Contents

1	**Class** — ***Singelton for querying the Pvm.***	2
2	**Custom** — ***Base class for classes with non-standard packing and unpacking.***	5
3	**EmptyStruct** — ***An empty derivation of Struct (→ , page 42).***	8
4	**Host** — ***Class representing a host in the PVM.***	10
5	**HostSet** — ***A set of hosts in the PVM.***	14
6	**ReceiveAction** — ***Class representing a certain action on receive.***	16
7	**Struct** — ***Base class for all classes to be transmitted by PVM++.***	17
8	**StructSet** — ***A set of structs to be received.***	26
9	**Task** — ***Class representing a task on the PVM.***	37
10	**TaskSet** — ***A set of tasks on the PVM.***	40
11	**Pvm** — ***Namespace for all objects of PVM++***	41
	Class Graph	42

1 class **Class**

Singelton for querying the Pvm.

Public Members

1.1	Task	**I** () const	*returns a Task (→ , page 42)-instance representing the current task.*	3
1.2	unsigned int	**NumOfArchs** () const	*returns the Number of different architectures, that are present on the pvm.*	3
1.3	void	**Hosts** (HostSet &Result) const	*a set of all hosts in the PVM is returned in the reference parameter Result.*	3
1.4	void	**Tasks** (TaskSet &Result) const	*a set of all tasks currently running on the PVM is returned in the reference parameter Result.*	3
1.5	void	**Update** () const	*this function handles all pending messages, for which receive actions other than normal receive are defined.*	4

Singelton for querying the Pvm.

This singelton class represents the global state of the Parallel Virtual Machine (PVM). During construction (this happens on first call to any PVM++ function) all necessary internal information is created and the current tasks is introduced to the PVM. Access to the class is given by the global function const PvmClass& Pvm().

1.1 Task **I** () const

returns a Task (→ , page 42)-instance representing the current task.

returns a Task (→ , *page* 42)-instance representing the current task. I.e. the task, this function is called from.

1.2

unsigned int **NumOfArchs** () const

returns the Number of different architectures, that are present on the pvm.

returns the Number of different architectures, that are present on the pvm.

1.3

void **Hosts** (HostSet &Result) const

a set of all hosts in the PVM is returned in the reference parameter Result.

a set of all hosts in the PVM is returned in the reference parameter Result.

1.4

void **Tasks** (TaskSet &Result) const

a set of all tasks currently running on the PVM is returned in the reference parameter Result.

a set of all tasks currently running on the PVM is returned in the reference parameter Result.

1.5

void **Update** () const

this function handles all pending messages, for which receive actions other than normal receive are defined.

this function handles all pending messages, for which receive actions other than normal receive are defined. Such are message handler, automatic unpack, swallow on receive. This is executed automatically by all Send() and Receive*() commands of the Struct (→ , *page* 42) and StructSet (→ 7, *page* 17) classes. If you have message handlers installed, it is important to call this function before using data, provided by such handlers. E.g. a task gets messages from time to time, telling it, which tasks it has to send messages to. It then has to provide this information to the Send()-function. Send() calls Update(), but when the message-handlers are invoked for the received message, this new information can't be provided to the Send()-function, as the old information is already there.

2 class **Custom**

Base class for classes with non-standard packing and unpacking.

Public Members

2.1 virtual void
Pack () const — *must be overridden with a function, that packs all relevant data of the derived class.* 6

2.2 virtual void
UnPack () — *must be overridden with a function, that unpacks all relevant data of the derived class.* 7

Base class for classes with non-standard packing and unpacking.

This class is the base class for all data, you want to transmit, for which there are no standard Register ()-Calls (see Struct (→ , *page* 42)). For all types, that can be registered via Register()-Calls, there are functions called Pack (const Type &) and Unpack (type), which can be used in the Pack () and UnPack ()-methods of Custom. You can also use this mechanism, if you want to transmit data "compressed". For example you could have a huge array, that is mostly used with little data:

```
struct MyHugeArray : public Pvm::Custom
{
  int Size;
  int Huge[100000];
  void Pack () const
    {
      Pvm::Pack (Size);
      for (int i = 0; i < Size; ++i)
        Pvm::Pack (Huge[i]);
    }
  void UnPack ()
    {
      Pvm::Unpack (Size);
      for (int i = 0; i < Size; ++i)
        Pvm::Unpack (Huge[i]);
    }
};
```

```
struct Test : public Pvm::Struct
{
  PvmSetStructId (43);
  PvmRegistration ()
  {
    Pvm::Register (Data);
  }
  MyHugeArray Data;
};

int
main ()
{
  Test A;
  A.Data.Size = 3;
  A.Data.Huge[ 0 ] = 4;
  A.Data.Huge[ 1 ] = 8;
  A.Data.Huge[ 2 ] = 16;
  A.Send (Pvm::Pvm ().I ().Parent ()); // Only 4 ints are sent, not 100001 !
}
```

2.1

virtual void **Pack** () const

must be overridden with a function, that packs all relevant data of the derived class.

must be overridden with a function, that packs all relevant data of the derived class.

2.2

virtual void **UnPack** ()

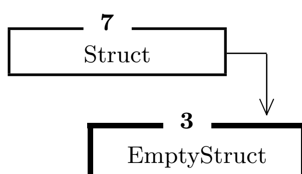
must be overridden with a function, that unpacks all relevant data of the derived class.

must be overridden with a function, that unpacks all relevant data of the derived class.

3 class **EmptyStruct** : public Struct

An empty derivation of Struct (→ , page 42).

Inheritance



Public Members

3.1 **EmptyStruct** (StructId Id)
constructs an instance of EmptyStruct with Id as the StructId (→ 7, page 17). 9

An empty derivation of Struct (→ , *page* 42).

Sometimes you only want to transmit a message without data, e.g. a command to end the receiver. Then it would be to much overhead (both for programming and for compiling) to define a new class for all those messages. As a solution this class is defined. You simply have to make instances of this class, constructed with the corresponding StructId (→ 7, *page* 17) (It still has to be unique, see vmStruct (→ , *page* 42)). As an example see the following:

```
Pvm::EmptyStruct Begin (BeginId);
Pvm::EmptyStruct End (EndId);

Pvm::Task Task;
Begin.Receive (Task);
End.Send (Task);
```

3.1

EmptyStruct (StructId Id)

constructs an instance of EmptyStruct with Id as the StructId (→ 7, page *17).*

constructs an instance of EmptyStruct with Id as the StructId (→ 7, *page* 17).

4 class **Host**

Class representing a host in the PVM.

Public Members

4.1		**Host** (unsigned int What)	*constructs a Host for the host with the host id What as used by PVM.*	11
4.2		**operator unsigned int** () const	*returns the host id as used by PVM.*	11
4.3	std::string	**Name** () const	*returns the name of the host.*	11
4.4	std::string	**Arch** () const	*returns the name of the architecture of this host.*	12
4.5	unsigned int	**Speed** () const	*returns the speed of this host.*	12
4.6	bool	**Running** () const	*returns, whether the host is up and running and still part of the PVM.*	12
4.7	void	**Spawn** (const std::string &Task, int Num, TaskSet &Result) const	*starts Num instances of the task with name Task on the current host and returns the set of started tasks in Result.*	12
4.8	Task	**Spawn** (const std::string &Task) const	*starts the task with name Task on the host and returns the corresponding instance of Task*	13
4.9	void	**Tasks** (TaskSet &Result) const	*returns a list of all tasks, currently running on this host, in the reference parameter Result.*	13

Class representing a host in the PVM.

This class represents a host, that is part of the PVM. Internally it just stores

a pointer to a data structure, so it is save and fast to use it as a value parameter and as a function result as well. Requests to the PVM, like Name() are cached, so you don't need to cache yourself.

There is an ostream& operator<<(ostream& Stream, Host What) defined as well. It outputs the host id as used by the PVM prepended by a "h".

4.1

Host (unsigned int What)

constructs a Host for the host with the host id What as used by PVM.

constructs a Host for the host with the host id What as used by PVM.

4.2

operator unsigned int () const

returns the host id as used by PVM.

returns the host id as used by PVM.

4.3

std::string **Name** () const

returns the name of the host.

returns the name of the host.

4.4

std::string **Arch** () const

returns the name of the architecture of this host.

returns the name of the architecture of this host.

4.5

unsigned int **Speed** () const

returns the speed of this host.

returns the speed of this host. It's the same as in PVM.

4.6

bool **Running** () const

returns, whether the host is up and running and still part of the PVM.

returns, whether the host is up and running and still part of the PVM.

4.7

void **Spawn** (const std::string &Task, int Num, TaskSet &Result) const

starts Num instances of the task with name Task on the current host and returns the set of started tasks in Result.

starts Num instances of the task with name Task on the current host and returns the set of started tasks in Result. The PVM rules regarding the default search path apply.

4.8

Task **Spawn** (const std::string &Task) const

starts the task with name Task on the host and returns the corresponding instance of Task

starts the task with name Task on the host and returns the corresponding instance of Task

4.9

void **Tasks** (TaskSet &Result) const

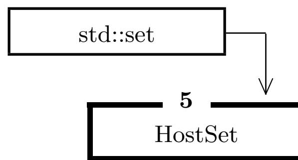
returns a list of all tasks, currently running on this host, in the reference parameter Result.

returns a list of all tasks, currently running on this host, in the reference parameter Result.

5 class **HostSet** : public std::set< Host >

A set of hosts in the PVM.

Inheritance



Public Members

5.1 void **Spawn** (const std::string &Task, int Num, TaskSet &Result) const
starts Num instances of the task with name Task on the hosts in the set and returns the set of started tasks in Result. 15

5.2 Task **Spawn** (const std::string &Task) const
starts the task with name Task on one of the hosts in the set and returns the corresponding instance of Task. 15

A set of hosts in the PVM.

This class represents a set of hosts in the PVM. As it is publically derived from the STL-set, all STL-features can be used. Here is a short example:

```
// get all hosts except the one, I'm on.
Pvm::HostSet AllHosts;
Pvm::Pvm ().Hosts (AllHosts);
AllHosts.erase (Pvm::Pvm ().I ().Host ());
// start one task on all those hosts
Pvm::TaskSet AllTasks;
Pvm::HostSet::iterator Current;
for (Current = AllHosts.begin (); Current != AllHosts.end (); ++Current)
  {
```

```
    // save all started tasks to AllTasks.
    AllTasks.insert (Current->Spawn (PROGNAME));
  }
```

5.1

void **Spawn** (const std::string &Task, int Num, TaskSet &Result) const

starts Num instances of the task with name Task on the hosts in the set and returns the set of started tasks in Result.

starts Num instances of the task with name Task on the hosts in the set and returns the set of started tasks in Result. The PVM rules regarding the default search path apply. The tasks are distributed evenly, taking into account the speed (as set by PVM) of the hosts.

5.2

Task **Spawn** (const std::string &Task) const

starts the task with name Task on one of the hosts in the set and returns the corresponding instance of Task.

starts the task with name Task on one of the hosts in the set and returns the corresponding instance of Task. This function is not very useful actually, but only here because of consistency.

6 class **ReceiveAction**

Class representing a certain action on receive.

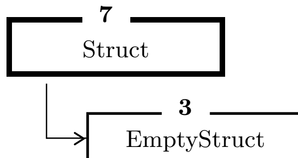
Class representing a certain action on receive. This class is simply a wrapper for the different receive policies, i.e. normal receive (you have to do a Receive*() to get a message), message handler (the handler is automatically called on receive of a message), automatic unpack (the received message is automatically unpacked into a fixed instance of the corresponding class) and swallow on receive (a received message is simply removed from the input queue).

See Also: Struct for Usage.

7 class **Struct**

Base class for all classes to be transmitted by PVM++.

Inheritance



Public Members

7.1 void **Send** (Task To) const
sends the message to task To. .. 20

7.2 void **Send** (const TaskSet &To) const
sends the message to all tasks in the set To. 21

7.3 void **Receive** (Task &From = IgnoreTask)
receives a message of the given type and returns the id of the sender in From. 21

7.4 void **ReceiveFrom** (const TaskSet &FromSet, Task &From = IgnoreTask)
receives a message of the given type, but only from one of the tasks given by FromSet, and returns the id of the sender in From. 21

7.5 void **ReceiveFrom** (Task From)
receives a message of the given type, but only from the task From. 22

7.6 bool **TimedReceive** (unsigned long int &Time, Task &From = IgnoreTask)

receives a message of the given type and returns the id of the sender in From. 22

7.7 bool **TimedReceiveFrom** (const TaskSet &FromSet, unsigned long int &Time, Task &From = IgnoreTask)
receives a message of the given type, but only from one of the tasks given by FromSet, and returns the id of the sender in From. 22

7.8 bool **TimedReceiveFrom** (Task From, unsigned long int &Time)
receives a message of the given type, but only from the task From. 23

7.9 ::Pvm::ReceiveAction **ReceiveAction** (const ::Pvm::ReceiveAction &What)
sets the new action on receive for the current message type to What. 23

7.10 ::Pvm::ReceiveAction **InstallHandler** (HandlerFunc Func)
installs the handler Func as a message handler for all messages of the current message type. 24

7.11 ::Pvm::ReceiveAction **AutomaticUnPack** ()
installs the "unpack on receive" action, ie. 24

7.12 ::Pvm::ReceiveAction **SwallowOnReceive** ()
installs the "swallow on receive" action, ie. 24

7.13 ::Pvm::ReceiveAction **NormalReceive** ()
switches back to normal mode of operation, ie. 25

Base class for all classes to be transmitted by PVM++.

Every struct (or class, for that matter) that should be sent between tasks

must be publically derived from Struct. And it has to register all its members, that should be transmitted, to PVM++. Finally it has to provide a tag, which is used internally in PVM++ to determine the type of a message upon arrival, just like in PVM. **It is very important not to use one number for different classes and to use the same number for the same type in the different communicating programs!** For example a definition could look like this:

```
struct Test : public Pvm::Struct
{
  PvmSetStructId (42); // Setting the Tag for this struct.
                       // Don't use twice!!
  PvmRegistration ()
  {
    Pvm::Register (Data);
    Pvm::Register (Character);
    Pvm::Register (DoubleArray, 187);
    Pvm::Register (Host);
    Pvm::Register (IntSet);
  }
  int Data;
  char Character;
  double DoubleArray[187];
  Pvm::Host Host;
  std::set< int > IntSet;
};
```

As you can see, STL types can be used and will be transmitted correctly, if they are registered and if the template arguments (i.e. int in set< int >) can be registered. The following list shows all types, that can be registered (and therefore transmitted):

- bool.
- char.
- unsigned char.
- short int.
- int.
- float.
- double.
- long int.

- unsigned short.
- unsigned int.
- unsigned long int.
- Pvm::Host (→ 11, *page* 41).
- Pvm::HostSet (→ 11, *page* 41).
- Pvm::Task (→ 11, *page* 41).
- Pvm::TaskSet (→ 11, *page* 41).
- std::string.
- std::complex< Type >, if Type can be registered.
- std::pair< First, Second >, if First and Second can be registered.
- std::vector< Key >, if Key can be registered.
- std::list< Key >, if Key can be registered.
- std::deque< Key >, if Key can be registered.
- std::set< Key >, if Key can be registered.
- std::multiset< Key >, if Key can be registered.
- std::map< Key >, if Key can be registered.
- std::multimap< Key >, if Key can be registered.
- Classes derived from Pvm::Struct (→ 11, *page* 41).
- Classes derived from Pvm::Custom (→ 11, *page* 41).

Arrays for all of the above types (except Struct (→ , *page* 42) and Custom (→ , *page* 42), where it isn't possible due to inheritance) are supported as well. The corresponding syntax is Register (type PointerToArray, int Size).

7.1

void **Send** (Task To) const

sends the message to task To.

sends the message to task To.

7.2

void **Send** (const TaskSet &To) const

sends the message to all tasks in the set To.

sends the message to all tasks in the set To.

7.3

void **Receive** (Task &From = IgnoreTask)

receives a message of the given type and returns the id of the sender in From.

receives a message of the given type and returns the id of the sender in From. The parameter can be omitted, if you're not interested in who was sending. The call blocks until a message is received.

7.4

void **ReceiveFrom** (const TaskSet &FromSet, Task &From = IgnoreTask)

receives a message of the given type, but only from one of the tasks given by FromSet, and returns the id of the sender in From.

receives a message of the given type, but only from one of the tasks given by FromSet, and returns the id of the sender in From. This parameter can be omitted. The call blocks until a message is received.

7.5

```
void ReceiveFrom (Task From)
```

receives a message of the given type, but only from the task From.

receives a message of the given type, but only from the task From. The call blocks until a message is received.

7.6

```
bool TimedReceive (unsigned long int &Time, Task &From = IgnoreTask)
```

receives a message of the given type and returns the id of the sender in From.

receives a message of the given type and returns the id of the sender in From. This parameter can be omitted. The parameter Time specifies the maximal blocking time in microseconds. If after this time no message is received, than the function returns false, otherwise it returns true and the remaining time in Time immediately after receiving the message.

7.7

```
bool TimedReceiveFrom (const TaskSet &FromSet, unsigned long int &Time, Task &From = IgnoreTask)
```

receives a message of the given type, but only from one of the tasks given by FromSet, and returns the id of the sender in From.

receives a message of the given type, but only from one of the tasks given by FromSet, and returns the id of the sender in From. This parameter can be omitted. The parameter Time specifies the maximal blocking time in microseconds.

If after this time no message is received, than the function returns false, otherwise it returns true and the remaining time in Time immediately after receiving the message.

7.8

bool **TimedReceiveFrom** (Task From, unsigned long int &Time)

receives a message of the given type, but only from the task From.

receives a message of the given type, but only from the task From. The parameter Time specifies the maximal blocking time in microseconds. If after this time no message is received, than the function returns false, otherwise it returns true and the remaining time in Time immediately after receiving the message.

7.9

::Pvm::ReceiveAction **ReceiveAction** (const ::Pvm::ReceiveAction &What)

sets the new action on receive for the current message type to What.

sets the new action on receive for the current message type to What. The old action is returned for later use with this function call.

7.10

::Pvm::ReceiveAction **InstallHandler** (HandlerFunc Func)

installs the handler Func as a message handler for all messages of the current message type.

installs the handler Func as a message handler for all messages of the current message type. Such messages can't be received after a call to that function, as every time, such a message arrives, the installed handler will be called. The old action is returned for later use with the method ReceiveAction(). The parameter Func is of type void (*HandlerFunc)(const class Struct (→ , *page* 42)&, const class Task (→ , *page* 42)&). It gets the received message and the sender of it as parameters. A message handler might change global variables and send messages to other tasks, but it should **never** try to receive messages! Message handlers are not called asynchronous, i.e. they are called, whenever the program calls a Send() or Receive*()-function or the Update()-function of Class (→ , *page* 42).

7.11

::Pvm::ReceiveAction **AutomaticUnPack** ()

installs the "unpack on receive" action, ie.

installs the "unpack on receive" action, ie. every subsequent arriving message of this type will automatically be unpacked into this instance. If this instance is deleted, NormalReceive is enabled again. The old action is returned for later use with the method ReceiveAction().

7.12

::Pvm::ReceiveAction **SwallowOnReceive** ()

installs the "swallow on receive" action, ie.

installs the "swallow on receive" action, ie. every subsequent arriving message of this type will be ignored. The old action is returned for later use with the method ReceiveAction().

7.13

::Pvm::ReceiveAction **NormalReceive** ()

switches back to normal mode of operation, ie.

switches back to normal mode of operation, ie. you have to receive messages yourself. The old action is returned for later use with the method ReceiveAction().

8 class **StructSet**

A set of structs to be received.

Public Members

8.1 void **insert** (Struct &What)
inserts a reference to the instance What into the set. 29

8.2 void **erase** (const Struct &What)
erases the potentially contained instance with the same StructId (→ 7, page *17) as What.* 29

8.3 void **erase** (StructId What)
erases the potentially contained instance with the StructId (→ 7, page *17) What.* 30

8.4 int **count** (const Struct &What) const
returns 1, if What (not just any instance with the same StructId (→ 7, page *17) as What) is in the set and 0 otherwise.* 30

8.5 int **count** (StructId What) const
returns 1, if an instance with the StructId (→ 7, page *17)' is in the set and 0 otherwise.* 30

8.6 StructId **Receive** (Task &From = IgnoreTask)
receives a message of one of the types in the set and returns the id of that message and the id of the sender in From. 31

8.7 StructId **ReceiveFrom** (const TaskSet &FromSet, Task &From = IgnoreTask)
receives a message of one of the types in the set, but only from one of the tasks given by FromSet, and returns the id of that message and the id of the sender in From. ... 31

8.8 StructId **ReceiveFrom** (Task From)

receives a message of one of the types in the set, but only from the task From. 32

8.9 StructId **TimedReceive** (unsigned long int &Time, Task &From = IgnoreTask)
receives a message of one of the types in the set and returns the the id of the sender in From. 32

8.10 StructId **TimedReceiveFrom** (const TaskSet &FromSet, unsigned long int &Time, Task &From = IgnoreTask)
receives a message of one of the types in the set, but only from one of the tasks given by FromSet, and returns the id of the sender in From. 33

8.11 StructId **TimedReceiveFrom** (Task From, unsigned long int &Time)
receives a message of one of the types in the set, but only from the task From. 33

8.12 typedef std::set< int > **FDSet**
the type representing a set of file descriptors. 33

8.13 FDSet& **ReadFDs** ()
returns a reference to the set of file descriptors, you want to read from. 34

8.14 const FDSet& **ReadFDs** () const
returns the set of file descriptors, you want to read from. 34

8.15 const FDSet& **ReadyReadFDs** () const
returns the set of file descriptors, that are ready to read from. 34

8.16 FDSet& **WriteFDs** ()
returns a reference to the set of file descriptors, you want to write to. 34

8.17 const FDSet&

		WriteFDs () const	*returns the set of file descriptors, you want to write to.*	35
8.18	const FDSet&	**ReadyWriteFDs** () const	*returns the set of file descriptors, that are ready to write to.*	35
8.19	FDSet&	**ExceptFDs** ()	*returns a reference to the set of file descriptors, that you're expecting to get into an exceptional condition.*	35
8.20	const FDSet&	**ExceptFDs** () const	*returns the set of file descriptors, that you're expecting to get into an exceptional condition.*	35
8.21	const FDSet&	**ReadyExceptFDs** () const	*returns the set of file descriptors, that have an exceptional condition pending.*	36
8.22	bool	**FDsReady** () const	*returns true if any file descriptor is ready.*	36

A set of structs to be received.

A StructSet is of course a set of Struct (→ , *page* 42) derivations. But in contrast to the classes TaskSet (→ 9, *page* 37) and HostSet (→ 4, *page* 10) it is not derived from the STL-class set. Yet some of the methods are available with the same name as in the STL. You can add a Struct (→ , *page* 42) to the StructSet and can remove one from there. If you want to receive messages of, let's say, three different types, then you simply add instances of all those three Struct (→ , *page* 42)-derivations to a StructSet and do a Receive*() (syntax and semantic are analogous to those in Struct.Receive*()). Then you get a return value with the id of the received message (as given with SetStructId()) or 0, in case no message has been received (e.g. if it is a timed receive). Now you find the received information in the previously added instance of the corresponding type. So a code-fragment looks like this:

```
StructA A; // derived from Pvm::Struct; Id = A_Id;
StructB B; // derived from Pvm::Struct; Id = B_Id;
Pvm::StructSet Awaited;
```

```
Awaited.insert (A);
Awaited.insert (B);
while (1)
  {
    Pvm::StructId Id = Awaited.ReceiveFrom (SendingTask);
    if (Id == A_Id)
    {
      cout << A.what_ever_data_is_in_A << endl;
    }
    else // now Id must be equal B_Id
    {
      cout << B.what_ever_data_is_in_B << endl;
    }
  }
```

8.1

void **insert** (Struct &What)

inserts a reference to the instance What into the set.

inserts a reference to the instance What into the set. There can inly be one instance of every type (determined by the corresponding StructId (→ 7, *page* 17)) inside such a set. So an instance overrides potentially contained ones of the same type.

8.2

void **erase** (const Struct &What)

erases the potentially contained instance with the same StructId (→ 7, page *17) as What.*

erases the potentially contained instance with the same StructId (→ 7, *page* 17) as What.

8.3

void **erase** (StructId What)

erases the potentially contained instance with the StructId (→ 7, page *17) What.*

erases the potentially contained instance with the StructId (→ 7, *page* 17) What.

8.4

int **count** (const Struct &What) const

returns 1, if What (not just any instance with the same StructId (→ 7, page *17) as What) is in the set and 0 otherwise.*

returns 1, if What (not just any instance with the same StructId (→ 7, *page* 17) as What) is in the set and 0 otherwise.

8.5

int **count** (StructId What) const

returns 1, if an instance with the StructId (→ 7, page *17)' is in the set and 0 otherwise.*

returns 1, if an instance with the StructId (→ 7, *page* 17)' is in the set and 0 otherwise.

8.6

StructId **Receive** (Task &From = IgnoreTask)

receives a message of one of the types in the set and returns the id of that message and the id of the sender in From.

receives a message of one of the types in the set and returns the id of that message and the id of the sender in From. The parameter can be omitted, if you're not interested in who was sending. The message is unpacked into the instance that was added to set at. The call blocks until a message is received. If one of the file descriptor sets is not empty, the function returns immediatly 0, if the corresponding action on the file desriptor is possible. Subsequent calls to the Ready*FDs()-methods return the ready file descriptors. The input sets are not changed.

8.7

StructId **ReceiveFrom** (const TaskSet &FromSet, Task &From = IgnoreTask)

receives a message of one of the types in the set, but only from one of the tasks given by FromSet, and returns the id of that message and the id of the sender in From.

receives a message of one of the types in the set, but only from one of the tasks given by FromSet, and returns the id of that message and the id of the sender in From. This parameter can be omitted. The message is unpacked into the instance that was added to set at.The call blocks until a message is received. If one of the file descriptor sets is not empty, the function returns immediatly 0, if the corresponding action on the file desriptor is possible. Subsequent calls to the Ready*FDs()-methods return the ready file descriptors. The input sets are not changed.

8.8

StructId **ReceiveFrom** (Task From)

receives a message of one of the types in the set, but only from the task From.

receives a message of one of the types in the set, but only from the task From. It returns the id of the message. The message is unpacked into the instance that was added to set at.The call blocks until a message is received. If one of the file descriptor sets is not empty, the function returns immediatly 0, if the corresponding action on the file desriptor is possible. Subsequent calls to the Ready*FDs()-methods return the ready file descriptors. The input sets are not changed.

8.9

StructId **TimedReceive** (unsigned long int &Time, Task &From = IgnoreTask)

receives a message of one of the types in the set and returns the the id of the sender in From.

receives a message of one of the types in the set and returns the the id of the sender in From. This parameter can be omitted. The parameter Time specifies the maximal blocking time in microseconds. If after this time no message is received, than the function returns 0, otherwise it returns the id of the message and the remaining time in Time immediately after receiving the message. If received, the message is unpacked into the instance that was added to set at. If one of the file descriptor sets is not empty, the function returns immediatly 0, if the corresponding action on the file desriptor is possible. Subsequent calls to the Ready*FDs()-methods return the ready file descriptors. The input sets are not changed.

8.10

StructId **TimedReceiveFrom** (const TaskSet &FromSet, unsigned long int &Time, Task &From = IgnoreTask)

receives a message of one of the types in the set, but only from one of the tasks given by FromSet, and returns the id of the sender in From.

receives a message of one of the types in the set, but only from one of the tasks given by FromSet, and returns the id of the sender in From. This parameter can be omitted. The parameter Time specifies the maximal blocking time in microseconds. If after this time no message is received, than the function returns 0, otherwise it returns the id of the message and the remaining time in Time immediately after receiving the message. If received, the message is unpacked into the instance that was added to set at. If one of the file descriptor sets is not empty, the function returns immediatly 0, if the corresponding action on the file desriptor is possible. Subsequent calls to the Ready*FDs()-methods return the ready file descriptors. The input sets are not changed.

8.11

StructId **TimedReceiveFrom** (Task From, unsigned long int &Time)

receives a message of one of the types in the set, but only from the task From.

receives a message of one of the types in the set, but only from the task From. The parameter Time specifies the maximal blocking time in microseconds. If after this time no message is received, than the function returns 0,otherwise it returns the id of the message and the remaining time in Time immediately after receiving the message. If received, the message is unpacked into the instance that was added to set at. If one of the file descriptor sets is not empty, the function returns immediatly 0, if the corresponding action on the file desriptor is possible. Subsequent calls to the Ready*FDs()-methods return the ready file descriptors. The input sets are not changed.

8.12

typedef std::set< int > **FDSet**

the type representing a set of file descriptors.

the type representing a set of file descriptors.

8.13

FDSet& **ReadFDs** ()

returns a reference to the set of file descriptors, you want to read from.

returns a reference to the set of file descriptors, you want to read from.

8.14

const FDSet& **ReadFDs** () const

returns the set of file descriptors, you want to read from.

returns the set of file descriptors, you want to read from.

8.15

const FDSet& **ReadyReadFDs** () const

returns the set of file descriptors, that are ready to read from.

returns the set of file descriptors, that are ready to read from.

8.16

FDSet& **WriteFDs** ()

returns a reference to the set of file descriptors, you want to write to.

returns a reference to the set of file descriptors, you want to write to.

8.17

const FDSet& **WriteFDs** () const

returns the set of file descriptors, you want to write to.

returns the set of file descriptors, you want to write to.

8.18

const FDSet& **ReadyWriteFDs** () const

returns the set of file descriptors, that are ready to write to.

returns the set of file descriptors, that are ready to write to.

8.19

FDSet& **ExceptFDs** ()

returns a reference to the set of file descriptors, that you're expecting to get into an exceptional condition.

returns a reference to the set of file descriptors, that you're expecting to get into an exceptional condition.

8.20

const FDSet& **ExceptFDs** () const

returns the set of file descriptors, that you're expecting to get into an exceptional condition.

returns the set of file descriptors, that you're expecting to get into an exceptional condition.

8.21

const FDSet& **ReadyExceptFDs** () const

returns the set of file descriptors, that have an exceptional condition pending.

returns the set of file descriptors, that have an exceptional condition pending.

8.22

bool **FDsReady** () const

returns true if any file descriptor is ready.

returns true if any file descriptor is ready.

9 class **Task**

Class representing a task on the PVM.

Public Members

9.1		**Task** (unsigned int What)	*constructs a Task for the host with the task id What as used by PVM.*	38
9.2		**operator unsigned int** () const	*returns the host id as used by PVM.*	38
9.3	bool	**HasParent** () const	*returns, whether the task has a parent.*	38
9.4	Task	**Parent** () const	*returns the parent of the task.*	38
9.5	::Pvm::Host	**Host** () const	*returns the host, the task is running on.*	39
9.6	std::string	**Name** () const	*returns the name of the task.*	39
9.7	bool	**Running** () const	*returns, whether the task is running.*	39
9.8	void	**Kill** () const	*kills the tasks.*	39

Class representing a task on the PVM.

This class represents a task, running (or preveously running) on the PVM. Internally it just stores a pointer to a data structure, so it is save and fast to use it as a value parameter and as a function result as well. Requests to the PVM, like Name() are cached, so you don't need to cache yourself.

There is an ostream& operator<< (ostream &Stream, Task What) defined as well. It outputs the task id as used by the PVM prepended by a "t".

9.1

Task (unsigned int What)

constructs a Task for the host with the task id What as used by PVM.

constructs a Task for the host with the task id What as used by PVM.

9.2

operator unsigned int () const

returns the host id as used by PVM.

returns the host id as used by PVM.

9.3

bool **HasParent** () const

returns, whether the task has a parent.

returns, whether the task has a parent.

9.4

Task **Parent** () const

returns the parent of the task.

returns the parent of the task. It is an error to call this function, if no parent exists. Use HasParent() to check.

9.5

::Pvm::Host **Host** () const

returns the host, the task is running on.

returns the host, the task is running on.

9.6

std::string **Name** () const

returns the name of the task.

returns the name of the task. The name is only available for programs started by PVM++ via a Spawn() call or started by PVM using the pvm_spawn() call. This is a limitation inherited from PVM.

9.7

bool **Running** () const

returns, whether the task is running.

returns, whether the task is running.

9.8

void **Kill** () const

kills the tasks.

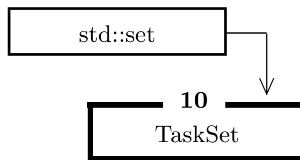
kills the tasks.

10

class **TaskSet** : public std::set< Task >

A set of tasks on the PVM.

Inheritance



A set of tasks on the PVM.

This class represents a set of tasks on the PVM. As it is derived from the STL-set, all STL-features can be used. See the example provided with HostSet (→ 4, *page* 10).

11 namespace **Pvm**

Namespace for all objects of PVM++

Names

11.1 typedef unsigned int **StructId** *type for the ID of a Struct (→ , page 42).* 41

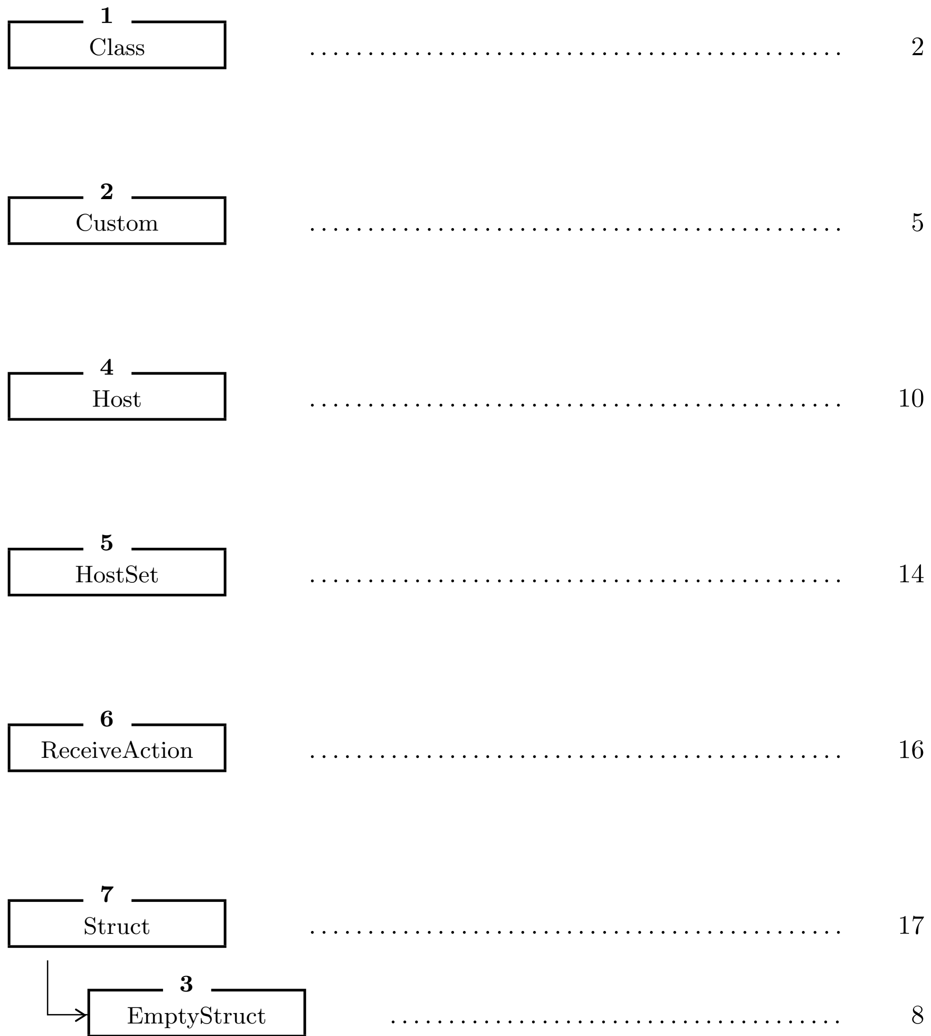
Namespace for all objects of PVM++

11.1 typedef unsigned int **StructId**

type for the ID of a Struct (→ , page 42).

type for the ID of a Struct (→ , *page* 42).

Class Graph



Class 2

Custom 5

Host 10

HostSet 14

ReceiveAction 16

Struct 17

EmptyStruct 8

8 StructSet 26

9 Task 37

10 TaskSet 40