Message Passing Toolkit: PVM Programmer's Manual

007–3686–003

CONTRIBUTORS

Written by Julie Boney

Edited by Susan Wilkening

Illustrations by Chris Wengelski

Production by Susan Gorski

PVM (Parallel Virtual Machine) is based on software that was developed by the Oak Ridge National Laboratory, the University of Tennessee, and Emory University. This work was supported in part by the Applied Mathematical Sciences subprogram of the Office of Energy research, U.S. Department of Energy, in part by the National Science Foundation, and in part by the State of Tennessee.

New Features in This Manual

This revision of the *Message Passing Toolkit: PVM Programmer's Manual*, supports the 1.4 release of the Message Passing Toolkit for IRIX (MPT). The MPT implementation of PVM for IRIX systems contained in this release is based on the Oak Ridge National Laboratories (ORNL) version 3.3.10.

Record of Revision

Version	Description
1.0	January 1996 Original Printing.
1.1	August 1996 This revision supports the Message Passing Toolkit (MPT) 1.1 release.
1.2	January 1998 This revision supports the Message Passing Toolkit (MPT) 1.2 release for UNICOS, UNICOS/mk, and IRIX systems.
1.3	February 1999 This revision supports the Message Passing Toolkit (MPT) 1.3 release for UNICOS, UNICOS/mk, and IRIX systems.
003	February 2000 This revision supports the Message Passing Toolkit (MPT) 1.4 release for IRIX systems.

Contents

Tables

Examples

About This Manual

This publication documents the Message Passing Toolkit for IRIX (MPT) 1.4 implementation of PVM-3 supported on SGI MIPS based systems running IRIX release 6.5 or later.

This implementation of PVM-3 is based on the public domain PVM product, version 3.3.10, developed by researchers at the Oak Ridge National Laboratory (ORNL), the University of Tennessee (UT), and Emory University (EU). It consists of a PVM library and several commands that support PVM.

Related Publications

The following documents contain additional information that might be helpful:

- *Message Passing Toolkit: MPI Programmer's Manual*
- *NQE User's Guide*
- *NQE Administration*
- *Application Programmer's Library Reference Manual*
- *Installing Programming Environment Products*

All of these publications can be ordered from the Minnesota Distribution Center. For ordering information, see "Obtaining Publications."

Other Sources

Material about PVM is available from the following other sources:

- *PVM: Parallel Virtual Machine: A User's Guide and Tutorial for Networked Parallel Computing*, available at the following URL:

  `http://www.netlib.org/pvm3/book/pvm-book.html`

- Usenet news group at `comp.parallel.pvm`

- PVM standard, available from the Computer Science and Mathematics Division of Oak Ridge National Laboratories.
- PVM related web pages from the following PVM home page:

  `http://www.epm.ornl.gov/pvm`

Obtaining Publications

To obtain SGI documentation, go to the SGI Technical Publications Library at `http://techpubs.sgi.com`.

Conventions

The following conventions are used throughout this document:

Convention	Meaning
`command`	This fixed-space font denotes literal items such as commands, files, routines, path names, signals, messages, and programming language structures.
`manpage`(*x*)	Man page section identifiers appear in parentheses after man page names. The following list describes the identifiers:

1	User commands
1B	User commands ported from BSD
2	System calls
3	Library routines, macros, and opdefs
4	Devices (special files)
4P	Protocols
5	File formats
7	Miscellaneous topics
7D	DWB-related information

	8 Administrator commands Some internal routines (for example, the `_assign_asgcmd_info`() routine) do not have man pages associated with them.
variable	Italic typeface denotes variable entries and words or concepts being defined.
`user input`	This bold, fixed-space font denotes literal items that the user enters in interactive sessions. Output is shown in nonbold, fixed-space font.
[ ]	Brackets enclose optional portions of a command or directive line.
...	Ellipses indicate that a preceding element can be repeated.

SGI systems include all MIPS based systems running IRIX 6.5 or later.

Reader Comments

If you have comments about the technical accuracy, content, or organization of this document, please tell us. Be sure to include the title and document number of the manual with your comments. (Online, the document number is located in the front matter of the manual. In printed manuals, the document number can be found on the back cover.)

You can contact us in any of the following ways:

- Send e-mail to the following address:

  `techpubs@sgi.com`

- Use the Feedback option on the Technical Publications Library World Wide Web page:

  `http://techpubs.sgi.com`

- Contact your customer service representative and ask that an incident be filed in the SGI incident tracking system.

- Send mail to the following address:

Technical Publications
SGI
1600 Amphitheatre Pkwy., M/S 535
Mountain View, California 94043–1351

- Send a fax to the attention of “Technical Publications” at +1 650 932 0801.

We value your comments and will respond to them promptly.

Overview

The Message Passing Toolkit for IRIX (MPT) is a software package that supports interprocess data exchange for applications that use concurrent, cooperating processes on a single host or on multiple hosts. Data exchange is done through *message passing*, which is the use of library calls to request data delivery from one process to another or between groups of processes.

The MPT 1.4 package contains the following components and the appropriate accompanying documentation:

- Parallel Virtual Machine (PVM)
- Message Passing Interface (MPI)
- Logically shared, distributed memory (SHMEM) data-passing routines

The Parallel Virtual Machine (PVM) software was initially developed to enable a collection of heterogeneous computer systems to be used as a coherent and flexible concurrent computation resource. SGI has taken this initial implementation and extended it in several ways.

This chapter provides an overview of the PVM software that is included in the toolkit.

The PVM Package

This manual contains instructions for building, installing, and using the MPT implementation of PVM-3 on IRIX systems. It consists of a PVM library and several commands that support PVM. The most important of these is a user-level daemon that runs on each computer system in the PVM system.

The MPT version of PVM has enhancements to use POSIX shared memory, which provides greater flexibility and robustness than did the previously used IRIX shared arenas.

Default communication is based on TCP sockets between processes on the same system and between different systems. Transfer speeds are relatively slow when sockets are used as the mechanism for communication. The MPT version of PVM also provides alternative mechanisms for communication. The socket communication has been optimized to utilize high-speed network devices more effectively. The different communication mechanisms are discussed further in the PVM man pages, and the

communication costs (in time, resources, and so on) associated with the different communication mechanisms are discussed in Chapter 2, "PVM Functionality", page 5.

PVM has been integrated with the Network Queuing Environment (NQE) so that you can use PVM within a batch job in isolation from other PVM jobs. For more information about NQE, see the *NQE User's Guide*, and *NQE Administration*.

PVM on SGI Systems

As described in this manual, SGI provides versions of PVM to support a variety of needs. These versions provide users with a single subroutine interface for message passing programming; this interface is portable and a de facto standard. PVM is available from its developers as public domain software and is being made available as vendor-supported software by SGI and a number of other computer vendors. By using PVM in your application, you can avoid being locked into a proprietary interface.

PVM is supported on all SGI systems. The PVM software system consists of a library and commands that support PVM. The PVM software provided by SGI has been developed specifically for each system on which it runs.

You can choose to use PVM to communicate among processes on a number of different computer systems. The following characteristics apply to all PVM system combinations:

- The user building an executable file for use on an SGI system links with a single PVM library, regardless of how PVM is used.
- The same standard library syntax and behavior are supported, regardless of how PVM is used (although certain releases may support features not appropriate to other releases).
- The performance of PVM in different basic scenarios differs significantly; this difference influences the communications strategy that should be used.

PVM Terminology

The following PVM terminology is used in this manual:

Term	Definition
task	The UNIX process that uses PVM for communications.
application	A number of tasks running the same program.
process	The entity running on the IRIX operating system or another UNIX system.

PVM Functionality

This chapter describes the Message Passing Toolkit (MPT) implementation of the Parallel Virtual Machine (PVM) software. The following concepts are discussed:

- Multiple computer systems as a virtual machine
- Applications and environments
- PVM program development
- Data types
- Environment variables

Multiple Computer Systems As a Virtual Machine

PVM is a software system that enables a collection of heterogeneous computer systems to be used as a coherent and flexible concurrent computation resource. The individual systems can be shared-memory or local-memory multiprocessors, vector supercomputers, specialized graphics engines, or scalar workstations interconnected by a variety of networks. From the user's point of view, the combination of these different systems can be treated as a single *virtual machine* when using PVM. The term *host* refers to one of the member computer systems.

PVM support software executes on each system in a user-configurable pool and presents a unified, general, and powerful computational environment for concurrent applications. User programs, written in C or Fortran programming languages, gain access to PVM in the form of library routines for functions such as the following:

- Process or task initiation
- Message transmission and reception
- Synchronization through the use of barriers or rendezvous

Optionally, users can control the execution location of specific application components; the PVM system transparently handles message routing, data conversion for incompatible architectures, and other tasks that are necessary for operation in a heterogeneous, networked environment.

Applications and Environments

PVM is ideally suited for concurrent applications composed of many interrelated subalgorithms, although performance is good even for traditional parallel applications. PVM is particularly effective for heterogeneous applications that exploit specific strengths of individual systems on a network. As a loosely coupled, concurrent supercomputing environment, PVM is a viable scientific computing platform.

PVM has been used for molecular dynamics simulations, superconductivity studies, distributed fractal computations, matrix algorithms, and as the basis for teaching concurrent programming.

PVM Program Development

To develop a program that uses PVM, you must perform the following steps:

Procedure 2-1

1. Add PVM function calls to your application for process initiation, communications, and synchronization. For syntax descriptions of these functions, see Chapter 3, "Functions and Subroutines", page 27.
2. Build executable files for the systems that you will use, as described in "Building PVM Executable Files", page 7.
3. Create a host file to define the virtual machine, as described in "Creating Host Files", page 7.
4. If your program is in distributed mode, execute the PVM daemon and your application in one of the following ways:
   - As described in "Starting and Stopping the PVM Daemon", page 12, for the PVM daemon, and as described in "Running PVM Applications", page 12, for your application
   - As an NQS job, as described in "Using NQS to Run PVM Applications", page 14
   - Through the PVM console by using the console `spawn` command, as described in Table 2-2, page 16
5. Troubleshoot the application, if necessary. For information on PVM troubleshooting, see "Troubleshooting PVM", page 19.

Building PVM Executable Files

After you have added PVM function calls, code can be linked, beginning with the source file or the object file.

If you begin with the source file, you must specify the `-I` (include) option and the Application Binary Interface (ABI) of the application development library (N32 or 64 ABIs), as follows:

```
cc -I /usr/array/PVM/include -64 -o compute compute.c -lpvm3
```

If you begin with an object file, the code can be linked as follows:

```
cc -64 -o compute compute.o -lpvm3
```

If you have the optional IRIX `mpt` module loaded, use the following command:

```
cc -64 -o compute compute.c -lpvm3
```

After the code is linked, you can install the executable files on the SGI systems you will be using. If you specified the `ep` option in the host file for a system, install the file in the specified directory. Otherwise, install it in the following directory:

```
$HOME/pvm3/bin/$PVM_ARCH
```

Creating Host Files

Each system in the PVM virtual machine must have a separate entry in the host file. Lines that begin with a hash symbol (#), possibly preceded by white space, are ignored.

If you do not want PVM to start a host immediately, but you might start it later by using the `pvm_addhosts`(3) function or the PVM console `add` command, you do not need to include the host in the host file. However, if you need to set any of the options described in Table 2-1, page 8, you should include the specified system in the host file, preceded by the ampersand (&) character.

This command starts the PVM daemon in the background and tells it that automatic host file selection should be used. Hosts can be excluded based on many different resources. For more information on NQE policies, see *NQE Administration*. If a host file is also specified, PVM uses the options specified in the host file. A host specified in the host file will be included in the virtual machine only if that host is available, as determined by the NQE policy.

Example 2-1 is an example of a host file that contains the names of systems, which is the basic information necessary in a host file.

Example 2-1 Simple Host File

```
# my first host file
thud
fred
wilma
gust.sgi.com
rain
```

You should verify that no system is listed more than once, and that the system on which the master `pvmd3`(1) daemon will run (the *master host*) is included in the host file (see "Starting and Stopping the PVM Daemon", page 12, for information on starting the `pvmd3` daemon). Automatic host file selection always includes the host running the master `pvmd3`(1) daemon.

The `$PVM_ROOT` and `$PVM_ARCH` environment variables are set for you automatically when you load the `mpt` module to access the Message Passing Toolkit software. To customize your environment, you can specify the options listed in Table 2-1, after any system name in the host file.

Table 2-1 Host File Options

Option	Description
`bx=` *dpath*	Specifies the debugger path. You can also set this path by using the `PVM_DEBUGGER` environment variable. The default debugger path is `$PVM_ROOT/lib/debugger`.
`dx=` *loc*	Specifies a location for `pvmd3` other than the default, `$PVM_ROOT/lib/$PVM_ARCH/pvmd3`. This option is useful in debugging new versions of PVM. The *loc* variable may be a simple file name, an absolute path name, or a path relative to the user's home directory on the remote system. The `pvmd3` daemon is installed in `$PVM_ROOT/lib/$PVM_ARCH/pvmd3` when the MPT version is installed on SGI systems.

Option	Description
`ep=` *paths*	Specifies a series of paths to search for application tasks. A percent sign (%) in the path expands to the architecture of the remote system. Multiple paths are separated by a colon (`:`). By default, PVM looks for application tasks in the following directories: `$HOME/pvm3/bin/$PVM_ARCH:$PVM_ROOT/bin/$PVM_ARCH`
`ip=` *network_name*	Specifies the network name to be used for communication. The default is determined by the network routing, as shown by the `netstat -i` command. You can use this option to specify HIPPI or another specific device.
`lo=` *userid*	Specifies an alternative login name for the system. The default is the login name on the master system.
`so=ms`	Causes the master `pvmd3` daemon to request that you manually start a `pvmd3` daemon on a slave system when the `rsh`(1) and `rexec`(1) network services are disabled but IP connectivity exists. The default is no request. You cannot start the master system from the PVM console or background when you specify this option. (This option is rarely used.)
`so=pw`	Causes PVM to prompt for a password on the remote system. This option is useful when you have a different login name and password on a remote system. The master host prompts you for your password, as in the following example: `Password(honk.cs.utk.edu:manchek):` Type your password for the remote system. The startup will then continue as normal. You cannot start the master host from the PVM console or background when you specify this option.
`sp=` *value*	Specifies the relative computational speed of this system compared to other systems in the configuration. *value* is an integer in the range 1 through 1,000,000. The default is 1000. (This option currently has no effect on PVM operation.)
`wd=` *path*	Specifies the path name of a working directory in which all spawned tasks on the host will execute. The default is `$HOME`.

A dollar sign (`$`) in an option introduces an environment variable name, for example, `$PVM_ARCH`. Each PVM daemon expands names from environment variables.

The simple host file in Example 2-1, page 8, works well if both of the following conditions are met:

- You have a login with the same name on all of the systems in your host file.
- The local system is listed in the `.rhosts` file on each of the remote systems.

To supply an alternative login name for the `thud` system, add the `lo` option to its host file entry, as follows:

```
thud lo=NAME
```

To be queried for your password on a system named `cyclone`, add `so=pw` to its host file entry, as follows:

```
cyclone so=pw
```

To specify the path of the daemon executable file for a system named `sun114`, add the `dx` option, as follows:

```
sun114 dx=/usr/fred/pvm3/lib/Sun/pvmd3
```

Note: By default, the MPT version of `pvmd3` is installed in `$PVM_ROOT/lib/$PVM_ARCH/pvmd3`, where `$PVM_ROOT` and `$PVM_ARCH` are set for you automatically when you load the `mpt` module.

The string specified in the previous example is passed to a shell so that variable expansion works. Following is another example that uses variable expansion:

```
sun114 dx=bin/$MYBIN/pvmd3
```

You can change the default value of any option for all hosts in a host file by specifying them on a line with an asterisk (`*`) in the host field, as in the following example:

```
thud.cs.utk.edu
gust.sgi.com
sun114 dx=/tmp/pvmd3
* lo=afriend so=pw
```

The preceding example sets the default login name (on remote systems) to `afriend` and queries for a password on each system. Defaults set in this way are effective forward from the location at which they occur in the host file. They can be changed with another `*` line.

You can override the location of executable files by adding the `ep` option to your host file entries, as in the following example:

```
ep=$HOME/pvm3/bin
```

Unlike the `dx` option, which names the daemon file, the `ep` option names a directory.

Example 2-2 shows a more complex host file in which host names are followed by options.

Example 2-2 Sample Host File with Host Name Options

```
# host file for testing on various platforms
# default to my executable
*                  dx=pvm/SUN4/pvmd3
fonebone
refuge
sigi.cs            dx=pvm/PMAX/pvmd3
# reset default for other systems
*                  dx=$PVM_ROOT/lib/$PVM_ARCH/pvmd3
# do not start this system, but define ep in case we add it later
& rain.sgi.com   ep=$(HOME)/bin ip=rain-hippi
# borrowed accts, "guest", don't trust fonebone
*                  lo=guest  so=pw
sn666.jrandom.com  ep=$(HOME)/bin
cubie.misc.edu     ep=pvm/IPSC/pvmd3
```

Specifying Architecture Types

Before you run a PVM executable file on an IRIX system, you must specify the architecture type by setting the `PVM_ARCH` environment variable. Four architecture types are supported for IRIX systems. With the software installed in the default locations, you must also set the `PVM_ROOT` environment variable to `/usr/array/PVM` and the `PATH` environment variable to `$PVM_ROOT/lib/$PVM_ARCH`. The following C shell example shows the setting of all three variables:

```
setenv PVM_ARCH SGIMP64
setenv PVM_ROOT /usr/array/PVM
setenv PATH ${PATH}:${PVM_ROOT}/lib/$PVM_ARCH
```

The architecture types shown in the following list are arranged in an approximate order of lowest to highest performance types:

Architecture type	Description
`SGI32`	N32 ABI/MIPS III version using sockets
`SGI32mips4`	N32 ABI/MIPS IV version using sockets
`SGIMP64mips3`	64 ABI/MIPS III version using POSIX shared memory and sockets

SGIMP64	64 ABI/MIPS IV version using POSIX shared memory and sockets

Starting and Stopping the PVM Daemon

After you have written a host file, you can start up the master pvmd3(1) daemon by passing it the host file as an argument. You must specify the appropriate path for pvmd3(1). For example, you can enter one of the following:

pvmd3 *hostfile* &

or

pvm [*hostfile*]

If you do not specify a host file when starting the PVM console, the PVM daemon found in the default location will be started on the local machine.

The ampersand (&) in the first line tells the operating system to run pvmd3(1) in the background, which is what you will normally want to do.

You should not run pvmd3(1) in the background if you have to enter passwords for any of the slave systems (that is, if you included the so=pw option for one or more systems). In this case, run pvmd3(1) in the foreground and then stop it (by pressing CONTROL-Z) and put it in the background (by entering bg at the prompt) after all systems have started up.

To shut off PVM, enter halt at a PVM console prompt. For detailed information on using console prompts, see "Using Console Commands", page 16.

If the master pvmd3(1) daemon has trouble starting a slave pvmd3(1) daemon on a system, the error message written to the PVM log file from the master pvmd3(1) may indicate the problem.

Running PVM Applications

When the pvmd3(1) daemon is running successfully, you can start your application. PVM provides the following methods of starting applications:

- Start the application from the shell command line.

With this method, you start the application as any command or application would be started. For example, if the application is named `a.out`, enter the following command at the shell command line prompt:

```
./a.out
```

- Start the application from the PVM console by using the `spawn` command.

  With this method, you first start the console. After the `pvm>` prompt has appeared, enter the `spawn` command followed by the application name or path, as needed. For example, to run an application named `cannon`, enter the following command at the console command line prompt:

  ```
  spawn cannon
  ```

You can obtain help for the `spawn` command by typing `help spawn` at the console command line prompt.

Once the application has started, it displays standard output and standard error information for the initial task, but not for the other tasks in the application. PVM captures this output information and sends it to the master daemon. The daemon, in turn, prefaces each line with a PVM task identifier that identifies its source, and writes it to the PVM log file.

The log file can contain very useful information about the virtual machine and its tasks. By default, the log file contains output from the PVM daemon, including error messages and output from tasks. Optionally, the log file can contain debugging output from the daemon.

When PVM is run without NQS, the log file is located in `/tmp`. The IRIX implementation allows overlapping PVM virtual machines. Therefore, more than one PVM daemon started by the same user can run on the same host. The log file is located in `/tmp/pvml.`*uid.vmid*, where *uid* is the user ID and *vmid* is the virtual machine ID. By default, *vmid* is 0, but if the `PVM_VMID` (formerly `PVMJID`) environment variable is set, *vmid* will equal the numeric value of `PVM_VMID`.

Instead of having the data written to the PVM log file, you can request that output be sent as a PVM message to another task's output device. For more information, see the `PvmOutputTid` and `PvmOutputCode` options on the `pvm_setopt`(3) man page.

You can also redirect output by using options on the console `spawn` command (see Table 2-2, page 16) or by using the `pvm_catchout`(3) function.

Using NQS to Run PVM Applications

PVM applications can be run as part of an NQS job script. Each NQS job has its own PVM daemon; therefore, the PVM daemon must be started within the NQS job script. This is different from interactive use, in which one daemon is run per user per system. Any application run as part of the same NQS job script uses the same PVM daemon. Using the `PVM_VMID` environment variable allows more than one daemon to run per user per system. A single user running multiple NQE jobs on a single host should set the `PVM_VMID` environment variable for each batch job.

PVM processes spawned by the daemon inherit the limits of the NQS job. This allows a user to run multiple NQS jobs that use PVM, each with limits of the NQS job being run. Previous versions of PVM used the same daemon for multiple NQS jobs.

The following example is an NQS job script to run the application `foo`:

```
module load mpt
pvmd3 hostfile &   # Start the daemon
sleep 60           # Wait for startup
foo                # Run application
pvm << EOF         # Start console to halt pvm
halt
EOF
```

Using the PVM Console

Using the PVM console is an alternative to using the `pvmd3`(1) command to start the daemon and execute your application. The `pvm`(1) command starts the console, which can be started and stopped multiple times on any of the systems on which PVM is running.

Starting the Console

Start the PVM console by using the following command line:

`pvm [`*hostfile*`]`

When the console is started, it checks to see if a PVM daemon is running. If so, it simply attaches itself to the daemon and can be used to monitor ongoing PVM processes as shown:

```
% pvm
pvmd already running
pvm>
```

If the daemon is not started, the pvm(1) command tries to start one, but the command must first find the daemon. (Currently, the pvm(1) command does not examine the *hostfile* argument, if provided, but simply passes its name to the daemon. Therefore, the pvm command cannot use information from this file.)

The logic used by the pvm command to start the daemon is as follows:

1. The command tries to execute $HOME/pvm3/lib/pvmd on all systems. $HOME/pvm3/lib/pvmd must be an executable file that is one of the following:
   - A shell script that starts up the PVM daemon, perhaps by using a host file. If you use this option, you may find it useful to have the script do other preparatory or related work.
   - A symbolic link to the PVM daemon. The following example shows how you can set up a link:

```
% mkdir ~/pvm3
% mkdir ~/pvm3/lib
% ln -s $PVM_ROOT/lib/$PVM_ARCH/pvmd3 ~/pvm3/lib/pvmd
```

2. If pvmd3(1) is not found or cannot be executed, the pvm(1) command explicitly tries to start $PVM_ROOT/lib/$PVM_ARCH/pvmd3.
   a. If a daemon is started, you see the following:

```
% pvm
pvm>
```

   b. If a daemon is not started, you see the following:

```
% pvm
libpvm [pid-1]: Console: Can't start pvmd
%
```

Using Console Commands

When you enter the pvm(1) command, the console responds with a prompt and accepts the commands described in Table 2-2.

Table 2-2 Console Commands

Command	Description
`add` *hostnames*	Adds systems to the virtual machine.
`alias[`*name command* `[`*args*`]]`	Defines or lists console command aliases.
`conf`	Lists the PVM system configuration. Fields in the output from `conf` are as follows: `HOST` Host name `DTID` PVM daemon task identifier `ARCH` PVM system name (architecture) `SPEED` Relative speed of this system
`delete` *hostnames*	Deletes systems from the virtual machine. PVM processes that are still running on these systems are lost.
`echo [` *args* `]`	Echoes arguments.
`halt`	Kills all PVM processes and shuts down PVM; all daemons exit. This is the best way to exit the console if you are done using PVM. See `quit`.
`help [` *command* `]`	Provides minimal information about the console commands. If you enter `help` followed by a command name, a brief description of the syntax is displayed.
`id`	Prints the `pvm_tid` task identifier of the console. (The console is simply another PVM task.)
`jobs [-l]`	Displays a list of running jobs. The `-l` option provides more detailed output.
`kill [-c ]`*taskids*	Kills a PVM user process. The `-c` option indicates that children of the task IDs should also be killed.
`mstat` *hostnames*	Gives status for each system listed.

Command	Description	
`ps [-a] [-h` *host*`] [-n` *host*`] [-l][-x]`	Gives a listing of current processes and their status. The following options are available:	
	`-a`	All systems (default is local)
	`-h` *host*	Task ID of the system (with no blanks)
	`-n` *host*	System name (with no blanks)
		This example illustrates `-n` *host* usage:
		`ps -ngust`
		This command requests the status of a system named `gust.`
	`-l`	Shows long output
	`-x`	Shows console task
	`ps` output includes the following fields:	
	`HOST`	System executing the process
	`A.OUT`	Executable name (if known to PVM)
	`TID`	Task identifier
	`PTID`	Parent's task identifier (`-l` only)
	`PID`	Task process identifier (`-l` only)
	`FLAG`	Process status. Can be one or more of the following:
		`a` Task is waiting for authorization.
		`c` Task is connected to `pvmd`.
		`o` Task connection is being closed.
		`H` Host starter task is identified.
		`R` Resource manager task is identified.
		`T` Task starter task is identified.
`pstat` *tid*	Displays the status of the specified PVM process.	
`quit` (or EOF)	Exits the console, but leaves the daemons and processes running. See `halt`.	

Command	Description
`reset`	Resets the virtual machine. Causes a `SIGKILL` signal to be sent to every running process. All message queues are cleared. The `pvmd` daemons are left in an idle state.
`setenv [`*name* `[` *value* `]]`	Displays or sets environment variables.
`sig` *num task*	Sends a signal to specified tasks.
`spawn [` *options* `]`*file*	Starts a PVM application for the specified file. Options are as follows: `-` *count* — Number of tasks (default is 1) `-` *host* — Spawn on *host* `-` *arch* — Spawn on hosts of *arch* `-?` — Enables debugging `->` — Redirects output of job to console `->` *file* — Redirects output of job to *file* `->>` *file* — Appends output of job to *file* If NQE load balancing is available, the `spawn` command places tasks based on the load balancer, but within the restrictions specified on the `spawn` command. In the following example, the `spawn` command spawns four instances of `a.out` on the system named `gust`. `pvm>` **`spawn -4 -gust a.out`**
`trace [` *names* `]`	Sets or displays a trace event mask. The *names* argument refers to names defined in the PVM include file, `$PVM_ROOT/include/pvmtev.h.` Alternatives are as follows: `trace [+]` *names* `trace [-]` *names* `trace [+] *` `trace [+] *`
`unalias` *name*	Undefines the specified command alias.
`version`	Displays the `libpvm` version.

Troubleshooting PVM

This section describes common problems encountered when using PVM and provides suggested solutions. There are several kinds of problems that can keep pvmd3(1) from building a virtual machine. The most common are permission problems.

If you do not specify the pw option for a particular system, your .rhosts file on that system must contain the name of the host from which you start the master pvmd. Otherwise, you will get a message like one of the following (although you may not get the entire message):

```
pvmd3@hostname: Permission denied
```

```
pvmd3@hostname: Login incorrect
```

To get the entire error message, enter the following command at a shell prompt:

```
rsh hostname daemon
```

daemon is the location of the PVM daemon (for example, /tmp/pvm/pvmd3 or $PVM_ROOT/lib/$PVM_ARCH/pvmd3).

Look at the output of the command and consult whichever of the following sections most closely applies.

PVM Already Running

When you start the pvmd3(1) daemon, you may receive a message that PVM is already running because a file exists in /tmp. If no pvmd3(1) is running, it is likely that the last time you used PVM you did not terminate pvmd3(1) by using the console halt command, or the previous execution of the pvmd3 daemon terminated abnormally, leaving the files in /tmp. Remove the file named in the message and start pvmd3(1) again.

pvmd3 Fails to Start on Remote System

If you use a shell (such as .kshrc) that does not automatically execute a startup script that sets $PVM_ROOT on added hosts, you can set the PVM_DPATH environment variable to the full or relative path of the pvmd startup script, or include the dx option in the host file to specify the path to the startup script. The pvmd startup script automatically sets $PVM_ROOT on the remote host.

The following command shows how to set the PVM_DPATH environment variable:

```
setenv PVM_DPATH $PVM_ROOT/lib/pvmd
```

The following command shows how to specify the pvmd startup script in the host file:

```
dx=/opt/ctl/mpt/mpt/pvm3/lib/pvmd
```

Note: The dx option in the host file overrides the PVM_DPATH environment variable, and $PVM_ROOT is not acknowledged for dx, so the dx path must be a full pathname.

Permission Denied

If you get a message denying you permission, it probably means that your .rhosts file on the remote system does not include your local system name. Add a line like the following to your .rhosts file on the remote system:

local-host-name your-local-user-name

Sometimes a system has more than one name, and the remote system may think your local system has a name that is different from the one that you have specified. To determine the name of your local system on the remote system, execute telnet(1) or rlogin(1) to get to the remote system and enter the following UNIX command:

```
% who am i
```

Look at the last column of the output of this command, which contains the first 16 characters of what the remote system (the one to which you connected) thinks is the name of your local system (the one on which you entered telnet(1) or rlogin(1)). Make sure you put that system name (the full name, not just the first 16 characters) in your .rhosts file on the remote system. Your /etc/hosts file should contain the full name. If you do not have this file, see your system administrator for the name. Some older systems require that you spell the name exactly the same, including the case; newer systems accept the name in either uppercase or lowercase.

Login Incorrect

If you get a message saying your login is incorrect, there is probably no account on the remote system that has the same login name as your login name on the local system. In this case, you need to add a lo= *username* option to your PVM host file.

Version Incorrect

If you get a message about a version mismatch, it indicates that the versions of PVM on the two systems were built from different PVM releases. You may be building with an old library, accessing an old PVM version built from the public domain version, or having some similar problem. Ensure that the versions of PVM on the two systems are compatible.

As a general rule, releases of the public domain implementation of PVM with the same second digit in the version number (for example, 3.2.0 and 3.2.6) will interoperate. Changes that result in incompatibility are held until a major version change (for example, from version 3.2 to version 3.3). For compatibility, you might need to upgrade one of your versions of PVM.

Failure of Spawn

A common application problem is the failure of a `pvm_spawn()` request. The PVM console command `tickle 6 4` enables tracing of spawn requests. The complete executable path is printed in the PVM log file.

Other Problems

If you get any other messages, ensure that your `.cshrc` file on the remote system is not printing something out when you log in or is not trying to set your terminal characteristics (usually by using the `stty`(1) or `tset`(1) commands).

If you want to print from your `.cshrc` file when you log in, put the relevant commands in an `if` statement in your `.cshrc` file, as in the following example:

```
if ( { tty -s } && $?prompt )  then
# example of printing something when you log in
 echo terminal type is $TERM
# example of setting terminal attributes
   stty erase '^?' kill '^u' intr '^c' echo endif
```

This statement ensures that printing occurs only when you log in from a terminal (and when you are not running a `csh`(1) command script).

Data Types

This section describes how PVM data types are implemented on IRIX systems. This discussion assumes that you are familiar with the functions used to pack and unpack data; for more information, see "Data Transmittal", page 32, and "Data Receipt", page 34.

Table 2-3 and Table 2-4 present basic information about data types on IRIX systems.

Table 2-3 N32 ABI Library Data Types

Data characteristics	C functions	Fortran names
8 bits, not typed	`pvm_pkbyte`	`BYTE1`
16 bits, signed integer	`pvm_pkshort`	`INTEGER2`
32 bits, signed integer	`pvm_pkint`, `pvm_pklong`	`INTEGER4`
16 bits, unsigned integer	`pvm_pkushort`	Not applicable
32 bits, unsigned integer	`pvm_pkuint`, `pvm_pkulong`	Not applicable
32 bits, floating-point	`pvm_pkfloat`,	`REAL4`
64 bits, floating-point	`pvm_pkdouble`	`REAL8`
Two 32 bits, floating-point	`pvm_pkcplx`	`COMPLEX8`
Two 64 bits, floating-point	`pvm_pkdcplx`	`COMPLEX16`
Null-terminated character string	`pvm_pkstr`	Not applicable
Fortran character constant or variable	Not applicable	`STRING`

Table 2-4 64 ABI Library Data Types

Data characteristics	C functions	Fortran names
8 bits, not typed	`pvm_pkbyte`	`BYTE1`
16 bits, signed integer	`pvm_pkshort`	`INTEGER2`

Data characteristics	C functions	Fortran names
32 bits, signed integer	pvm_pkint	INTEGER4
64 bits, signed integer	pvm_pklong	Not applicable
16 bits, unsigned integer	pvm_pkushort	Not applicable
32 bits, unsigned integer	pvm_pkuint	Not applicable
64 bits, unsigned integer	pvm_pkulong	Not applicable
32 bits, floating-point	pvm_pkfloat,	REAL4
64 bits, floating-point	pvm_pkdouble	REAL8
Two 32 bits, floating-point	pvm_pkcplx	COMPLEX8
Two 64 bits, floating-point	pvm_pkdcplx	COMPLEX16
Null-terminated character string	pvm_pkstr	Not applicable
Fortran character constant or variable	Not applicable	STRING

Environment Variables

To customize your PVM environment, you can use the environment variables described in Table 2-5, page 24.

Table 2-5 Environment Variables

Variable	Description	Default
NLB_SERVER	Specifies the location of the NQE load balancer. This host is known as the *master server*. Your system administrator might have this set automatically in the nqeinfo file. If NQE load balancing is enabled on your system, it is used automatically by PVM. To disable NQE load balancing for PVM applications, set the NLB_SERVER environment variable to 0. For more information, see the *NQE User's Guide*. **Note:** Support for this environment variable is deferred on UNICOS/mk and IRIX systems.	Value in the nqeinfo file
PVM_DEBUGGER	Specifies the debugger script to use when pvm_spawn(3) is called with PvmTaskDebug set.	$PVM_ROOT/lib/debugger
PVM_DPATH	Specifies the path of the pvmd3(1) command or the startup script. If you use a shell (such as .kshrc) that does not automatically execute a startup script that sets PVM_ROOT on added hosts, you can set PVM_DPATH to the full or relative path of the pvmd startup script, such as $PVM_ROOT/lib/pvmd. This startup script automatically sets PVM_ROOT.	$PVM_ROOT/lib/pvmd. You can override this setting by using the dx= *loc* option in the host file.
PVM_EXPORT	Names the environment variables that a parent task exports to its children by using the pvm_spawn(3) function. Multiple names must be separated by a colon.	None

Variable	Description	Default
PVM_POLICY	Specifies the NQE policy used for load balancing. For more information on specifying policies, see *NQE Administration*. **Note:** Support for this environment variable is deferred on UNICOS/mk and IRIX systems.	PVM
PVM_ROOT	Specifies the path where PVM libraries and system programs are installed. For PVM to function, this variable must be set on each PVM system.	Set automatically when you load the mpt module to access the Message Passing Toolkit software
PVM_RSH	Specifies that an alternative remote shell command, such as krsh (a Kerberos version of rsh), can be selected. PVM_RSH can specify the full path or relative path to the alternative remote command.	IRIX: If using Array Services, /usr/sbin/arshell. If not using Array Services, /usr/bsd/rsh.
PVM_SHMEM_DIR	Directory location of the POSIX shared memory files.	/usr/tmp (Only valid for SGIMP64 and SGIMP64mips3 architecture types)
PVM_SLAVE_STARTUP_TIMEOUT	Specifies the length of time that the master daemon will wait for a slave daemon to make contact after the slave daemon is started.	60 seconds

Variable	Description	Default
PVM_VMID	Sets the virtual machine identification (VMID) number for the host. This environment variable allows a host to be included in more than one virtual machine by using one pvmd3 command per virtual machine per host. The virtual machine number is appended to the file name of the PVM log and daemon socket files, so that they appear as pvml.*uid.vmid* and pvmd.*uid.vmid*. The previous name of this variable is PVMJID. This name is supported in the MPT 1.3 release, but will not be supported in subsequent releases. **Note:** This environment variable prevents IRIX PVM from interoperating with any implementation other than SGI IRIX PVM implementations.	0
PVMBUFSIZE	Specifies the size of the shared memory buffer for each task and daemon.	1 Mbyte

Functions and Subroutines

This chapter provides general information about PVM error messages and include files, and briefly describes tasks and associated functions.

You can use the C and Fortran interfaces to the PVM library functions to perform the following kinds of tasks:

- Basic operations (see "Basic Operations", page 29)
- Task control (see "Task Control", page 30)
- Option management (see "Option Management", page 30)
- Dynamic system configuration (see "Dynamic System Configuration", page 31)
- Dynamic task group management (see "Dynamic Task Group Management", page 31)
- Data transmittal (see "Data Transmittal", page 32)
- Data receipt (see "Data Receipt", page 34)
- Barrier synchronization (see "Barrier Synchronization", page 36)
- Global operations (see "Global Operations", page 36)
- Signaling (see "Signaling", page 37)
- Error handling (see "Error Handling", page 37)

This chapter briefly describes these tasks. The functions associated with each task are listed in a table. In each table, the functions are grouped as they are described on the man pages, and the groups are listed in the order you usually use them to perform the tasks.

In most cases, each logical PVM function is represented by a C function and a Fortran subroutine. For more information about a specific function or subroutine, use the `man`(1) command to view the associated man page online. To simplify references, this discussion refers to C functions, C++ functions, and Fortran subroutines as *functions* unless individual differences require documentation.

When the C interfaces specify `char *` as a data type, the Fortran interfaces generally permit specification of Fortran character variables or constants. However, these

Fortran values are processed as C strings; therefore, a null character in the middle of the character sequence, which is valid in Fortran, terminates the string.

Error Messages

For a complete list of the PVM error messages and the value associated with each, see "PVM Error Messages", page 39. In general, PVM functions return `PvmOk` (0) or a negative number for errors. Some functions return positive values with other meanings or have special return codes. Error checks should be coded as less than 0, rather than not equal to 0.

You can control the actions that PVM takes when it detects an error. The default is to print an ASCII message and return an error code to the caller. For more information, see the `pvm_setopt`(3) man page for a description of the `PvmAutoErr` option.

Process Identifiers

All processes that enroll in PVM are represented by an integer task identifier, a `pvm_tid`. Because `pvm_tid` values must be unique across the entire virtual machine, they are supplied by PVM and are not chosen by the user. The following routines return `pvm_tid` values:

`pvm_bufinfo`(3)
`pvm_gettid`(3)
`pvm_mytid`(3)
`pvm_parent`(3)
`pvm_spawn`(3)

PVM Include Files

PVM include files for the MPT release are installed in the `$PVM_ROOT/include` directory. If the `mpt` module has been loaded, this include file directory will be searched before any standard include directories.

For better portability, you can refer to PVM include files in your source and specify the include file directory on the compiler command line, as follows:

From C:

```
#include <pvm3.h>
cc -I $PVM_ROOT/include
```

From Fortran:

```
include "fpvm3.h"
f90 -I $PVM_ROOT/include
```

Basic Operations

You can perform basic PVM operations by using the functions in Table 3-1.

Table 3-1 Basic Operations Functions

C and C++ function	Fortran subroutine	Description
_my_pe	MY_PE	Returns the PE number of the PVM task that calls it
_num_pes	NUM_PES	Returns the total number of PEs (or PVM tasks) in the program
pvm_freezegroup	PVMFFREEZEGROUP	Freezes dynamic group membership and caches information locally
pvm_get_PE	PVMFGETPE	Converts a task ID into a PE number
pvm_hostsync	PVMFHOSTSYNC	Gets the time-of-day clock from the PVM host
pvm_mytid	PVMFMYTID	Returns the pvm_tid of the calling task
pvm_parent	PVMFPARENT	Returns the pvm_tid for the task that spawned the calling task
pvm_tidtohost	PVMFTIDTOHOST	Returns the pvm_tid for the PVM daemon task

Task Control

You can control PVM process creation and termination by using the task control functions in Table 3-2.

Table 3-2 Task Control Functions

C and C++ function	Fortran subroutine	Description
pvm_catchout	PVMFCATCHOUT	Catches output from child tasks
pvm_exit	PVMFEXIT	Exits PVM
pvm_halt	PVMFHALT	Shuts down the entire PVM system
pvm_kill	PVMFKILL	Terminates a PVM task
pvm_pstat	PVMFPSTAT	Determines if a PVM task is executing
pvm_reg_hoster	(Not applicable)	Registers a task as the PVM host starter
pvm_reg_tasker	(Not applicable)	Registers a task as the PVM task starter
pvm_spawn	PVMFSPAWN	Starts a new PVM task

Option Management

You can control PVM options by using the functions in Table 3-3.

Table 3-3 Option Management Functions

C and C++ function	Fortran subroutine	Description
pvm_setopt	PVMFSETOPT	Sets a PVM option
pvm_getopt	PVMFGETOPT	Returns the current value of a PVM option

Dynamic System Configuration

The dynamic system configuration functions, described in Table 3-4, allow PVM to be dynamically configured by the application. Systems may be added or removed from the virtual machine, and information can be obtained about a particular system or about the virtual machine as a whole.

Table 3-4 Dynamic System Configuration Functions

C and C++ function	Fortran subroutine	Description
`pvm_addhosts` `pvm_delhosts`	`PVMFADDHOST` `PVMFDELHOST`	Adds or deletes one or more systems
`pvm_config`	`PVMFCONFIG`	Returns the configuration of the virtual machine
`pvm_mstat`	`PVMFMSTAT`	Returns the status of the specified system
`pvm_tasks`	`PVMFTASKS`	Returns information about tasks

Dynamic Task Group Management

A PVM application can form dynamic groups of tasks during its execution. Usually, these groups are established to simplify *multicasting* (the broadcast of data to a number of tasks) and barrier synchronization. Tasks can join and leave groups as desired.

A group is identified by a character string that is assigned by the user. All tasks that want to join a group must specify the same character string.

Dynamically joining and leaving a group must be done with care. Synchronization problems can arise if, for example, one task is joining a group at the same time another task is broadcasting a message to the group. Participating tasks should synchronize at a barrier before trying to use a group.

Dynamic task group management functions are described in Table 3-5.

Table 3-5 Dynamic Task Group Management Functions

C and C++ function	Fortran subroutine	Description
`pvm_getinst`	`PVMFGETINST`	Returns the instance number of a task
`pvm_gettid`	`PVMFGETTID`	Returns the `pvm_tid` for a task
`pvm_gsize`	`PVMFGSIZE`	Returns the number of tasks in a group
`pvm_joingroup` `pvm_lvgroup`	`PVMFJOINGROUP` `PVMFLVGROUP`	Joins or leaves a dynamic group

Data Transmittal

There are two methods in PVM for sending messages. The simpler method, which involves the use of the `pvm_psend`(3) function, lets you make a single call to transmit a contiguous block of data to another PVM task.

The more complex method involves three steps:

1. Initializing a send buffer
2. Packing one or more blocks of data into the buffer
3. Transmitting the buffer to one or more tasks

The second method is more powerful and flexible than the first, but runs more slowly. Messages can be sent to a particular task, can be broadcast to all members of a group, can be broadcast to all tasks, or can be multicast to a list of tasks.

You can use the data transmittal functions in Table 3-6, to transmit data.

Table 3-6 Data Transmittal Functions

C and C++ function	Fortran subroutine	Description
pvm_bcast	PVMFBCAST	Broadcasts a message to all tasks in a group.
pvm_getsbuf	PVMFGETSBUF	Returns the buffer identifier of the current send buffer.
pvm_initsend	PVMFINITSEND	Initializes a send buffer.
pvm_mcast	PVMFMCAST	Broadcasts a message to all tasks in an array.
pvm_mkbuf pvm_freebuf	PVMFMKBUF PVMFFREEBUF	Creates send buffers or releases buffers.
pvm_psend	PVMFPSEND	Packs and sends data in one call.
pvm_pkint pvm_pkshort pvm_pklong pvm_pkuint pvm_pkushort pvm_pkulong pvm_pkfloat pvm_pkdouble pvm_pkcplx pvm_pkdcplx pvm_pkbyte pvm_pkstr pvm_packf	PVMFPACK	Inserts data values into the send buffer. See pvm_pk(3).
pvm_send	PVMFSEND	Sends a message to a single task.
pvm_setsbuf	PVMFSETSBUF	Specifies a new buffer as the current send buffer.

Data Receipt

There are two methods in PVM for receiving messages. The simpler method, which involves the use of the `pvm_precv`(3) function, lets you make a single call to receive a message and store its data into a contiguous block of data. This is a *blocking receive*; the calling task does not return until an appropriate message arrives.

The more complex method involves two steps:

1. Receiving a message. (You can choose either a blocking or a nonblocking form of receive.)
2. Unpacking one or more blocks of data from the message.

Both methods allow you to choose the message to receive. You can choose to receive a message of any of the following types:

- A message with a specific message tag sent by a specific PVM task
- Any message sent by a specific PVM task
- A message with a specific message tag sent by any PVM task
- Any message at all

In addition, PVM provides an optional capability that lets you select a message based on any criteria (including the contents of the message itself). To use this feature, you must write a comparison function (in C) and call `pvm_recvf`(3) or `pvm_trecv`(3). PVM then calls this comparison function on each subsequent `pvm_recv`(3) or `pvm_nrecv`(3) call to identify the message that should be selected.

After a message has been received, the data is available in an internal receive buffer, and additional functions must be called to transfer (and convert) this data into user buffers. Any combination and number of calls to the unpacking functions may be made to move this data into user memory, but it is recommended that the sequence of unpacking calls match the sequence of packing calls that built up the data for the message. It may be possible to use a different sequence, but you should be aware that this depends on undocumented, underlying data packing and transfer mechanisms. (This is particularly dangerous if you use `pvm_pkstr`(3) or if you use `pvm_pkbyte`(3) with a byte count that is not a multiple of 8. Also, if you ever anticipate using this code on another system or across heterogeneous systems, you should avoid using a different sequence.)

The data receipt functions are described in Table 3-7.

Table 3-7 Data Receipt Functions

C and C++ function	Fortran subroutine	Description
pvm_bufinfo	PVMFBUFINFO	Returns information about a message.
pvm_freebuf	PVMFFREEBUF	Releases receive buffers. See pvm_mkbuf(3).
pvm_getrbuf	PVMFGETRBUF	Returns the buffer identifier of the current receive buffer.
pvm_precv	PVMFPRECV	Receives a message directly into a buffer.
pvm_recv pvm_nrecv pvm_probe	PVMFRECV PVMFNRECV PVMFPROBE	Receives a message or probes for a message.
pvm_recvf	(Not applicable)	Supplies a user-written comparison function.
pvm_setrbuf	PVMFSETRBUF	Specifies a new buffer as the current receive buffer.
pvm_trecv	PVMFTRECV	Receives a message with a time-out.
pvm_upkint pvm_upkshort pvm_upklong pvm_upkuint pvm_upkushort pvm_upkulong pvm_upkfloat pvm_upkdouble pvm_upkcplx pvm_upkdcplx pvm_upkbyte pvm_upkstr pvm_unpackf	PVMFUNPACK	Extracts values from received messages. See pvm_upk(3).

Barrier Synchronization

The `pvm_barrier`(3) function described in Table 3-8 lets PVM tasks explicitly synchronize with one another. Calling this function causes the task to *block* (wait) until a specified number of tasks in a group have called the function. When this occurs, all waiting tasks are unblocked. The calling task must be a member of the group, and the *count* argument must be the same for all tasks that use the same barrier.

The `barrier`(3) function described in Table 3-8 lets multitasked PVM tasks explicitly synchronize with one another. This function is useful when PVM is being used in stand-alone mode for global synchronization between all multitasked PVM tasks.

Table 3-8 Barrier Synchronization Function

C and C++ function	Fortran subroutine	Description
`barrier`	`BARRIER`	Creates a barrier to synchronize multitasked PVM tasks
`pvm_barrier`	`PVMFBARRIER`	Creates a barrier to synchronize tasks

Global Operations

The functions in Table 3-9 allow the tasks in a group to participate in a global operation. All tasks in the group must call the same function at the same time.

The `pvm_reduce`(3) function supports sum, product, max, and min operations, as well as user-defined operations.

Table 3-9 Global Operations Functions

C and C++ function	Fortran subroutine	Description
`pvm_gather`	`PVMFGATHER`	Gathers data from group members into an array
`pvm_reduce`	`PVMFREDUCE`	Performs a reduction operation across a group
`pvm_scatter`	`PVMFSCATTER`	Sends a section of an array to each member of the group

Signaling

The functions in Table 3-10 support sending signals of different kinds to PVM tasks.

Table 3-10 Signaling Functions

C and C++ function	Fortran subroutine	Description
`pvm_notify`	`PVMFNOTIFY`	Notifies tasks of specific events
`pvm_sendsig`	`PVMFSENDSIG`	Sends a signal to a task

Error Handling

The function in Table 3-11 provides simple help for handling PVM-generated errors.

Table 3-11 Error Handling Function

C and C++ function	Fortran subroutine	Description
`pvm_perror`	`PVMFPERROR`	Outputs a PVM error message

For more information on controlling PVM behavior, see the `pvm_setopt`(3) man page.

PVM Error Messages

lists the errors detected by PVM. These error message descriptions include the following information:

- Text of the error message written to standard error by PVM functions
- Numeric value of the error returned by PVM functions
- Symbol name for each error, as defined within the PVM include files
- Additional information about the error

Be cautious in your use of the numeric values, because the values assigned to the symbols may change at any time and without any notice.

Errors with numeric values of –100 and below are SGI extensions.

Table A-1 Error Messages Issued by PVM Functions

Error text	Value	Symbol	Additional information
	0	`PvmOk`	
	–1		Reserved
`Bad parameter`	–2	`PvmBadParam`	A bad parameter was passed to the function.
`Count mismatch`	–3	`PvmMismatch`	The count parameter does not match the count used in peer tasks.
`Value too large`	–4	`PvmOverflow`	A value is too large to be packed or unpacked.
`End of buffer`	–5	`PvmNoData`	The end of a message buffer was reached while trying to unpack data.
`No such host`	–6	`PvmNoHost`	There is no host in the virtual machine with the specified name, or the name could not be resolved to an address.

Error text	Value	Symbol	Additional information
`No such file`	–7	`PvmNoFile`	The specified executable file does not exist.
	–8		Reserved
	–9		Reserved
`Malloc failed`	–10	`PvmNoMem`	`malloc` failed to get memory for `libpvm`.
	–11		Reserved
`Can't decode message`	–12	`PvmBadMsg`	The received message has a data format native to another machine, which cannot be decoded by `libpvm`.
	–13		Reserved
`System error`	–14	`PvmSysErr`	`libpvm` could not contact a `pvmd` daemon on the local host, or the `pvmd` failed during an operation.
`No current buffer`	–15	`PvmNoBuf`	There is no current message buffer to pack or unpack.
`No such buffer`	–16	`PvmNoSuchBuf`	There is no message buffer with the specified buffer handle.
`Null group name`	–17	`PvmNullGroup`	A null group name was passed to a function.
`Already in group`	–18	`PvmDupGroup`	The task is already a member of the group it attempted to join.
`No such group`	–19	`PvmNoGroup`	The specified group does not exist.
`Not in group`	–20	`PvmNotInGroup`	The specified group has no such member task.
`No such instance`	–21	`PvmNoInst`	The specified group has no member with this instance.
`Host failed`	–22	`PvmHostFail`	A foreign host in the virtual machine failed during the requested operation.
`No parent task`	–23	`PvmNoParent`	This task has no parent task.

Error text	Value	Symbol	Additional information
Not implemented	–24	PvmNotImpl	This libpvm function or option is not implemented.
Pvmd system error	–25	PvmDSysErr	An internal mechanism in the pvmd daemon failed during the requested operation.
Version mismatch	–26	PvmBadVersion	Two PVM components (a pvmd daemon and a task, two pvmd daemons, or two tasks) have incompatible protocol versions and cannot interoperate.
Out of resources	–27	PvmOutofRes	The requested operation could not be completed due to lack of resources.
Duplicate host	–28	PvmDupHost	An attempt was made to add the same host to a virtual machine more than once, or to add a host already a member of another virtual machine owned by the same user.
Can't start pvmd	–29	PvmCantStart	A pvmd daemon could not be started on the local host, or a slave pvmd daemon could not be started on a remote host.
Already in progress	–30	PvmAlready	The requested operation requires exclusive access, and another operation was already in progress.
No such task	–31	PvmNoTask	No task exists with the given TID.
No such entry	–32	PvmNoEntry	The class server has no entry matching the lookup request.
Duplicate entry	–33	PvmDupEntry	The class server already has an entry matching the insert request.
Name too long	–100	PvmTooLong	
Async transfers still active	–101	PvmStillActive	
Precision lost on default pack	–102	PvmLostPrecision	

Error text	Value	Symbol	Additional information
`Out of buffers`	–103	`PvmOutOfResBuf`	The requested operation could not be completed due to lack of data buffer resources.
`Out of shared memory pool`	–104	`PvmOutOfResSMP`	The requested operation could not be completed due to lack of SMP resources.
`Too many group members`	–105	`PvmOutOfResGmems`	The requested operation could not be completed due to lack of resources.
`Too much data packed`	–106	`PvmTooMuchData`	
`Hit PVM_TOTAL_PACK limit`	–107	`PvmMemLimit`	
`Cannot communicate`	–200	`PvmNoCom`	A multitasked task cannot communicate with the PVM daemon.

PVM Man Pages

The following list shows the online PVM man pages, which document the specified commands and functions (arranged alphabetically).

`man1` pages:

- `pvm_intro`(1)
- `pvm`(1)
- `pvmd3`(1)

`man3` pages:

- `pvm_addhosts`(3)
- `pvm_barrier`(3)
- `pvm_bcast`(3)
- `pvm_bufinfo`(3)
- `pvm_catchout`(3)
- `pvm_channels`(3)
- `pvm_config`(3)
- `pvm_disptrace`(3)
- `pvm_exit`(3)
- `pvm_freezegroup`(3)
- `pvm_gather`(3)
- `pvm_getfds`(3)
- `pvm_get_PE`(3)
- `pvm_getinst`(3)
- `pvm_getrbuf`(3)
- `pvm_getsbuf`(3)

- pvm_gettid(3)
- pvm_gsize(3)
- pvm_halt(3)
- pvm_hostsync(3)
- pvm_initsend(3)
- pvm_joingroup(3)
- pvm_kill(3)
- pvm_mcast(3)
- pvm_mkbuf(3)
- pvm_mstat(3)
- pvm_mytid(3)
- pvm_notify(3)
- pvm_parent(3)
- pvm_perror(3)
- pvm_pk(3)
- pvm_precv(3)
- pvm_psend(3)
- pvm_pstat(3)
- pvm_recv(3)
- pvm_recvf(3)
- pvm_reduce(3)
- pvm_reg_hoster(3)
- pvm_reg_tasker(3)
- pvm_scatter(3)
- pvm_send(3)

- pvm_sendsig(3)
- pvm_setopt(3)
- pvm_setrbuf(3)
- pvm_setsbuf(3)
- pvm_spawn(3)
- pvm_tasks(3)
- pvm_tidtohost(3)
- pvm_trecv(3)
- pvm_upk(3)

Glossary

asynchronous

An asynchronous operation or function proceeds in parallel with its initiator. The initiator must check later to see if the operation or function has completed.

blocking

A blocking function is one that does not return until the function is complete.

broadcast

To send messages to multiple tasks. Often, a *broadcast* is used in the sense of sending to all tasks, whereas *multicast* is used in the sense of sending to an arbitrary set of tasks.

`cplx`

A data item consisting of two successive `float` types.

`dcplx`

A data item consisting of two successive `double` types.

dynamic groups

Groups in which tasks can join and leave groups at any time.

EU

Emory University.

global groups

A group consisting of all the tasks (or PEs) in the MPP partition.

message passing

A parallel programming style in which explicit messages (containing a user-defined, integer message type and data) are sent between tasks.

multicast
To send messages to multiple tasks. See also *broadcast*.

nonblocking
A nonblocking function is one that returns immediately.

NQE
Network Queuing Environment.

ORNL
Oak Ridge National Laboratory.

PE
Processing element.

probe
A message passing concept in which a check is made to see if a message is available, though the message is not actually received at that time.

PVM
Parallel Virtual Machine.

PVM console
A user-level command that lets you monitor and control your PVM system. The console is run with the command `pvm`.

PVM daemon
A user-level process that controls and manages PVM activity on a given host machine. The daemon is run with the command `pvmd3`.

pvm_tid
The name used in this manual to refer to a PVM task identifier, which is used to reference a specific PVM task.

RPC
Remote Procedure Call.

SIMD
Single instruction, multiple data.

SPMD
Same program, multiple data.

Stand-alone mode
PVM is used for communication between tasks within a single executable file with no PVM daemon present.

stride
The spacing between elements.

synchronous
A synchronous operation or function does not return control to its initiator until it has completed the requested operation or function.

task
An independent, parallel process.

task identifier
A 32-bit integer uniquely identifying a PVM task.

UDP
User datagram protocol.

UT
University of Tennessee.

XDR
eXternal Data Representation.

Index