be started on the same machine the user happens to start PVM.

Stdout and stderr appear on the screen for all manually started PVM tasks. The standard error from spawned tasks is written to the log file `/tmp/pvml.<uid>` on the host where PVM was started. The easiest way to see standard output from spawned PVM tasks is to use the redirection available in the pvm console. If standard output is not redirected at the pvm console, then this output also goes to the log file.

Users sometimes want to run their programs with a *nice* value that is at a lower priority so the programs impinge less on workstation owners. There are a couple of ways to accomplish this. The first method, which works with both Fortran and C applications, is to replace your program with a shell script that starts your program. Here is an example two line script:

```
#!/bin/sh
exec nice -10 your_program $*
```

Then when you spawn the shell script it will exec your program at a *nice* level. The second method is to call the UNIX function setpriority() in your program.

A whole series of applications may be run on the existing PVM configuration. It is not necessary to start a new PVM for each application, although it may be necessary to reset PVM if an application crashes.

It is also possible to compile PVM with -DOVERLOADHOST defined. This allows a user to create overlapping virtual machines. The next sections will describe how to write PVM application programs.

5. User Interface

An alphabetical listing of all the routines is given in Appendix A. Appendix A contains a detailed description of each routine, including a description of each argument in each routine and the possible error codes a routine may return and the possible reasons for the error. Each listing includes examples of both C and Fortran use.

A concise summary of the PVM 3.3 routines can be found on the PVM quick reference guide.

In this section we give a brief description of the routines in the PVM 3.3 user library. This section is organized by the functions of the routines. For example, in the subsection on *Dynamic Configuration* is a discussion of the purpose of dynamic configuration, how a user might take advantage of this functionality, and the C and Fortran PVM routines that pertain to this function.

In PVM 3 all PVM tasks are identified by an integer supplied by the local pvmd. In the following descriptions this identifier is called tid. It is similar to the process ID (PID) used in the UNIX system except the tid has encoded in it the location of the process in the virtual machine. This encoding allows for more efficient communication routing, and allows for more efficient integration into multiprocessors.

All the PVM routines are written in C. C++ applications can link to the PVM library. Fortran applications can call these routines through a Fortran 77 interface supplied with the PVM 3 source. This interface translates arguments, which are passed by reference in Fortran, to their values if needed by the underlying C routines. The interface also takes into account Fortran character string representations and the various naming conventions that different Fortran compilers use to call C functions.

5.1. Process Control

```
int tid = pvm_mytid( void )
```

```
call pvmfmytid( tid )
```

The routine pvm_mytid() enrolls this process into PVM on its first call and generates a unique `tid` if the process was not started with pvm_spawn(). It returns the `tid` of this process and can be called multiple times. Any PVM system call (not just pvm_mytid) will enroll a task in PVM if the task is not enrolled before the call.

```
int info = pvm_exit( void )
```

```
call pvmfexit( info )
```

The routine pvm_exit() tells the local pvmd that this process is leaving PVM. This routine does not kill the process, which can continue to perform tasks just like any other UNIX process.

```
int numt = pvm_spawn(  char *task, char **argv, int flag, char *where,
                       int ntask, int *tids )
```

```
call pvmfspawn(  task, flag, where, ntask, tids, numt )
```

The routine pvm_spawn() starts up `ntask` copies of an executable file `task` on the virtual machine. `argv` is a pointer to an array of arguments to `task` with the end of the array specified by NULL. If task takes no arguments then `argv` is NULL. The `flag` argument is used to specify options, and is a sum of

PvmTaskDefault - PVM chooses where to spawn processes,

PvmTaskHost - the `where` argument specifies a particular host to spawn on,

PvmTaskArch - the `where` argument specifies a PVM_ARCH to spawn on,

PvmTaskDebug - starts these processes up under debugger,

PvmTaskTrace - the PVM calls in these processes will generate trace data.

PvmMppFront - starts process up on MPP front-end/service node.

PvmHostCompl - starts process up on complement host set.

PvmTaskTrace is a new feature in PVM 3.3. To display the events, a graphical interface, called XPVM has been created. XPVM combines the features of the PVM console, the Xab debugging package, and ParaGraph to display real-time or post mortem executions. XPVM is available on netlib.

On return `numt` is set to the number of tasks successfully spawned or an error code if no tasks could be started. If tasks were started, then pvm_spawn() returns a vector of the spawned tasks' `tids` and if some tasks could not be started the corresponding error codes are placed in the last $(ntask - numt)$ positions of the vector.

pvm_spawn() can also start tasks on multiprocessors. In the case of the Intel iPSC/860 the following restrictions apply. Each spawn call gets a subcube of size `ntask` and loads the program `task` on all of these nodes. The iPSC/860 OS has an allocation limit of 10 subcubes across all users, so it is better to start a block of tasks on an iPSC/860 with a single pvm_spawn() call rather than several calls. Two different blocks of tasks spawned separately on the iPSC/860 can still communicate with each other as well as any other PVM tasks even though they are in separate subcubes. The iPSC/860 OS has a restriction that messages going from the nodes to the outside world be less than 256 Kbytes.

```
int info = pvm_kill( int tid )
```

```
call pvmfkill( tid, info )
```

The routine pvm_kill() kills some other PVM task identified by `tid`. This routine is not designed to kill the calling task, which should be accomplished by calling pvm_exit() followed by exit().

5.2. Information

```
int tid = pvm_parent( void )
```

```
call pvmfparent( tid )
```

The routine pvm_parent() returns the `tid` of the process that spawned this task or the value of PvmNoParent if not created by pvm_spawn().

```
int pstat = pvm_pstat( int tid )
```

```
call pvmfpstat( tid, pstat )
```

The routine pvm_pstat() returns the status of a PVM task identified by `tid`. It returns PvmOk if the task is running, PvmNoTask if not, or PvmBadParam if `tid` is invalid.

```
int mstat = pvm_mstat( char *host )
```

```
call pvmfmstat( host, mstat )
```

The routine pvm_mstat() returns PvmOk if `host` is running, PvmHostFail if unreachable, or PvmNoHost if `host` is not in the virtual machine. This information can be useful when implementing application level fault tolerance.

```
int info = pvm_config(  int *nhost, int *narch,
                        struct pvmhostinfo **hostp )
```

```
call pvmfconfig( nhost, narch, dtid, name, arch, speed, info )
```

The routine pvm_config() returns information about the virtual machine including the number of hosts, `nhost`, and the number of different data formats, `narch`. `hostp` is a pointer to an array of `pvmhostinfo` structures. The array is of size `nhost`. Each `pvmhostinfo` structure contains the pvmd tid, host name, name of the architecture, and relative cpu speed for that host in the configuration. PVM does not use or determine the speed value. The user can set this value in the hostfile and retrieve it with pvm_config() to use in an application. The Fortran function returns information about one host per call and cycles through all the hosts. Thus, if pvmfconfig is called nhost times, the entire virtual machine will be represented. The Fortran function does not reset itself until the end of a cycle. If the virtual machine is changing rapidly, pvmfconfig will not report the change until it is reset. The user can manually reset pvmfconfig in the middle of a cycle by calling pvmfconfig with `nhost = -1`.

```
int info = pvm_tasks(  int which, int *ntask,
                       struct pvmtaskinfo **taskp )
```

```
call pvmftasks( which, ntask, tid, ptid, dtid, flag, aout, info )
```

The routine pvm_tasks() returns information about the PVM tasks running on the virtual machine. The integer `which` specifies which tasks to return information about. The present options are (0), which means all tasks, a pvmd tid, which means tasks running on that host, or a tid, which means just the given task.

The number of tasks is returned in `ntask`. `taskp` is a pointer to an array of `pvmtaskinfo` structures. The array is of size `ntask`. Each `taskinfo` structure contains the tid, pvmd tid, parent tid, a status flag, and the spawned file name. (PVM doesn't know the file name of manually started tasks.) The Fortran function returns information about one task per call and cycles through all the tasks. Thus, if where = 0, and pvmftasks is called ntask times, all tasks will be represented. The Fortran function does not reset itself until the end of a cycle. If the number of tasks is changing rapidly, pvmftasks will not report the change until it is reset. The user can manually reset pvmftasks in the middle of a cycle by calling pvmftasks with `ntask = -1`.

```
int dtid = pvm_tidtohost( int tid )
```

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

If all a user needs to know is what host a task is running on, then pvm_tidtohost() can return this information.

5.3. Dynamic Configuration

```
int info = pvm_addhosts( char **hosts, int nhost, int *infos)
int info = pvm_delhosts( char **hosts, int nhost, int *infos)
```

```
call pvmfaddhost( host, info )
call pvmfdelhost( host, info )
```

The C routines add or delete a set of `hosts` in the virtual machine. The Fortran routines add or delete a single `host` in the virtual machine. In the Fortran routine `info` is returned as 1 or a status code. In the C version `info` is returned as the number of hosts successfully added. The argument `infos` is an array of length `nhost` that contains the status code for each individual host being added or deleted. This allows the user to check if only one of a set of hosts caused a problem rather than trying to add or delete the entire set of hosts again.

5.4. Signaling

```
int info = pvm_sendsig( int tid, int signum )
```

```
call pvmfsendsig( tid, signum, info )
```

pvm_sendsig() sends a signal `signum` to another PVM task identified by `tid`.

```
int info = pvm_notify( int what, int msgtag, int cnt, int tids )
```

```
call pvmfnotify( what, msgtag, cnt, tids, info )
```

The routine `pvm_notify` requests PVM to notify the caller on detecting certain events. The present options are:

PvmTaskExit - notify if a task exits.

PvmHostDelete - notify if a host is deleted (or fails).

PvmHostAdd - notify if a host is added.

In response to a notify request, some number of messages (see Appendix A) are sent by PVM back to the calling task. The messages are tagged with the code (msgtag) supplied to notify. The tids array specifies who to monitor when using TaskExit or HostDelete. The array contains nothing when using HostAdd. Outstanding notifies are consumed by each notification. For example, a HostAdd notification will need to be followed by another call to pvm_notify() if this task is to be notified of further hosts being added. If required, the routines pvm_config and pvm_tasks can be used to obtain task and pvmd tids.

If the host on which task A is running fails, and task B has asked to be notified if task A exits, then task B will be notified even though the exit was caused indirectly.

5.5. Setting and Getting Options

```
int oldval = pvm_setopt( int what, int val )
int val = pvm_getopt( int what )
```

```
call pvmfsetopt( what, val, oldval )
call pvmfgetopt( what, val )
```

The routines `pvm_setopt` and `pvm_getopt` are a general purpose function to allow the user to set or get options in the PVM system. In PVM 3 pvm_setopt can be used to set several options including: automatic error message printing, debugging level, and communication routing method for all subsequent PVM calls. pvm_setopt returns the previous value of set in oldval. The PVM 3.3 `what` can take have the following values:

Option value		MEANING
`PvmRoute`	1	routing policy
`PvmDebugMask`	2	debugmask
`PvmAutoErr`	3	auto error reporting
`PvmOutputTid`	4	stdout device for children
`PvmOutputCode`	5	output msgtag
`PvmTraceTid`	6	trace device for children
`PvmTraceCode`	7	trace msgtag
`PvmFragSize`	8	message fragment size
`PvmResvTids`	9	allow messages to be sent to reserved tags and tids

See Appendix A for allowable values for these options. Future expansions to this list are planned.

pvm_setopt() can set several communication options inside of PVM such as routing method or fragment sizes to use. It can be called multiple times during an application to selectively set up direct task-to-task communication links, but typical use is to call it once after the initial call to pvm_mytid(). For example:

```
CALL PVMFSETOPT( PvmRoute, PvmRouteDirect )
```

The advantage of direct links is the observed factor of two boost in communication performance. The drawback is the small number of direct links allowed by some UNIX systems, which makes their use unscalable.

When large messages are being sent over FDDI or HiPPI networks, communication performance can sometimes be improved by setting a large fragment size such as 64K.

5.6. Message Passing

Sending a message is composed of three steps in PVM. First, a send buffer must be initialized by a call to pvm_initsend() or pvm_mkbuf(). Second, the message must be "packed" into this buffer using any number and combination of pvm_pk*() routines. (In Fortran all message packing is done with the pvmfpack() subroutine.) Third, the completed message is sent to another process by calling the pvm_send() routine or multicast with the pvm_mcast() routine. In addition there are collective communication functions that operate over an entire group of tasks, for example, broadcast and scatter/gather.

PVM also supplies the routine, pvm_psend(), which combines the three steps into a single call. This allows for the possibility of faster internal implementations, particularly by MPP vendors. pvm_psend() only packs and sends a contiguous array of a single data type. pvm_psend() uses its own send buffer and thus doesn't affect a partially packed buffer to be used by pvm_send().

A message is received by calling either a blocking or non-blocking receive routine and then "unpacking" each of the packed items from the receive buffer. The receive routines can be set to accept ANY message, or any message from a specified source, or any message with a specified message tag, or only messages with a given message tag from a given source. There is also a probe function that returns whether a message has arrived, but does not actually receive it.

PVM also supplies the routine, pvm_precv(), which combines a blocking receive and unpack call. Like pvm_psend(), pvm_precv() is restricted to a contiguous array of a single data type. Between tasks running on an MPP such as the Paragon or T3D the user should receive a pvm_psend() with a pvm_precv(). This restriction was done because much faster MPP implementations are possible when pvm_psend() and pvm_precv() are matched. The restriction is only required within a MPP. When communication is between hosts, pvm_precv() can receive messages sent with pvm_psend(), pvm_send(), pvm_mcast(), or pvm_bcast(). Conversely, pvm_psend() can be received by any of the PVM receive routines.

If required, more general receive contexts can be handled by PVM 3. The routine pvm_recvf() allows users to define their own receive contexts that will be used by the subsequent PVM receive routines.

5.6.1. Message Buffers

The following message buffer routines are required only if the user wishes to manage multiple message buffers inside an application. Multiple message buffers are not required for most message passing between processes. In PVM 3 there is one *active* send buffer and one *active* receive buffer per process at any given moment. The developer may create any number of message buffers and switch between them for the packing and sending of data. The packing, sending, receiving, and unpacking routines only affect the *active* buffers.

```
int bufid = pvm_mkbuf( int encoding )
```

```
call pvmfmkbuf( encoding, bufid )
```

The routine pvm_mkbuf creates a new empty send buffer and specifies the encoding method used for packing messages. It returns a buffer identifier `bufid`.

The `encoding` options are:

PvmDataDefault - XDR encoding is used by default because PVM can not know if the user is going to add a heterogeneous machine before this message is sent. If the user knows that the next message will only be sent to a machine that understands the native format, then he can use *PvmDataRaw* encoding and save on encoding costs.

PvmDataRaw - no encoding is done. Messages are sent in their original format. If the receiving process can not read this format, then it will return an error during unpacking.

PvmDataInPlace - data left in place. Buffer only contains sizes and pointers to the items to be sent. When pvm_send() is called the items are copied directly out of the user's memory. This option decreases the number of times the message is copied at the expense of requiring the user to not modify the items between the time they are packed and the time they are sent. Another use of this option would be to call pack once and modify and send certain items (arrays) multiple times during an application. An example would be passing of boundary regions in a discretized PDE implementation.

```
int bufid = pvm_initsend( int encoding )
```

```
call pvmfinitsend( encoding, bufid )
```

The routine pvm_initsend clears the send buffer and creates a new one for packing a new message. The encoding scheme used for this packing is set by `encoding`. The new buffer identifier is returned in `bufid`. If the user is only using a single send buffer then pvm_initsend() must be called before packing a new message into the buffer, otherwise the existing message will be appended.

```
int info = pvm_freebuf( int bufid )
```

```
call pvmffreebuf( bufid, info )
```

The routine pvm_freebuf() disposes of the buffer with identifier `bufid`. This should be done after a message has been sent and is no longer needed. Call pvm_mkbuf() to create a buffer for a new message if required. Neither of these calls is required when using pvm_initsend(), which performs these functions for the user.

```
int bufid = pvm_getsbuf( void )
```

```
call pvmfgetsbuf( bufid )
```

pvm_getsbuf() returns the active send buffer identifier.

```
int bufid = pvm_getrbuf( void )
```

```
call pvmfgetrbuf( bufid )
```

pvm_getrbuf() returns the active receive buffer identifier.

```
int oldbuf = pvm_setsbuf( int bufid )
```

```
call pvmfsetsbuf( bufid, oldbuf )
```

This routine sets the active send buffer to `bufid`, saves the state of the previous buffer, and returns the previous active buffer identifier `oldbuf`.

```
int oldbuf = pvm_setrbuf( int bufid )
```

```
call pvmfsetrbuf( bufid, oldbuf )
```

This routine sets the active receive buffer to `bufid`, saves the state of the previous buffer, and returns the previous active buffer identifier `oldbuf`.

If `bufid` is set to 0 in pvm_setsbuf() or pvm_setrbuf() then the present buffer is saved and there is no active buffer. This feature can be used to save the present state of an application's messages so that a math library or graphical interface which also use PVM messages will not interfere with the state of the application's buffers. After they complete, the application's buffers can be reset to active.

It is possible to forward messages without repacking them by using the message buffer routines. This is illustrated by the following fragment.

```
bufid = pvm_recv( src, tag );
oldid = pvm_setsbuf( bufid );
info  = pvm_send( dst, tag );
info  = pvm_freebuf( oldid );
```

5.6.2. Packing Data

Each of the following C routines packs an array of the given data type into the active send buffer. They can be called multiple times to pack a single message. Thus a message can contain several arrays each with a different data type. There is no limit to the complexity of the packed messages, but an application should unpack the messages exactly like they were packed. C structures must be passed by packing their individual elements.

The arguments for each of the routines are a pointer to the first item to be packed, `nitem` which is the total number of items to pack from this array, and `stride` which is the stride to use when packing. An exception is pvm_pkstr() which by definition packs a NULL terminated character string and thus does not need `nitem` or `stride` arguments.

```
int info = pvm_pkbyte(   char   *cp, int nitem, int stride )
int info = pvm_pkcplx(   float  *xp, int nitem, int stride )
int info = pvm_pkdcplx(  double *zp, int nitem, int stride )
int info = pvm_pkdouble( double *dp, int nitem, int stride )
int info = pvm_pkfloat(  float  *fp, int nitem, int stride )
int info = pvm_pkint(    int    *np, int nitem, int stride )
int info = pvm_pklong(   long   *np, int nitem, int stride )
int info = pvm_pkshort(  short  *np, int nitem, int stride )
int info = pvm_pkuint(  unsigned int   *np, int nitem, int stride )
int info = pvm_pkushort(unsigned short *np, int nitem, int stride )
int info = pvm_pkulong( unsigned long  *np, int nitem, int stride )
int info = pvm_pkstr(    char   *cp )
```

```
int info = pvm_packf( const char *fmt, ... )
```

PVM also supplies a packing routine pvm_packf() that uses a printf-like format expression to specify what and how to pack data into the send buffer. All variables are passed as addresses if count and stride are specified; otherwise, variables are assumed to be values. A description of the format syntax is given in Appendix A.

A single Fortran subroutine handles all the packing functions of the above C routines.

```
call pvmfpack( what, xp, nitem, stride, info )
```

The argument `xp` is the first item of the array to be packed. Note that in Fortran the number of characters in a string to be packed must be specified in `nitem`. The integer `what` specifies the type of data to be packed. The supported options are:

STRING	0	REAL4	4
BYTE1	1	COMPLEX8	5
INTEGER2	2	REAL8	6
INTEGER4	3	COMPLEX16	7

These names have been predefined in parameter statements in the include file `pvm3/include/fpvm3.h`. Some vendors may extend this list to include 64 bit architectures in their PVM implementations. We will be adding INTEGER8, REAL16, etc. as soon as XDR support for these data types is available.

5.6.3. Sending and Receiving Data

```
int info = pvm_send( int tid, int msgtag )
```

```
call pvmfsend( tid, msgtag, info )
```

The routine pvm_send() labels the message with an integer identifier `msgtag` and sends it immediately to the process `tid`.

```
int info = pvm_mcast( int *tids, int ntask, int msgtag )
```

```
call pvmfmcast( ntask, tids, msgtag, info )
```

The routine pvm_mcast() labels the message with an integer identifier `msgtag` and broadcasts the message to all tasks specified in the integer array `tids` (except itself). The `tids` array is of length `ntask`.

`int info = pvm_psend( int tid, int msgtag, void *vp, int cnt, int type )`

`call pvmfpsend( tid, msgtag, xp, cnt, type, info )`

The routine pvm_psend() packs and sends an array of the specified datatype to the task identified by `tid`. The defined datatypes for Fortran are the same as for pvmfpack(). In C the `type` argument can be any of the following:

PVM_STR	PVM_FLOAT
PVM_BYTE	PVM_CPLX
PVM_SHORT	PVM_DOUBLE
PVM_INT	PVM_DCPLX
PVM_LONG	PVM_DCPLX
PVM_USHORT	PVM_UINT
PVM_ULONG	

These names are defined in `pvm3/include/pvm3.h`.

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

`call pvmfrecv( tid, msgtag, bufid )`

This blocking receive routine will wait until a message with label `msgtag` has arrived from `tid`. A value of -1 in `msgtag` or `tid` matches anything (wildcard). It then places the message in a new active receive buffer that is created. The previous active receive buffer is cleared unless it has been saved with a pvm_setrbuf() call.

`int bufid = pvm_nrecv( int tid, int msgtag )`

`call pvmfnrecv( tid, msgtag, bufid )`

If the requested message has not arrived, then the non-blocking receive pvm_nrecv() returns `bufid` = 0. This routine can be called multiple times for the same message to check if it has arrived while performing useful work between calls. When no more useful work can be performed the blocking receive pvm_recv() can be called for the same message. If a message with label `msgtag` has arrived from `tid`, pvm_nrecv() places this message in a new active receive buffer which it creates and returns the ID of this buffer. The previous active receive buffer is cleared unless it has been saved with a pvm_setrbuf() call. A value of -1 in `msgtag` or `tid` matches anything (wildcard).

`int bufid = pvm_probe( int tid, int msgtag )`

`call pvmfprobe( tid, msgtag, bufid )`

If the requested message has not arrived, then pvm_probe() returns `bufid` = 0. Otherwise, it returns a bufid for the message, but does not "receive" it. This routine can be called multiple times for the same message to check if it has arrived while performing useful work between calls. In addition pvm_bufinfo() can be called with the returned bufid to determine information about the message before receiving it.

`int info = pvm_bufinfo( int bufid, int *bytes, int *msgtag, int *tid )`

`call pvmfbufinfo( bufid, bytes, msgtag, tid, info )`

`int bufid = pvm_trecv( int tid, int msgtag, struct timeval *tmout )`

`call pvmftrecv( tid, msgtag, sec, usec, bufid )`

PVM also supplies a timeout version of receive. Consider the case where a message is never going to arrive (due to error or failure). The routine pvm_recv would block forever. There are times when the user wants to give up after waiting for a fixed amount of time. The routine pvm_trecv() allows the user to specify a timeout period. If the timeout period is set very large then pvm_trecv acts like pvm_recv. If the timeout period is set to zero then pvm_trecv acts like pvm_nrecv. Thus, pvm_trecv fills the gap between the blocking and nonblocking receive functions.

The routine pvm_bufinfo() returns msgtag, source tid, and length in bytes of the message identified by `bufid`. It can be used to determine the label and source of a message that was received with wildcards specified.

```
int info = pvm_precv(  int tid, int msgtag, void *vp, int cnt,
                       int type, int *rtid, int *rtag, int *rcnt )
```

```
call pvmfprecv( tid, msgtag, xp, cnt, type, rtid, rtag, rcnt, info )
```

The routine pvm_precv() combines the functions of a blocking receive and unpacking the received buffer. It does not return a `bufid`. Instead, it returns the actual values of `tid`, `msgtag`, and `cnt` in `rtid, rtag, rcnt` respectively.

```
int (*old)() = pvm_recvf(int (*new)(int buf, int tid, int tag))
```

The routine pvm_recvf() modifies the receive context used by the receive functions and can be used to extend PVM. The default receive context is to match on source and message tag. This can be modified to any user defined comparison function. (See Appendix A for an example of creating a probe function with pvm_recf().) There is no Fortran interface routine for pvm_recvf().

5.6.4. Unpacking Data

The following C routines unpack (multiple) data types from the active receive buffer. In an application they should match their corresponding pack routines in type, number of items, and stride. `nitem` is the number of items of the given type to unpack, and `stride` is the stride.

```
int info = pvm_upkbyte(   char   *cp, int nitem, int stride )
int info = pvm_upkcplx(   float  *xp, int nitem, int stride )
int info = pvm_upkdcplx(  double *zp, int nitem, int stride )
int info = pvm_upkdouble( double *dp, int nitem, int stride )
int info = pvm_upkfloat(  float  *fp, int nitem, int stride )
int info = pvm_upkint(    int    *np, int nitem, int stride )
int info = pvm_upklong(   long   *np, int nitem, int stride )
int info = pvm_upkshort(  short  *np, int nitem, int stride )
int info = pvm_upkuint(  unsigned int   *np, int nitem, int stride )
int info = pvm_upkushort(unsigned short *np, int nitem, int stride )
int info = pvm_upkulong( unsigned long  *np, int nitem, int stride )
int info = pvm_upkstr(    char   *cp )
```

```
int info = pvm_unpackf( const char *fmt, ... )
```

The routine pvm_unpackf() uses a printf-like format expression to specify what and how to unpack data from the receive buffer.

A single Fortran subroutine handles all the unpacking functions of the above C routines.

```
call pvmfunpack( what, xp, nitem, stride, info )
```

The argument `xp` is the array to be unpacked into. The integer argument `what` specifies the type of data to be unpacked. (Same `what` options as for pvmfpack()).

6. Dynamic Process Groups

The dynamic process group functions are built on top of the core PVM routines. There is a separate library `libgpvm3.a` that must be linked with user programs that make use of any of the group functions. The pvmd does not perform the group functions. This is handled by a group server that is automatically started when the first group function is invoked. There is some debate about how groups should be handled in a message passing interface. There are efficiency and reliability issues. There are tradeoffs between static verses dynamic groups. And some people argue that only tasks in a group can call group functions.

In keeping with the PVM philosophy, the group functions are designed to be very general and transparent to the user at some cost in efficiency. Any PVM task can join or leave any group at any time without having to inform any other task in the affected groups. Tasks can broadcast messages to groups of which they are not a member. And in general any PVM task may call any of the following group functions at any time. The exceptions are pvm_lvgroup(), pvm_barrier(), and pvm_reduce() which by their nature require the calling task to be a member of the specified group.

```
int inum  = pvm_joingroup( char *group )
int info  = pvm_lvgroup( char *group )
```

```
call pvmfjoingroup( group, inum )
call pvmflvgroup( group, info )
```

These routines allow a task to join or leave a user named group. The first call to pvm_joingroup() creates a group with name `group` and puts the calling task in this group. pvm_joingroup() returns the instance number (`inum`) of the process in this group. Instance numbers run from 0 to the number of group members minus 1. In PVM 3 a task can join multiple groups.

If a process leaves a group and then rejoins it that process may receive a different instance number. Instance numbers are recycled so a task joining a group will get the lowest available instance number. But if multiple tasks are joining a group there is no guarantee that a task will be assigned its previous instance number.

To assist the user in maintaining a contiguous set of instance numbers despite joining and leaving, the pvm_lvgroup() function does not return until the task is confirmed to have left. A pvm_joingroup() called after this return will assign the vacant instance number to the new task. It is the users responsibility to maintain a contiguous set of instance numbers if his algorithm requires it. If several tasks leave a group and no tasks join, then there will be gaps in the instance numbers.

```
int tid = pvm_gettid( char *group, int inum )
```

```
call pvmfgettid( group, inum, tid )
```

The routine pvm_gettid() returns the `tid` of the process with a given group name