

# S-Lang PVM Module Reference

---

John C. Houck, [houck@space.mit.edu](mailto:houck@space.mit.edu)

Oct 28, 2005



# Contents

<b>1</b>	<b>Introduction to the PVM Module</b>	<b>5</b>
<b>2</b>	<b>Using the PVM Module</b>	<b>7</b>
<b>3</b>	<b>Examples</b>	<b>9</b>
3.1	Example 1: A Simple <i>Hello World</i> Program . . . . .	9
3.1.1	The <code>hello_master</code> program . . . . .	9
3.1.2	The <code>hello_slave</code> program . . . . .	11
3.2	Example 2: Using the Master-Slave Interface . . . . .	11
3.2.1	The <code>master</code> program . . . . .	12
3.2.2	The <code>slave</code> program . . . . .	12
<b>4</b>	<b>Master-Slave Function Reference</b>	<b>13</b>
4.1	<code>pvm_ms_kill</code> . . . . .	13
4.2	<code>pvm_ms_set_num_processes_per_host</code> . . . . .	13
4.3	<code>pvm_ms_set_debug</code> . . . . .	14
4.4	<code>pvm_ms_slave_exit</code> . . . . .	14
4.5	<code>pvm_ms_run_slave</code> . . . . .	15
4.6	<code>pvm_ms_run_master</code> . . . . .	15
4.7	<code>pvm_ms_add_new_slave</code> . . . . .	16
4.8	<code>pvm_ms_set_message_callback</code> . . . . .	16
4.9	<code>pvm_ms_set_slave_exit_failed_callback</code> . . . . .	17
4.10	<code>pvm_ms_set_slave_spawned_callback</code> . . . . .	18
4.11	<code>pvm_ms_set_idle_host_callback</code> . . . . .	18
4.12	<code>pvm_ms_set_hosts</code> . . . . .	19
<b>5</b>	<b>PVM Module Function Reference</b>	<b>21</b>
5.1	<code>pvm_send_obj</code> . . . . .	21

5.2	<code>pvm_recv_obj</code> . . . . .	21
5.3	<code>pvm_config</code> . . . . .	22
5.4	<code>pvm_kill</code> . . . . .	22
5.5	<code>pvm_initsend</code> . . . . .	23
5.6	<code>pvm_pack</code> . . . . .	23
5.7	<code>pvm_send</code> . . . . .	23
5.8	<code>pvm_recv</code> . . . . .	24
5.9	<code>pvm_unpack</code> . . . . .	24
5.10	<code>pvm_psend</code> . . . . .	25
5.11	<code>pvm_addhosts</code> . . . . .	25
5.12	<code>pvm_delhosts</code> . . . . .	25
6	<b>Module Symbols Lacking Documentation</b>	<b>27</b>

# Chapter 1

## Introduction to the PVM Module

PVM (Parallel Virtual Machine) is a software package which permits a heterogeneous collection of Unix and/or Windows computers, connected by a network, to be used as a single large parallel computer. The PVM module provides a **S-lang** interface to this package. By performing distributed computations with **S-lang** one can make better use of available computer resources yet still retain the advantages of programming in an interpreted language.

This document briefly describes how to use the **S-lang** interface to PVM. It assumes that the reader is already familiar with the PVM package itself.

For complete details on obtaining, installing and using the PVM package, see the *PVM documentation* <http://www.csm.ornl.gov/pvm/> . Note that, once the PVM package is properly installed on your computer, the PVM **man** pages will provide detailed documentation on all the PVM library functions.

Although the **S-lang** PVM module functions often have slightly different interfaces, the differences are usually minor so the PVM documentation is quite helpful. Because the **S-lang** interface is not yet fully documented, it will be necessary to consult the PVM documentation directly to make full use of the **S-lang** PVM module.

Because PVM processes require running programs on remote hosts, it is necessary to provide each host with the full path to the relevant executables. To simplify this process, it may be useful to create a directory, e.g. `$HOME/bin/PVM`, on every host and put relevant executables in that directory so that the same relative path will work on all machines. This PVM path may be specified in the `$HOME/.pvmhosts` configuration file; for a detailed description of the contents of this file, see the `pvm` **man** page.

The usage examples discussed in this manual assume that the PVM has already been initialized by running a command such as

```
unix> pvm ~/.pvmhosts
```

This starts the PVM *console* and also starts the PVM daemon, `pvm`, on each remote host. This daemon runs all PVM slave processes and handles all communications with the parent process and the rest of the PVM.

The execution environment of the PVM slave processes is inherited from the corresponding `pvm`

process which, in turn, is inherited from the parent process which started the PVM *console*. However, it is sometimes useful to configure the environment of the remote `pvmd` process using a startup script, `$HOME/.pvmpfile`. This is a Bourne shell script which, if present, is run when `pvmd` is started. For a detailed description of the contents of this file, see the `pvmd` man page.

## Chapter 2

# Using the PVM Module

To use the PVM module in a **S-lang** script, it is first necessary to make the functions in the package known to the interpreter via

```
() = evalfile ("pvm");
```

or, if the application embedding the interpreter supports the **require** function,

```
require ("pvm");
```

may be used. If there is a namespace conflict between symbols in the script and those defined in the module, it may be necessary to load the PVM package into a namespace, e.g.,

```
() = evalfile ("pvm", "p");
```

will place the PVM symbols into a namespace called **p**.

Once the PVM module has been loaded, the functions it defines may be used in the usual way, e.g.,

```
require ("pvm");  
.  
.  
variable master_tid = pvm_mytid ();
```

where **pvm\_mytid** is the PVM function which returns the task identifier of the calling process.





## Chapter 3

# Examples

This section presents examples of two alternate methods of using the PVM module. The source code for these examples is included in the PVM module source code distribution in the `examples` subdirectory. The first method uses PVM library routines to manage a simple distributed application. The second method uses the higher-level master-slave interface. This interface can provide a high degree of tolerance to failure of slave machines which proves useful in long-running distributed applications.

### 3.1 Example 1: A Simple *Hello World* Program

In programming language tutorials, the first example is usually a program which simply prints out a message such as *Hello World* and then exits. The intent of such a trivial example is to illustrate all the steps involved in writing and running a program in that language.

To write a *Hello World* program using the PVM module, we will write two programs, the master (`hello_master`), and the slave (`hello_slave`). The master process will spawn a slave process on different host and then wait for a message from that slave process. When the slave runs, it sends a message to the master, or parent, and then exits. For the purpose of this example, we will assume that the PVM consists of two hosts, named `vex` and `pirx`, and that the slave process will run on `pirx`.

#### 3.1.1 The `hello_master` program

First, consider the master process, `hello_master`. Conceptually, it must specify the full path to the slave executable and then send that information to the slave host (`pirx`). For this example, we assume that the master and slave executables are in the same directory and that the master process is started in that directory. With this assumption, we can construct the path to the slave executable using the `getcwd` and `path_concat` functions. We then send this information to the slave host using the `pvm_spawn` function:

```
path = path_concat (getcwd(), "hello_slave");
slave_tid = pvm_spawn (path, PvmTaskHost, "pirx", 1);
```

The first argument to `pvm_spawn` specifies the full path to the slave executable. The second argument is a bit mask specifying options associated with spawning the slave process. The `PvmTaskHost` option indicates that the slave process is to be started on a specific host. The third argument gives the name of the slave host and the last argument indicates how many copies of this process should be started. The return value of `pvm_spawn` is an array of task identifiers for each of the slave processes; negative values indicate that an error occurred.

Having spawned the `hello_slave` process on `pirx`, the master process calls the `pvm_recv` function to receive a message from the slave.

```
bufid = pvm_recv (-1, -1);
```

The first argument to `pvm_recv` specifies the task identifier of the slave process expected to send the message and the second argument specifies the type of message that is expected. A slave task identifier `-1` means that a message from any slave will be accepted. Similarly, a message identifier of `-1` means that any type of message will be accepted. In this example, we could have specified the slave task id and the message identifier explicitly:

```
bufid = pvm_recv (slave_tid, 1);
```

When a suitable message is received, the contents of the message are stored in a PVM buffer and `pvm_recv` returns the buffer identifier which may be used by the PVM application to retrieve the contents of the buffer.

Retrieving the contents of the buffer normally requires knowing the format in which the information is stored. In this case, because we accepted all types of messages from the slave, we may need to examine the message buffer to find out what kind of message was actually recieved. The `pvm_bufinfo` function is used to obtain information about the contents of the buffer.

```
(,msgid,) = pvm_bufinfo (bufid);
```

Given the buffer identifier, `pvm_bufinfo` returns the number of bytes, the message identifier and the task identifier sending the message.

Because we know that the slave process sent a single object of `Struct_Type`, we retrieve it by calling the `pvm_recv_obj` function.

```
variable obj = pvm_recv_obj();
vmessage ("%s says %s", obj.from, obj.msg);
```

This function is not part of the PVM package but is a higher level function provided by the PVM module. It simplifies the process of sending **S-lang** objects between hosts by handling some of the bookkeeping required by the lower level PVM interface. Having retrieved a **S-lang** object from the message buffer, we can then print out the message. Running `hello_master`, we see:

```
vex> ./hello_master
pirx says Hello World
```

Note that before exiting, all PVM processes should call the `pvm_exit` function to inform the `pvm` daemon of the change in PVM status.

```
pvm_exit();  
exit(0);
```

At this point, the script may exit normally.

### 3.1.2 The hello\_slave program

Now, consider the slave process, `hello_slave`. Conceptually, it must first determine the location of its parent process, then create and send a message to that process.

The task identifier of the parent process is obtained using the `pvm_parent` function.

```
variable ptid = pvm_parent();
```

For this example, we will send a message consisting of a **S-lang** structure with two fields, one containing the name of the slave host and the other containing the string "Hello World".

We use the `pvm_send_obj` function to send this message because it automatically handles packaging all the separate structure fields into a PVM message buffer and also sends along the structure field names and data types so that the structure can be automatically re-assembled by the receiving process. This makes it possible to write code which transparently sends **S-lang** objects from one host to another. To create and send the structure:

```
variable s = struct {msg, from};  
s.msg = "Hello World";  
s.from = getenv ("HOST");  
  
pvm_send_obj (ptid, 1, s);
```

The first argument to `pvm_send_obj` specifies the task identifier of the destination process, the second argument is a message identifier which is used to indicate what kind of message has been sent. The remaining arguments contain the data objects to be included in the message.

Having sent a message to the parent process, the slave process then calls `pvm_exit` to inform the `pvmd` daemon that its work is complete. This allows `pvmd` to notify the parent process that a slave process has exited. The slave then exits normally.

## 3.2 Example 2: Using the Master-Slave Interface

The PVM module provides a higher level interface to support the master-slave paradigm for distributed computations. The symbols associated with this interface have the `pvm.ms` prefix to distinguish them from those symbols associated with the PVM package itself.

The `pvm.ms` interface provides a means for handling computations which consist of a predetermined list of tasks which can be performed by running arbitrary slave processes which take command-line arguments. The interface provides a high degree of robustness, allowing one to add or delete hosts from the PVM while the distributed process is running and also ensuring that the task list will be completed even if one or more slave hosts fail (e.g. crash) during the computation. Experience has shown that this failure tolerance is surprisingly important. Long-running distributed computations

experience failure of one or more hosts with surprising frequency and it is essential that such failures do not require restarting the entire distributed computation from the beginning.

Scripts using this interface must initialize it by loading the `pvm.ms` package via, e.g.

```
require ("pvm.ms");
```

As an example of how to use this interface, we examine the scripts master and slave.

### 3.2.1 The master program

The master script first builds a list of tasks each consisting of an array of strings which provide the command line for each slave process that will be spawned on the PVM. For this simple example, the same command line will be executed a specified number of times. First, the script constructs the path to the slave executable, (`Slave.Pgm`), and then the command line (`Cmd`), that each slave instance will invoke. Then the array of tasks is constructed:

```
variable pgm_argvs = Array_Type[N];  
variable pgm_argv = [Slave_Pgm, Cmd];  
  
pgm_argvs[*] = pgm_argv;
```

The distribution of these tasks across the available PVM is automatically handled by the `pvm.ms` interface. The interface will simultaneously start as many tasks as possible up to some maximum number of processes per host. Here we specify that a maximum of two processes per host may run simultaneously and then submit the list of tasks to the PVM:

```
pvm_ms_set_num_processes_per_host (2);  
exit_status = pvm_ms_run_master (pgm_argvs);
```

As each slave process is completed, its exit status is recorded along with any messages printed to `stdout` during the execution. When the entire list of tasks is complete, an array of structures is returned containing status information for each task that was executed. In this example, the master process simply prints out this information.

### 3.2.2 The slave program

The slave process in this example is relatively simple. Its command line arguments provide the task to be completed. These arguments are then passed to `pvm.ms_run_slave`

```
pvm_ms_run_slave (__argv[[1:]]);
```

which spawns a subshell, runs the specified command, communicates the task completion status to the parent process and exits.

## Chapter 4

# Master-Slave Function Reference

### 4.1 pvm\_ms\_kill

#### Synopsis

Send a "task complete" message to a specific task

#### Usage

```
pvm_ms_kill (Int_Type mtid, Int_Type stid)
```

#### Description

This function may be used to send a "task complete" message to a specific PVM process. The first argument gives the task identifier of the destination process. The second argument gives the task identifier of the sending process.

#### Example

```
tid = pvm_mytid ();  
ptid = pvm_parent ();  
pvm_ms_kill (ptid, tid);
```

#### See Also

[4.4](#) (pvm\_ms\_slave\_exit)

### 4.2 pvm\_ms\_set\_num\_processes\_per\_host

#### Synopsis

Set the maximum number of simultaneous processes per host

#### Usage

```
pvm_ms_set_num_processes_per_host (Int_Type num_processes)
```

**Description**

This function is used to set the maximum number of simultaneous processes per host. The master process normally runs as many simultaneous processes as possible; by setting the maximum number of simultaneous processes per host, one can limit the processing load per host.

**Example**

```
pvm_ms_set_num_processes_per_host (2);
```

**See Also**

[4.6](#) (`pvm_ms_run_master`)

## 4.3 `pvm_ms_set_debug`

**Synopsis**

Set the debug flag

**Usage**

```
pvm_ms_set_debug (Int_Type debug)
```

**Description**

This function may be used to control whether debugging information is printed out during execution. Debugging information is printed if the flag is non-zero.

**Example**

```
pvm_ms_set_debug (1);
```

**See Also**

[4.2](#) (`pvm_ms_set_num_processes_per_host`)

## 4.4 `pvm_ms_slave_exit`

**Synopsis**

Cause a normal exit of a slave process from the PVM

**Usage**

```
pvm_ms_slave_exit (Int_Type exit_status)
```

**Description**

To exit the PVM, a slave process calls this function to send its exit status to the parent process and to notify the local `pvm`d of its exit.

**Example**

```
pvm_ms_slave_exit (exit_status);
```

**See Also**

[4.5](#) (`pvm_ms_run_slave`)

## 4.5 `pvm_ms_run_slave`

### Synopsis

Execute the slave's assigned task in a subshell, then exit the PVM

### Usage

```
pvm_ms_run_slave (String_Type argv[])
```

### Description

A slave process calls this function to run a command in a subshell and then exit the PVM. The command line is constructed by concatenating the elements of an array of strings, `argv`, delimited by spaces. The integer return value from the `system` call provides the exit status for the slave process. After sending this value to its parent process, the slave notifies the PVM and exits.

### Example

```
pvm_ms_run_slave (argv);
```

### See Also

[4.4](#) (`pvm_ms_slave_exit`)

## 4.6 `pvm_ms_run_master`

### Synopsis

Submit a list of tasks to the PVM

### Usage

```
Struct_Type exit_status[] = pvm_ms_run_master (String_Type pgms[])
```

### Description

This function is used to submit a managed list of tasks to the PVM. The task list manager will try to ensure that all tasks are completed and, upon completion of the task list, will return an array of structures containing information about the results of each task.

### Example

To run the Unix command `ps xu` on a number of different hosts:

```
variable slave_argv = Array_Type[n];  
slave_argv[*] = ["ps", "axu"];  
exit_status = pvm_ms_run_master (slave_argv);
```

### See Also

[4.7](#) (`pvm_ms_add_new_slave`)

## 4.7 pvm\_ms\_add\_new\_slave

### Synopsis

Add a new slave to the managed list

### Usage

```
pvm_ms_add_new_slave (String_Type argv[])
```

### Description

This function may be used to add a new slave process while `pvm_ms_run_master()` is running, usually as a result of handling a message.

### Example

```
pvm_ms_add_new_slave ("vex");
```

### See Also

[4.6](#) (`pvm_ms_run_master`)

## 4.8 pvm\_ms\_set\_message\_callback

### Synopsis

Set a callback for handling user-defined messages

### Usage

```
pvm_ms_set_message_callback (Ref_Type func)
```

### Description

This function may be used to handle user-defined messages be sent from slave processes back to the master process.

### Example

```
static define handle_user_message (msgid, tid)
{
    switch (msgid)
    {
        case USER_SLAVE_RESULT:
            recv_results (tid);
            start_task (tid);
        }
    {
        case USER_SLAVE_READY:
            start_task (tid);
        }
    {
        % default:
            return 0;
    }
```



```

    }
    return 1;
}

pvm_ms_set_message_callback (&handle_user_message);

```

See Also

[4.11](#) (pvm\_ms\_set\_idle\_host\_callback), [4.9](#) (pvm\_ms\_set\_slave\_exit\_failed\_callback)

## 4.9 pvm\_ms\_set\_slave\_exit\_failed\_callback

### Synopsis

Set a hook to be called when a slave exits on failure

### Usage

```
pvm_ms_set_slave_exit_failed_callback (Ref_Type func)
```

### Description

This function may be used to have the master process perform a specified action whenever a slave process exits without having completed its assigned task.

This is primarily useful in the context where each command-line submitted to `pvm_ms_run_master` represents a task which itself communicates with the PVM, performing potentially many additional tasks which are independently managed by the process that called `pvm_ms_run_master`.

For example, consider a case in which initialization of slave processes is very expensive but, once initialized, a single slave process may perform many tasks. In this case, the master process may spawn a small number of slaves and then repeatedly send each slave a task to perform. Each slave performs its task, sends the result to the master, and then waits for another task. The managing process must keep track of which tasks have been completed and which remain. If a slave exits while working on a task, it is important that the manager process be notified that that task in progress was not completed and that it should be reassigned to another slave.

### Example

```

static define slave_exit_failed_callback (msgid, tid)
{
    variable t = find_task_tid (tid);

    if (orelse {t == NULL} {t.status == FINISHED})
        return;

    % mark the unfinished task "READY" so that it will
    % be assigned to another slave

    t.tid = -1;
    t.status = READY;
}

```

```
pvm_ms_set_slave_exit_failed_callback (&slave_exit_failed_callback);
```

#### See Also

[4.8](#) (`pvm_ms_set_message_callback`)

## 4.10 `pvm_ms_set_slave_spawned_callback`

### Synopsis

Set the slave spawned callback hook

### Usage

```
pvm_ms_set_slave_spawned_callback (Ref_Type func)
```

### Description

This function may be used to specify a callback function to be called whenever a slave process has been spawned. The callback function will be called with three arguments: the slave task id, the name of the host running the slave process, and an array of strings representing the argument list passed to the slave.

### Example

```
static define slave_spawned_callback (tid, host, argv)
{
    vmessage ("Slave running %s spawned on %s with task-id %d",
              argv[0], host, tid);
}
pvm_ms_set_slave_spawned_callback (&slave_spawned_callback);
```

#### See Also

[4.8](#) (`pvm_ms_set_message_callback`)

## 4.11 `pvm_ms_set_idle_host_callback`

### Synopsis

Set the idle host hook

### Usage

```
pvm_ms_set_idle_host_callback (Ref_Type func)
```

### Description

This function may be used to specify a callback function to be called whenever a new host is added to the virtual machine.

### Example

```
static define idle_host_callback ()
{
    loop (Max_Num_Processes_Per_Host)
    {
        variable slave_argv = build_slave_argv (0);
        pvm_ms_add_new_slave (slave_argv);
    }
}
pvm_ms_set_idle_host_callback (&idle_host_callback);
```

See Also

[4.8](#) (`pvm_ms_set_message_callback`)

## 4.12 pvm\_ms\_set\_hosts

### Synopsis

Set list of hosts to use

### Usage

```
pvm_ms_set_hosts (String-Type hosts[])
```

### Description

This function may be used to specify which hosts will be used to perform distributed calculations. The default is to use all hosts in the current PVM.

### Example

```
pvm_ms_set_hosts (["vex", "pirx", "aluche"]);
```

See Also

[5.11](#) (`pvm_addhosts`)



## Chapter 5

# PVM Module Function Reference

### 5.1 pvm\_send\_obj

#### Synopsis

Pack and send data objects

#### Usage

```
pvm_send (Int_Type tid, Int_Type msgid, object [...])
```

#### Description

This function is much like `pvm_psend` except that it sends additional type information with each object. Using this function paired with `pvm_recv_obj` simplifies sending aggregate data objects such as structures and removes the need for the receiver to specify datatypes explicitly.

#### Example

To send a **S-lang** structure to another process:

```
variable obj = struct {name, x, y, data};  
...  
pvm_send_obj (tid, msgid, obj);
```

#### See Also

[5.2](#) (`pvm_recv_obj`), [5.10](#) (`pvm_psend`), [5.9](#) (`pvm_unpack`)

### 5.2 pvm\_recv\_obj

#### Synopsis

Receive data objects from `pvm_send_obj`

#### Usage

```
obj = pvm_recv_obj ()
```

**Description**

This function receives an object sent by `pvm_send_obj` and returns a slang object of the same type that was sent. It simplifies sending aggregate data types such as structures.

**Example**

To receive a **S-lang** object sent by another process via `pvm_send_obj`:

```
obj = pvm_rcv_obj ();
```

**See Also**

[5.1](#) (`pvm_send_obj`), [5.10](#) (`pvm_psend`), [5.9](#) (`pvm_unpack`)

## 5.3 `pvm_config`

**Synopsis**

Returns information about the present virtual machine configuration

**Usage**

```
Struct_Type = pvm_config ()
```

**Description**

See the PVM documentation.

**Example**

```
h = pvm_config ();
```

**See Also**

[5.4](#) (`pvm_kill`)

## 5.4 `pvm_kill`

**Synopsis**

Terminates a specified PVM process

**Usage**

```
pvm_kill (Int_Type tid)
```

**Description**

See the PVM documentation.

**Example**

```
pvm_kill (tid);
```

**See Also**

[5.3](#) (`pvm_config`)

## 5.5 pvm\_initsend

### Synopsis

Clear default send buffer and specify message encoding

### Usage

```
bufid = pvm_initsend (Int_Type encoding)
```

### Description

See the PVM documentation.

### Example

```
bufid = pvm_initsend (PvmDataDefault);
```

### See Also

[5.7](#) (pvm\_send)

## 5.6 pvm\_pack

### Synopsis

Pack the active message buffer with arrays of prescribed data type

### Usage

```
pvm_pack (object)
```

### Description

See the PVM documentation.

### Example

```
pvm_pack (x);
```

### See Also

[5.9](#) (pvm\_unpack)

## 5.7 pvm\_send

### Synopsis

Immediately sends the data in the active message buffer

### Usage

```
pvm_send (Int_Type, tid, Int_Type msgid)
```

### Description

See the PVM documentation.

**Example**

```
pvm_send (tid, msgid);
```

**See Also**

[5.8](#) (`pvm_recv`)

## 5.8 `pvm_recv`

**Synopsis**

Receive a message

**Usage**

```
bufid = pvm_recv (Int_Type tid, Int_Type msgtag)
```

**Description**

See the PVM documentation.

**Example**

```
bufid = pvm_recv (tid, msgtag);
```

**See Also**

[5.7](#) (`pvm_send`)

## 5.9 `pvm_unpack`

**Synopsis**

Unpack the active message buffer into arrays of prescribed data type

**Usage**

```
item = pvm_unpack (Int_Type type_id, Int_Type num)
```

**Description**

See the PVM documentation.

**Example**

```
item = pvm_unpack (type, num);
```

**See Also**

[5.6](#) (`pvm_pack`)



## 5.10 pvm\_psend

### Synopsis

Pack and send data

### Usage

```
pvm_psend (Int_Type tid, Int_Type msgid, object [...])
```

### Description

See the PVM documentation.

### Example

```
pvm_psend (tid, msgid, data);
```

### Notes

Unlike the `pvm_send` function in the PVM library, this function does not operate asynchronously.

### See Also

[5.7](#) (`pvm_send`), [5.5](#) (`pvm_initsend`), [5.6](#) (`pvm_pack`), [5.8](#) (`pvm_recv`)

## 5.11 pvm\_addhosts

### Synopsis

Add one or more hosts to the PVM server

### Usage

```
Int_Type[] = pvm_addhosts (String_Type[] hosts)
```

### Description

See the PVM documentation.

### Example

```
tids = pvm_addhosts (["vex", "verus", "aluche"]);
```

### See Also

[5.11](#) (`pvm_addhosts`), [5.3](#) (`pvm_config`), [5.12](#) (`pvm_delhosts`)

## 5.12 pvm\_delhosts

### Synopsis

Delete one or more hosts from the PVM server

### Usage

```
pvm_delhosts (String_Type[] hosts)
```

**Description**

See the PVM documentation.

**Example**

```
pvm_delhosts (["vex", "verus"]);
```

**See Also**

[5.12](#) (pvm\_delhosts), [5.3](#) (pvm\_config), [5.4](#) (pvm\_kill)

## Chapter 6

# Module Symbols Lacking Documentation

Although many more low-level PVM intrinsic functions are provided by the S-Lang module, not all of S-Lang interfaces have been documented. See the PVM documentation for information on the following functions:

```
pvm_delhost  
pvm_export  
pvm_freebuf  
pvm_freecontext  
pvm_getcontext  
pvm_newcontext  
pvm_setcontext  
pvm_getopt  
pvm_nrecv  
pvm_sendsig  
pvm_tidtohost  
pvm_setopt  
pvm_config  
pvm_getrbuf  
pvm_getsbuf  
pvm_halt  
pvm_tasks  
pvm_kill  
pvm_mstat  
pvm_pstat  
pvm_mcast  
pvm_addhost  
pvm_archcode  
pvm_probe  
pvm_bufinfo  
pvm_notify  
pvm_unpack  
pvm_send
```

pvm\_recv  
pvm\_pack  
pvm\_initsend  
pvm\_exit  
pvm\_mytid  
pvm\_parent  
pvm\_spawn  
pvm\_barrier  
pvm\_getinst  
pvm\_bcast  
pvm\_gettid  
pvm\_gsize  
pvm\_joingroup  
pvm\_lvgroup  
pvm\_settmask  
pvm\_tev\_mask\_init  
pvm\_tev\_mask\_set  
pvm\_sigterm\_enable

Similarly, the following PVM intrinsic constants are provided by the S-Lang module but are documented only through the PVM documentation.

PvmDataDefault  
PvmDataRaw  
PvmDataInPlace  
PvmDataTrace  
PvmTaskDefault  
PvmTaskHost  
PvmTaskArch  
PvmTaskDebug  
PvmTaskTrace  
PvmMppFront  
PvmHostCompl  
PvmNoSpawnParent  
PvmTaskExit  
PvmHostDelete  
PvmHostAdd  
PvmRouteAdd  
PvmRouteDelete  
PvmNotifyCancel  
PvmRoute  
PvmDontRoute  
PvmAllowDirect  
PvmRouteDirect  
PvmDebugMask  
PvmAutoErr  
PvmOutputTid  
PvmOutputCode  
PvmTraceTid  
PvmTraceCode  
PvmTraceBuffer

---

PvmTraceOptions  
PvmTraceFull  
PvmTraceTime  
PvmTraceCount  
PvmFragSize  
PvmResvTids  
PvmSelfOutputTid  
PvmSelfOutputCode  
PvmSelfTraceTid  
PvmSelfTraceCode  
PvmSelfTraceBuffer  
PvmSelfTraceOptions  
PvmShowTids  
PvmPollType  
PvmPollConstant  
PvmPollSleep  
PvmPollTime  
PvmOutputContext  
PvmTraceContext  
PvmSelfOutputContext  
PvmSelfTraceContext  
PvmNoReset  
PvmTaskSelf  
PvmTaskChild  
PvmBaseContext  
PvmMboxDefault  
PvmMboxPersistent  
PvmMboxMultiInstance  
PvmMboxOverWritable  
PvmMboxFirstAvail  
PvmMboxReadAndDelete  
PvmMboxWaitForInfo  
PvmOk  
PvmBadParam  
PvmMismatch  
PvmOverflow  
PvmNoData  
PvmNoHost  
PvmNoFile  
PvmDenied  
PvmNoMem  
PvmBadMsg  
PvmSysErr  
PvmNoBuf  
PvmNoSuchBuf  
PvmNullGroup  
PvmDupGroup  
PvmNoGroup  
PvmNotInGroup  
PvmNoInst  
PvmHostFail

PvmNoParent  
PvmNotImpl  
PvmDSysErr  
PvmBadVersion  
PvmOutOfRes  
PvmDupHost  
PvmCantStart  
PvmAlready  
PvmNoTask  
PvmNotFound  
PvmExists  
PvmHostrNMstr  
PvmParentNotSet  
PvmNoEntry  
PvmDupEntry  
TEV\_MCAST  
TEV\_SEND  
TEV\_RECV  
TEV\_NRECV