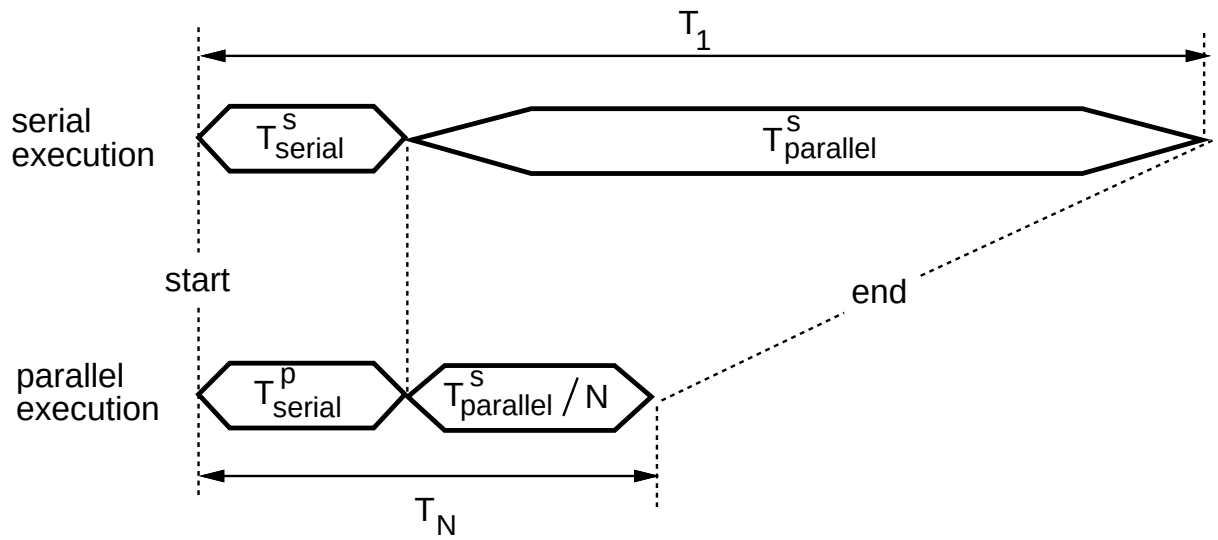


Figure 5. Serial and Parallel Execution Times

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

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

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 processor 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 processor nodes

Communication overhead refers to the time spent in communicating among processor 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 processor 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 processor 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 processors 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 processor nodes. This is because you can distribute small parallel tasks continuously, in turn, to each processor node, preventing a situation from arising where one or more processor 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 processor nodes in a predetermined and ordered way. With *dynamic load balancing*, you decide at execution which processor 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 processor 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 processor 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. However, it allows us to classify both IBM RISC System/6000* *clusters* and the 9076 SP2 as 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 processors, 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 processor, computing them locally will save time (unless you do not have enough local memory on your processor 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.

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 AIX PVMe'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).

- Do not use the format conversion functions if it is not necessary, because this can be time consuming.
- Use asynchronous transmission, provided hardware and software support it, because this can free up the central processor.

Chapter 3. Product Installation

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

Requirements

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

- The IBM 9076 SP2 system, configured with the High-Performance Switch feature.
- AIX level 3.2.5 (or later).
- The Parallel System Support Programs (5765-296), included with the 9076 SP2 system, configured and installed (IBM AIX PVMe uses the *Resource Manager* and *low-level communication protocol* which is part of this software).

Before You Begin

Before installing IBM AIX PVMe, be sure that the root id appears in the **.rhosts** file of each SP2 node.

Installing the IBM AIX PVMe Software

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

To begin the installation process:

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

3. Specify from what device you are installing the IBM AIX PVMe 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.

© Copyright IBM Corp. 1993, 1994

Creating Directory Structure

After restoring the IBM AIX PVMe package from the installation media, the appropriate directory structure is created under /**usr**/**lpp**/**pvm3**.

Next, on all the nodes, some system files are updated in order to load and configure a new kernel extension at nodes startup. This means that if one node (or the whole system) is switched off and then switched on again, the kernel extension will be loaded automatically, because the proper lines have been added in those files that will be run at boot time.

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

```
###PVMe
###PVMe
/usr/lpp/pvm3/etc/addkex /usr/lpp/pvm3/etc/pvm3e.kex     ###PVMe
```

Finally, some symbolic links are created in order to let you link applications without specifying the full path to the IBM AIX PVMe installation directory.

Note: All these updates may be reversed by rejecting the installation using the **smit** command.

If new nodes are added to an existing 9076 SP2 system, the IBM AIX PVMe software must be deinstalled and then reinstalled again, according to the procedures above.

Customizing Your Installation

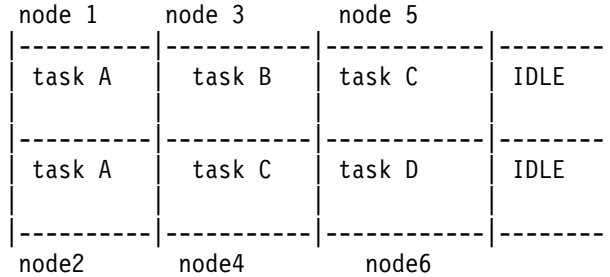
Once the IBM AIX PVMe program product has been successfully installed and configured, a few additional steps may be performed in order to customize the installation.

The **pvm6kcnf** file, located under the /**usr**/**lpp**/**pvm3** directory, contains a few parameters that may be changed to suit your particular needs. This file, by default, contains two lines:

```
MAXCOMP=4
MAXINST=4
```

The two lines above specify that the system wide maximum number of components (distinct executables) is 4 (MAXCOMP=4) and for each of them there may be 4 instances (MAXINST=4).

To further explain, consider the following example:



The number of components is 3 (A, B, C are three different executables). The number of instances of component A is 2 (there are 2 occurrences of program A), for B it is 1, and so on.

MAXCOMP sets the maximum number of distinct components supported by the IBM AIX PVMe program product. MAXINST sets the maximum number of instances of the same component supported by the IBM AIX PVMe program product.

At startup, the IBM AIX PVMe daemon (**pvmd3e**) looks for the **pvm6kcnf** file, first in your home directory and then, if not found there, under /**usr**/**lpp**/**pvm3**. By editing the main copy of this file or by placing an updated copy under the home directory, you can override the default for both 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 shell scripts open new windows on the screen, run **dbx** on them and run the user process. These scripts are run by the daemon whenever a new process must be started under **debugger**. You can customize **debugger** and **debugger2** by using the two templates provided with the IBM AIX PVMe program product package. The templates can be found under the /**usr**/**lpp**/**pvm3**/**lib** directory.

Chapter 4. Product Startup

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

The IBM AIX PVMe program product interfaces both with the Resource Manager and with the LoadLeveler* program products, so that on the 9076 SP2 system, you can easily run an IBM AIX PVMe session either in *interactive* or *batch* mode. This allows the Resource Manager to allocate the computational resources.

IBM AIX PVMe libraries rely on the Communication Subsystem (CSS) for communication between processes over the High-Performance Switch.

Running an IBM AIX PVMe Interactive Session

To run an IBM AIX PVMe session the IBM AIX PVMe daemon must be started. The daemon executable is **pvmd3e** and is located under the /**usr**/**lpp**/**pvm3**/ directory. There are two methods for starting an IBM AIX PVMe interactive session:

1. It can be started directly with the command:

   ```
   /usr/lpp/pvm3/pvmd3e
   ```

   Or

   ```
   /usr/lpp/pvm3/pvmd3e &
   ```

   Or by using the IBM AIX PVMe console (**pvm**):

   ```
   /usr/lpp/pvm3/pvm
   ```

   The following should appear and different options can be used:

   ```
   pvmd already running
   pvm>
   ```

2. Start the daemon and console at the same time by entering:

   ```
   /usr/lpp/pvm3/pvm <number of nodes > &
   ```

The **pvm** program is able to start the daemon on the local machine, if you have no other instances of the daemon running. If you have other instances of the daemon running, it contacts the active daemon. Then **pvm** allows you to get information from the daemon and to send it commands or requests.

Without arguments, the commands **pvm** and **pvmd3e** start the IBM AIX PVMe program product only on the local host. However, using the console, other nodes can be added to the same session.

To start the daemon on a set of processors, one of the following procedures must be used:

1. Start either **pvm** or **pvmd3e** by supplying the number of processors that are going to be used as an argument. The daemon sends 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). The

© Copyright IBM Corp. 1993, 1994

Resource Manager allocates nodes in **exclusive** mode for IBM AIX PVMe jobs, so that those nodes cannot be allocated for other jobs until your program finishes. This mechanism prevents nodes from having to participate in more than one IBM AIX PVMe session at a time, as well as users from competing for CPU and High-Performance Switch resources.

The default *pool_id* for nodes allocation is **1**; if you want nodes to be allocated from another *pool_id* you must use the **-p** option (see “Using the -p Option” on page 29).

The daemon obtains the list of available processors from the Resource Manager that will be printed on the standard output and on the **pvmnodelist** file in the working directory.

Note: The local host is not necessarily included in the list.

Once the daemon is started and the initialization is complete, your parallel program must be run on one of the allocated nodes. If the local host is not in the list, your programs must be started on a remote node (for example, using **rsh** or **rexec** system commands).

If the daemon executable is located in an alternative directory, in place of the default one, an environment variable must be set and exported:

```
export PVMDPATH = <absolute path to the directory for the PVMe
                   daemon executable>
```

Note: The default is /**usr**/**lpp**/**pvm3**/.

The IBM AIX PVMe daemon will search the **$(HOME)**/**pvm3**/**bin**/**RS6K** directory for your executables, whenever it receives the spawning new processes request. You can specify an alternative directory to be searched by setting and exporting another environment variable:

```
export PVMEPATH = <absolute path to the directory for the executables>
```

Note: The default is **$HOME**/**pvm3**/**bin**/**RS6K**.

Following are two examples:

```
pvmuser@sp2n1< export PVMEPATH = $HOME/bin
pvmuser@sp2n1< /usr/lpp/pvm3/pvm 4
RM_CLIENT : The RM server is running on sp2n1
PVMD: trying to get 4 nodes
PVMD: using sp2n8 with dx=/usr/lpp/pvm3/pvmd3e ep=/u/pvmuser/bin
PVMD: using sp2n9 with dx=/usr/lpp/pvm3/pvmd3e ep=/u/pvmuser/bin
PVMD: using sp2n10 with dx=/usr/lpp/pvm3/pvmd3e ep=/u/pvmuser/bin
PVMD: using sp2n11 with dx=/usr/lpp/pvm3/pvmd3e ep=/u/pvmuser/bin
PVM session 1 on 4 available
pvmd3e ready for 4 hosts....
pvm>
```

The session number shows you how many IBM AIX PVMe sessions are running on the nodes. The number **1** indicates that there is only one session running, while the numbers **2**, **3**, and **4** indicate that one or more nodes are being shared by multiple IBM AIX PVMe sessions. In general, the only safe session is session **1**.

```
pvmuser@sp2n1> export PVMEPATH = $HOME/bin
pvmuser@sp2n1> /usr/lpp/pvm3/pvmd3e 2 &
RM_CLIENT : The RM server is running on sp2n1
PVMD: trying to get 2 nodes
PVMD: using sp2n1 with dx=/usr/lpp/pvm3/pvmd3e ep=/u/pvmuser/bin
PVMD: using sp2n2 with dx=/usr/lpp/pvm3/pvmd3e ep=/u/pvmuser/bin
PVM session 1 on 4 available
pvmd3e ready for 2 hosts....
```

Then the console can be started, either on **sp2n1** or on **sp2n2**, to connect to the running daemon:

```
pvmuser@sp2n2> /usr/lpp/pvm3/pvm
pvmd already running
pvm>
```

2 Start either **pvm** or **pvmd3e** by supplying the name of a configuration file as an argument. It will be called **hostlist**, although that filename is not mandatory. In that file you must specify the names of the hosts that will be used for running the parallel application.

Example:

```
pvmuser@sp2n1> /usr/lpp/pvm3/pvm hostlist
```

In this case, the node where the daemon is run *must* be listed in the **hostlist** file.

Understanding IBM AIX PVMe Options

The **hostlist** file contains a line for each host participating in the virtual machine, including the *hostname* and possibly one or more of the following options:

- **ep=** <absolute path of the directory for the executables>
- **dx=** <full pathname for the IBM AIX PVMe daemon executable>
- **lo=** <user login name on the remote machine>

The meaning of the three 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**/**pvm3**/**pvmd3**.
- If the **lo=** option is not set, the login name is assumed to be the same as the user starting the daemon.

It is possible to 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 the following hosts until they are overridden by another set-defaults line. Hosts that you do not want in the initial configuration (but may add later) can

be specified in the **hostlist** by beginning those lines with an & (ampersand). The following example shows a **hostlist** file using the * (asterisk) and & an (ampersand):

```
# Lines starting with "#" are comment lines
# PVMe configuration
#
# Set a default option
* dx=/u/pvmuser/bin/pvmd3e
#
sp2n10
sp2n11       ep=/u/pvmuser1/bin
sp2n12       ep=/u/pvmuser2/bin
# sp2n13
#
# This host will be added later
&sp2n14  ep=/u/pvmuser3/bin dx=/u/pvmuser3/bin/pvmd3 lo=pvmuser3
```

Note: The **pw** option of the public domain PVM is not supported here.

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.

The following is an example using the configuration file of the previous example:

```
user@sp2n10> /usr/lpp/pvm3/pvmd3e hostlist &
PVM session 1 on 4 available
pvmd3e ready for 3 hosts...
```

Both for interactive and batch sessions, it is strongly recommended that you create the proper **.rhosts** file in your home directory before starting the IBM AIX PVMe program product. This is to allow you to run the **rsh** command from remote nodes without being asked to type your password. That file must have read/write permission only for the owner and must contain a line for each remote host to run the **rsh** command on the local host, using the following syntax:

<hostname>.<domainname> <username>

Running an IBM AIX PVMe Batch Session

When a parallel job is started in batch mode, LoadLeveler requests the desired number of nodes from the Resource Manager. The shell script submitting a parallel job to LoadLeveler must contain additional information with respect to a shell script for a serial job, that is, the number of nodes that are going to be used.

If the supplied number of nodes are available, the script must start the IBM AIX PVMe daemon by supplying the number of nodes, so that the daemon connects to the Resource Manager and obtains the list of allocated processors.

When the startup operations have been completed, the same script must start your program on one of the allocated nodes.

Below is an example of a script file that can be used to submit a parallel job to LoadLeveler. It can be modified or customized by you. The following variables can be set:

initialdir	Directory where LoadLeveler runs this script.
program	Name of the executable file you want to run.
parallelmachines	Number of nodes requested by LoadLeveler for this job.

These variables are concatenated in a single one (*load_var* in the example), then exported to the shell.

Note: The same script file can be used to directly run a IBM AIX PVMe session. If you do this, the three variables described above *must* be supplied as arguments.

```
#!/bin/ksh
# The following section contains statements for LoadLeveler only; it
# will be ignored if the script is executed directly by the user.
#
# @ input = /dev/null
# @ program = hostbenc
# @ initialdir = /u/loadl/pvm
# @ output = $(initialdir)/$(program).$(Cluster).$(Process).out
# @ error = $(initialdir)/$(program).$(Cluster).$(Process).err
# @ requirements = (Arch == "R6000") && (OpSys == "AIX32")
# @ notify_user = pvmuser@sp1serv1.ecsec.romesc.ibm.com
# @ class = PVMe1
# @ parallelmachines = 2
# @ account_no = system
# @ notification = complete
# @ environment = loadl_var=$(parallelmachines):$(initialdir):$(program)
# @ checkpoint = no
# @ queue
#
# The following section will be used both when the script is submitted
# to LoadLeveler and when it is executed directly.
#
######  Get the running shell name without the path
#
export CMD="${0##*/}"
#
######  Extract the 3 variables from the "load_var" variable
######  or parse arguments from the command line.
#
if [ -n "${loadl_var}" ]
then
        OFS=$IFS
        IFS="$IFS:"
        set ${loadl_var}
        IFS=$OFS
        job=yes
elif [ $# -ne 3 ]
then
        print -u2 "Interactive Usage: $CMD nodes initialdir program"
        exit 1
fi
if [ -n "$1" ]
then
```

```
  node=$1
else
  print -u2 "$CMD: missing number of nodes"
  exit 1
fi
if [ -n "$2" ]
then
  initialdir=$2
else
  print -u2 "$CMD: missing initial directory name"
  exit 1
fi
if [ -n "$3" ]
then
  program=$3
else
  print -u2 "$CMD: missing program name"
  exit 1
fi
#
######  Set up environmental variables for the PVMe demon ...
#
export PVMDPATH=/usr/lpp/pvm3
export PVMEPATH=$initialdir
#
######  ... and for convenience:
#
export PVMD=$PVMDPATH/pvmd3e
export PROGRAM=$PVMEPATH/$program
#
######  Store in PVMLIST the filename where the daemon will write
######  the list of nodes allocated by the Resource Manager.
#
export PVMLIST="pvmnodelist"
#
######  Get also the hostname, and remove the domain name:
#
export HOST= hostname
HOST=${HOST%%.*}
#
echo "Running $CMD on node $HOST"
echo "INITIAL DIRECTORY =$initialdir"
echo "NUMBER OF NODES =$node"
echo "PROGRAM NAME =$program"
#
######  Remove the <PVMLIST> file of a previous run (if any),
######  and creat it again, to be sure that it contains 0 lines.
#
echo "$CMD: Removing previous pvmnodelist file"
rm -f $PVMLIST
echo "$CMD: Creating file $PVMLIST"
touch $PVMLIST
#
######  Start the daemon; before continuing, we must be sure that
######  the daemon has completed the initialization step on all the
######  allocated nodes. One possible strategy is to wait for N seconds,
######  where N must be properly set. Another possibility, adopted in the
######  example, is to wait until the daemon has written <node>
```

```
######  line in <PVMLIST>: in fact the daemon writes each line in
######  the file when the initialization of the corresponding node is
######  completed. If in the meanwhile the daemon died (e.g.: it was not
######  able to get enough nodes) we stop waiting and exit.
#
echo "$CMD: starting $PVMD demon on $node nodes"
($PVMD $node )  &
PVMD_PID= jobs -p
if [ -n "$PVMD_PID" ]
then
  echo "$CMD: started $PVMD demon process_id=$PVMD_PID"
  #
  echo "$CMD: Waiting for $PVMD ready"
  while [  wc -l $PVMLIST | cut -c1-7  -ne $node ]
  do
        sleep 2
        echo "$CMD: - 2 seconds"
        ps $PVMD_PID > /dev/null
        if [ $? -ne 0 ]
        then
          print -u2 "$CMD: Error: $PVMD dead"
          exit 1
        fi
  done
#
######  Now the daemon is ready; check whether the local host is in
######  the list or not.
#
  echo "$CMD: checking current node in the list"
  grep $HOST $PVMLIST > /dev/null 2>&1
#
######  If it is not in the list, we read the name of the first node
######  in the list and start remotely the user program on it.
#
  if [ $? -ne 0 ]
  then
    echo "$CMD: Warning current node=$HOST not in nodes pool"
    exec 3<"$PVMLIST"
    read -u3 RHOST REST
    exec 3<&-
    echo "$CMD: Starting $PROGRAM on $RHOST:$initialdir"
    ( rsh $RHOST "$PROGRAM; exit" )
#
######  If it is in the list, we can start the program.
#
  else
    if [ -x $PROGRAM ]
    then
      echo "$CMD: Starting user $PROGRAM"
      ($PROGRAM )
    else
      print -u2 "$CMD: error $PROGRAM not found"
    fi
  fi
  #
#
######  Before exiting, kill the daemon ....
#
```

```
  echo "$CMD: killing $PVMD"
  kill %1
  #
  echo "$CMD: Waiting for $PVMD pid=$PVMD_PID"
  ps $PVMD_PID > /dev/null
  while [ $? -eq 0  ]
  do
        sleep 2
        echo "$CMD: - 2 seconds"
        ps $PVMD_PID > /dev/null
  done
#
######  .... and run it again with the "reset" option to cleanup.
#
  if [ job = "yes" ]
  then
    echo "$CMD: Cleaning $PVMD on the previous allocated nodes"
    $PVMD -r $node
  fi
#
###### We were not able to start the daemon
#
else
  print -u2 "Error $PVMD is not started"
fi
exit 0
```

Using the IBM AIX PVMe Console

In the public domain version, the PVM console may be started and stopped multiple times on any of the hosts on which PVM is running. In the IBM AIX PVMe program product a single console per node is supported. The console allows interactive starting and killing of the IBM AIX PVMe processes and accepts the following commands from the standard input:

Command	Description
add	Followed by one or more *hostnames*, it adds these hosts to the virtual machine.
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	Followed by one or more *hostnames*, it deletes these hosts. The IBM AIX PVMe processes still running on these hosts are killed.
echo	Prints the supplied argument(s).
halt	Halts all IBM AIX PVMe processes including consoles and then shuts IBM AIX PVMe down.
help or ?	Shows a list of the available commands.
help [command]	Displays an explanation of the command and its syntax.

id	Prints the console 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 AIX PVMe process.
quit	Exits the console leaving daemons and IBM AIX PVMe jobs running.
reset	Kills all IBM AIX PVMe processes except consoles and resets all the internal IBM AIX PVMe 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 a IBM AIX PVMe process.
spawn	Starts one or more IBM AIX PVMe 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]
```

Options for the IBM AIX PVMe Daemon

The IBM AIX PVMe daemon accepts three options which are used to query and reset the status of an IBM AIX PVMe session. The options are:

```
-q, -c, -r
```

The options are mutually exclusive, that is, only one can be used at a time. The daemon *must* be started in the foreground when one of these options is used.

Another option, **-p**, controls the interaction between IBM AIX PVMe and the Resource Manager.

Using the -q Option

The **-q** option allows you to query the status of your IBM AIX PVMe session. The syntax is:

```
$pvmd3e hostlist -q
```

Where:

hostlist
Is the file which contains the list of the processors used by the IBM AIX PVMe program product. The structure of this file is described in “Running an IBM AIX PVMe Interactive Session” on page 19.

Notes:

1. The **-q** option shows the ID of the IBM AIX PVMe processes belonging to you (for each processor in the list).
2. The **-q** option also shows the number of the first IBM AIX PVMe session available, which allows you to see if there are (active) IBM AIX PVMe sessions.
3. If the number of the first session available is two, this means that at least one processor of the set listed in the **hostlist** file is already involved in another session.

If the nodes have been allocated by the Resource Manager, the daemon creates a file (**pvmnodelist**) that contains the names of the nodes allocated by the Resource Manager. This file can then be used as **hostlist**, running the daemon with the **-q** option.

Using the -c Option

The **-c** option is used to reset your IBM AIX PVMe session. The syntax is the same as the **-q** option:

```
$pvmd3e hostlist -c
```

Where:

hostlist
Is the file which contains the list of the processors used by the IBM AIX PVMe program product. The structure of this file is described in “Running an IBM AIX PVMe Interactive Session” on page 19.

Notes:

1. The **-c** option terminates all of *your* IBM AIX PVMe processes.
2. The output for this option is the number of the first session available after termination.

Using the -r Option

The **-r** option is used to reset *all* the IBM AIX PVMe sessions running on the processors nodes listed in the **hostlist** file.

The syntax is:

```
$pvmd3e hostlist -r
```

Where:

hostlist
Is the file which contains the list of the processors used by the IBM AIX PVMe program product. The structure of this file is described in “Running an IBM AIX PVMe Interactive Session” on page 19.

Notes:

1. With the **-r** option, *all* IBM AIX PVMe processes are terminated (but not the IBM AIX PVMe daemons), regardless of ownership. In this way, the first available IBM AIX PVMe session becomes the number 1.
2. For a user other than *root*, running the daemon with the **-r** option is equivalent to using the **-c** option.

Using the -p Option

The **-p** option 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. Example:

```
user@spn1n1> /user/lpp/pvm3/pvm 5 -p 2
```

Resolving Startup Problems

If, after several startup attempts, an error message continues to appear, it might mean that an IBM AIX PVMe 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 **-c** option or ask the system administrator to run the daemon with the **-r** option. See “Options for the IBM AIX PVMe Daemon” on page 27 for more information.

When you start the **pvmde** 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 AIX PVMe daemon creates the **.pvmdname** file in the default directory of the executables, in your **HOME** directory and, if any, in the alternative directory supplied by you for the executables (see page 20). 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 AIX PVMe 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 AIX PVMe sessions, on the same set of nodes or on a disjointed 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 AIX PVMe 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 ambiguity among different sessions, you can specify an alternate filename where the daemon writes its address, and the parallel program reads it. You must set the environment variable **PVMHFN** to a filename of your choice and export that variable. 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.

Testing the IBM AIX PVMe Program Product

A sample application is available under the **sys** subdirectory to test the installation and to run simple performance tests.

There are two programs, **hostbenc** and **nodebenc**, an **include** file and a **makefile** for testing.

To test the IBM AIX PVMe program product:

1. Set the variable **EXEDIR** to the directory where executable files should be placed.
2. Set the **hostlist** file.
3. Start the IBM AIX PVMe daemon and run the **hostbenc** program.

   It prompts you for three inputs:

   a. Number of words **k**
   b. Number of times **n**
   c. Transmission mode

      Note: This is significant with the public domain PVM but can be any value with the IBM AIX PVMe program product.

4. It then makes a ping-pong with the **nodebenc** program giving the time it takes to send and receive *k* words *n* times as output.

Chapter 5. Using the IBM AIX PVMe Program Product

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

- An overview of the IBM AIX PVMe features and of the routines that are callable by user applications. Its purpose is to explain how the IBM AIX PVMe 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 6, "IBM AIX PVMe Reference: C and FORTRAN Subroutines" on page 77.
- 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 AIX PVMe Features

The IBM AIX PVMe 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 host-node 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 AIX PVMe 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
  - Signaling
- The IBM AIX PVMe daemon is also included to support running programs. The IBM AIX PVMe daemon looks like a supervisor and a coordinator for your IBM AIX PVMe 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:
  - The hosts where they are running
  - The synchronization events
  - The groups configuration

© Copyright IBM Corp. 1993, 1994

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

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

Managing Tasks

The primitives for task management allow you to:

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

Primitives for Managing Tasks

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

Table 1. IBM AIX PVMe Task Management Routines

C	**FORTRAN**
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 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 taskinfo **taskp*)	call pvmftasks (*where*, *ntasks*, *tid*, *ptid*, *htid*, *flag*, *name*, *rc*)

Examples for Managing Tasks

FORTRAN Example

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

```
      program master

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

c Enroll to PVM3
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 by 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 AIX PVMe routine (with the exception of the **pvm_start_pvmd** routine) it must enroll with the IBM AIX PVMe daemon. This task is accomplished by calling the **pvm_mytid** routine. It returns the *task identifier* of the process within the IBM AIX PVMe environment.

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

The IBM AIX PVMe 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 AIX PVMe 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.

Unenrolling a Process

When a process no longer needs the IBM AIX PVMe services, or is ready to exit, it should detach the IBM AIX PVMe 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 AIX PVMe program product. The daemon changes the process status by setting it to **FALSE**.

When all processes have left the IBM AIX PVMe program product without errors, the IBM AIX PVMe database is reset and the daemon loses memory of the previous parallel job executions. In IBM AIX PVMe terms, a new *epoch* is started. 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 AIX PVMe console during an interactive session.

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

Starting a Process

An IBM AIX PVMe 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 AIX PVMe Interactive Session” on page 19 for more information.
- By another IBM AIX PVMe process.

The IBM AIX PVMe 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 17 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 public domain PVM. In fact, IBM AIX PVMe supports only one architecture, RS6K. RS6K is the name of the architecture RISC System/6000 within the IBM AIX PVMe program product, and it is the string to be used when the *architecture* parameter must be supplied to an IBM AIX PVMe routine.

When a process calls the **pvm_spawn** routine, a request is sent to the IBM AIX PVMe daemon. If a *hostname* is not specified, the daemon chooses a host for the new instance, following the round-robin rule. 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 AIX PVMe environment.

Searching for Executable Files

The IBM AIX PVMe 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. 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 should be less than the number of new instances required by you.

Terminating a Process

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

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

Querying 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 AIX PVMe 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 AIX PVMe process.

Accessing Storage Devices for Read/Write Operations

Like any other UNIX process, an IBM AIX PVMe process may access storage devices for **read**/**write** operations. However, you must keep in mind that processes started by the IBM AIX PVMe daemon search your **HOME** directory for data files. This implies that you must either locate data files in your **HOME** directory or specify the full *pathname* to the application to be sure that calls like **open**, **read**, **write**,and **close** complete successfully.

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 AIX PVMe daemon cannot.
2. **stdout** and **stderr** (for processes started by the IBM AIX PVMe program product), are redirected, by default, to **stderr** 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 AIX PVMe 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.

Managing Hosts

The primitives for task management allow you to:

- Start/stop the IBM AIX PVMe 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 AIX PVMe host management routines:

Table 2. IBM AIX PVMe Host Management Routines

C	FORTRAN
int nh = pvm_addhosts (*char **hosts*, *int nhosts*, *int* rcs*)	call pvmfaddhost (*host*, *rc*)
int rc = pvm_config (*int *nhost*, *int *narch*, *struct hostinfo **hosts*)	call pvmfconfig (*nhost*, *narch*, *tid*, *name*, *arch*, *mtu*, *speed*, *rc*)
int nh = pvm_delhosts (*char **hosts*, *int nhosts*, *int *rcs*)	call pvmfdelhost (*host*, *rc*)
int rc = pvm_halt ()	call pvmfhalt (*rc*)
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*)

Starting and Stopping the Virtual Machine

Your programs are allowed to start up the IBM AIX PVMe 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 AIX PVMe 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 19. 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 19), then the IBM AIX PVMe 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.

Adding/Deleting Hosts from the Virtual Machine

The virtual machine configuration can be modified at any time during a global search of an IBM AIX PVMe session, both from the console and from your applications.

The **pvm_addhosts** routine allows you to add one or more hosts with a single call, while the corresponding FORTRAN routine adds a single host with each call. You must supply the *hostnames* for the new hosts. When the full name, (for example, **hostA**<*domainname*>) is supplied, the routine removes the <*domainname*>, so that it will be successful only if the short name (for example, **hostA**) has been defined.

Special statements allow you to define your options (such as, **ep**, **dx**, or **lo**) in the configuration file for hosts that can be added later. For further information on these options, see “Understanding IBM AIX PVMe Options” on page 21.

In the IBM AIX PVMe program product, there is always a single daemon running throughout the whole virtual machine. When a host is added a new daemon is not started on it. Hosts, as well as processes, have unique identifiers within the virtual machine. This identifier can be used to request information about a particular host or about the processes running on it. The **pvm_addhosts** routine returns an array containing the identifiers of the newly added hosts.

The **pvm_delhosts** routine allows you to delete one or more hosts from the virtual machine (the FORTRAN routine deletes a single host). You cannot delete the *master host*, that is, the host where the daemon is running.

CAUTION:

Special attention must be taken when using the pvm_delhosts routine, because all IBM AIX PVMe processes running on these hosts to be deleted will be killed.

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 AIX PVMe runs in an homogeneous environment, but it is compliant with the public domain PVM.
- The host name
- The host identifier
- The host architecture
- The maximum transfer unit
- The relative CPU speed

  Note: The last data is the same for all the hosts, because IBM AIX PVMe 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 *host-node* fashioned application. The host program starts both the IBM AIX PVMe environment on a set of hosts and the node 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 = { "sp2n01",
                   "sp2n02",
                    .......
                   "sp2n15",
                   "sp2n16" };

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

  int tasks[MAXHOST], ntasks;
  int nhosts, narchs;
  int me, i, rc;
  struct hostinfo *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);
  }

  /* Before starting the computation, add, if */
  /* possible, new computing resources        */
```

```
  for (i=0; i< MAXHOST; i++) {
     rc = pvm_mstat(allhosts[i]);
     if (rc==PvmNoHost) {
     /* The hosts is not running in the virtual machine */
         rc = pvm_addhosts (&allhosts[i], 1, &rc);
         if (rc == 1 ) {
            printf ("Host %s successfully added \n",allhosts[i]);
         } else
            printf ("Error adding host %s \n",allhosts[i]);
     }
  }

  /* Get updated 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 processors are available. One processor */
  /* (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 AIX PVMe 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 AIX PVMe 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 AIX PVMe 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 AIX PVMe data communication routines:

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

C	**FORTRAN**
int rc = pvm_advise (*int route*)	call pvmfadvise (*route*, *rc*)
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 AIX PVMe 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 bufid = pvm_probe (*int tid*, *int msgtype*)	call pvmfprobe (*tid*, *msgtype*, *bufid*)
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 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*)

Preparing a Message

The IBM AIX PVMe 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 AIX PVMe 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 (**PvmDataRow** and **PvmDataDefault**) were used in public domain PVM to enable/disable data conversion to/from the XDR format. Both of them are supported in the IBM AIX PVMe

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 away 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_pknint**, 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 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 60 kilobytes, 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.

The **pvm_advise** routine has been implemented only for compatibility. In the public domain PVM you are allowed to select one out of two possible communication paths for data exchange through the daemons or by using direct process-to-process communication links. With the IBM AIX PVMe program product, data exchange always occurs directly between the sending and receiving processes. In addition, the high-level protocols (TCP-UDP/IP) are bypassed and the low-level communication libraries are interfaced, to maximize performance in communication.

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.

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.

pvm_send and **pvm_mcast** are nonblocking routines, that is, they return quickly, whether the recipient task performs a receive or not.

Broadcast to all members of a given group is discussed in "Understanding the Broadcast Operation" on page 58.

Receiving a Message

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

- The **pvm_recv** routine is a blocking routine, that blocks until the proper message arrives (it returns immediately if the proper message has already arrived).
- The **pvm_nrecv** routine 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.

Both 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 receving 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 B, “IBM AIX PVMe Intrinsic Limits: Summary” on page 173.

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 AIX PVMe 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.

This routine receives only one parameter (*encoding*), which can assume three different values, two of them (**PvmDataRow** and **PvmDataDefault**) were used in public domain PVM to enable/disable data conversion to/from the XDR format. Both of them are supported in the IBM AIX PVMe 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. However, the same data can be sent multiple times, with modifications between the sends, without having to pack it each time. When no more buffers are available a special code (**PvmNoMem**) is returned.

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 AIX PVMe 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'

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 processors */

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);

for (;;) {

  /* receive messages from all nodes and save buffers  */

  for (i=0; i<nproc; i++) {
    if ( (pvm_receive (-1, -1, msgtype)) < 0 )
        pvm_perror ("error in receive routine");
```

```
        goto errorlabel;
      }
      bufs[i] = pvm_setrbuf (0);
    }

    /* 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 Chapter 6, “IBM AIX PVMe Reference: C and FORTRAN Subroutines” on page 77.

Primitives for Error Handling

All the IBM AIX PVMe 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 positive value if the routine was successful. All the routines, (with the exception of **pvm_spawn**, **pvm_addhosts**, and **pvm_delhosts**, discussed below) return a negative value if they were unsuccessful.

Table 4. IBM AIX PVMe Error Handling Routines

C	**FORTRAN**
int val = pvm_getopt (*int opt*)	call pvmfgetopt (*opt val*)
int rc = pvm_perror (*char *msg*)	call pvmfperror (*msg*, *rc*)
int oldset = pvm_serror (*int set*)	call pvmfserror (*set*, *oldset*)
int oldval = pvm_setopt (*int opt*, *int val*)	call pvmfsetopt (*opt*, *val*, *oldval*)

The three routines cited above, **pvm_spawn**, **pvm_delhosts**, **pvm_addhosts**, can return a negative error code if invalid parameters were supplied. If valid parameters were supplied, they return the actual number of spawned processes, or hosts deleted or hosts added. 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**, **delete**, and **add** attempts were successful.

Automatic Printing of Error Messages

The **pvm_serror** routine allows you to set a flag indicating whether to enable or disable automatic printing of error messages by the IBM AIX PVMe routines. When a flag is set to 1, it enables the IBM AIX PVMe routines to print a message on **stderr** whenever an error occurs. When a flag is set to 0, it disables this action.

Whenever an error occurs, **libpvm.a** routines set an internal variable to the error code, so that this variable contains the last error code.

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.

There are two utility routines, **pvm_setopt** and **pvm_getopt** that allow you to set options and obtain their current value. These routines have been implemented only for compatibility with the public domain PVM. This is because only a subset of the public domain PVM options are supported, and only one of them is meaningful.

Note: The **PvmAutoErr** error code can be set either to 0 or to 1, with the same effect as calling the **pvm_serror** routine.

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 AIX PVMe 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_serror** 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_serror (1);

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

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

  ......

}
```

Signaling

The IBM AIX PVMe 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 AIX PVMe signaling routines:

Table 5. IBM AIX PVMe 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*)

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 */
/****************/

#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 AIX PVMe processes can be organized into task groups. These task groups run specific operations on subsets of tasks only. *Broadcasts* and *barriers* are typical operations involving only members of a given group. Managing task groups allows you to structure complex applications in an organized fashion.

Managing Task Groups with Primitives

The basic concepts of group management in IBM AIX PVMe 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 disjointed.

Each group is identified with a string, containing the group name. All processes belonging to a group have an instance number within that group. An **include** file (**pvmgs.h** for C programs) contains the definitions of the maximum group number, maximum group size and maximum length of group names.

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

Table 6. IBM AIX PVMe 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 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_getsize (*char *group*)	call pvmfgetsize (*group*, *size*)
int inst = pvm_joingroup (*char *group*)	call pvmfjoingroup (*group*, *inst*)
int rc = pvm_lvgroup (*char *group*)	call pvmflvgroup (*group*, *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.

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.

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.

Understanding the Barrier Routine

The **pvm_barrier** routine is the only routine provided by the IBM AIX PVMe program product that imposes 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 imposes a set of processes belonging to the same group to block until a specified number of processes have called the same barrier. It is not necessary that all processes belonging to the same group be involved in the synchronization.

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();
}
```

Understanding the Differences between the IBM AIX PVMe Program Product and PVM

The IBM AIX PVMe program product was designed to optimize performance while maintaining the same parallel programming paradigm offered by the public domain PVM. However, there are some inherent differences between the IBM AIX PVMe program product and the public domain package. IBM AIX PVMe could almost be considered a *stateless* system while PVM could be considered a *statefull* one.

Comparing a snd Operation

At the user level, the differences in the two packages impact the behavior of a process only when a **snd** 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 AIX PVMe process will keep trying to complete the operation, while a process that uses the public domain 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 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 AIX PVMe 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_exit** 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. As before, if the communication path passing through the daemons is exploited, the behavior of the IBM AIX PVMe 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 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 AIX PVMe daemon. With the IBM AIX PVMe 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.

Starting New Tasks

There are also differences in the way the IBM AIX PVMe program product and public domain PVM start new tasks through the **pvm_spawn** routine, which is based on the **rcmd** utility of the AIX Operating System.

Some of these differences are:

- Public domain PVM does not pass any arguments to the new tasks, beyond those supplied by you.
- For the IBM AIX PVMe program product, the daemon passes its own arguments to the new processes. However, such arguments can be easily distinguished, because they are all of the form **-_pvm**_*xxxx=yyyy*, and always follow the arguments supplied by you.

Compiling and Linking a Parallel Program

The IBM AIX PVMe libraries for C and FORTRAN programs were stored in the /**usr**/**lpp**/**pvm3** directory during installation. Symbolic links were created so that you would not be required to specify the full *pathname* for the libraries.

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 AIX PVMe libraries:

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

The example below shows a **makefile** for compiling host and node programs:

```
FLAGS = -O
FC = xlf
CLIB  = libpvm3.a
FLIB  = libfpvm3.a
LIBS  = $(CLIB) $(FLIB)
IMPORT= -bI:/usr/lib/pvm3e.exp
TARGETDIR = $(HOME)/pvm3/bin/RS6K

LFLAGS=
.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 nodepvmr $(TARGETDIR)
```

Since the CSS is a shared library, at run time the **LIBPATH** environment variable must contain, at the beginning, the path to the CSS library that you are going to use.

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

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

Running a Parallel Program

The IBM AIX PVMe program product may run either in batch or interactive mode. The procedure for starting a IBM AIX PVMe session was discussed in Chapter 4, “Product Startup” on page 19. 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 AIX PVMe program product searches for programs to be started
- An alternate directory, provided that you tell the IBM AIX PVMe 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. For example:

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

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

```
hostA   ep=$HOME/benchmark/gauss
hostB   ep=$HOME/benchmark/gauss
hostC
```

A mechanism is available which allows environment variables to be passed to processes spawned by the IBM AIX PVMe 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. Example:

```
user@sp2n1> export INPUT_PATH=$HOME/data/input
user@sp2n1> export DISPLAY=xstat12:0
user@sp2n1> export PVM_EXPORT=DISPLAY:INPUT_PATH
user@sp2n1>
user@sp2n1> /usr/lpp/pvm3/pvm 8
```

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 AIX PVMe routines. They are recommended during code development and debugging and can be removed when the program works as expected.

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.

   Run the parallel program in parallel on a single host (as a set of tasks all running on the same processor). Provided the IBM AIX PVMe 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.

2. Gradually increase the number of processes (and number of processors) 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.

   A single host must be included in the *hostfile*, or a single node must be requested 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. The communication protocol is not involved at all. In this way, errors derived from bugs in the synchronization or in the data exchange mechanisms can be detected. The program may be run under a standard debugging tool (such as **dbx**), like any other serial program.

   For example, you will be able to recognize the following types of errors:

   - That a process is blocked waiting for a message with a given message type

- That a message is never sent
- A process is waiting for a message with a similar message type
- More messages with the same message type arrive

  Note: Messages may be received in different order from execution to execution, causing unpredictable results.
- Errors due to wrong use of the synchronization routines can be detected in this phase.

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 AIX PVMe 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 the same number of windows as there are new tasks to be started. Run **dbx** on them, supplying the executable *filename* for the new programs, and all the proper parameters.

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/pvm3/lib** directory, and must be customized by you. For more information, see “Customizing the debugger and debugger2 Shell Scripts” on page 17.

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 host-node 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 Befor 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. IBM AIX PVMe Reference: C and FORTRAN Subroutines

This chapter contains an alphabetical listing of all the IBM AIX PVMe 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 naming conventions that the FORTRAN compiler uses to call C routines. 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

Conventions for Syntax Diagrams

Syntax diagrams use the following conventions:

- Required keywords and values appear on the main path line. These items include the routine name, flags, and literal characters.
- Optional keywords and values appear below the main path line.

© Copyright IBM Corp. 1993, 1994

- Diagram items representing variables that must be replaced by a name are in *italics*. These items include parameters that follow flags and parameters that the subroutine reads, such as *Files* and *Directories*.

Notes:

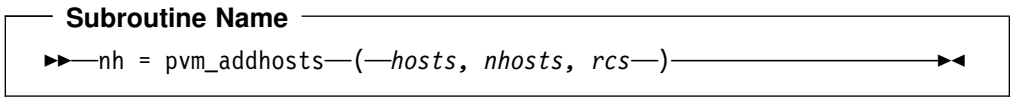
1. FORTRAN is not case sensitive, therefore, you can enter the FORTRAN subroutines in either upper, lower, or mixed case. C is case sensitive, and the subroutines must be entered in lower case only.
2. In this book both C and FORTRAN syntax appear in all lower case.
3. The variables in the routines in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

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:




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

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

pvm_addhosts

Description

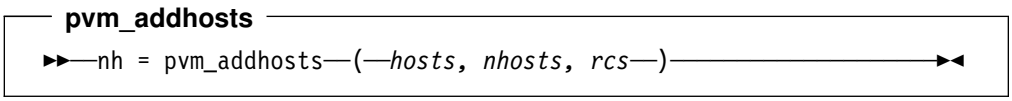
The C routine allows you to add one or more hosts to the virtual machine configuration. The FORTRAN routine allows you to add a single host at each call. At any time your processes can call the **pvm_config** or the **pvm_mstat** routines to be informed about the virtual machine configuration.

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.

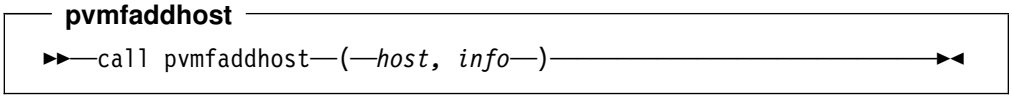
The C routine also returns the number of hosts successfully added. If a failure occurred, this number is less than the value of the *nhosts* parameter. The error codes are stored in the proper elements of the *rcs* array. You can identify an error as a negative number, greater than -256, because the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

C Syntax




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

FORTRAN Syntax




```
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

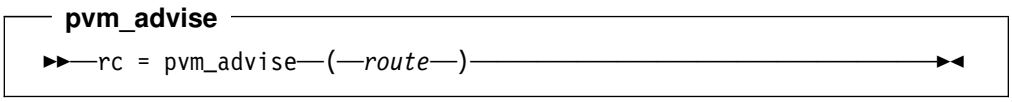
PvmSysErr	System error. The maximum number of hosts has been reached. No more sessions available on the host to be added. Process not enrolled.
PvmBadParam	*Hostname* longer than 64 characters.
PvmNoHost	Invalid *hostname*.
PvmDupHost	Host to be added is already running in the virtual machine.
PvmOutOfRes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

pvm_advise

Description

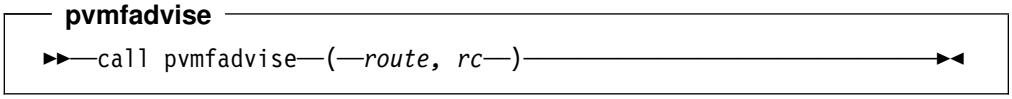
The **pvm_advise** routine is a dummy routine that has been implemented only to maintain compatibility with the public domain PVM. The routine was used to set up direct communication links (TCP sockets) between user processes. In IBM AIX PVMe data exchange always occurs using direct process-to-process links. To improve performance, these communication paths are built directly on top of the low-level communication protocol, and the high-level protocols (TCP/IP) are bypassed.

C Syntax




```
int route;
int rc;
```

FORTRAN Syntax




```
integer route
integer rc
```

Parameters

route
Integer flag which may take any of the following values:

PvmDontRoute (PVMDONTROUTE)	1
PvmAllowDirect (PVMALLOWDIRECT)	2
PvmRouteDirect (PVMROUTEDIRECT)	3

Note: The strings can be used as well. Strings in parenthesis are to be used in FORTRAN programs.

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

```
#include "pvm3.h"
 ...
int rc;

rc = pvm_advise (PvmRouteDirect);
```

FORTRAN Example

```
integer rc
include 'fpvm3.h'

call pvmfadvise (PVMROUTEDIRECT, rc)
```

Error Conditions

PvmSysErr	System error. Process not enrolled.
PvmBadParam	Invalid route value.

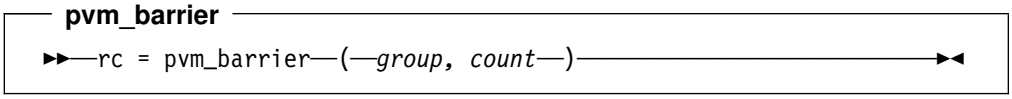
pvm_barrier

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 has been designed to be a synchronization among all processes belonging to a group. Although not strictly required, the *count* parameter should contain the total number of members of the group.

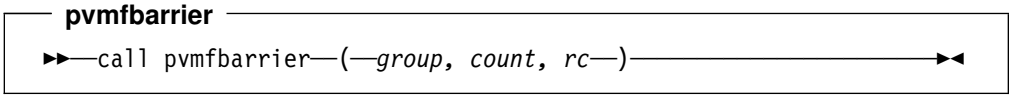
If a `-1` is passed as *count* value, then the result of the **pvm_gsize** routine (that is the total number of members of the group), will be used.

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 required, it is expected to be 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, nslaves
parameter (nslaves=12)

call pvmfjoingroup ("slaves", rc)

if (rc.ge.0) then
   call pvmfbarrier ("slaves", nslaves, 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.
PvmTooLong	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

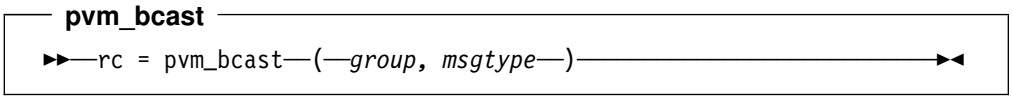
Description

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. However, if it is a member of the supplied group, it will not receive the message.

pvm_bcast is a nonblocking routine. It returns when the message has been delivered to all recipient tasks, even though the recipient tasks may not have called a **pvm_recv** or **pvm_nrecv** routine (recipient tasks interrupt their processing to store incoming messages in a pre-receiving area).

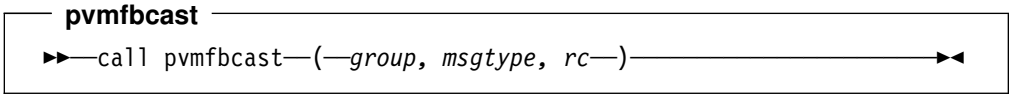
The **pvm_bcast** routine first determines the *tids* of the group members, then runs a multicast to these *tids*. If the group size is changed during the call, the change is not reflected in the broadcast.

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

Error Conditions

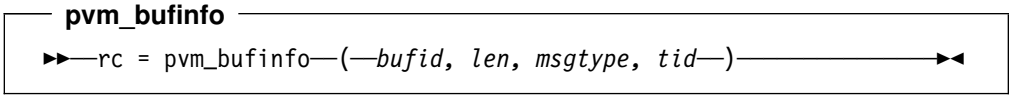
PvmSysErr	System error. Process not enrolled.
PvmBadParam	*group* is a NULL pointer.
PvmTooLong	*group* is longer than 32 characters.

pvm_bufinfo

Description

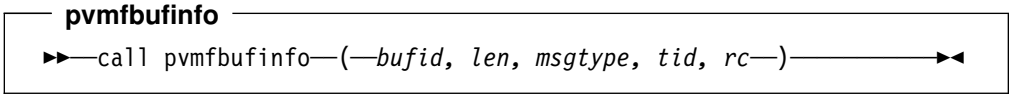
The **pvm_bufinfo** routine returns information about the message received last in the supplied *bufid*. It returns the message size, the message type and the identifier of the sending process. It is useful when the application receives any incoming message without following an established order. At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

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.

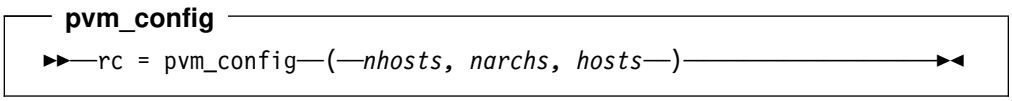
Note: This parameter (for the number of different architectures) is always set to 1; it has been introduced only for compatibility with the public domain PVM.

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).

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.

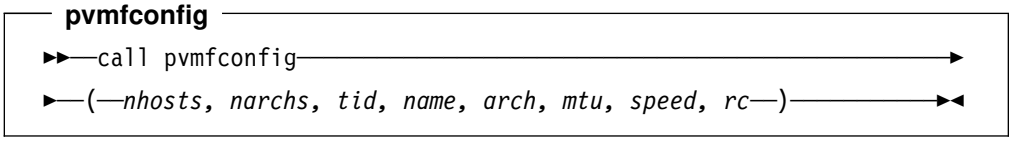
Note: Data returned from this routine is the same as that shown by the **conf** command from the IBM AIX PVMe console.

C Syntax




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

FORTRAN Syntax




```
integer nhosts
integer narchs
integer tid
character(*) name
character(*) arch
integer mtu
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 AIX PVMe program runs on a homogeneous platform.

hosts
Address of an array of *hostinfo* structures. The array will be *nhosts* long. Each structure contains data about a single host, and has the following shape:

```
struct hostinfo {
   int   hi_tid                  /* host id            */
   char *hi_name                 /* host name          */
   char *hi_arch                 /* host architecture  */
   int   hi_mtu                  /* max transfer unit  */
   int   hi_speed                /* relative CPU speed */
} taskp
```

tid
Integer where the host identifier of a particular host is returned (see the *Description* on page 89).

name
Character string where the *hostname* is returned (see the *Description* on page 89).

arch
Character string where host architecture is returned (it will be always RS6K).

mtu
Maximum transfer unit over the low-level communication protocol.

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 hostinfo *hosts;
  .....
if (!(rc = pvm_config (&nhosts, &narchs, &hosts))) {
  printf ("%d hosts are currently running\n", nhosts);
  printf ("   HTID    HNAME     HARCH     MTU    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_mtu, hosts->hi_speed);

    hosts++;
  }
}
```

FORTRAN Example

```
      integer nhosts, narchs, rc, i
      integer htid(MAX), mtu(MAX), speeds(MAX)
      character*32 names(MAX), arches(MAX)
          .....
      i=1
      print *, '   HTID     NAME     ARCH      MTU    SPEED'
66    call pvmfconfig (nhosts, narchs, htid(i), names(i), arches(i),
     $                 mtu(i), speed(i)
      print *, htid(i), name(i), arches(i), mtu(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

Description

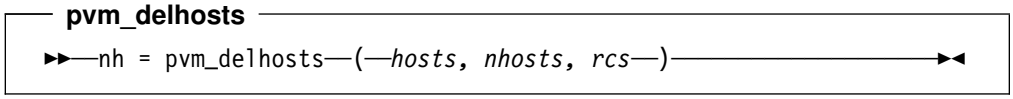
The C routine allows you to delete one or more hosts from the virtual machine. The FORTRAN routine allows you to delete a single host at each call. At any time, your processes can call the **pvm_config** or the **pvm_mstat** routines to get information about the virtual machine configuration.

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.

Notes:

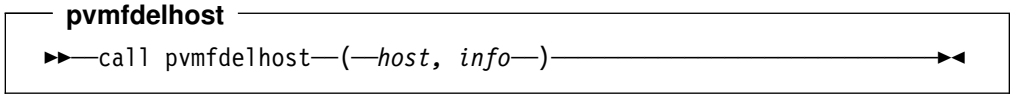
1. Special attention should be taken when using this routine because, before one or more hosts are deleted from the virtual machine configuration, all the IBM AIX PVMe processes running on them will be terminated.
2. It is an error to delete the master host the IBM AIX PVMe daemon is actually running on.

C Syntax




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

FORTRAN Syntax




```
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*.

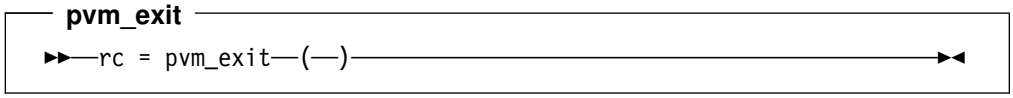
pvm_exit

Description

The **pvm_exit** routine informs the daemon that the process is leaving the IBM AIX PVMe program environment. It should be called by all the IBM AIX PVMe processes before they stop or exit.

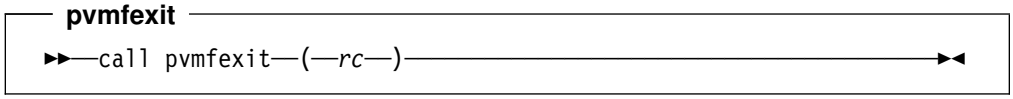
This routine does not kill the process, which can continue to run tasks just like any other serial UNIX process. Processes that leave the IBM AIX PVMe program may enroll again. Entries in the processes database are reused. If a process calls the **pvm_exit** routine and then enrolls again, (and no other 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

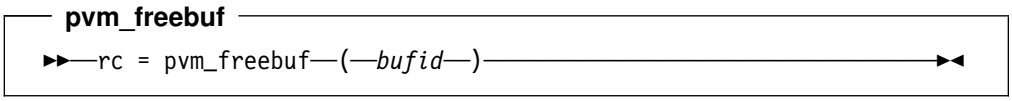
The **pvm_freebuf** routine unlocks a buffer and makes it available again to the IBM AIX PVMe routines. The buffer identifier must be supplied.

This routine should be used to free:

- Send buffers created by the **pvm_mkbuf** routine, and are no longer needed
- Send buffers created by the **pvm_initsend** routine and are no longer needed
- Buffers that were saved with the **pvm_setsbuf** and **pvm_setrbuf** routines

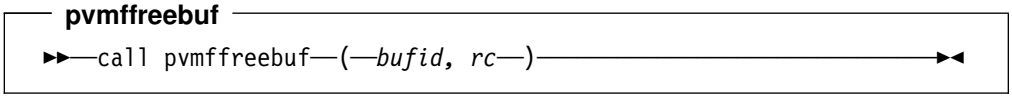
Most of your applications do not need to manage multiple buffers. If multiple buffers are required, the default send and receive buffers are automatically created by the IBM AIX PVMe routines.

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.
PvmNoSuchBuf	Invalid buffer identifier.

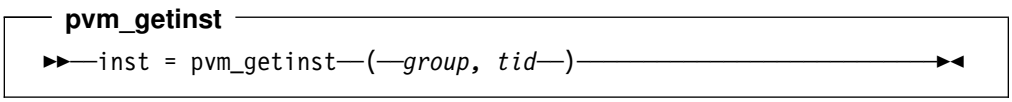
pvm_getinst

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*.

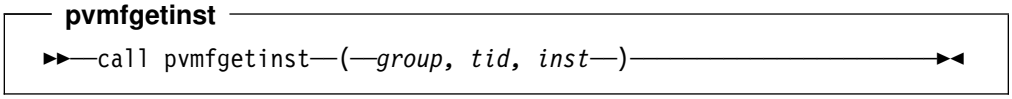
If the same process joins and leaves the same group multiple times, it is not guaranteed to receive the same instance number within the group. It receives the same instance number only if no other processes join or leave the same group in the meantime.

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 ("slaves", ltid, inst)
```

Error Conditions

PvmSysErr	System error. Process not enrolled.
PvmTooLong	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.

pvm_getopt

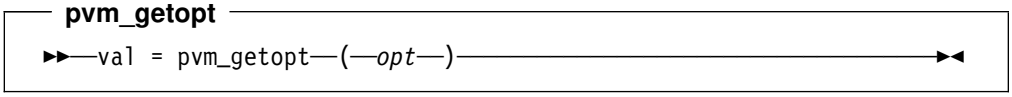
Description

The **pvm_getopt** routine allows you to obtain the current value of an option.

The **PvmRoute** option has been introduced only to maintain compatibility with the public domain PVM, where it allowed you to select direct communication links (TCP sockets) among tasks. The syntax is exactly the same, but in the IBM AIX PVMe program this option does not produce any effect, since in this implementation, data exchange among tasks always exploits direct process-to-process links.

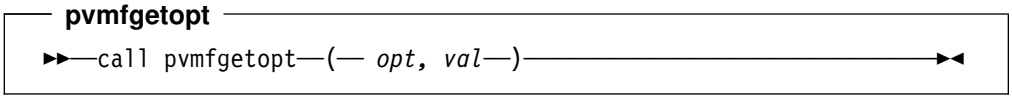
The **PvmAutoErr** option allows you to enable/disable automatic error printings whenever an error occurs in a IBM AIX PVMe routine. The effect is the same as when the **pvm_serror** routine is called.

C Syntax




```
int val;
int opt;
```

FORTRAN Syntax




```
integer val
integer opt
```

Parameters

opt
Integer identifying the option. The following options can be set:

- **PvmRoute**
- **PvmAutoErr**

val
Return code. A positive number identifying the current value of the option if the call was successful. 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*.

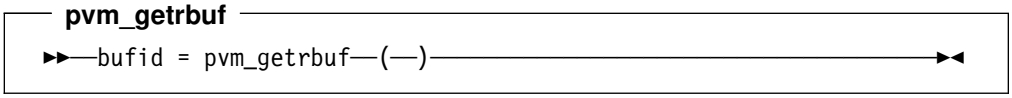
pvm_getrbuf

Description

The **pvm_getrbuf** routine returns the identifier of the current receive buffer. In the IBM AIX PVMe program, only one receive buffer is active at any time.

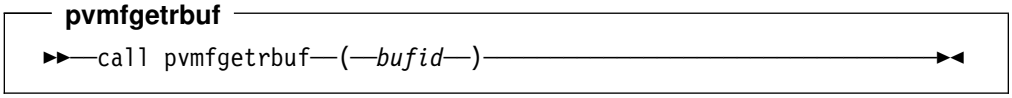
If there is no active buffer, the routine returns `0`. If a receive is attempted in this condition, the receiving routines set a new active buffer.

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

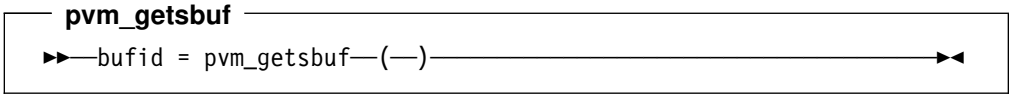
The **pvm_getsbuf** routine returns the identifier of the current send buffer.

In the IBM AIX PVMe 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.

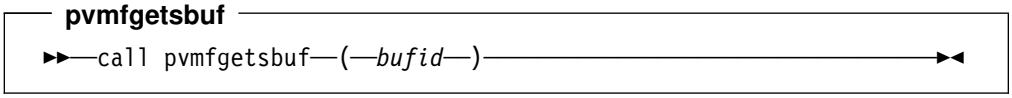
Note: If there is no active buffer, the routine returns `0`.

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

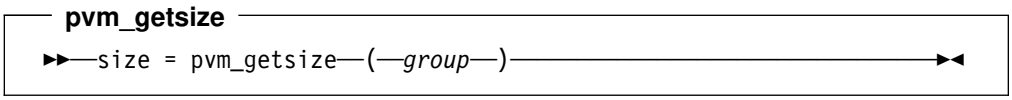
PvmSysErr	System error. Process not enrolled.

pvm_getsize

Description

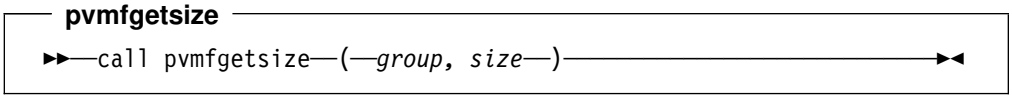
The **pvm_getsize** routine queries for the current number of members of the supplied group. Group size can change during the program execution.

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 ("slaves");
  ....
size = pvm_getsize ("slaves");
```

FORTRAN Example

```
integer size

call pvmfgetsize ("slaves", size)
```

Error Conditions

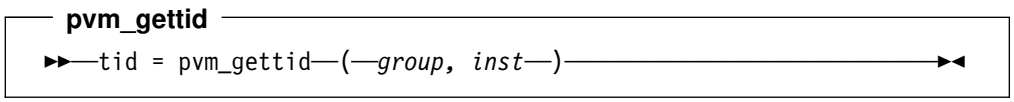
PvmSysErr	System error. Process not enrolled.
PvmTooLong	Group name longer than 32 characters.
PvmNullGroup	*group* name is NULL.
PvmNoGroup	Group named *group* does not exist.

pvm_gettid

Description

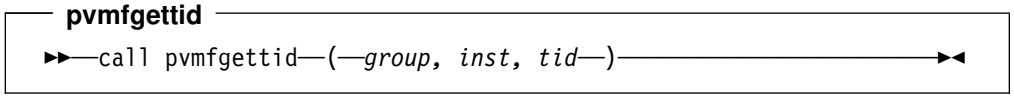
The **pvm_gettid** routine queries for the task identifier of the process that has the instance number *inst* within the group *group*.

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 ("slaves");
  ....
up_tid = pvm_gettid ("slaves", my_gtid+1);
if (up_tid > 0)
   pvm_send (up_tid, msgtype);
```

FORTRAN Example

```
integer ltid, rtid, ..

call pvmfgettid ("slaves", 0, tid)
```

Error Conditions

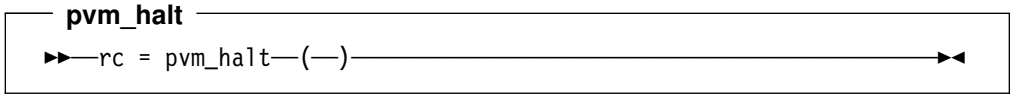
PvmSysErr	System error. Process not enrolled.
PvmTooLong	Group name longer than 32 characters.
PvmNullGroup	*group* name is NULL.
PvmNoGroup	Group named *group* does not exist.
PvmNoInst	Process identified with *inst* instance number does not exist within the supplied group.

pvm_halt

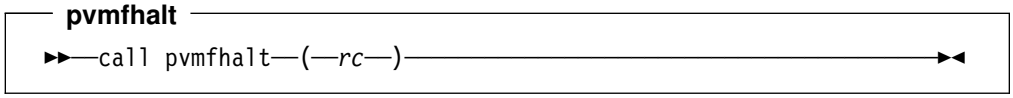
Description

The **pvm_halt** routine shuts the entire IBM AIX PVMe system down including the remote tasks, the IBM AIX PVMe 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_initsend

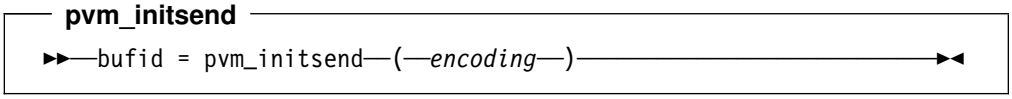
Description

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.

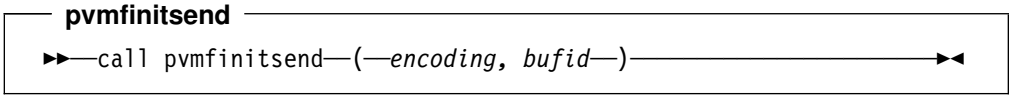
If the **pvm_initsend** routine is not called after a **pvm_send** routine, the result of successive packing operations is the append of new data to the previous message. If a **pvm_send** routine is called after a **pvm_initsend** routine, but before any pack operation, a zero-length message is sent.

The *encoding* parameter can assume three different values: two of them (**PvmDataRow** and **PvmDataDefault**) were used in public domain PVM to enable/disable data conversion to/from the XDR format. Both of them are supported in the IBM AIX PVMe 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. However, the same data can be sent multiple times, with modifications intervening between the sends, without having to pack it each time.

C Syntax



FORTRAN Syntax



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 (0);
```

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

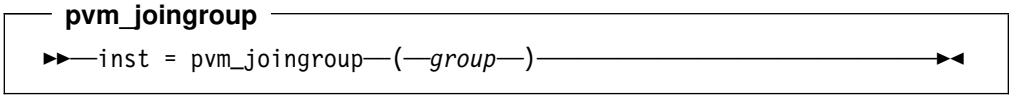
The **pvm_joingroup** routine enrolls the calling process to the supplied *group* and returns its instance number within the group. If the named *group* does not exist, the routine creates it.

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 AIX PVMe program reuses entries in the groups database when they are no longer needed.

Notes:

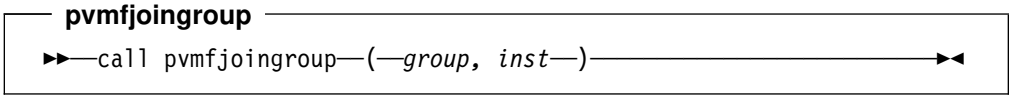
1. At any time, a process can belong to one group, multiple groups, or to no group.
2. The maximum group size is 128.
3. The maximum number of groups is 8.

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 (positive 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 ("slaves", gid)
if (gid.ge.0) then
  print *, 'Successfully joined group, gid=',gid
endif
```

Error Conditions

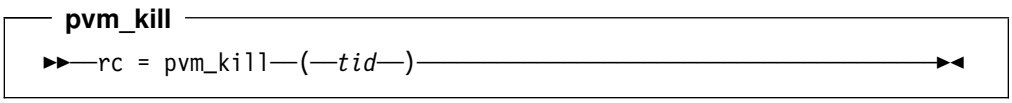
PvmSysErr	System error. Process not enrolled.
PvmTooLong	*group* longer than 32 characters.
PvmNulGroup	*group* equal to NULL.
PvmOutOfRes	*group* cannot have more than 128 members. No more than 8 groups can exist.
PvmDupGroup	The calling process is already member of *group*.

pvm_kill

Description

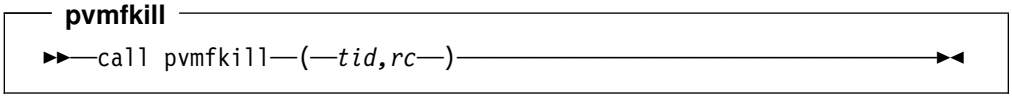
The **pvm_kill** routine kills the IBM AIX PVMe process identified by *tid*. The **pvm_kill** routine has not been designed to kill the calling process. To do that, call the **pvm_exit (pvmfexit)** routine followed by an **exit (STOP)**.

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

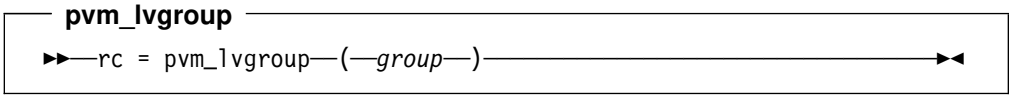
PvmSysErr	System error. Process not enrolled.
PvmBadParam	*tid* identifies the calling process. *tid* process does not exist.

pvm_lvgroup

Description

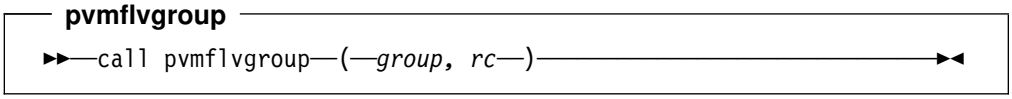
The **pvm_lvgroup** routine unenrolls the calling process from the supplied *group*. The process must already have called the **pvm_joingroup** routine to become a member of that group. Processes can join and leave groups at any time during the program execution, but they are not guaranteed to get the same instance number within each group if they leave and then enroll again. In fact, the IBM AIX PVMe 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 ("slaves", gid)
if (gid.ne.0) then
  print *, 'Error leaving group: all'
endif
```

Error Conditions

PvmSysErr	System error. Process not enrolled.
PvmTooLong	*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

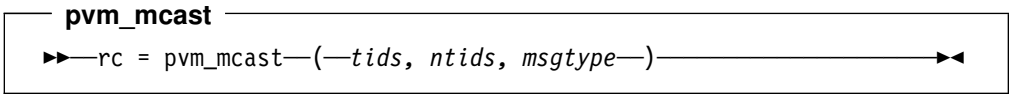
Description

The **pvm_mcast** routine sends a message to a set of IBM AIX PVMe 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: This list of recipient processes can also include the sending one.

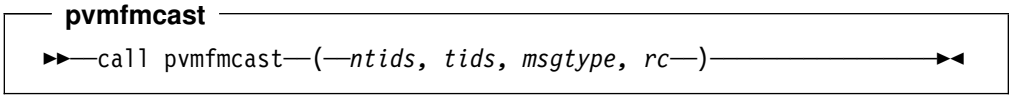
This routine is nonblocking. It returns when the message has been delivered to all recipient's tasks, even though the recipient tasks may not have called a **pvm_recv** or **pvm_nrecv** routine (recipient tasks interrupt their processing to store incoming messages in a pre-receiving area). However, the recipient processes must still exist as IBM AIX PVMe program processes. They can call either the **pvm_recv** or the **pvm_nrecv** routines to receive 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.
PvmNoBuf	No active send buffer.
PvmNoTask	Recipient process does not exist (not yet enrolled or already exited).

pvm_mkbuf

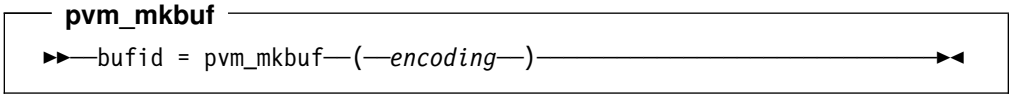
Description

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.

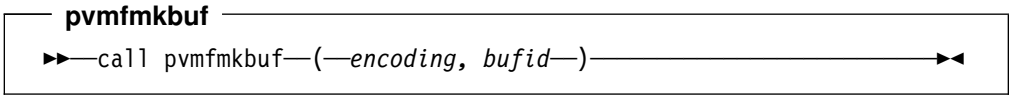
Receive buffers are automatically created by the **pvm_recv** and **pvm_nrecv** routines, and you are not required to free them, unless they have been explicitly saved (see “pvm_setrbuf” on page 152 for more information).

The *encoding* parameter can assume three different values: two of them (**PvmDataRow** and **PvmDataDefault**) were used in public domain PVM to enable/disable data conversion to/from the XDR format. Both of them are supported in the IBM AIX PVMe 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



FORTRAN Syntax



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(0);
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.

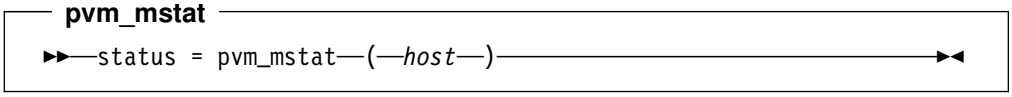
pvm_mstat

Description

The **pvm_mstat** routine returns the status of the supplied host. The first argument must be the name known by the IBM AIX PVMe daemon. If the host is properly running the return code is `0`.

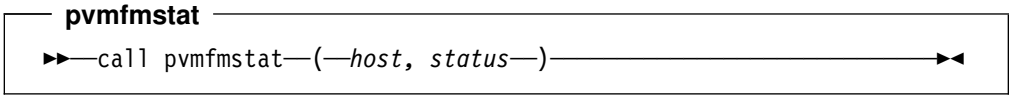
Note: When full names (for example, **hostA.domain**) are supplied to an IBM AIX PVMe routine, the domain name is skipped, and the host is registered with only the short name (**hostA**). This implies that the short alias must always be defined.

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= {'RISC600'};
int stat;
  .....
if (!(stat = pvm_pstat (host))
  printf ("Host %s is running\n", host);
```

FORTRAN Example

```
integer status;
  .....
call pvmfpstat ('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

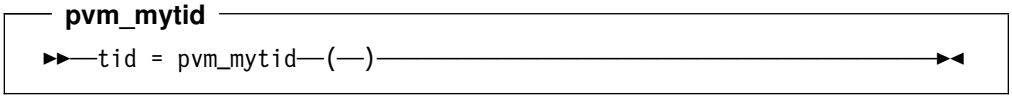
The **pvm_mytid** routine enrolls the calling process to the IBM AIX PVMe program on its first call, and returns the task identifier. On all successive calls, it simply returns the task identifier of the calling process.

Note: This routine should be the first PVM call of any IBM AIX PVMe program.

The *tid* is generated by the IBM AIX PVMe daemon during the first call to the **pvm_mytid** routine, or, if the process was started by the IBM AIX PVMe daemon, during the spawning of that process.

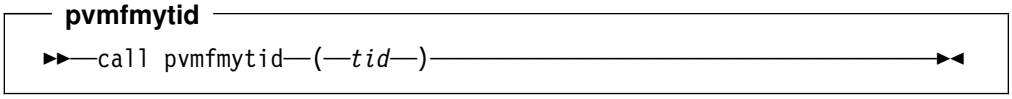
If a value less than `0` is returned, an error occurred. You should check to see if the IBM AIX PVMe program has been started on the proper host node.

C Syntax




```
int tid;
```

FORTRAN Syntax




```
integer tid
```

Parameters

tid
Address of an integer where the routine 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

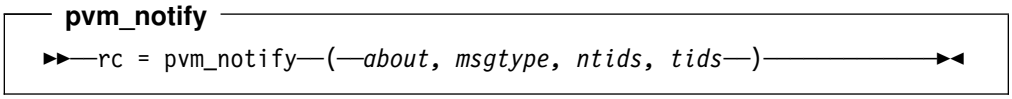
PvmSysErr	System error. The IBM AIX PVMe program is not active on the host node. Process not enrolled.

pvm_notify

Description

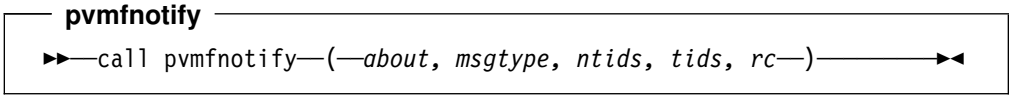
The **pvm_notify** routine informs the daemon that the calling process wants to be notified about certain events. The only supported event is **PvmTaskExit**. A list of task identifiers, their number and a particular message type must be supplied. The calling process will receive a message with the supplied 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 the public domain 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.

pvm_nrecv

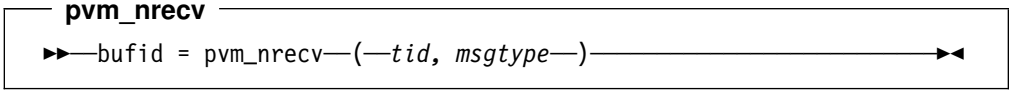
Description

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`.

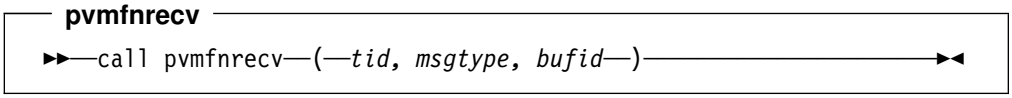
If the routine returns a positive number (the buffer identifier), data is available in the buffer and can be unpacked into your memory space by using the **pvm_upk***xxx* routines.

C Syntax




```
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 AIX PVMe 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.

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 is:

format null | 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:

- Format:
  - **+** 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: Variables *must* be passed as addresses.

C Syntax

pvm_packf

```
►►─rc = pvm_packf─(─fmt, ....─)──────────────────────────────►◄
```

```
const char *fmt;
int rc;
```

FORTRAN Syntax

Not Implemented

Parameters

fmt
Format expression like **printf** indicating data and data types to be packed.

info
Return code. It returns a `0` (zero) if the call was successful, or a negative number if unsuccessful.

C Example

```
double float LFLOAT_array[MAX];
int INT_array[MAX];
int count, rc;

rc = pvm_packf ("%+ %d %*d %4lf", PvmDataRow, count, count,
                INT_array, LFLOAT_array);
```

Error Conditions

PvmSysErr	System Error. Process not enrolled.
PvmNoMem	Data exceeded available memory in the send buffer.
PvmBadParam	*count* less than `0` (zero). *stride* less than `0` (zero). Invalid data type.
PvmNoBuf	No active send buffer.

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

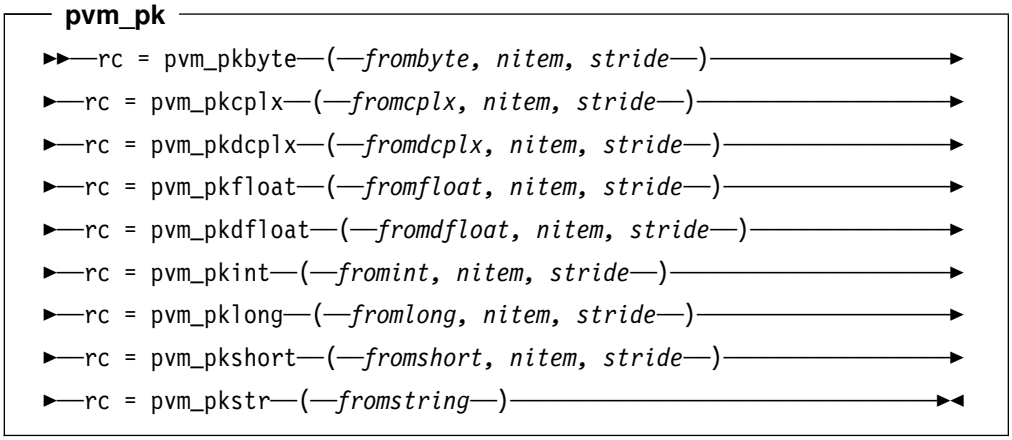
The only exception is the **pvm_pkstr** routine, that packs a character string and does not need either the *nitem* nor the *stride* parameters.

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.

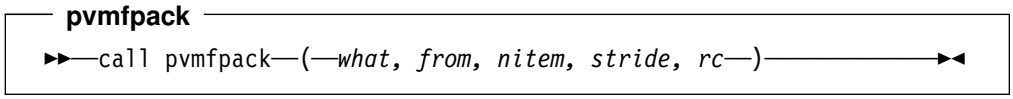
The maximum size for a single message is 32 megabytes. If more data is going to be packed into a buffer, the **PvmNoMem** error code is returned.

C Syntax




```
char *frombyte;
float *fromcplx;
double *fromdcplx;
float *fromfloat;
double *fromdfloat;
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

frombyte
fromcplx
fromdcplx
fromfloat
fromdfloat
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.

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

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.

info

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_pkdfloat(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

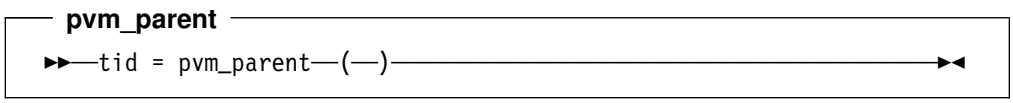
PvmSysErr	System Error. Process not enrolled.
PvmNoMem	Data exceeded available memory in the send buffer.
PvmBadParam	*count* less than `0` (zero). *stride* less than `0` (zero). Invalid data type.
PvmNoBuf	No active send buffer.

pvm_parent

Description

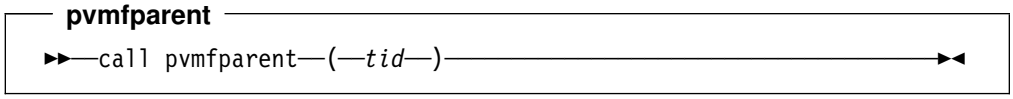
The **pvm_parent** routine can be used to determine the task identifier of the IBM AIX PVMe process that spawned the calling one. If the calling was not started by a **pvm_spawn** routine, the return code is **PvmNoParent**.

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 AIX PVMe process, then the *tid* will be set to **PvmNoParent**. **PvmNoParent** is also 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

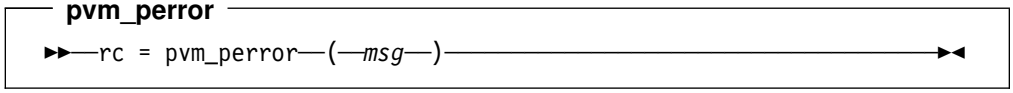
PvmSysErr	System Error. Process not enrolled.
PvmNoParent	The calling process was not started by a **pvm_spawn** routine.

pvm_perror

Description

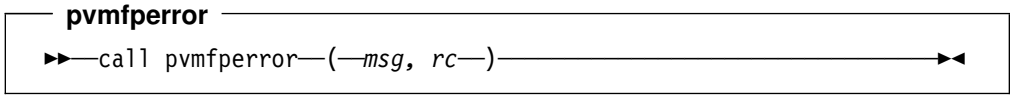
The **pvm_perror** routine prints a line on *stderr* containing a message supplied by you (if any), a colon (:) and an error string corresponding to the error code of the last error that occurred. The **pvm_perror** routine is intended for errors in the IBM AIX PVMe library routines.

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.

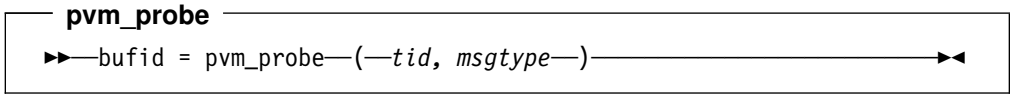
pvm_probe

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 AIX PVMe 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.

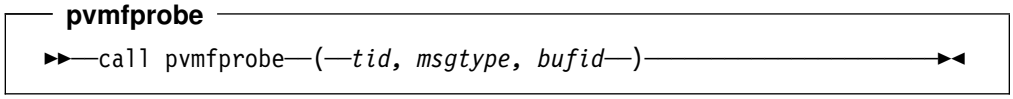
The **pvm_probe** routine may be invoked multiple times. Once it returns a valid *bufid*, a **pvm_recv** routine must be called before unpacking data.

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.

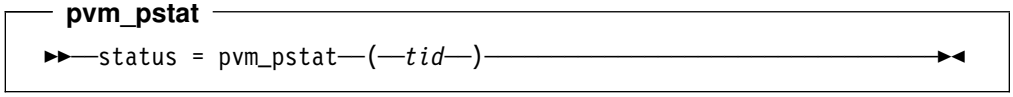
pvm_pstat

Description

The **pvm_pstat** routine returns the status of the specified process. It returns **PvmNoTask**, if the supplied process already left the IBM AIX PVMe program or exited. It returns a `0` if the process is properly running.

C Syntax

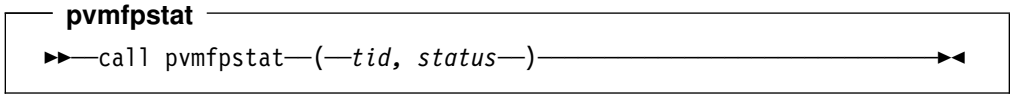
pvm_pstat




```
int tid;
int status;
```

FORTRAN Syntax

pvmfpstat




```
integer tid
integer status
```

Parameters

tid
Task identifier of the process requiring status.

status
Return code. It returns the process status, if the call was successful. It returns a negative value (different from `-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*, has arrived. 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

pvm_recv

```
►►—bufid = pvm_recv—(—tid, msgtype—)———————————————►◄
```

```
int tid;
int msgtype;
int bufid;
```

FORTRAN Syntax

pvmfrecv

```
►►—call pvmfrecv—(—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 AIX PVMe 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.

pvm_send

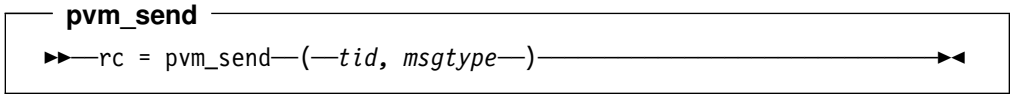
Description

The **pvm_send** routine sends a message to another IBM AIX PVMe 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.

This routine is nonblocking. 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 AIX PVMe process. It can either call the **pvm_recv** or the **pvm_nrecv** routines to receive the message.

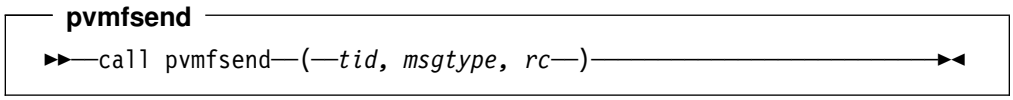
Note: The same buffer may be sent to multiple processes.

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.
PvmNoBuf	No active send buffer.
PvmNoTask	Recipient process does not exist (not yet enrolled or already exited).

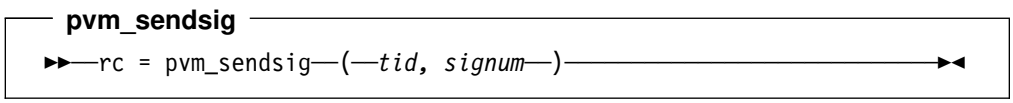
pvm_sendsig

Description

The **pvm_sendsig** routine sends the UNIX signal number (*signum*) to another IBM AIX PVMe process.

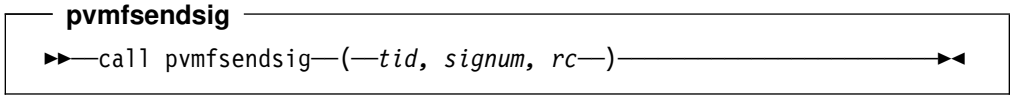
Note: This routine should only be used by programmers experienced with UNIX signal handling.

C Syntax




```
int tid;
int signum;
int rc;
```

FORTRAN Syntax




```
integer tid
integer signum
integer rc
```

Parameters

tid
Task identifier (>0) of the process the signal is to be sent to.

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

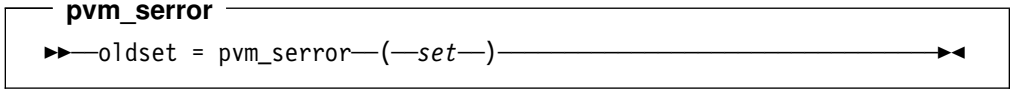
PvmSysErr	System error. Process not enrolled.
PvmBadParam	Invalid *signum*.

pvm_serror

Description

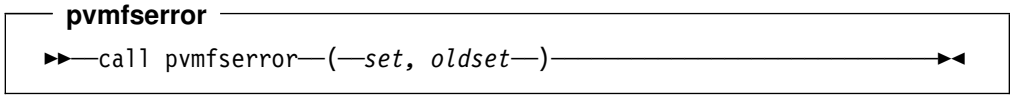
The **pvm_serror** routine enables or disables automatic error message printings on the *stderr*, whenever an error in a IBM AIX PVMe library routine occurs. You must specify the new setting value, while the old value is returned if the call is successful.

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 AIX PVMe processes.

The **PvmRoute** option has been introduced only to maintain compatibility with the public domain PVM, where it allowed you to select direct communication links (TCP sockets) among tasks. The syntax is exactly the same, but in the IBM AIX PVMe 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 a IBM AIX PVMe routine. The effect is the same as when the **pvm_serror** routine is called.

C Syntax

pvm_setopt

```
►►—oldval = pvm_setopt—(—opt, val—)———————————————————►◄
```

FORTRAN Syntax

pvmfsetopt

```
►►—call pvmfsetopt—(—opt, val, oldval—)———————————————►◄
```

Parameters

opt
Integer identifying the option. The following options can be set:

- **PvmRoute**
- **PvmAutoErr**

val
Value assigned to the supplied option. All the valid values for the options are listed:

PvmRoute	**PvmDontRoute** **PvmAllowDirect** **PvmRouteDirect**
PvmAutoErr	0 1

oldval
Return code. It returns a positive number identifying the previous value of the option if the call was successful. 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

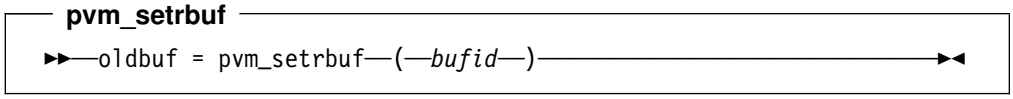
Description

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.

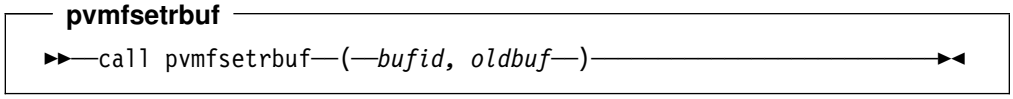
If the value of *bufid* is `0`, then the previous buffer is saved, and no other buffer becomes active. If a receive operation is attempted, the receive routine will set a new buffer to active.

Only values returned by previous **pvm_recv**, **pvm_nrecv**, **pvm_setrbuf** routines can be supplied. For example, the return value of a **pvm_mkbuf** routine is invalid.

C Syntax



FORTRAN Syntax



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	Invalid *bufid*.

pvm_setsbuf

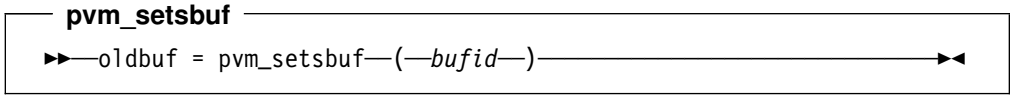
Description

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.

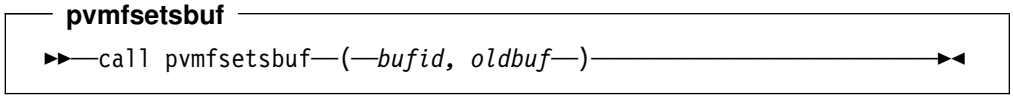
If the value of *bufid* is `0`, then the previous buffer is saved, and no other buffer becomes active. Then a new **pvm_setsbuf** or a new **pvm_initsend** routine must be called to set a new active buffer to send messages.

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



FORTRAN Syntax



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	Invalid *bufid*.

pvm_spawn

Description

The **pvm_spawn** routine starts *ntasks* copies of the *task* executable and enrolls them into the IBM AIX PVMe 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 19).

If the *flag* parameter does not contain the **PvmTaskHost** option, the IBM AIX PVMe 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.

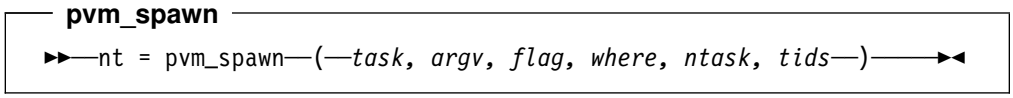
The return code of each spawn attempt is returned in the elements of the *tids* array. If it was successful, it is the task identifier of the new process. If it was not successful, it returns a negative number. The return code of the routine is the actual number of processes started, so that it will 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 AIX PVMe 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 AIX PVMe 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 17).

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* can be set to NULL.

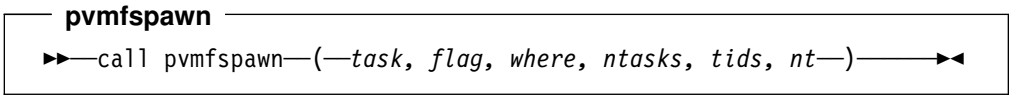
The IBM AIX PVMe daemon by itself passes a set of arguments to the spawned processes, to provide them with all the necessary information. These kinds of arguments all start with the **pvm_** substring, so that you can easily distinguish them from the other arguments. However, arguments supplied by you 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 AIX PVMe 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 the public domain PVM. It was used there to specify the architecture of the host where the new process was to be started. Because the IBM AIX PVMe 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. Valid values are: `'RS6K'`, `'.'`, `'*'`, `'NULL'`, an empty string.

ntask
Number of processes to be started.

tids
Points to an array of integers. 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[i]
  for (i=0; i<n;i++) {
    if (procs[i] <0 )
       printf ("error spawning task %d",i);
  }
```

FORTRAN Example

```
int procs(MAXPRC)

call pvmfspawn ('node', 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.
PvmTooLong	Process name too long.
PvmOutOfRes	An IBM AIX PVMe process is already running on the supplied host. There are no available hosts.

pvm_start_pvmd

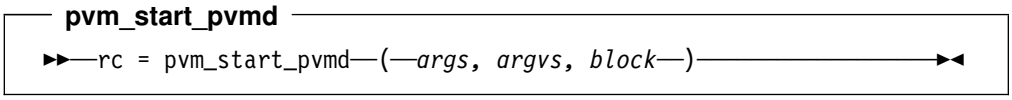
Description

The **pvm_start_pvmd** routine allows a process to start an IBM AIX PVMe session, that is, to start the IBM AIX PVMe 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 AIX PVMe routine immediately afterwards.

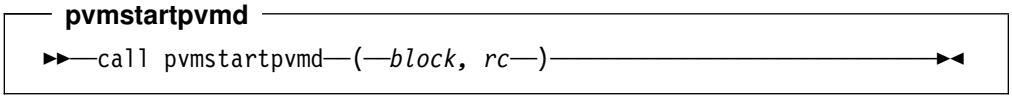
If an IBM AIX PVMe session is already active, no new session is started, and no error message is returned.

C Syntax




```
int args;
char **argvs;
int block;
int rc;
```

FORTRAN Syntax




```
integer block
integer rc
```

Parameters

args
Number of arguments that are passed to the daemon.

argvs
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.

pvm_tasks

Description

The **pvm_tasks** routine returns information about the IBM AIX PVMe 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

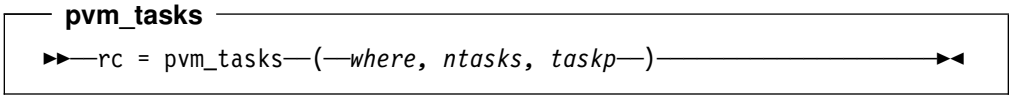
In the second argument (*ntasks*) the routine returns the number of processes you were interested in.

The C routine allocates memory space for a set of **taskinfo** 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.

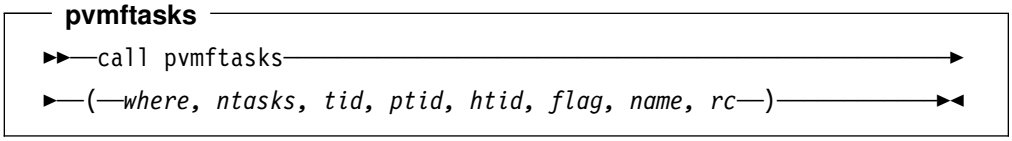
Data returned from this routine is the same as data shown by the **ps** command from the IBM AIX PVMe console.

C Syntax




```
int where;
int *ntasks;
struct taskinfo **taskp;
int rc;
```

FORTRAN Syntax




```
integer where
integer ntasks
integer tid
integer ptid
integer htid
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 AIX PVMe processes is returned.
pvmd *tid*	Information about the tasks running only on the host identified by the daemon *tid* 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 **taskinfo** structures. The array will be *ntasks* long. Each structure contains data about a single task, and has the following shape:

```
struct taskinfo {
   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 */
} taskp
```

tid

Integer where the process identifier of a particular task is returned (see the *Description* on page 161).

ptid

Integer where the identifier of the parent of the task *tid* is returned (see the *Description* on page 161).

htid

Integer where the identifier of the daemon on the host *tid* is running on is returned (see the *Description* on page 161)

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 taskinfo *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	*where* is an invalid task identifier.
PvmNoHost	*where* is an invalid host identifier.

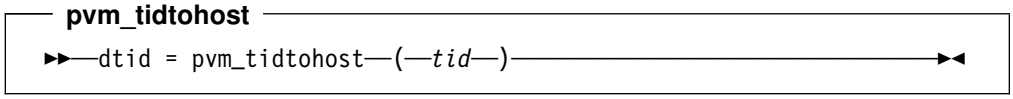
pvm_tidtohost

Description

The **pvm_tidtohost** routine returns the task identifier of the IBM AIX PVMe daemon on the supplied process it is running on. The identifiers for the daemon are always negative numbers (less than -256).

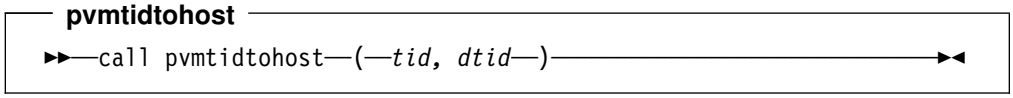
The routine properly masks the supplied *tid* to obtain the daemon *tid*, but does not check if the supplied process exists.

C Syntax




```
int tid;
int dtid;
```

FORTRAN Syntax




```
integer tid
integer dtid
```

Parameters

tid
Task identifier of a IBM AIX PVMe process.

dtid
Return code. Task identifier of the IBM AIX PVMe daemon on the host the supplied process is running on.

C Example

```
int dtid, tid;

dtid = pvm_tidtohost (tid);
```

FORTRAN Example

```
integer dtid, tid
call pvmftidtohost (tid, dtid);
```

Error Conditions

There are no error conditions.

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 null | 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

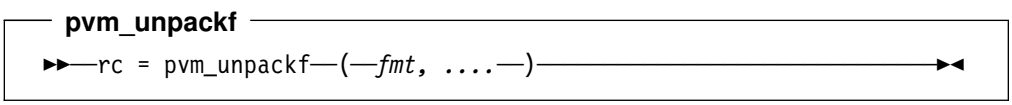
mchar 'h' | 'l' | 'u'

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




```
const char *fmt;
int rc;
```

FORTRAN Syntax

Not Implemented

Parameters

fmt
A format expression like **printf** indicating data and data types that will be unpacked.

info
Return code. It returns a `0` if the call was successful. It returns a negative number if it was unsuccessful.

C Example

```
double float LFLOAT_array[MAX];
int INT_array[MAX];
int count, rc;

rc = pvm_unpackf ("%*d %4lf", count, 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:

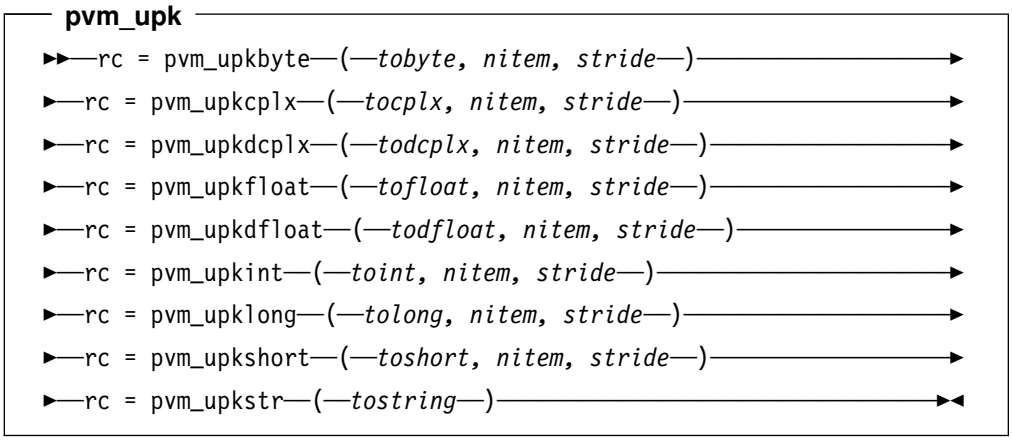
- A pointer to the array to be unpacked into
- The total number of items to be unpacked
- The stride to be used to unpack data

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.

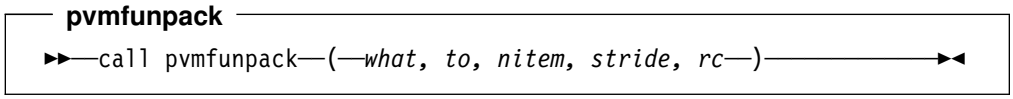
Note: A single message may contain different data types. Data must be unpacked exactly as it was packed to insure data integrity.

C Syntax




```
char *tobyte;
float *tocplx;
double *todcplx;
float *tofloat;
double *todfloat;
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

tobyte
todcplx
tofloat
todfloat
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, and the allocated memory region must contain at least *nitem*stride* items of that data type.

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

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.

info

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_upkdfloat(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). *stride* less than 0 (zero). Invalid data type.
PvmNoData	*count* greater than the actual message length.
PvmNoBuf	No active receive buffer.

pvm_upk

Appendix A. IBM AIX PVMe Error Code Summary

All IBM AIX PVMe 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: Three routines behave differently. The **pvm_spawn** (and the corresponding FORTRAN subroutine **pvmfspawn**), **pvm_addhosts** and **pvm_delhosts** routines return the actual number of tasks started, and hosts added and 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.

Analog 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 AIX PVMe Error Code Summary

Error Code	Value	Meaning
PvmOk	0	No error
PvmBadParam	-2	Bad parameter
PvmMismatch	-3	Barrier count mismatch
PvmTooLong	-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	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

Table 7 (Page 2 of 2). IBM AIX PVMe 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 slave pvmd
PvmAlready	-30	Slave pvmd already running
PvmNoTask	-31	Task does not exist
PvmNoEntry	-32	No such (group, instance)
PvmDupEntry	-33	(group, instance) already exists

Appendix B. IBM AIX PVMe Intrinsic Limits: Summary

There are a few limitations in the present release of the IBM AIX PVMe program product. Those that are of interest to you are shown in the following table.

Table 8. IBM AIX PVMe Intrinsic Limits Summary

Maximum length of a host name	64 characters
Maximum length of a barrier name	32 characters
Maximum number of distinct components	4
Maximum number of instances for a single component	64
Maximum number of hosts	64
Maximum size of a single message	32 megabytes
Maximum number of concurrent sessions on the same set of processors	4
pw option for the daemon	Not supported
Maximum number of consoles per host	1
pvm_recvf routine	Not implemented
PvmHostAdd and **PvmHostDelete** suboptions of the **pvm_notify** routine	Not Supported

There are also limitations for the number of outstanding messages (messages which are ready for a **pvm_rcv** or **pvm_nrcv** operation). The maximum number of outstanding messages for the three message classes supported by the IBM AIX PVMe program (the message size defines the class of the message) are provided in the following table:

Table 9. IBM AIX PVMe: Limits (Number and Size for the Outstanding 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 small messages ...
```

or

```
Too many outstanding big messages ...
```

When this message appears, the suggested action is to stop the application and reset the IBM AIX PVMe program product. 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>
```

© Copyright IBM Corp. 1993, 1994

A final limitation is that the 9076 SP2 system supports only one process per node when the High-Performance Switch is used. This limits the usage of the High-Performance Switch to a single process per node. This means that the number of nodes reserved for an IBM AIX PVMe session must be at least equal to the total number of the IBM AIX PVMe processes that will be concurrently active during that session.

Appendix C. IBM AIX PVMe Quick Reference

Use this appendix to quickly find out in what order you need to enter the arguments of a C or FORTRAN subroutine. For detailed information, see Chapter 6, “IBM AIX PVMe Reference: C and FORTRAN Subroutines” on page 77.

The following table lists the C subroutines:

Table 10 (Page 1 of 2). IBM AIX PVMe Quick Reference: C Routines			
int nh = pvm_addhosts (char **hosts, int nhosts, int* rcs)			
int rc = pvm_advise (int route)	route	**PvmDontRoute** **PvmAllowDirect** **PvmRouteDirect**	1 2 3
int rc = pvm_barrier (char *group, int count)			
int rc = pvm_bcast (char *group, int msgtype)			
int rc = pvm_bufinfo (int bufid, int *len, int *msgtype, int* tid)			
int rc = pvm_config (int *nhost, int *narch, struct hostinfo **hosts)			
int nh = pvm_delhosts (char **hosts, int nhosts, int *rcs)			
int rc = pvm_exit ()			
int rc = pvm_freebuf (int bufid)			
int inst = pvm_getinst (char *group, int tid)			
int val = pvm_getopt (int opt)			
int bufid = pvm_getrbuf ()			
int bufid = pvm_getsbuf ()			
int tid = pvm_gettid (char *group, int inst)			
int size = pvm_getsize (char *group)			
int rc = pvm_halt ()			
int bufid = pvm_initsend (int encoding)	encoding	**PvmDataDefault** **PvmDataRaw** **PvmDataInPlace**	0 1 2
int inst = pvm_joingroup (char *group)			
int rc = pvm_kill (int tid)			
int rc = pvm_lvgroup (char *group)			
int rc = pvm_mcast (int *tids, int ntids, int msgtype)			
int bufid = pvm_mkbuf (int encoding)			
int status = pvm_mstat (char *host)			
int tid = pvm_mytid ()			
int rc = pvm_notify (int about, int msgtype, int ntids, int *tids)	about	**PvmTaskExit**	1
int bufid = pvm_nrecv (int tid, int msgtype)			
int rc = pvm_packf (const char *fmt, ...)			

© Copyright IBM Corp. 1993, 1994

Table 10 (Page 2 of 2). IBM AIX PVMe Quick Reference: C Routines			
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)			
int tid = pvm_parent ()			
int rc = pvm_perror (char *msg)			
int bufid = pvm_probe (int tid, int msgtype)			
int status = pvm_pstat (int tid)			
int bufid = pvm_recv (int tid, int msgtype)			
int rc = pvm_send (int tid, int msgtype)			
int rc = pvm_sendsig (int tid, int signum)			
int oldset = pvm_serror (int set)			
int oldval = pvm_setopt (int opt, int val)			
int oldbuf = pvm_setrbuf (int bufid)			
int oldbuf = pvm_setsbuf (int bufid)			
int nt =pvm_spawn (char *task, char **argv, int flag, char *where, int ntask, int *tids)	flag	**PvmTaskDefault** 0 **PvmTaskHost** 1 **PvmTaskArch** 2 **PvmTaskDebug** 4	
int dtid = pvm_start_pvmd (int args, char **argvs, int block)			
int rc = pvm_tasks (int where, int *ntasks, struct taskinfo **taskp)			
int dtid = pvm_tidtohost (int tid)			
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)			

The following table lists the FORTRAN subroutines:

Table 11 (Page 1 of 2). IBM AIX PVMe Quick Reference: FORTRAN Routines			
call pvmfaddhost (host, rc)			
call pvmfadvise (route, rc)	route	**PVMDONTROUTE** **PVMALLOWDIRECT** **PVMROUTEDIRECT**	1 2 3
call pvmfbarrier (group, count, rc)			
call pvmfbcast (group, msgtype, rc)			
call pvmfbufinfo (bufid, len, msgtype, tid, rc)			
call pvmfconfig (nhost, narch, tid, name, arch, mtu, speed, rc)			
call pvmfdelhost (host, rc)			
call pvmfexit (info)			
call pvmffreebuf (bufid, rc)			
call pvmfgetinst (group, tid, inst)			
call pvmfgetopt (opt, val)			
call pvmfgetrbuf (bufid)			
call pvmfgetsbuf (bufid)			
call pvmfgettid (group, inst, tid)			
call pvmfgetsize (group, size)			
call pvmfhalt (rc)			
call pvmfinitsend (encoding, bufid)	encoding	**PVMDEFAULT** **PVMRAW** **PVMINPLACE**	0 1 2
call pvmfjoingroup (group, inst)			
call pvmfkill (tid, rc)			
call pvmflvgroup (group, rc)			
call pvmfmcast (ntids, tids, msgtype, rc)			
call pvmfmkbuf (encoding, bufid)			
call pvmfmstat (host, status)			
call pvmfmytid (tid)			
call pvmfnotify (about, msgtype, ntids, tids, rc)	about	**PVMTASKEXIT**	1
call pvmfnrecv (tid, msgtype, bufid)			
call pvmfpack (what, from, nitem, stride, rc)	what	**STRING** **BYTE1** **INTEGER2** **INTEGER4** **REAL4** **COMPLEX8** **REAL8** **COMPLEX16**	0 1 2 3 4 5 6 7
call pvmfparent (tid)			
call pvmfperror (msg, rc)			
call pvmfprobe (tid, msgtype, bufid)			

Table 11 (Page 2 of 2). IBM AIX PVMe Quick Reference: FORTRAN Routines			
call pvmfpstat (tid, status)			
call pvmfrecv (tid, msgtype, bufid)			
call pvmfsend (tid, msgtype, rc)			
call pvmfsendsig (tid, signum, rc)			
call pvmfserror (set, oldset)			
call pvmfsetopt (opt, val, oldval)			
call pvmfsetrbuf (bufid, oldbuf)			
call pvmfsetsbuf (bufid, oldbuf)			
call pvmfspawn (task, flag, where, ntask, tids, nt)	flag	**PVMDEFAULT** **PVMHOST** **PVMARCH** **PVMDEBUG**	0 1 2 4
call pvmfstartpvmd (block, rc)			
call pvmftasks (where, ntasks, tid, ptid, htid, flag, name, rc)			
call pvmftidtohost (tid, dtid)			
call pvmfunpack (what, to, nitem, stride, rc)			

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

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 processors 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 processor 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 processor 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 processors each accessing only its own private memory. 9076 SP2 systems, as well as a cluster of workstations, belong to this class.

dynamic load balancing. Allows you decide at run time which processor node should run the specified parallel task.

E

enroll. In IBM AIX PVMe terminology, it is the registration of a process to the IBM AIX PVMe 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 processors used by the IBM AIX PVMe program.

hot spot. These are the parts of a program that take the most time to run.

I

instance number. Process identifier. In IBM AIX PVMe 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 processors 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 AIX PVMe 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 processors, 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 processors 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 processor nodes in a predetermined and ordered way.

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

-c option, for resetting your PVM session 28
-p option, for selecting the pool_id 29
-q option, for querying the status of the PVM session 27
-r option, for resetting all the PVM sessions 28

A

accessing storage devices for read/write operations 38
adding hosts 39
AIX, definition 180
application program, definition 180
automatic printing of error messages 53

B

bandwidth
  definition 180
  how it affects performance 4
barrier routine 58
barrier routine, definition 180
batch mode
  running an IBM AIX PVMe session 22
bit rate, definition 180
blocking routine, definition 180
broadcast operation 58, 61
broadcast operation, definition 180
buffer, definition 180
buffers, multiple 48

C

C example 60
checking the status of a target process 62
cluster, definition 180
commands
  dbx 64
  rcmd 36
  used for interactive starting and killing of PVM processes 26
communication
  for sharing data 3
  overhead 8—11
  tuning 13
communication overhead, definition 180
communication protocol, definition 180
comparing a broadcast operation 61
comparing a snd operation 61
compiling and linking a parallel program 62
components of cooperation 3
computation-to-communication ratio, definition 180
computational independence, definition 180
concepts, parallel programming 2
configuration files 16
control data, definition 180
cooperation
  components of 3
  concepts
    communication, for sharing data 3
    message passing 3
    synchronization 3
  for task execution 3
coupling scheme of the processor 3
creating directory structure 16

D

daemon, definition 180
data communication
  definition 180
  examples 49
  preparing a message 45
  primitives used for 44
  probe for message arrival 48
  receiving a message 47
  sending a message 46
data parallelism, definition 180
data type, definition 180
dbx command 64
debugging a parallel program 64
  finding errors in program logic 64
decomposing a serial program 6—7
decomposition, definition 180
default, definition 180
definitions
  AIX 180
  application program 180
  bandwidth 180
  barrier routine 180
  bit rate 180
  blocking routine 180
  broadcast operation 180
  buffer 180
  cluster 180
  communication overhead 180
  communication protocol 180
  computation-to-communication ratio 180
  computational independence ratio 180
  control data 180
  daemon 180
  data communication 180
  data parallelism 180

© Copyright IBM Corp. 1993, 1994

definitions *(continued)*
data type 180
decomposition 180
default 180
distributed memory systems 180
dynamic load balancing 180
enroll 180
fork-exec mechanism 181
functional parallelism 181
granularity 181
group management 181
hostlist file 181
hot spot 181
instance number 181
latency 181
load balancing 181
local host 181
low-level communication protocol 181
master host 181
menu 181
message passing 181
MPMD 181
nonblocking routines 181
parallel overhead 181
parallel processing 181
parallelize 181
parameter 181
PID 181
primitive 181
process control 181
PVM 181
server 182
shared memory segments 182
shell 182
SMIT 182
speed-up 182
SPMD 182
static load balancing 182
synchronization 182
task 182
task identifier 182
TCP 182
TCP/IP 182
workstation 182
deleting hosts 39
description of syntax diagrams 77
design considerations 8
determining computational independence 5
differences between PVM and IBM AIX PVMe 2
differences between PVM and the IBM AIX PVMe program product 61
differences between tightly and loosely coupled systems 3
directory structure, creating 16
distributed memory systems, definition 180
dynamic load balancing 12
dynamic load balancing, definition 180

E

enroll, definition 180
enrolling a process 35
error code summary 171, 172
error handling, primitives 52
examples
C 60
data communication 49
for managing groups 59
FORTRAN 59, 65
managing error conditions 53
managing hosts 41
managing tasks 33
signaling 55
executable files 37
executing tasks using cooperation 3
explanation, of subroutines 77

F

features and routines 31
finding errors in program logic 64
fork-exec mechanism, definition 181
format, explanation of subroutines 77
FORTRAN
example 59, 65
interface usage 77
functional parallelism, definition 181

G

getting started with IBM AIX PVMe 1
gprof command, using 8
granularity 11
granularity, definition 181
group management, definition 181
groups
examples 59
information on status 58
joining and leaving 58
managing 57
primitives 57

H

High-Performance Switch feature 3
hostlist file 27
hostlist file, definition 181
hosts
adding 39
deleting 39
example, managing hosts 41
information 40

hosts *(continued)*
managing 38, 57
primitives, using 39
querying information 40, 58
hot spot, definition 181
hot-spot analysis 8

I

I/O for IBM AIX PVMe processes 38
IBM AIX PVMe
bandwidth 4
configuration 19
customizing your installation 16
daemon, accepting options 27
determining computational independence 5
differences between PVM and IBM AIX PVMe 2, 61
error code summary 171, 172
High-Performance Switch feature 3
I/O 38
installation overview 15, 16
intrinsic limits summary 173
latency 4
managing tasks
examples 41
managing error conditions 53
using primitives 32
options 21
overview of features and routines 31
overview, getting started 1
parallel programming concepts 2
product startup 19—30
quick reference 175—178
resolving startup problems 29
starting
new tasks 62
running an IBM AIX PVMe session 19
testing 30
using
cooperation for executing tasks 3
the console 26
the program product 31
using the program product 75
writing parallel programs 5
information on tasks 37
installation
configuration files 16
creating directory structure 16
customizing your installation 16
IBM AIX PVMe software 15
overview 16
requirements 15
instance number, definition 181
interconnection network 3

J

joining groups 58

L

latency, definition 181
latency, how it affects performance 4
leaving groups 58
linking and compiling a parallel program 62
load balancing 11
load balancing, definition 181
local host, definition 181
loosely coupled system 3
low-level communication protocol, definition 181

M

managing
error conditions 53
example, hosts 41
group examples 57, 59
hosts 38
multiple buffers 48
primitives 57
tasks 32
master host, definition 181
memory, private 3
menu, definition 181
message
definition 181
passing 3
preparing 45
probe for arrival 48
receiving 47
sending 46
MPMD, definition 181
multiple buffers 48
multiple program, multiple data (MPMD) 1, 4

N

network, interconnection 3
new tasks, starting 62
nodes, processor 3
nonblocking routines, definition 181

O

optimizing the use of system functions 13

P

packing routine, using 46
parallel overhead, definition 181
parallel processing, definition 181

parallel programming
  communication overhead 8—11
  compiling and linking 62
  concepts 2
  debugging a parallel program 64
  decomposing a serial program 6—7
  design considerations 8
  granularity, fine and coarse 11
  load balancing 11
  paradigms 3
  performance considerations 8
  running a parallel program 63
  tuning 12
  understanding the differences between serial and parallel programs 2
  writing parallel programs 5
parallelization, tuning 13
parallelize, definition 181
parameter, definition 181
performance considerations 8
PID, definition 181
preparing a message 45
primitive, definition 181
primitives
  for error handling 52
  for joining and leaving groups 58
  for managing groups 57
  for managing tasks 32
  for signaling 55
  for the barrier routine 58
  for the broadcast operation 58
printing, error messages 53
private memory 3
probe for message arrival 48
process
  enrolling 35
  I/O 38
  starting 36, 37
  terminating 37
  unenrolling 36
process control, definition 181
processor nodes, synchronization of 3
prof command, using 8
profiling a serial program 8
programming models
  MPMD 1, 4
  SPMD 1, 4
pvm_addhosts routine 79
pvm_advise routine 81
pvm_barrier routine 83
pvm_bcast routine 85
pvm_bufinfo routine 87
pvm_config routine 89
pvm_delhosts routine 92
pvm_exit routine 94
pvm_freebuf routine 95
pvm_getinst routine 97
pvm_getopt routine 99
pvm_getrbuf routine 101
pvm_getsbuf routine 103
pvm_getsize routine 105
pvm_gettid routine 106
pvm_halt routine 108
pvm_initsend routine 109
pvm_joingroup routine 111
pvm_kill routine 113
pvm_lvgroup routine 115
pvm_mcast routine 117
pvm_mkbuf routine 119
pvm_mstat routine 121
pvm_mytid routine 123
pvm_notify routine 125
pvm_nrecv routine 127
pvm_packf routine 129
pvm_parent routine 134
pvm_perror routine 136
pvm_pk routines 131
pvm_probe routine 138
pvm_pstat routine 140
pvm_recv routine 142
pvm_send routine 144
pvm_sendsig routine 146
pvm_serror routine 148
pvm_setopt routine 150
pvm_setrbuf routine 152
pvm_setsbuf routine 154
pvm_spawn routine 156
pvm_start_pvmd routine 159
pvm_tasks routine 161
pvm_tidtohost routine 164
pvm_unpackf routine 165
pvm_upk routines 167
PVM, definition 181

Q

querying an IBM AIX PVMe session, using the -q option 27
quick reference 175—178

R

rcmd command 36
reading syntax diagrams 77
receiving a message 47
requirements, installation 15
resetting an IBM AIX PVMe session, using the -c and -r options 27
resolving startup problems 29
running
  a parallel program 63

running *(continued)*
an IBM AIX PVMe batch session 22

S

searching for executable files 37
sending a message 46
serial program 2
decomposing 6—7
profiling 8
server, definition 182
session
batch 22—26
interactive 19—22
shared memory segments, definition 182
shell, definition 182
signaling
examples 55
primitives 55
routines 55
single program, multiple data (SPMD) 1, 4
SMIT, definition 182
software, required for installation 15
speed-up, definition 182
SPMD, definition 182
starting
a process 36, 37
and killing processes, using commands 26
new tasks 62
the virtual machine 39
startup
IBM AIX PVMe batch session 22
information 19
options used to query and reset a session 27
resolving problems 29
running an IBM AIX PVMe batch session 22
testing the IBM AIX PVMe program product 30
understanding IBM AIX PVMe options 21
using the console 26
static load balancing 12
static load balancing, definition 182
status, of a target process 62
stopping the virtual machine 39
subroutines, C and FORTRAN
pvm_addhosts 79
pvm_advise 81
pvm_barrier 83
pvm_bcast 85
pvm_bufinfo 87
pvm_config 89
pvm_delhosts 92
pvm_exit 94
pvm_freebuf 95
pvm_getinst 97
pvm_getrbuf 101
pvm_getsbuf 103
subroutines, C and FORTRAN *(continued)*
pvm_getsize 105
pvm_gettid 106
pvm_halt 108
pvm_initsend 109
pvm_joingroup 111
pvm_kill 113
pvm_lvgroup 115
pvm_mcast 117
pvm_mkbuf 119
pvm_mstat 121
pvm_mytid 123
pvm_notify 125
pvm_nrecv 127
pvm_packf 129
pvm_parent 134
pvm_perror 136
pvm_pk 131
pvm_probe 138
pvm_pstat 140
pvm_recv 142
pvm_send 144
pvm_sendsig 146
pvm_serror 148
pvm_setopt 99, 150
pvm_setrbuf 152
pvm_setsbuf 154
pvm_spawn 156
pvm_start_pvmd 159
pvm_tasks 161
pvm_tidtohost 164
pvm_unpackf 165
pvm_upk 167
summary, error code 171, 172
summary, intrinsic limits 173
Switch, High-Performance 3
synchronization, definition 182
syntax diagrams, description of 77
system functions, optimizing the use of 13

T

target process, checking status 62
task identifier, definition 182
task, definition 182
tasks
information 37
managing 32
using primitives to manage tasks 32
TCP, definition 182
TCP/IP, definition 182
terminating a process 37
testing the IBM AIX PVMe program
using the hostbenc and nodebenc programs 30
using the include and makefile programs 30

tightly coupled system 3
tuning
  a parallel application 12
  communication 13
  parallelization 13

U

understanding IBM AIX PVMe options 21
unenrolling a process 36
unpacking data 47
using
  a packing routine 46
  the -c option 27, 28
  the -p option 29
  the -q option 27
  the -r option 28
  the IBM AIX PVMe console 26
  the IBM AIX PVMe program product 31

V

virtual machine
  dynamic configuration 39
  starting 39
  stopping 39

W

what is parallel processing 2
workstation, definition 182
writing parallel programs
  communication overhead 8—11
  decomposing a serial program 6—7
  design and performance considerations 8
  determining computational independence 5
  granularity, fine and coarse 11
  load balancing 11
  optimizing the use of system functions 13
  overview 5
  parallelization 13
  profiling a serial program 8

Readers' Comments — We'd Like to Hear from You

IBM AIX PVMe
User's Guide and
Subroutine Reference
Release 3.0

Publication No. SH23-0019-02

	Very Satis-fied	Satis-fied	Neutral	Dissatisfied	Very Dissatisfied
Overall satisfaction	☐	☐	☐	☐	☐

How satisfied are you that the information in this book is:

	Very Satis-fied	Satis-fied	Neutral	Dissatisfied	Very Dissatisfied
Accurate	☐	☐	☐	☐	☐
Complete	☐	☐	☐	☐	☐
Easy to find	☐	☐	☐	☐	☐
Easy to understand	☐	☐	☐	☐	☐
Well organized	☐	☐	☐	☐	☐
Applicable to your tasks	☐	☐	☐	☐	☐

Please tell us how we can improve this book:

Thank you for your responses. May we contact you? ☐ Yes ☐ No

Use this form to tell us what you think about this manual. If you prefer, you may send comments electronically to your IBM branch office, **Attention: Dept. 52QA-911, Kingston, NY**. Or you may fax your comments to us directly at: **FAX (914) 385-0662**.

To request additional publications, or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer.

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

Name _____ Address _____

Company or Organization _____

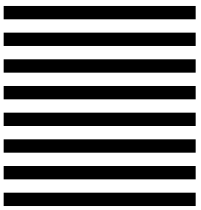
Phone No. _____

Readers' Comments — We'd Like to Hear from You
SH23-0019-02

Cut or Fold
Along Line

Fold and Tape | **Please do not staple** | Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES



BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Department 52QA, Mail Station 911
1 Enterprise Drive
Kingston, NY 12401-1099

Fold and Tape | **Please do not staple** | Fold and Tape

Cut or Fold
Along Line

Printed in U.S.A.

SH23-0019-02