CPPvm:
C++ Interface to PVM
(Parallel Virtual Machine)

Version 1.5.0

Steffen Görzig
Steffen.Goerzig@informatik.uni-stuttgart.de

Preface

The Parallel Virtual Machine (PVM) system allows to combine a heterogeneous collection of hosts running various operating systems to be used as a single large parallel computer. The programming languages supported by PVM are Fortran and C. The CPPvm (C Plus Plus PVM) library was written to add C++ to this list.

Although PVM can be used directly in C++ programs due to its C–functions, it does not support C++ specific features. CPPvm enlarges PVM with these C++ features like classes, inheritance, overloaded operators, exception handling and streams. CPPvm also hides some details of PVM from the user and thus makes it even easier to write parallel programs.

I hope CPPvm will bring more people to parallel programming. It should be very easy to write parallel software with CPPvm even for unexperienced programmers. On the other hand, CPPvm is sufficient for very complex parallel problems. So a user of CPPvm can make his way from the very first parallel steps to the innermost secrets of MIMD programming.

I want to thank (in alphabetical order) Axel Gern, Udo Gleich, Heike and Stefan Krauß, Christina and Jürgen Mehl as well as Jürgen Wakunda for reviewing the software. They all spent a lot of their spare time for this project and gave me plenty of ideas for what they called "improvements". I called it "work". Anyway, thank you very much. Also many thanks to (in order of appearance) José Manuel Palomares Munoz, Marc Barette, Jim McCusker, Jan van der Veen, John Galbraith, Patrick Gundlach, Alexander Vorobiev, James Arthur Kohl, Thomas Ullmann, André Schmidt, Roland Ortloff, and Andrew Bradley for new ideas and bugfixes. Of course I want to thank all the people who have implemented PVM. Keep up your good work.

Contents

Chapter 1

Quick Start

"Please, I don´t want to read the complete manual, I want to see something working at first!"

Ok, ok, calm down. Here is a short introduction for impatient people.

1.1 UNIX

First get the PVM3.4.x software:

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

Extract the PVM files:

```
>cd $HOME
>gunzip pvm3.4.x.tar.gz
>tar xvf pvm3.4.x.tar
```

Set the environment variables `PVM_ROOT` and `PVM_ARCH`:

- `PVM_ROOT`: Location of the `pvm3` installation directory. For a local installation as user `smith` the directory can be for example `/home/smith/pvm3`.
- `PVM_ARCH`: Architecture of the host. The architecture can be determined using the shell script `pvmgetarch` in the directory `$PVM_ROOT/lib` (e.g. `LINUX` or `SUN4SOL2`).

Compile PVM:

```
>cd $PVM_ROOT
>make
```

Extract the files, compile and install CPPVM (modify the `cppvm/makefile` if necessary):

```
>cd $HOME
>gunzip cppvm.tar.gz
>tar xvf cppvm.tar
>cd cppvm
>make
>make install
```

Compile the CPPVM examples (modify the `cppvm/examples/makefile` if necessary):

```
>cd $HOME
>cd cppvm/examples
>make all
>make install
```

Execute the examples, e.g. `hello`:

```
>cd $PVM_ROOT/bin/$PVM_ARCH
>hello
```

1.2 Windows

- First get the PVM3.4.x software:

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

- Extract the PVM files with `winzip`. The Windows version of PVM has a `setup` program to install the software.
- Extract the CPPVM files with `winzip`.
- Copy the sempahore server `cppvms.exe` from `cppvm/bin/WIN32` to `$PVM_ROOT/bin/WIN32`.
- Copy the `cppvm/examples/bin/WIN32/*` files to `$PVM_ROOT/bin/WIN32`
- Start the `$PVM_ROOT/bin/WIN32/hello` program.

Chapter 2

Introduction

CPPVM (C Plus Plus PVM) is a C++ class library for message passing. It provides an easy–to–use C++ interface to the parallel virtual machine system PVM[1]. CPPVM is available on many architectures, from Windows up to several UNIX derivatives. CPPVM allows to:

- Combine a heterogenous collection of computers
- Spawn and kill processes dynamically
- Detect failed processes and hosts
- Send/receive C++ objects
- Use distributed C++ objects
- Use C++ objects together with a message mailbox
- Write your own message C++ classes
- Use C++ templates
- Use standard template library (STL) classes
- Use semaphores, mutual exclusions, and barriers
- Use CPPVM together with existing PVM software

This manual describes how to use CPPVM, gives detailed information about the CPPVM classes and explains how to write your own message passing classes. A lot of examples – from the very simple "hello world" program up to more complex application/server programs – will guide you through the facilities of CPPVM.

[1] University of Tennessee, Knoxville TN., Oak Ridge National Laboratory, Oak Ridge TN., Emory University, Atlanta GA., Authors: J. J. Dongarra, G. E. Fagg, G. A. Geist, J. A. Kohl, R. J. Manchek, P. Mucci, P. M. Papadopoulos, S. L. Scott, and V. S. Sunderam

2.1 PVM

To work with CPPVM it is necessary to have a basic knowledge about the parallel virtual machine system PVM [Geist et al. 94]. For CPPVM PVM Version 3.4.x is needed. The latest PVM software can be obtained via the Internet:

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

PVM is a software package that permits a heterogeneous collection of hosts hooked together by a network to be used as a single large parallel virtual machine. By adding a host to the virtual machine, the PVM–daemon is started on that host. Processes can connect to their local PVM–daemon and become part of the virtual machine system. The processes can spawn other processes on every host in the system and pass messages to each other. PVM can send and receive data of standard C types like integers or arrays of standard C types.

For more detailed information about PVM read the documentation coming along with the PVM package.

2.1.1 PVM Installation

The installation of PVM is quite easy[2]:

- Set the environment variables `PVM_ROOT` and `PVM_ARCH` :
  - `PVM_ROOT`: Location of the `pvm3` installation directory. For a local installation as user `smith` the directory can be for example `/home/smith/pvm3`.
  - `PVM_ARCH`: Architecture of the host. The architecture can be determined using the shell script `pvmgetarch` in the directory `$PVM_ROOT/lib` (e.g. `LINUX` or `SUN4SOL2`).
- Execute `make` in the `$PVM_ROOT` directory.

The most comfortable way is to set the environment variables in your shell setup file (`.cshrc`, `.profile`, `.kshrc` or `.bashrc`). The scripts `$PVM_ROOT/lib/*.stub` can also be used to determine the architecture. Here is an example for the `.bashrc` file:

```
export PVM_ROOT=/home/smith/pvm3
#set PVM_ARCH explicit
#export PVM_ARCH=LINUX
#bashrc.stub determines the architecture and sets PVM_ARCH
source $PVM_ROOT/lib/bashrc.stub
```

[2]This description is for the UNIX versions, the Windows version has a setup program.

Example for the `.cshrc` file:

```
setenv PVM_ROOT /home/smith/pvm3
#cshrc.stub determines the architecture and sets PVM_ARCH
#source $PVM_ROOT/lib/cshrc.stub
#set PVM_ARCH explicit
setenv PVM_ARCH LINUX
```

If you have problems with pre-installed versions of PVM install your own version in your home directory.

2.1.2 Start PVM Programs

The PVM-daemon tries to start PVM–programs in the directories (in this order):

```
<WorkingDirectory>/pvm3/bin/$PVM_ARCH
$PVM_ROOT/bin/$PVM_ARCH
```

This path list can be changed with the host-file option `ep=` (see 2.1.5). The default working directory is `$HOME`. You can change the working directory by using the host-file option `wd=`.

This list allows PVM to find the programs on every host which are part of the virtual machine. When you write your own PVM programs, you have to move the executables to one of these directories. If your programs aren´t on a filesystem shared between the hosts, you must compile or copy them to each host.

Before you can start the programs, you have to start the PVM–daemon. This is done by running the PVM–console (see subsection 2.1.4).

2.1.3 Stop PVM Programs

There are two ways to shut down a parallel virtual machine:

- within PVM–programs call the function `pvm_halt`.
- run the PVM–console (see subsection 2.1.4) and type in the `halt` command.

`halt` will kill all PVM–processes including the console, shutdown PVM, exit all running PVM–daemons.

2.1.4 PVM **Console**

The local PVM–daemon of a host can be interactively controlled by the PVM-console. The console is located in the directory

$PVM_ROOT/console/$PVM_ARCH and can be started by typing:

```
> pvm
```

The PVM–daemon of the local host will be started if it is not already running. A prompt appears and the console allows the user to type in commands. The basic commands of the console are:

version	PVM version
add <hostname>	add a host to the virtual machine
delete <hostname>	delete a host from the virtual machine
conf	see the hosts which are part of the virtual machine
ps -a	see the processes running on the virtual machine
reset	kill all PVM processes except consoles, reset all the internal PVM tables and message queues
quit	exit console, leaving daemons and processes running
halt	kill all PVM–processes including console, shutdown PVM, exit all PVM–daemons

2.1.5 PVM **Host-File**

The host-file defines the configuration of a parallel virtual machine. The host-file is only needed when special settings have to be used for the hosts of the virtual machine. For example special locations of the PVM daemon or different user logins.

The host-file is a text file and is used as an argument of the PVM console:

```
> $PVM_ROOT/console/pvm myhostfile.txt
```

When you are using a host-file, you have to call the console with the host-file before calling your PVM or CPPVM program. You can also use the host-file within your program (see subsection 2.2.5.1).

The syntax of a host-file is:

`<blank_line>`	Blank lines are ignored.
`# [<text>]`	Lines that begin with a # are comment lines.
`<hostname> [<options>]`	The host `<hostname>` is added to the virtual machine. The host options are described below and are optional.
`&<hostname> [<options>]`	The host `<hostname>` is not added to the virtual machine. When the host is added later on (e.g. within a PVM- or CPPVM-program) this options are set.
`* <options>`	Set default options for all following hosts in the host-file. They can be overwritten by another default option line.

The options for a hostname are separated by white spaces can can be:

`lo=<user>`	Specify alternative login name. Otherwise your login name on the start-up machine is used.
`so=pw`	PVM will prompt you for a password on this host. PVM uses `rsh` by default to start up remote PVM daemons. When `so=` is specified, PVM will use `rexec` instead.
`so=ms`	Specifies that a slave PVM daemon will be started manually on this host. This is useful if `rsh` and `rexec` network services are disabled but IP connectivity exists.
`dx=<name>`	Location of the PVM daemon. This allows you to specify a location other than the default for this host.
`ep=<path>`	Paths to user executables. Allows you to specify a series of paths to search down to find the requested files to spawn on this host. Multiple paths are separated by a colon. If `ep=` is not specified, then PVM looks in `$HOME/pvm3/bin/$PVM_ARCH` for the application task
`sp=<value>`	Relative computational speed of the host compared to other hosts in the configuration. This number is used for load balancing if PVM can choose the host for a spawned task. The range of possible values is 1 to 1000000 with 1000 as the default.
`bx=<script>`	Location of the debugger script on this host if debugging is requested in the spawn routine. The environment variable `PVM_DEBUGGER` can also be set. The default debugger script is `$PVM_ROOT/lib/debugger`.
`wd=<dir>`	Working directory in which all spawned tasks on this host will execute. The default is `$HOME`.
`ip=<name>`	Specifies an alternate name to resolve to the host IP address.

Here is an example host-file `myhostfile.txt`:

```
# add standard hosts
sun1
lin1
```

```
# set default options for following hosts
* ep=/super-project/bin:/even-better-project/bin
sun2

# replace default options with new values
* lo=goerzig so=pw
lin2

# hosts for (maybe) adding later but which need options when added
&win1 dx="C:/programs/pvm3.4/lib/pvmd"
&lin3 wd=$HOME/config sp=100000
&c64  sp=1
```

2.1.6 PVM Environment Variables

To work with PVM or CPPVM the user has only to set the environment variables `PVM_ROOT` and `PVM_ARCH` (see subsection 2.1.1). Here is a list of PVM environment variables, which may be set by the user:

`PVM_ROOT`	Root installation directory.
`PVM_ARCH`	Architecture name.
`PVM_EXPORT`	Names of environment variables to inherit through spawn. See also the `exportEnvVar` method of cppvmConnection in section 11.1.
`PVM_DPATH`	Default slave PVM daemon path.
`PVM_DEBUGGER`	Debugger script used by spawn.

The following variables are set by PVM and should not be modified:

`PVMSOCK`	Address of the PVM daemon local socket.
`PVMEPID`	Expected PID of a spawned task.
`PVMMASK`	Libpvm trace mask.

2.2 CPPVM

CPPVM is a C++ interface to PVM. It hides some details of PVM from the user and thus makes it even easier to write parallel programs. CPPVM allows to use C++ objects as messages and it provides distributed and mailbox C++ objects. A user can easily write new CPPVM C++ classes for message passing. CPPVM includes also semaphores, mutual exclusions, barriers, and a uniform exception handling. The CPPVM software can be obtained via the Internet:

`http://www.informatik.uni-stuttgart.de/ipvr/bv/cppvm`

A mailing list and the latest beta version can be found at:

`http://groups.yahoo.com/group/CPPvm`

The basic functionality of CPPVM is to pass messages between several processes running in parallel (see figure 2.1). The processes can run on a network of heterogenous architectures and different operating systems. The parallel virtual machine combines these hosts to just one machine. This allows a transparent message passing between all CPPVM processes running on these hosts. CPPVM offers special classes to make a process part of the parallel virtual machine.

Usually, the processes within CPPVM have a parent–child relationship. For example in figure 2.1 the process `P1` has spawned the process `P2` on host `A` and the process `P3` on host `B`. These relationships are used to define CPPVM message passing streams. In the example the process `P1` has opened a send stream to its child process `P2`. The message passing streams are used to send or receive C++ objects. Objects can be send blocking or nonblocking, i.e. wait until the other process has received the object or go on without waiting. The modes of the receive stream are blocking, nonblocking and timeout receive. A timeout receive tries to receive an object for a specified time, otherwise the process continues without having the object received. The explicit message passing is described in chapter 4.

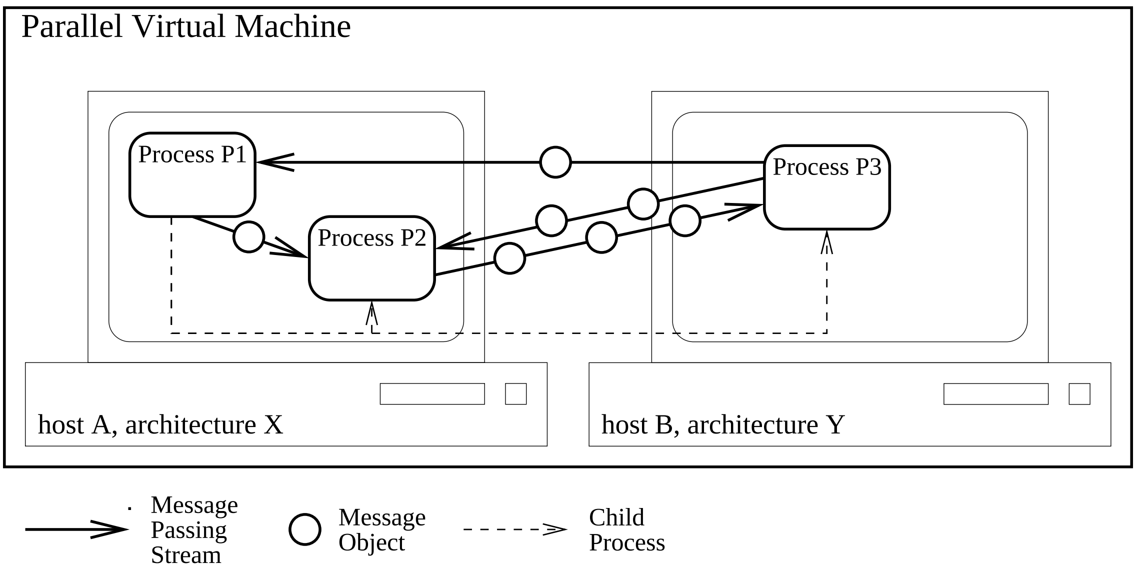


Figure 2.1: Explicit Message Passing.

Another possibility to exchange data between processes are the distributed objects (see figure 2.2). A distributed object is a C++ object existing in several processes.

A global database contains one instance of every distributed object. Processes can have a local instance of this object. A read method of the object can be used to update the local object, a write method updates the object in the database. Using distributed objects there is no need for send/receive streams. For a closer look on distributed objects see chapter 5.

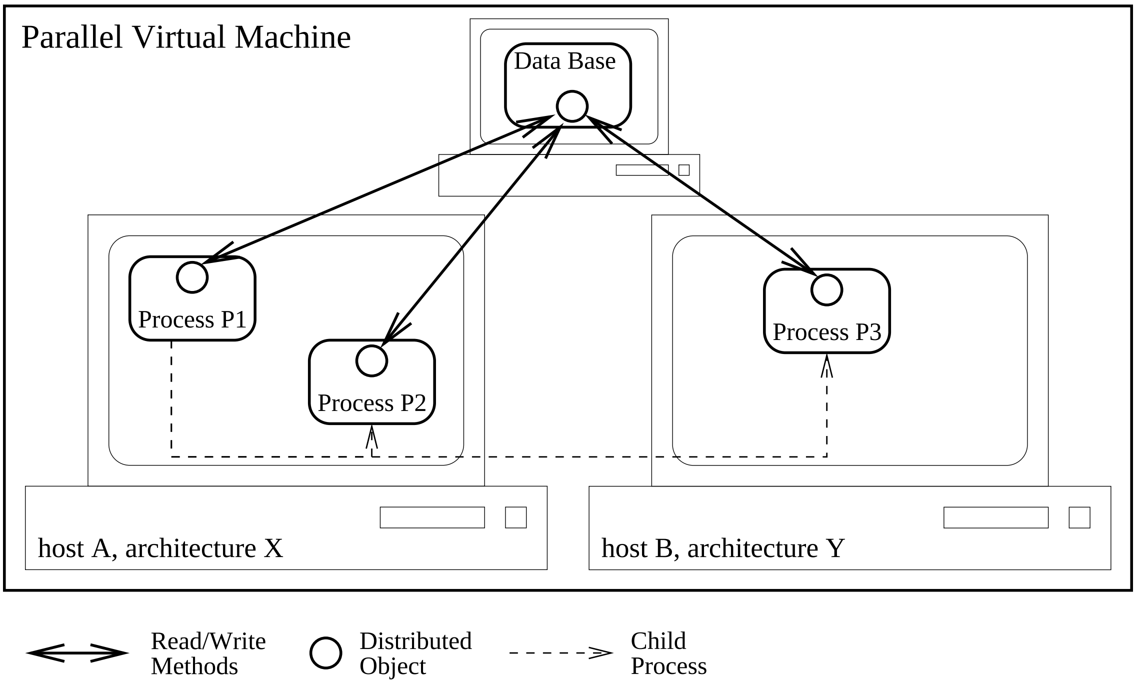


Figure 2.2: Distributed Objects.

Distributed objects are a small subset of mailbox objects. Mailbox objects allow to generate more than one instance in the global database (see figure 2.3). The local object can access the instances in the database. To learn more about mailbox objects see chapter 6.

CPPvm includes also an implementation of the semaphore concept proposed by Dijkstra [Dijkstra 68] a mutual exclusion concept, and barriers. These concepts can be used to synchronize parallel processes (see chapter 7).

2.2.1 CPPvm Installation

The main steps to install CPPvm on UNIX systems are:

- Install PVM version 3.4.x (see subsection 2.1.1).
- Ensure that the environment variables `PVM_ROOT` and `PVM_ARCH` are set.
- Execute `make` in the CPPvm directory. This will compile the CPPvm library and the semaphore server.

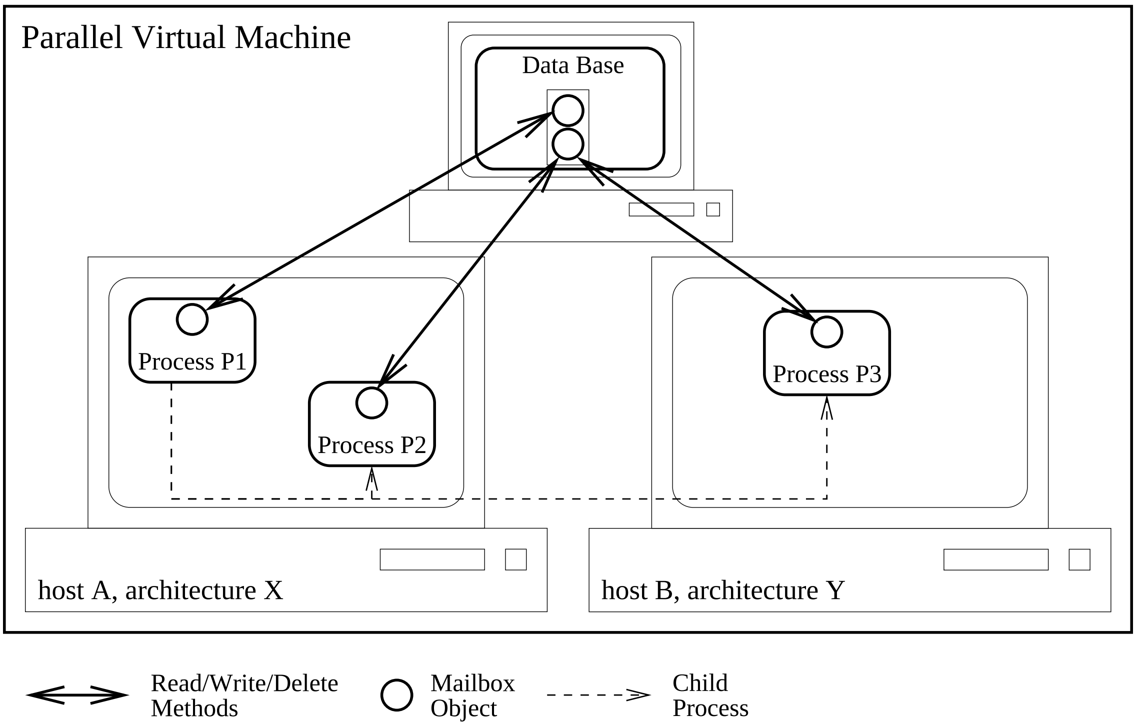


Figure 2.3: Mailbox Objects.

- Execute `make install` in the CPPvm directory. This will copy the semaphore server to `$HOME/pvm3/bin/$PMV_ARCH` and `$PVM_ROOT/bin/$PMV_ARCH`.

If you have problems to install CPPvm, follow the instructions described in the `README` file coming with the CPPvm package. For further information read also the comments in the makefiles and see subsection 2.2.5.

The Windows CPPvm library and semaphore server `cppvms` are already compiled. The library can be found in the directory `cppvm/lib/WIN32`. Copy the sempahore server `cppvms.exe` from `cppvm/bin/WIN32` to `$PVM_ROOT/bin/WIN32`. If you want to compile your own version of CPPvm you have to create your own Windows projects with your favorite compiler. See the file `cppvm/README` for more informations.

2.2.2 Start CPPvm Programs

As described in 2.1.2, the executable programs must be located by default in one of the directories

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

You can also change these paths using a host-files (see subsection 2.1.5 and 2.2.5.1).

In contrast to normal PVM–programs, a CPPVM–program starts the PVM–daemon automatically on the local host. Therefore there is no need to start the PVM–console, just execute the CPPVM–program. Of course you may start the console for interactive control of the local PVM–daemons.

2.2.3 Stop CPPVM Programs

As described in 2.1.3, there are two ways to shut down a parallel virtual machine:

- within CPPVM–programs call the method `halt`. The method is available for the classes cppvmConnecion, cppvmSpawnConnection, cppvmSpawnMultiConnection, and cppvmTask.
- run the PVM–console (see subsection 2.1.4) and type in the `halt` command.

2.2.4 Debugging and Monitoring

Parallel Programs are very nice as long as they do what you expect them to do. But unfortunately a lot of problems may occur: a spawned process can crash, messages can be miss-leaded, or the load balancing between the involved hosts is anything but optimal.

The next subsections describe some useful tools for debugging and monitoring of PVM/ CPPVM programs. There are a lot more tools available on the PVM homepage:

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

2.2.4.1 Debugging

PVM provides a very easy to use debugging interface. A process started from a console can use a debugger like every other program. But what about spawned child processes?

All processes spawned in CPPVM can be started within a debugger by adding the flag `PvmTaskDebug` to the cppvmSpawnConnectionor cppvmSpawnMultiConnection constructor call:

```
// spawn child 'myproggy' on 'sun1'
cppvmSpawnConnection child("myproggy", "",
                           PvmTaskHost+PvmTaskDebug, "sun1");
// spawn four children 'progs' on 'lin2'
cppvmSpawnMultiConnection child("progs", 4, "",
                                PvmTaskHost+PvmTaskDebug, "lin2");
```

The flag `PvmTaskDebug` is not inherited so it has to be added to each spawn call. The process is spawned in a debugger specified in the script file:

`$PVM_ROOT/lib/debugger`

By default the script opens an xterm and runs the program under a debugger specified in `$PVM_ROOT/lib/debugger2`. You can modify the `debugger` script to use your favorite debugger. For example by replacing the line

```
exec xterm -title "$hn:$argv[1]" -e $PVM_ROOT/lib/debugger2 $argv
```

by

```
exec xxgdb $argv
```

the debugger `xxgdb` is used instead.

The client/server architecture of X-Window systems must be configured to allow X programs running on other hosts (like `xxgdb`) to open a window on the local host. The access to the local server is enabled by the `xhost` command. E.g. by typing in on a local console

```
xhost sun2
```

the host `sun2` is allowed to access the local X server. There are two ways telling the spawned processes which X server to connect:

- set the `PVM_EXPORT` environment variable before starting the program, e.g.:

  ```
  setenv PVM_EXPORT DISPLAY
  ```

- use the `exportEnvVar` method of the class cppvmConnection within your program:

  ```
  cppvmConnection pvmConn;
  pvmConn.exportEnvVar("DISPLAY");
  // spawn child 'myproggy' on 'lin1'
  cppvmSpawnConnection child("myproggy", "",
                             PvmTaskHost+PvmTaskDebug, "lin1");
  ```

Ensure that the environment variable `DISPLAY` is set correctly (for example `setenv DISPLAY lin1:0.0`).

2.2.4.2 Tracer

The `tracer` program is part of the PVM package. It is located in the directory `$PVM_ROOT/bin/$PMV_ARCH`. If the binary is missing, you have to compile it:

```
>cd $PVM_ROOT
>make t
```

`tracer` allows to log detailed information about message passing with PVM. The program tries to connect to the local PVM daemon. Therefore the daemon must be started before calling `tracer`. You can do this for example by starting a PVM console:

```
>cd $PVM_ROOT/console/$PVM_ARCH
>pvm <hostfile>
```

The hostfile is an optional parameter (see subsection 2.1.5). After that, you can call the program:

```
>cd $PVM_ROOT/tracer/$PVM_ARCH
>tracer
```

`tracer` uses a command line interface. The logging data is specified by a trace mask. The mask can be modified with the `mask` command, e.g.:

```
tracer>mask + addhosts
tracer>mask - joingroup
```

The `help` command will give you more information about the use of the program. Once `tracer` is started, it is time to run the program for which you want to get the trace information, for example:

```
>cd $PVM_ROOT/bin/$PVM_ARCH
>hello
```

The trace output is logged in the file `/tmp/tracefile.<username>`. Be aware that the program `tracer` is also part of the Parallel Virtual Machine. By using the `halt` method, `tracer` will also be killed. It must be restarted if you want to trace your program again.

2.2.4.3 XPVM

XPVM is a graphical console and monitor for PVM. It provides a graphical interface to the PVM console commands and information, along with several animated views to monitor the execution of PVM programs. These views provide information about the interactions among tasks in a parallel PVM program, to assist in debugging and performance tuning.

XPVM can be obtained from the PVM homepage. For XPVM the Tcl/Tk package is needed. To install XPVM follow the instructions in the `xpvm/README` file. XPVM should be started before the program you want to inspect. If your program needs a hostfile (see subsection 2.1.5), copy your hostfile to `$HOME/.xpvm_hosts`. Now you can start XPVM:

```
>cd xpvm/src/$PVM_ARCH
>xpvm
```

XPVM logs the trace information in the file `/tmp/xpvm.trace.<username>`. You can also inspect the trace information online in the event history view. XPVM allows to start tasks and to add hosts, so you can start your parallel programs within XPVM. Be aware that XPVM is also part of the Parallel Virtual Machine. By using the `halt` method, `xpvm` will also be killed. To avoid this, do not use `halt` in your parallel program when monitoring it with XPVM. You can also halt the Virtual Machine by calling the menu entry `File->Halt PVM`.

2.2.5 Advanced Topics

2.2.5.1 CPPvm Console, Host-File and Environment Variables

CPPvm has no separate console, host-file or environment variables. Just use the console, host-file and the environment variables of PVM (see subsection 2.1.4, 2.1.5, and 2.1.6).

The host-file option can also be used within a CPPvm program. Use a cppvmConnection object to start the PVM daemon with arguments:

```
static char *pvmdArgv[] = {
  "/home/smith/host-file.txt",
};
int pvmdArgc = 1;
cppvmConnection myProcess(pvmdArgc, pvmdArgv);
```

Ensure that this is the first CPPvm process handling object used in your parallel program. If another process on the same host has created a cppvmConnection object first, this object will have no effect (the PVM daemon is already running).

There are several other options you can hand over to the PVM daemon. See the man pages of the PVM-functions `pvm_start_pvmd` and `pvmd` for more informations.

2.2.5.2 C++ Exceptions

CPPVM is using C++ exceptions to indicate internal errors. Exceptions are part of the new C++ standard. Up to now not all C++ compilers have implemented this feature. Some compiler runtime environments answer a thrown exception by segmentation faults or I/O traps.

By adding the flag `-DCPPvmDoNotUseExceptions` to `CFLAGS_ALL` in the file `cppvm/makefile` all CPPVM exceptions are replaced by error messages. Whenever an exception occurs, the error message is printed and the process stops with `exit(1)`.

By using this flag it is not possible to catch exceptions any more. Since CPPVM exceptions usually indicate only serious errors, this is a sufficient workaround for the most problems. The best solution is of course to use a compiler who knows how to handle exceptions.

2.2.5.3 C++ Templates and the STL

CPPVM uses C++ templates. If your C++ compiler does not support templates very well you can switch the templates off. This is done by adding `-DCPPvmDoNotUseTemplates` to `CFLAGS_ALL` in the file `cppvm/makefile`. If you are using the CPPVM library in your program you have to add the flag `-DCPPvmDoNotUseTemplates` when compiling your program as well. This is necessary due to CPPVM templates are defined in a header file, which will be implicitly included by your program.

The flag `-DCPPvmDoNotUseTemplates` disables also support for the standard template library (STL). If you want to keep template support but not STL support, use the flag `-DCPPvmDoNotUseStl` instead.

The STL classes bitset, slist, and valarray are not supported by default. This is due to limitations of the current STL implementations (see subsection 11.17.1). To enable support for these classes, use the compiler switches `-DCPPvmUseStlBitset`, `-DCPPvmUseStlSlist`, and `-DCPPvmUseStlValarray`. After that recompile the CPPVM library. You have to use these switches in your program as well.

2.2.5.4 Standard C Include Files

CPPVM uses by default the namespace-aware C++-style include files (e.g. `<cstdio>`, `<ctime>`, `<cstring>`, etc.). If you have to include the C-style include files (`<stdio.h>`, `<time.h>`, `<string.h>`, etc.) due to compiler restrictions you must add the compiler switch `-DCPPvmUseStandardCIncludeFiles` to `CFLAGS_ALL`. You have to add this flag when compiling your program as well, since this affects include files.

If you want to use the C++-style include files, but you have to use some selected C-style include files, you have to add one of these compiler switches to `CFLAGS_ALL`:

```
-DCPPvmUseStandardCIncludeFileCtype
```

```
-DCPPvmUseStandardCIncludeFileIomanip
-DCPPvmUseStandardCIncludeFileIostream
-DCPPvmUseStandardCIncludeFileLimits
-DCPPvmUseStandardCIncludeFileStdarg
-DCPPvmUseStandardCIncludeFileStdio
-DCPPvmUseStandardCIncludeFileStdlib
-DCPPvmUseStandardCIncludeFileString
-DCPPvmUseStandardCIncludeFileTime
```

By using `-DCPPvmUseStandardCIncludeFileIomanip`, for example, the C-style include file `iomanip.h` is used instead of `iomanip`. Again you have to add this flag when compiling your program as well.

2.2.5.5 Standard Namespace `std`

By default CPPvm uses several types in the namespace `std`:

```
using std::ostream;
using std::istream;
using std::cout;
using std::cin;
using std::cerr;
using std::endl;
using std::bitset;
using std::complex;
using std::deque;
using std::list;
using std::map;
using std::multimap;
using std::multiset;
using std::pair;
using std::priority_queue;
using std::queue;
using std::set;
using std::slist;
using std::stack;
using std::string;
using std::valarray;
using std::vector;
```

If your compiler has problems with the `using` commands, you can switch them off by adding `-DCPPvmDoNotUseStdNamespace` to `CFLAGS_ALL`. Be aware that this can cause type name problems (e.g. the compiler can complain that `ostream` is an unknown type).

2.2.5.6 Multithreading

CPPVM is not threadsafe. First steps have been done in this direction but it is possible that there will occur problems in multithreaded applications. You can try the multithreaded version of CPPVM on UNIX systems with POSIX–threads and on Windows systems with the MFC classes. To do this, add the flag `-DCPPvmThread` to `CFLAGS_ALL` in the file `cppvm/makefile` and recompile the CPPVM library and the semaphore server. It is also necessary to use the flag `-DCPPvmThread` when compiling own programs, since the flag is used in some CPPVM header files.

In the multithread–version all message passing methods are locked against each other using two mutual exclusion (mutex) variables. The first mutex locks `cppvmMutex` function calls against each other (see section 11.10). The second mutex locks the CPPVM message passing of the threads within one process (not the message passing of all CPPVM–processes on the current host).

This causes a restriction: Two threads in one process cannot use blocking send/receive methods to pass messages among them. The send method of one thread will lock the mutex variable of this process and does not allow to enter the receive method of the other thread.

If you want to pass messages between two threads within one process, you have to use nonblocking send/receive methods.

2.2.5.7 Boolean Type

The boolean type is also part of the new C++ standard. But again, not all C++ compilers have implemented this new type `bool`. When the compilation stops at a line with a boolean command, the compiler probably does not know the type `bool`.

Add the flag `-DCPPvmCompilerWithoutBoolType` to `CFLAGS_ALL` in the file `cppvm/makefile` to define the type `bool`, `true`, and `false` within CPPVM.

Chapter 3

CPPvm Classes and Functions

The classes and functions of CPPvm can be grouped by their functionality. First of all, the processes must get part of the parallel virtual machine. This is done by using the process handling CPPvm classes. They enable processes to connect to PVM and to spawn child processes. Furthermore send and receive stream classes are needed for explicit message passing. The messages objects resp. the distributed objects are CPPvm classes as well. Additionally a class and a function support the synchronization of processes. The notification classes inform about modifications of the virtual machine. Finally the exception handling classes are used for CPPvm exceptions.

This chapter should give you a short impression of the CPPvm class library – why the class hierarchy is designed this way and for what the classes are used to. A detailed description of the CPPvm classes can be found in chapter 11.

3.1 Process Handling Classes

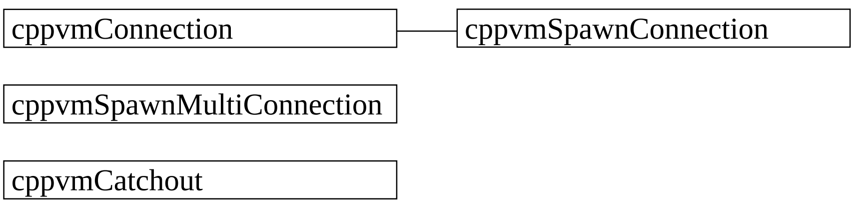


Figure 3.1: Process handling classes.

There are several classes to handle the processes in the parallel virtual machine:

- cppvmConnection: This class is used to connect process to PVM.

- cppvmSpawnConnection: This class allows to spawn a child process. Additionally the calling process is connected to PVM.
- cppvmSpawnMultiConnection: Objects of this class can spawn several child processes. The calling process is connected to PVM.
- cppvmCatchout: The output of spawned processes can be redirect by this class.

By creating an object of the class cppvmConnection, the process connects to the parallel virtual machine. From now on the process can use the message passing features of CPPVM. Child processes are spawned by using the classes cppvmSpawnConnection and cppvmSpawnMultiConnection. The output of these child processes can be redirected (e.g. to standard out or into a file) by using a cppvmCatchout object before spawning the processes.

3.2 Stream Classes

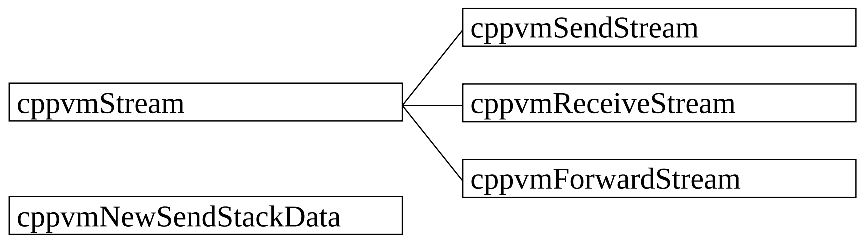


Figure 3.2: Stream classes.

The message passing stream classes are:

- cppvmSendStream: Send data.
- cppvmReceiveStream: Receive data.
- cppvmForwardStream: Forward received data.
- cppvmNewSendStackData: Allocate memory for `PvmDataInPlace` sending of external (non-CPPVM) classes (see section 9.1 and section 11.8)

The stream classes are used for explicit message passing. Each process can define streams to other processes to exchange data. The processes can send blocking or nonblocking. The receive modes are blocking, nonblocking and timeout receive. The forward stream redirects a incoming message to another process without receiving the message.

3.3 Message Classes

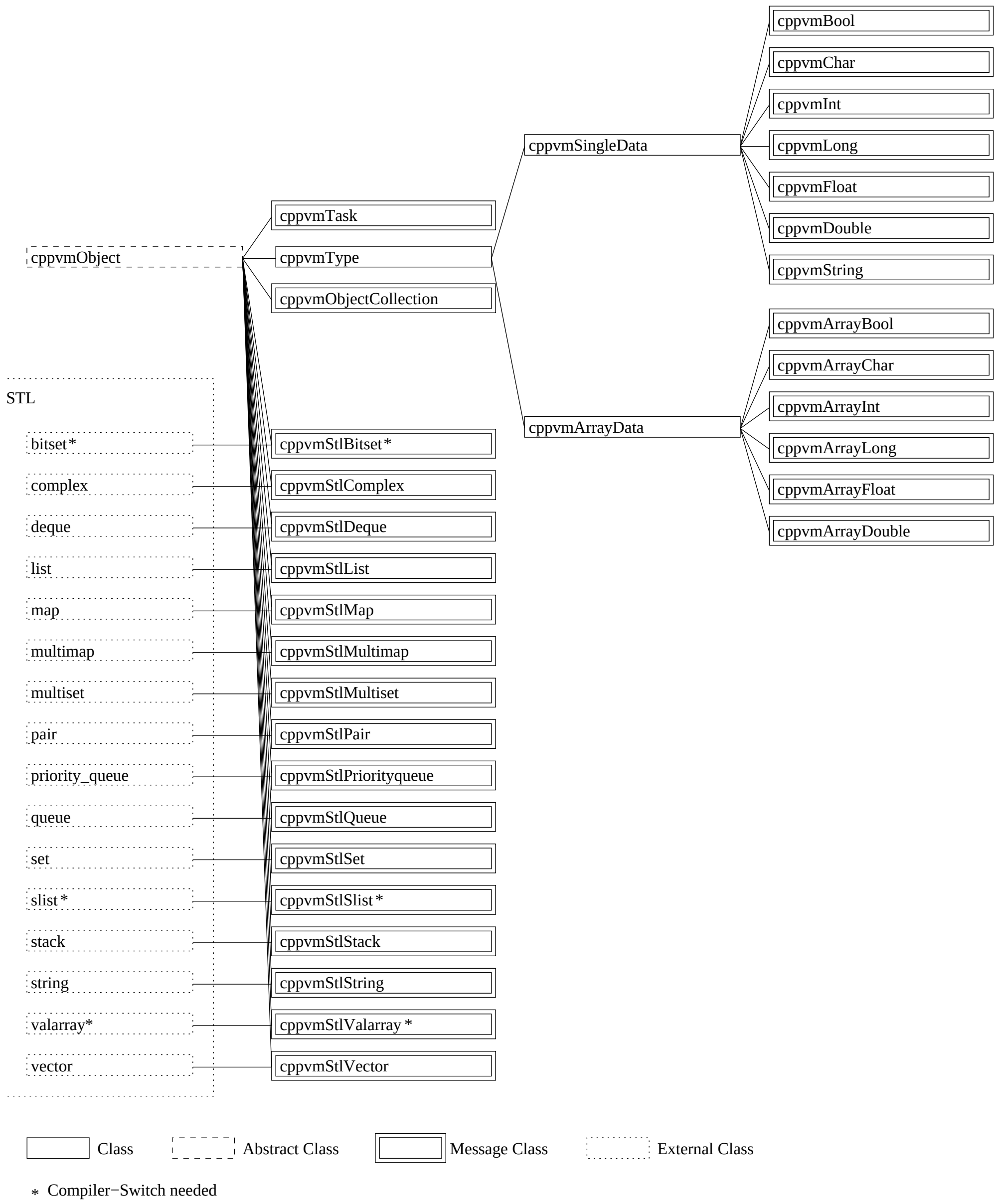


Figure 3.3: Message classes.

The message classes can be used for explicit message passing as well as for distributed and mailbox objects. The cppvmSingleData and cppvmArrayData classes can be used equivalent to the standard C++ data types. All message classes must be derived from the class cppvmObject.

Figure 3.3 shows the hierarchy of transferable message classes. The message classes are:

- cppvmObject: Base class for all message classes.
- cppvmObjectCollection: Collection for message classes.
- cppvmSingleData: Standard basic types: cppvmBool, cppvmChar, cppvmInt, cppvmLong, cppvmFloat, cppvmDouble, cppvmString.
- cppvmArrayData: Standard array types: cppvmArrayBool, cppvmArrayChar, cppvmArrayInt, cppvmArrayLong, cppvmArrayFloat, cppvmArrayDouble.
- cppvmTask: The description of a PVM process. Objects of this class can be used to open a send/receive stream to this process.
- cppvmStlBitset: Transferable STL bitset class (only available with a special compiler switch, see subsection 2.2.5.3)
- cppvmStlComplex: Transferable STL complex class
- cppvmStlDeque: Transferable STL deque class
- cppvmStlList: Transferable STL list class
- cppvmStlMap: Transferable STL map class
- cppvmStlMultimap: Transferable STL multimap class
- cppvmStlMultiset: Transferable STL multiset class
- cppvmStlPair: Transferable STL pair class
- cppvmStlPriorityqueue: Transferable STL priority_queue class
- cppvmStlQueue: Transferable STL queue class
- cppvmStlSet: Transferable STL set class
- cppvmStlSlist: Transferable STL slist class (only available with a special compiler switch, see subsection 2.2.5.3)
- cppvmStlStack: Transferable STL stack class
- cppvmStlString: Transferable STL string class

- cppvmStlValarray: Transferable STL valarray class (only available with a special compiler switch, see subsection 2.2.5.3)
- cppvmStlVector: Transferable STL vector class
- One can enhance this set of classes with new user defined message classes.

3.4 Process Synchronization Classes/Function

Figure 3.4: Process Synchronization classes and function.

Except for message passing there are three additional concepts to synchronize processes:

- cppvmSemaphore: Semaphore variable class.
- cppvmMutex: Mutual exclusion function.
- cppvmBarrier: Block calling processes.

Semaphores are a quite old approach to synchronize parallel tasks. Nevertheless they are very useful for example to solve consumer/producer problems. The mutual exclusion (mutex) concept can be applied e.g. for critical sections. Another concept to synchronize processes are barriers. The processes have to wait until all participating processes reached the barrier.

3.5 Notification Classes

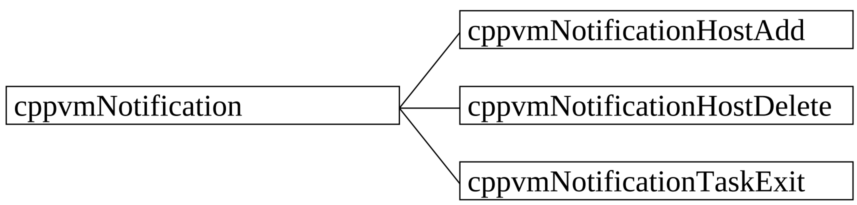


Figure 3.5: Notification classes.

The notification classes give informations about modifications of the virtual machine. By using these classes, PVM sends notification messages for special events to the calling process:

- cppvmNotification: Base class for all notifications.
- cppvmNotificationHostAdd: New hosts are added.
- cppvmNotificationHostDelete: Hosts are deleted or crashed.
- cppvmNotificationTaskExit: A PVM task exits or is killed.

3.6 Exception Handling Classes

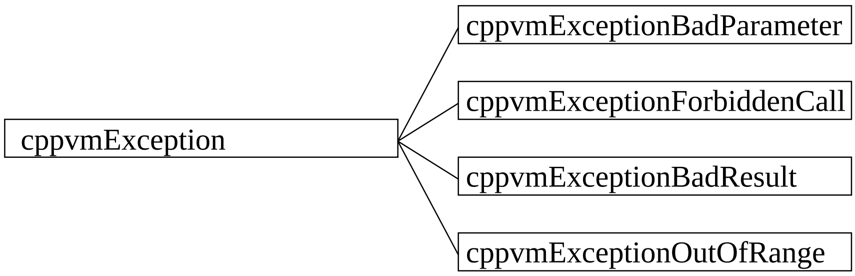


Figure 3.6: Exception Handling Classes.

The CPPvm exception handling classes are:

- cppvmException: Base class for all exceptions.
- cppvmExceptionForbiddenCall: Un-allowed method call.
- cppvmExceptionBadParameter: A wrong parameter was given to a method.
- cppvmExceptionOutOfRange: A parameter is not in the allowed range.
- cppvmExceptionBadResult: A method or function call within a CPPvm method returned an error.

CPPvm uses the C++ exception handling. Whenever an error occurs within CPPvm, an appropriate object of one of the exception handling classes is thrown. The user can catch the occurred errors if he wants to.

Chapter 4

Explicit Message Passing

The first example is of course a "hello world" program. The program consists of the two processes `hello` and `hello2` (see figure 4.1). The process `hello` is started on the host `lin1` and spawns the process `hello2` on the host `sun1`. `hello2` sends a message to `hello`. `hello` prints the message and PVM is halted.

The executable of `hello` is placed on the host `lin1` in the directory `$PVM_ROOT/bin/LINUX` and the executable `hello2` is located on `sun1` in the directory `$PVM_ROOT/bin/SUN4SOL2`. The example is started by calling the executable `hello`. Lets have a look on the source code:

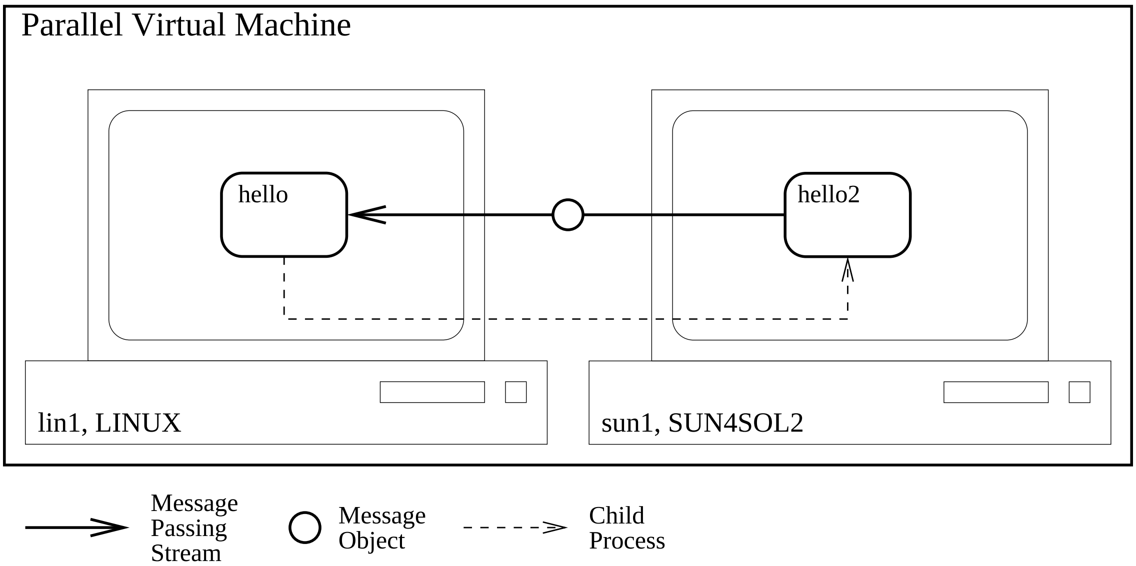


Figure 4.1: Once more in the history of computer programs: hello world.

hello.cpp

```
// cppvm 'hello world'
#include "cppvm.h"

int
main()
{
  // connect to pvm and
  // spawn child 'hello2' on 'sun1'
  cppvmSpawnConnection child("hello2", "",
    PvmTaskHost, "sun1");

  // receive descriptor
  // (blocking receive
  //  from child process)
  cppvmReceiveStream recStrm(child,
    CPPvmRBchild);

  // empty messages object
  cppvmString childMsg;

  // receive message
  recStrm >> childMsg;

  cout << childMsg << endl;

  // halt the virtual machine
  child.halt();

  return 0;
}
```

hello2.cpp

```
#include "cppvm.h"

int
main()
{
  // connect to pvm
  cppvmConnection pvmConn;

  // send descriptor
  // (blocking send
  //  to parent process)
  cppvmSendStream sendStrm(pvmConn,
    CPPvmSBparent);

  // message
  cppvmString sendMsg("hello world!");

  // send message
  sendStrm << sendMsg;

  return 0;
}
```

To use the functionality of CPPvm, every program must include `cppvm.h`. For explicit message passing you have to:

- Enroll the processes into the parallel virtual machine
- Define message passing streams
- Define message objects

To enroll a program into PVM, you just have to create a cppvmConnection, cppvmSpawnConnection or cppvmSpawnMultiConnection object. From now on the process is part of PVM until the last of these objects is deleted.

You can not use any CPPvm functionality before creating one of these objects. To avoid unexpected errors when using CPPvm, it is also a good idea to create a cppvmConnection at the beginning of the program.

The process `hello` starts its child process `hello2` by creating a cppvmSpawnConnection object. In the example the process `hello2` is started with an empty command line option `""`. The parameter `PvmTaskHost` indicates PVM that the process should be started on a host specified by the next parameter `"sun1"`.

By deleting the cppvmSpawnConnection object, the child process is not killed. If you want to kill a child process, you have to call the method `killChild()` of the class cppvmSpawnConnection. The parallel virtual machine (i.e. the PVM daemons on the participating hosts) is still running even when all processes are terminated. To avoid problems with these zombie parallel virtual machines, the machine should be halted by using the `halt()` method of the class cppvmConnection. By calling the `halt()` method, all PVM daemons and all PVM processes are killed. In the example `hello`, `hello2` and the PVM daemons on `lin1` and `sun1` are terminated.

The receive stream in `hello` is defined by the cppvmReceiveStream object `recStrm`. To establish a connection to the child process, the source process must be specified. This is usually done by using the CPPVM process handling classes cppvmConnection, cppvmSpawnConnection, and cppvmSpawnMultiConnection.

In the process `hello` the `child` object is used to specify the stream partner process. The second parameter specifies the receive mode and the source. The scheme is quite easy:

```
CPPvm<receive mode><source>
```

The possible receive modes are:

- Receive blocking (`RB`)
- Receive nonblocking (`RNB`)
- Receive timeout (`RT`)

The messages can be received from the sources:

- Parent process (`parent`)
- Child process (`child`)
- Task with a specified id (`tid`, `task`)
- Any process (`any`)
- Same process as the last received (`same`)

In our example the process `hello` defines a blocking receive stream from its child process `hello2` so the parameter has to be `CPPvmRBchild`.

The process `hello2` defines a send stream. Again the first parameter specifies the PVM stream partner process, in this case the parent process specified by a `pvmConnection` object. The second parameter has the scheme:

```
CPPvm<send mode><destination>
```

The send modes are:

- Send blocking (`SB`)
- Send nonblocking (`SNB`)

The destinations can be:

- Parent process (`parent`)
- Child process (`child`)
- Task with a specified id (`tid`, `task`)
- All processes which have joined a group (`group`)
- Set of processes (`taggedChild`, `allChild`)
- Process from which a message was received (`back`)

In the example, the parameter `CPPvmSBparent` defines a blocking send to the parent process.

The streams can now be used to transfer every object derived from cppvmObject. This is easily done like using standard I/O streams: The operator << is used to send objects, the operator >> is used for receiving objects. It is also possible to send or receive more than one object:

```
sendStrm << stringObj << integerObj << floatArray;
recStrm >> longObj >> charArray;
```

The cppvmString in process `hello2` is set to `"hello world!"` and send to the process `hello`. The message is received into the object `childMsg`, and printed to `cout`.

The next sections will explain more specific examples for explicit message passing. If you just want to get a short overview how to write CPPVM programs, you can continue with chapter 5.

4.1 Send/Receive Standard C++ types

CPPVM allows to apply the standard C++ types `bool`, `char`, `double`, `float`, `int` and `long` as well as constants[1] for explicit message passing. It is not possible to use standard C++ type variables or constants as distributed or mailbox variables. Standard C++ types

[1]Constants can of course only be send and not received because they can not change their values.

and constants can also not be used for message passing with the collection cppvmObjectCollection. In these cases, use the corresponding CPPVM classes (cppvmChar, cppvmDouble, ...) instead. Use the CPPVM classes cppvmString, cppvmArrayChar, cppvmArrayDouble etc. for strings and arrays.

cpptypes.cpp

```
// explicit message passing with
// standard C++ types:
// bool, char, double, float, int, long
// and constants
#include "cppvm.h"

int
main()
{
  // spawn child process
  cppvmSpawnConnection child("cpptypes2");

  // receive descriptor
  // (blocking receive
  //  from child process)
  cppvmReceiveStream recStrm(child,
    CPPvmRBchild);

  bool bVar;
  char cVar;
  double dVar;
  float fVar;
  int iVar;
  long lVar;
  int iCVar;
  float fCVar;
  double dCVar;

  recStrm >> bVar >> cVar >> dVar
        >> fVar >> iVar >> lVar
        >> iCVar >> fCVar >> dCVar;

  cout << "received bVar: " << bVar
     << " cVar: " << cVar
     << " dVar: " << dVar
     << " fVar: " << fVar
     << " iVar: " << iVar
     << " lVar: " << lVar
     << " iCVar: " << iCVar
     << " fCVar: " << fCVar
     << " dCVar: " << dCVar
     << endl;

  // halt the virtual machine
  child.halt();

  return 0;
}
```

cpptypes2.cpp

```
#include "cppvm.h"

#define iCVar 5
#define fCVar 6.6f
#define dCVar 7.7

int
main()
{
  // connect to pvm
  cppvmConnection pvmConn;

  // send descriptor
  // (non-blocking send
  //  to parent process)
  cppvmSendStream sendStrm(pvmConn,
    CPPvmSNBparent);

  bool bVar = true;
  char cVar = (char)97;
  double dVar = 1.1;
  float fVar = 2.2;
  int iVar = 3;
  long lVar = 4;

  sendStrm << bVar << cVar << dVar
         << fVar << iVar << lVar
         << iCVar << fCVar << dCVar;

  return 0;
}
```

The standard C++ types (`bool`, `int`, ...) are internally mapped to the corresponding CPPVM classes (cppvmBool, cppvmInt, ...). Therefore the modification of a class message tag e.g. for cppvmFloat will affect the message passing of the corresponding standard C++

type `float` (method `setCPPvmClassMsgTag`, see section 4.7). Object message tags do not affect standard C++ types.

4.2 Standard Template Library Classes

You can use the standard template library (STL) classes bitset, complex, deque, list, map, multimap, multiset, pair, priority_queue, queue, set, slist, stack, string, valarray, and vector directly for explicit message passing (see subsection 11.17.1 for limitations):

stl.cpp

```
// simple stl example

#include "cppvm.h"

// stl namespace
using namespace std;

int
main()
{
  // connect to pvm
  cppvmConnection pvmConn;

  vector<int> vect;

  // spawn child process
  cppvmSpawnConnection child("stl2");

  // receive descriptor (blocking
receive from child process)
  cppvmReceiveStream recStrm(child,
CPPvmRBchild);

  // receive object from child process
  recStrm >> vect;

  // halt the virtual machine
  pvmConn.halt();

  return 0;
}
```

stl2.cpp

```
#include "cppvm.h"

// stl namespace
using namespace std;

int
main()
{
  // connect to pvm
  cppvmConnection pvmConn;

  vector<int> vect(3);

  // send descriptor (blocking
send to parent process)
  cppvmSendStream sendStrm(pvmConn,
CPPvmSBparent);

  // send object to parent process
  sendStrm << vect;

  return 0;
}
```

The template parameters of the STL classes can be standard C++ types (`bool`, `char`, ...) as well as classes derived from cppvmObject. This list can be enhanced by writing new `cppvmTransferTemplate` functions (see chapter 9).

The template type of the sending object must be the same as the template type of the receiving object. E.g. do not try to receive an object of type `deque<int>` when sending an object of type `deuqe<double>`.

The STL classes (complex, deque, ...) are internally mapped to the corresponding CPPvm classes (cppvmStlComplex, cppvmStlDeque, ...). So like for standard C++ classes the modification of a class message tag e.g. for cppvmStlMap will affect the message passing of the corresponding STL class `map` (see section 4.7). Object message tags do not affect the STL classes.

See subsection 11.17.1 for limitations of the CPPvm STL support.

4.3 Send/Receive Process Descriptions

To set up a CPPvm stream it is necessary to specify the process to connect to. This is usually done by objects of the classes cppvmConnection, cppvmSpawnConnection, and cppvmSpawnMultiConnection. Objects of these classes cannot be transfered because they contain information about the relationship (parent/child) to other processes. These informations would be false in the context of other than the current process.

Now consider the relationship shown in figure 4.2. Process `peter` is the parent process of `paul`, `paul` of `mary`. Process `peter` wants to receive an object from process `mary`. But the receive stream needs a process handling object of the process `mary` to connect to this object. The process `paul` can not send its child description of process `mary` because it is of class `cppvmSpawnConnection`. So how can the process `peter` send an object to `mary`?

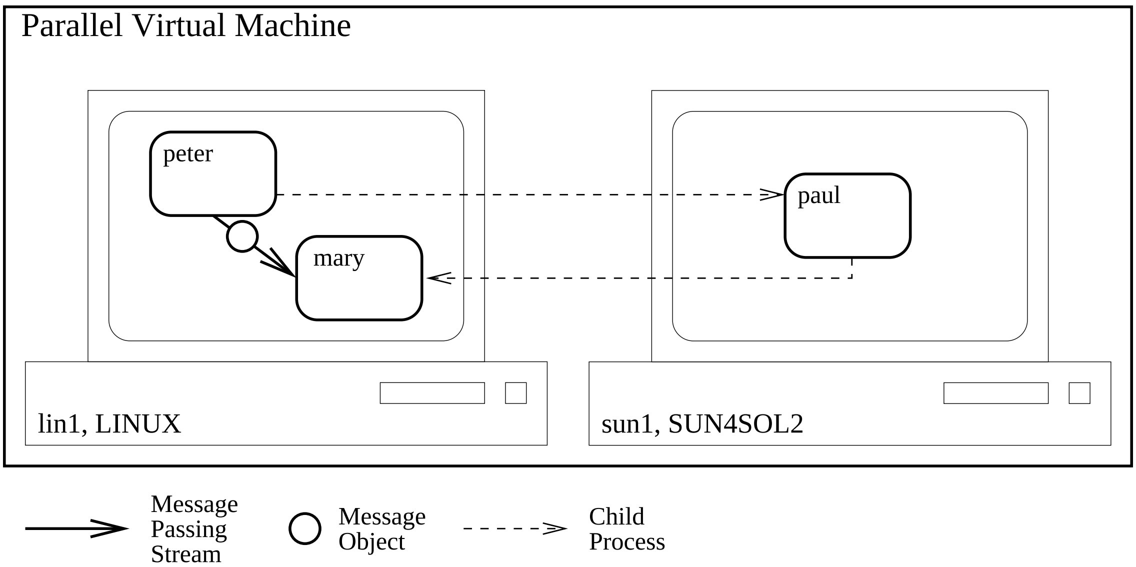


Figure 4.2: Explicit message passing between far relatives.

There are several possibilities to solve this problem:

- use the class cppvmTask to transfer informations about a process
- specify a process by its PVM task id

- add processes to a group (see section 4.4)

The classes cppvmConnection, cppvmSpawnConnection and cppvmSpawnMultiConnection can generate objects of the class `cppvmTask` by using the methods `getMyTask()`, `getParentTask()` and `getChildTask()`. These objects can be used to connect to other processes.

The process `paul` sends a cppvmTask description of its child process `mary` to `peter`. With this object `peter` is able to open a send stream to `mary`. On the other hand `mary` opens a receive stream to receive from any process.

peter.cpp

```
#include "cppvm.h"

int
main()
{
  // connect to pvm
  cppvmConnection pvmConn;

  // direct all couts of the
  // children to stdout
  cppvmCatchout catchOut(CPPvmStdout);

  // spawn child 'paul' on 'sun1'
  cppvmSpawnConnection child("paul",
    "", PvmTaskHost, "sun1");

  // receive descriptor
  // (blocking receive
  //  from child process)
  cppvmReceiveStream recStrm(child,
    CPPvmRBchild);

  // empty task description
  cppvmTask taskDesc;

  // receive task description
  recStrm >> taskDesc;

  // send descriptor
  // (blocking send
  //  to task)
  cppvmSendStream sendStrm(taskDesc,
    CPPvmSBtask);

  // message
  cppvmString sendMsg("hello mary!");

  // send message
  sendStrm << sendMsg;

  return 0;
}
```

paul.cpp

```
#include "cppvm.h"

int
main()
{
  // connect to pvm
  cppvmConnection pvmConn;

  // spawn child 'mary' on 'lin1'
  cppvmSpawnConnection child("mary",
    "", PvmTaskHost, "lin1");

  // let pvm choose the host
  cppvmSpawnConnection child("mary");

  // send descriptor
  // (blocking send
  //  to parent process)
  cppvmSendStream sendStrm(pvmConn,
    CPPvmSBparent);

  // task description
  cppvmTask taskDesc;

  // set task description
  taskDesc = child.getChildTask();

  // send task description
  sendStrm << taskDesc;

  return 0;
}
```

mary

```
#include "cppvm.h"

int
main()
{
  // connect to pvm
  cppvmConnection pvmConn;

  // receive descriptor
  // (blocking receive
  //  from any process)
  cppvmReceiveStream recStrm(CPPvmRBany);

  // empty messages object
  cppvmString msg;
 ...
```

mary

```
...
  // receive message
  recStrm >> msg;

  cout << msg << endl;

  // halt the virtual machine
  pvmConn.halt();

  return 0;
}
```

Every PVM process has a unique task id. A transfer stream can also be specified by this task id. The easiest way to get the task id of a process is to call the method `getTidOfTask(char *)`. This method returns the task id (integer) for the task with the given name[2].

Only the task spawned within PVM can be asked for their task id with this method. PVM does not know the names of the processes started by the user. In the second example process `peter2` is executed by the user. Therefore the method `getTidOfTask("peter2")` will return a negative integer, i.e. the task id can not be determined. The processes `paul2` and `mary2` are spawned within PVM. They can be asked for their task id's.

By using task id's, the program gets even easier:

[2]On Windows systems add the suffix `.exe` to the name of the process, e.g. `getTidOfTask("paul2.exe")`.

peter2.cpp

```
#include "cppvm.h"

int
main()
{
  // connect to pvm
  cppvmConnection pvmConn;

  // direct all couts of the
  // children to stdout
  cppvmCatchout catchOut(CPPvmStdout);

  // spawn child 'paul2' on 'sun1'
  cppvmSpawnConnection child("paul2",
    "", PvmTaskHost, "sun1");

  int taskId=-1;

  while(taskId<0){
    taskId= child.getTidOfTask("mary2");
  }


  // (blocking send
  //  to task)
  cppvmSendStream sendStrm(taskId,
    CPPvmSBtid);

  // message
  cppvmString sendMsg("hello mary2!");

  // send message
  sendStrm << sendMsg;

  return 0;
}
```

paul2.cpp

```
#include "cppvm.h"

int
main()
{
  // spawn child 'mary2' on 'lin1'
  cppvmSpawnConnection child("mary2",
    "", PvmTaskHost, "lin1");

  return 0;
}
```

mary2.cpp

```
#include "cppvm.h"

int
main()
{
  // connect to pvm
  cppvmConnection pvmConn;

  // receive from any process
  // timeout every 1.2 seconds
  // recStrm(<mode>,<sec>,<usec>)
  cppvmReceiveStream recStrm(CPPvmRTany,
    1, 200000);

  // empty messages object
  cppvmString msg;

  // receive message
  recStrm >> msg;

 ...
```

mary2.cpp

```
...
  if(recStrm.received()){
    // write message
    cout << msg << endl;
  }
  else{
    cout << "timeout: no message"
         << " received." << endl;
  }

  // halt the virtual machine
  pvmConn.halt();

  return 0;
}
```

Again `peter2` spawns the process `paul2`, `paul` spawns `mary2`. `peter2` tries to get the task id of `mary2`. This is possible when `mary2` has connected to PVM. `peter2` sends a message to `mary2`. In the example `mary2` uses a timeout receive. The receive stream tries to receive a message for 1.2 seconds. If the message is received within this time, the process continues immediately. Otherwise the process continues after 1.2 seconds without having the message received. The result can be determined by the `received()` method.

4.4 Group Broadcast

Processes can be combined to groups. Every process can join a specific group by calling the method `joinGroup()`. A process can be a member of multiple groups simultaneously.

By using groups it is possible to send a message to all members of a group (see figure 4.3). Important is that broadcasted messages are not send to the caller, even if the calling process is member of the specified group.

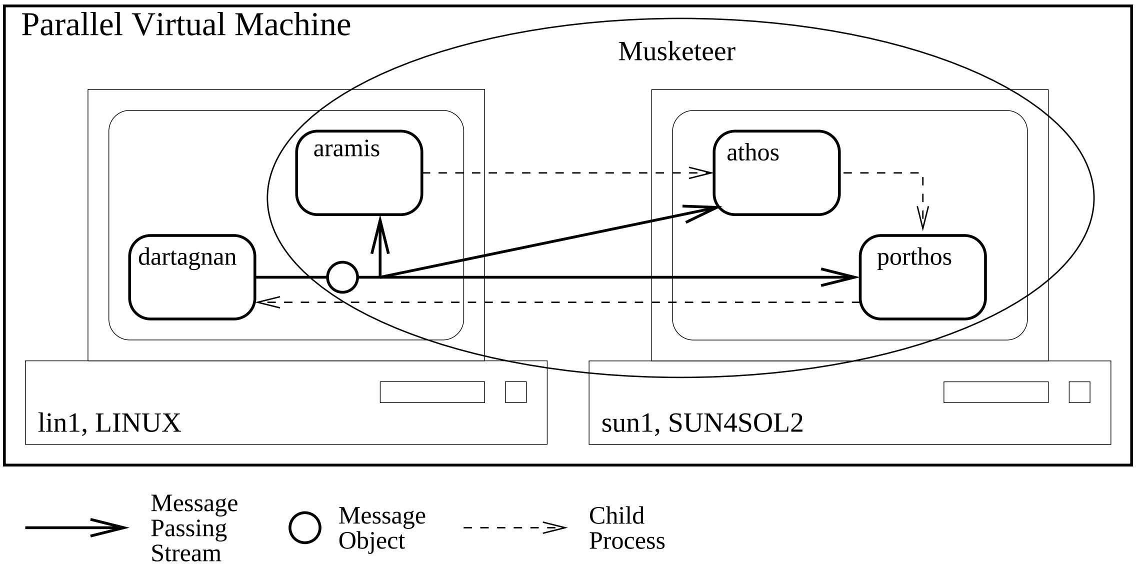


Figure 4.3: Group broadcast example.

In the example the processes `aramis`, `athos` and `porthos` join the group `Musketeer`. The process `dartagnan` sends a message non-blocking to all group members.

The child processes in the example are not spawned on a specific host. In this case PVM will select the most appropriate host. Due to that the distribution of the processes can differ from figure 4.3 on real hosts. New hosts can be added to the virtual machine with the method `addHost()` and deleted by using `delHost()`.

The distribution of the processes shown in figure 4.3 depends on the performance of the hosts. For example if `lin1` is much faster than `sun1`, it is possible that three processes are running on `lin1` and only one on `sun1`. The speed of the hosts can be set in the PVM host file. Please refer the PVM documentation for more details.

aramis.cpp

```
#include "cppvm.h"

int
main()
{
  // connect to pvm
  cppvmConnection pvmConn;

  // join the group "Musketeer"
  pvmConn.joinGroup("Musketeer");

  // add host "sun1" to pvm
  pvmConn.addHost("sun1");

  // spawn child 'athos'
  // let pvm choose the host
  cppvmSpawnConnection child("athos");

  // blocking receive
  // from any process
  cppvmReceiveStream recStrm(CPPvmRBany);

  // empty messages object
  cppvmString msg;

  // receive message
  recStrm >> msg;

  // write message
  cout << msg << endl;

  // halt the virtual machine
  pvmConn.halt();

  return 0;
}
```

athos.cpp

```
#include "cppvm.h"

int
main()
{
  // connect to pvm
  cppvmConnection pvmConn;

  // join the group "Musketeer"
  pvmConn.joinGroup("Musketeer");

  // spawn child 'porthos'
  // let pvm choose the host
  cppvmSpawnConnection child("porthos");

  // blocking receive
  // from any process
  cppvmReceiveStream
recStrm(CPPvmRBany);

  // empty messages object
  cppvmString msg;

  // receive message
  recStrm >> msg;

  return 0;
}
```

porthos.cpp

```
#include "cppvm.h"

int
main()
{
  // connect to pvm
  cppvmConnection pvmConn;

  // join the group "Musketeer"
  pvmConn.joinGroup("Musketeer");

  // spawn child 'dartagnan'
  // let pvm choose the host
  cppvmSpawnConnection child("dartagnan");

  // blocking receive
  // from any process
  cppvmReceiveStream recStrm(CPPvmRBany);

  // empty messages object
  cppvmString msg;

  // receive message
  recStrm >> msg;

  return 0;
}
```

dartagnan.cpp

```
#include "cppvm.h"

int
main()
{
  // connect to pvm
  cppvmConnection pvmConn;

  // (non-blocking send
  //  to all group members)
  cppvmSendStream sendStrm("Musketeer",
                           CPPvmSNBgroup);

  // message
  cppvmString sendMsg("One for all - all for
one!");

  cout << sendMsg;

  // send message
  sendStrm << sendMsg;

  return 0;
}
```

4.5 Multicast

Another possibility to send messages to multiple processes is multi-casting. A process can spawn several child processes at one time, and send messages to them (see figure 4.4).

The process `dartagnan2` spawn three processes `musketeer` by using the class cppvmSpawnMultiConnection. After that, `dartagnan2` sends a message to all children. Each child receives the message and sends a message back to the sender.

The class cppvmSpawnMultiConnection allows also to send messages to a subset of spawned children. See section 11.3 and section 12.12 for more details.

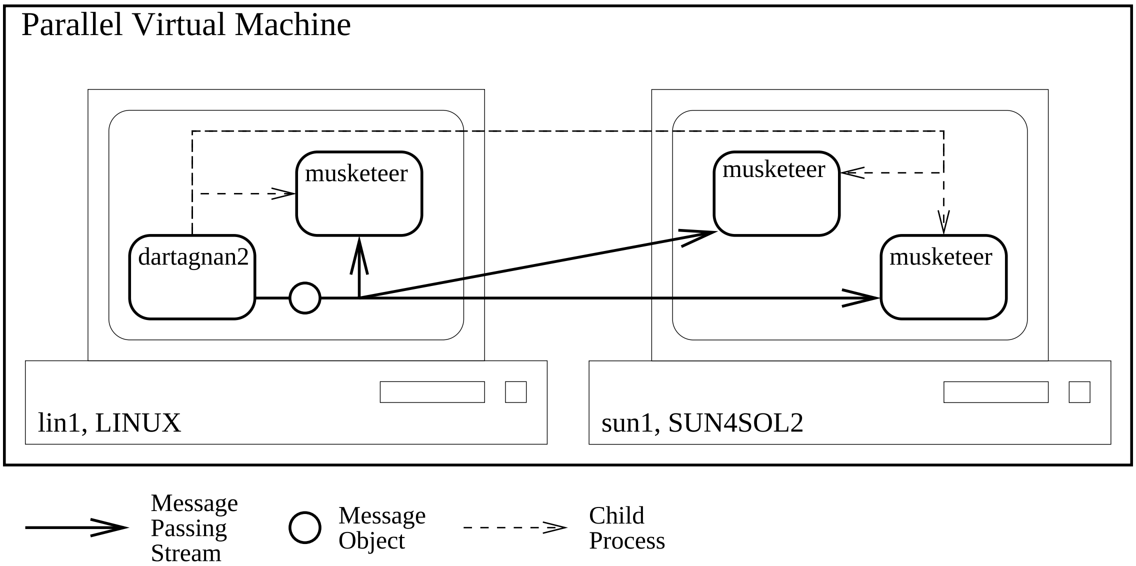


Figure 4.4: Multicast example.

dartagnan2.cpp

```
#include "cppvm.h"

int
main()
{
  // connect to pvm
  cppvmConnection pvmConn;

  // spawn three child processes
  cppvmSpawnMultiConnection
    children("musketeer", 3);

  // blocking send to all
  // children
  cppvmSendStream sendStrm(children,
    CPPvmSBallChildren);

  // message
  cppvmString msg("One for all!");

  cout << msg << endl;

  // send message
  sendStrm << msg;

  // blocking receive from any
  // process
  cppvmReceiveStream recStrm(CPPvmRBany);

  int i;
  for(i=0;i<3;i++){
    recStrm >> msg;
    cout << msg << endl;
  }

  // halt pvm
  pvmConn.halt();

  return 0;
}
```

musketeer.cpp

```
#include "cppvm.h"

int
main()
{
  // connect to pvm
  cppvmConnection pvmConn;

  // receive from parent process
  cppvmReceiveStream recStrm(pvmConn,
    CPPvmRBparent);

  // empty messages object
  cppvmString msg;

  // receive message
  recStrm >> msg;

  // send message back
  cppvmSendStream sendStrm(recStrm,
    CPPvmSBback);

  msg = "All for one!";

  sendStrm << msg;

  return 0;
}
```

4.6 Message Passing Context

Within PVM/CPPvm it is possible to produce miss-leaded messages. This is especially true when working on large parallel programs, reusing old PVM/CPPvm programs or using parallel libraries from other developers.

There are several ways to produce miss-leaded messages. A "receive from any host" might catch more messages than it was supposed to do. A process waiting for a message with a certain message tag can receive a wrong message, if two different message types have the same message tag.

A message passing context is helpful to avoid these problems. A process can be spawned in new context. Messages sent within one context can not be received in another context.

The context of a process is determined when spawning the process. It is not possible to change the context later on. By default all new processes are spawned in only one context. Here are some examples how to use a context (see section 11.2 and section 11.3 for details):

```
// spawn child in new context
cppvmSpawnConnection child1("proggy1", "",
                            PvmTaskDefault, "",
                            CPPvmNewContext);
// get context of child1
int child1Context = child1.getContextId();

// spawn child2 into the same context as child1
cppvmSpawnConnection child2("proggy2", "",
                            PvmTaskDefault, "",
                            CPPvmGivenContext,
                            child1Context);

// create one new context and spawn all 100 children
// within this new context
cppvmSpawnMultiConnection children1("nodeprg1", 100, "",
                                    PvmTaskDefault, "",
                                    CPPvmNewContext,0
                                    CPPvmOneNewContext);
// spawn every child in a new context
cppvmSpawnMultiConnection children2("nodeprg2", 30, "",
                                    PvmTaskDefault, "",
                                    CPPvmNewContext,0
                                    CPPvmAllNewContext);
// spawn every child into the given context '333'
cppvmSpawnMultiConnection children3("nodeprg3", 20, "",
                                    PvmTaskDefault, "",
                                    CPPvmGivenContext, 333);
```

Please note that distributed objects and message mailbox objects are not bound to the context of a process. They can be accessed from any process connected to PVM.

Consider now following problem. An administrator "admin" spawns two servers ("gimme5" and "sayhi") and three application programs ("app"). Both servers are sending objects of the class cppvmString to the applications. The applications are waiting for two messages: one from "gimme5" and one from "sayhi". If the applications would use a "receive from any host" the messages can be miss-leaded. E.g. instead of getting a string from the server "gimme5" a string from "sayhi" can be received.

The solution is to spawn "gimme5" and "sayhi" in a different context. The receive command can now be transformed into a "receive from any host within a certain context". The applications will now receive only the messages from the right servers, even when both servers are sending the same object types.

The example will show you also some other CPPvm techniques. A distributed collection of cppvmTask objects is used to enable the applications access to the servers. Every application and server is performing an exception handling. These exceptions are reported to the administrator who can now handle these failed child processes.

adminis.cpp

```
// cppvm 'adminis'
// IMPORTANT: Multicasted (array of
tids) and broadcasted
//           (to group members)
messages are NOT send to
//           the caller even if the
caller is in the
//           array of tids resp. member
of the group

#include "cppvm.h"

int
main()
{
  // connect to pvm
  cppvmConnection pvmConn;

  ////////////////////////
  // start servers    //
  // every server gets //
  // its own context!  //
  ////////////////////////
```

adminis.cpp

```
  cout << "starting servers...";
  cout.flush();
  cppvmSpawnMultiConnection servers;
  servers.add(new
cppvmSpawnConnection("gimme5", "",

PvmTaskDefault, "",
CPPvmNewContext));
  servers.add(new
cppvmSpawnConnection("sayhi", "",

PvmTaskDefault, "",
CPPvmNewContext));

  // collection of server objects
  cppvmObjectCollection *serverCollection;
  // get the pvm task descriptions
of all servers
  serverCollection =
servers.getAllChildTasks();
  // servers collection as
distributed object

serverCollection->cppvmDistObj("servers");
  cout << "...done." << endl;

  ////////////////////////////////
  // start applications
  // all cout/cerr
  // will be redirected to stdout
  ////////////////////////////////
```

adminis.cpp

```
  cppvmCatchout catchOut(CPPvmStdout);
  // catchout in file
  //cppvmCatchout catchOut(CPPvmFile,
"out.txt");

  cout << "starting applications...";
  cout.flush();
  // string object
  cppvmString resultStr("");
  int numApps = 3;
  int numResults = 0;
  cppvmSpawnMultiConnection apps("app",
numApps);
  cout << "...done." << endl;

  // receive a string from any process
  cppvmReceiveStream recStrm(CPPvmRNBany);

  // distributed object to indicate
  // server/application errors
  cppvmInt errorNotify(0);
  errorNotify.cppvmDistObj("error notify");

  bool error = false;
  bool result;
  // leave the loop if
  // 1) all results are sent back
  // 2) a server or an application
  //    has set the "error notify" != 0

  while(numResults<numApps &&
        error == false){
    recStrm >> resultStr;
    result = recStrm.received();
    if(result==true){
      numResults++;
    }
    if(errorNotify.read()!=0){
      cout << "ADMIN: critical error in
server/application occurred!" << endl;
      error = true;
    }
  }
```

adminis.cpp

```
  if(error==true){
    // an application or an server
has
    // caused an error
    // get the error message from
message box
    // distributed objects are not
restricted to contexts
    cppvmString errStr("");
    errStr.cppvmDistObj("exception error
message");
    // and write it to error out
    cerr << "ADMIN: error occurred: "
         << errStr.read() << endl;
  }

  // kill all servers and
applications
  servers.killAllChildren();
  apps.killAllChildren();

//   unsigned int i;
//   for(i=0;
i<servers.numElems(); i++){
//     servers[i].killChild();
//   }
//   for(i=0; i<apps.numElems();
i++){
//     apps[i].killChild();
//   }

  cout << "admin: The End." << endl;

  // halt the virtual machine
  pvmConn.halt();

  return 0;
}
```

gimme5.cpp

```
// cppvm 'gimme5'

#include "cppvm.h"

void
gimme5(void)
{
  // connect to pvm
  cppvmConnection pvmConn;

  cppvmString recStr("");
  cppvmString resStr("5!");
  cppvmSendStream sendStrm;

  // blocking receive from any process
  cppvmReceiveStream recStrm(CPPvmRBany);

  // endless cycle
  while(1){
    // blocking wait until a message
arrives
    recStrm >> recStr;
    // send a message back to the sender
    sendStrm.set(recStrm, CPPvmSNBback);

    sendStrm << resStr;

    ///////////////////////////
    // uncomment the next line
    // to cause an exception
    //cppvmString((char*) NULL);
    ///////////////////////////
  }
}

void errorHandling(void){
  // an exception occurred
  cppvmConnection pvmConn;
  // fill distributed error message
  // distributed objects are not
restricted to contexts
  cppvmString errStr("");
  errStr.cppvmDistObj("exception error message");
  errStr.write("exception occurred in gimme5");
  // signal exception to administrator
  cppvmInt indicateError;
  indicateError.cppvmDistObj("error notify");
  // set error notifier
  indicateError.write(1);

  // endless wait (until admin has
  // killed all processes)
  int counter=0;
  while(1){
    counter++;
  }
}
```

gimme5.cpp

```
int
main(){
  try {
    gimme5();
  }
  catch (cppvmException& exception) {
    cerr << exception;
    errorHandling();
  }
  catch (...) {
    cerr << "exception occurred (not part of
cppvm)" << endl;
    errorHandling();
  }
  return 0;
}
```

sayhi.cpp

```
// cppvm 'sayhi'

#include "cppvm.h"

void
sayhi()
{
  // connect to pvm
  cppvmConnection pvmConn;

  cppvmString recStr("");
  cppvmString resStr("hi!");
  cppvmSendStream sendStrm;

  // blocking receive from any process
  cppvmReceiveStream recStrm(CPPvmRBany);

  // endless cycle
  while(1){
    // blocking wait until a message
arrives
    recStrm >> recStr;
    // send a message back to the sender
    sendStrm.set(recStrm, CPPvmSNBback);

    sendStrm << resStr;

    //////////////////////////
    // uncomment the next line
    // to cause an exception
    //cppvmString((char*) NULL);
    ////////////////////////////
  }
}

void errorHandling(void){
  // an exception occurred
  cppvmConnection pvmConn;
  // fill distributed error message
  // distributed objects are not
restricted to contexts
  cppvmString errStr("");
  errStr.cppvmDistObj("exception error message");
  errStr.write("exception occurred in sayhi");
  // signal exception to administrator
  cppvmInt indicateError;
  indicateError.cppvmDistObj("error notify");
  // set error notifier
  indicateError.write(1);

  // endless wait (until admin has
  // killed all processes)
  int counter=0;
  while(1){
    counter++;
  }
}
```

sayhi.cpp

```
int
main(){
  try {
    sayhi();
  }
  catch (cppvmException& exception) {
    cerr << exception;
    errorHandling();
  }
  catch (...) {
    cerr << "exception occurred (not part of
cppvm)" << endl;
    errorHandling();
  }
  return 0;
}
```

app.cpp

```
// cppvm 'app'

#include "cppvm.h"

void
app(void)
{
  // connect to pvm
  cppvmConnection pvmConn;

  // collection of server objects
  cppvmObjectCollection serverCollection;
  // servers collection as distributed
object
  serverCollection.cppvmDistObj("servers");

  cout << serverCollection << endl;

  int i;
  cppvmString orderStr("Your result, please:");
  cppvmString resStr("");
  cppvmSendStream sendStrmGimme5,
sendStrmSayhi;
  cppvmReceiveStream recStrmGimme5,
recStrmSayhi;

  // send to 'gimme5'
  sendStrmGimme5.set(serverCollection.
                  getTaskWithName("gimme5"),
                  CPPvmSNBtask);
  // receive blocking from any process
  // with context of 'gimme5'
  recStrmGimme5.set((serverCollection.
                  getTaskWithName("gimme5"))
                  .getContextId(),
                  CPPvmRBany);

  // send to 'sayhi'
  sendStrmSayhi.set(serverCollection.
                  getTaskWithName("sayhi"),
                  CPPvmSNBtask);
  // receive blocking from any process
  // with context of 'sayhi'
  recStrmSayhi.set((serverCollection.
                  getTaskWithName("sayhi"))
                  .getContextId(),
                  CPPvmRBany);

  for(i=0; i<3; i++){
    /////////////////////////////
    // contact server 'gimme5' //
    /////////////////////////////

    // send order
    sendStrmGimme5 << orderStr;

    /////////////////////////////
    // contact server 'sayhi'  //
    /////////////////////////////
```

app.cpp

```
    // send order
    sendStrmSayhi << orderStr;

    // although the receive
descriptors
    // accepts sent strings from
any
    // processes, the strings sent
by the servers
    // to the application cannot
be received in
    // a false order here!
    // This is because the servers
are started
    // in different contexts and
message passing
    // is only possible within one
context.

    // receiveFromAny result from
"sayhi" server
    recStrmSayhi >> resStr;
    cout << "Result from sayhi: " << resStr
<< endl;

    // receiveFromAny result from
"gimme5" server
    recStrmGimme5 >> resStr;
    cout << "Result from gimme5: " <<
resStr << endl;

  }

  //////////////////////////
  // uncomment the next line
  // to cause an exception
  //cppvmString((char*) NULL);
  ////////////////////////////

  // blocking send to parent
process
  cppvmSendStream
sendStrmParent(pvmConn,
CPPvmSBparent);

  sendStrmParent << resStr;
}
```

app.cpp

```
void errorHandling(void){
  // an exception occurred
  cppvmConnection pvmConn;
  // fill distributed error message
  // distributed objects are not
restricted to contexts
  cppvmString errStr("");
  errStr.cppvmDistObj("exception error message");
  errStr.write("exception occurred in app");
  // signal exception to administrator
  cppvmInt indicateError;
  indicateError.cppvmDistObj("error notify");
  // set error notifier
  indicateError.write(1);

  // endless wait (until admin has
  // killed all processes)
  int counter=0;
  while(1){
    counter++;
  }
}
```

app.cpp

```
int
main(){
  try {
    app();
  }
  catch (cppvmException& exception) {
    cerr << exception;
    errorHandling();
  }
  catch (...) {
    cerr << "exception occurred (not part of
cppvm)" << endl;
    errorHandling();
  }
  return 0;
}
```

4.7 Message Tags

Whether an incoming message can be received by an object or not is determined by a message tag. Usually there is only one message tag for all objects of a class. There are two methods to get the message tag of an object: `cppvmClassMsgTag` and `cppvmMsgTag`. By default both methods will return the same message tag.

It is possible to change class message tag during runtime. After changing the class message tag both methods will return the same, new value. This allows for example to differ between messages of the same class from several child processes:

```
cppvmInt intObj1;
cppvmInt intObj2;
cppvmReceiveStream recStrm(CPPvmRBany);
intObj1.setCPPvmClassMsgTag(555);
// receive from process one
recStrm >> intObj1;
intObj2.setCPPvmClassMsgTag(666);
// receive from process two
recStrm >> intObj2;
```

Since both objects have different tags, the incoming messages can not be miss-leaded. Another possibility to differ between messages from child processes is the message passing context as described above.

It is also possible to have different message tags for each object. In this case the method `cppvmClassMsgTag` will return the class message tag, and `cppvmMsgTag` returns the message tag of the object. Here is an example how to set and deactivate object message tags:

```
cppvmInt intObj1;
cppvmInt intObj2;
cppvmReceiveStream recStrm(CPPvmRBany);
intObj1.setCPPvmObjectMsgTag(555);
intObj2.setCPPvmObjectMsgTag(666);
// receive from process one and two
recStrm >> intObj1 >> intObj2;
// use class message tag again instead
// of the object message tag
intObj1.cppvmDeactivateObjectMsgTag();
```

In contrast to the example above the message tags can both be set before receiving the messages. This is due to the change of the message tag with the method `setCPPvmObjectMsgTag` affects only the object itself, not all objects of a certain class (like using `cppvmClassMsgTag`).

The program `newtag` is an example how to modify message tags during runtime (see section 12.14). For information how to use object and class message tags within user written classes please refer section 8.5.

4.8 Advanced Topics

There are several advanced possibilities for explicit message passing which you might find useful. The output of the child processes can be redirected into a file (see section 12.5). Messages can not only be sent or received but also forwarded to other processes (see section 12.16). It is possible to pass messages between processes which do not have a parent/(grand-)child relationship (see section 12.8).

For further information see also the description of the classes cppvmConnection, cppvmSpawnConnection and cppvmSpawnMultiConnection in chapter 11.

Chapter 5

Distributed Objects

All classes derived from `cppvmObject` can be used for distributed objects. Distributed CPPVM objects are objects which can be accessed from every process connected to PVM. Distributed objects consists of a local object that can be synchronized with an object in the PVM message mailbox, i.e. a global data base (see figure 5.1). Distributed objects can only share values of variables via the data base. It is not possible to share methods. All method calls for distributed objects are executed within the local process.

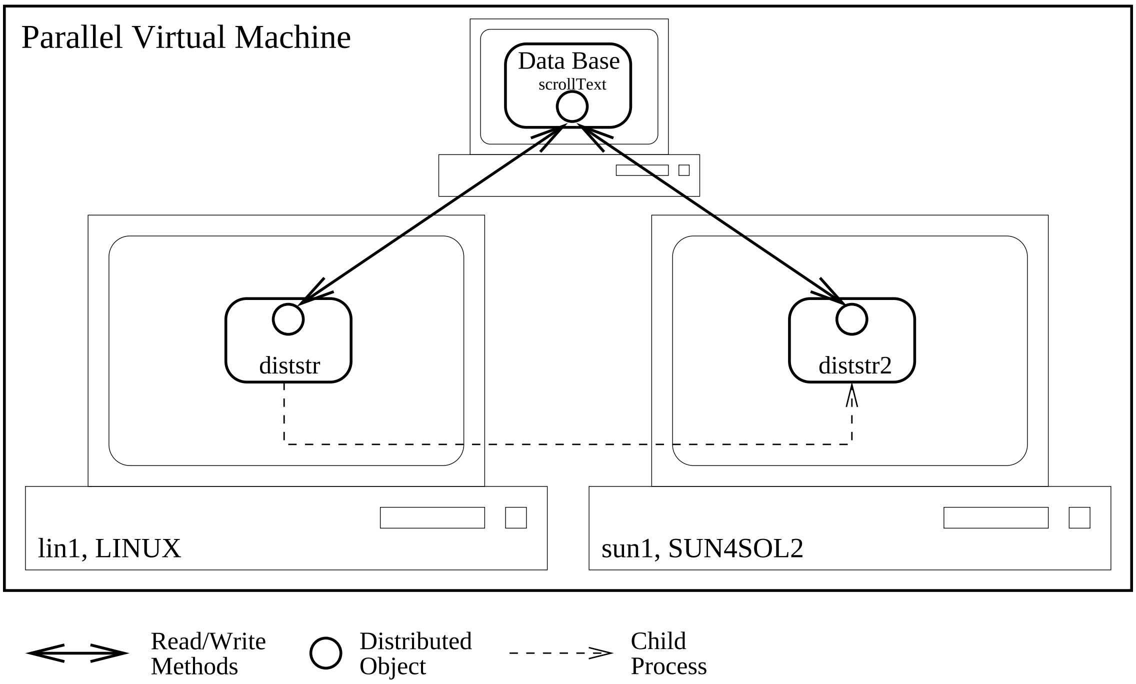


Figure 5.1: distributed string.

To create a distributed object you have to:

- Connect the process to PVM
- Register the object as a distributed object using a key name (method `cppvmDistObj(key)`)

Now you can access to the distributed object with the methods:

- `cppvmReadMboxInfo`: Read object values from the global database and store them in the local object
- `cppvmWriteMboxInfo`: Write the values of the local object into the global database

A good way to synchronize these accesses are semaphores and mutual exclusion calls (see chapter 7).

It is not necessary to have explicit informations about other processes. When using distributed objects you don´t need a process description of other processes or a send/receive stream. The process also don´t have to care about new spawned processes or processes leaving the parallel virtual machine. The distributed objects can still be accessed.

Usually distributed objects exist until the parallel virtual machine is halted. See section 5.2 how to get rid of distributed objects. The registration of an object as distributed object does not exclude explicit message passing. The object can still be sent/received as described in chapter 4.

Here is a small example:

diststr.cpp

```
// cppvm 'distributed string object'
#include "cppvm.h"

int
main()
{
  int i;

  // connect to pvm
  cppvmConnection pvmConn;

  // spawn child process 'diststr2'
  cppvmSpawnConnection child("diststr2");

  // string object
  cppvmString distStr("");
  // register the object as
  // a distributed object
  // with key name "scrollText"
  distStr.cppvmDistObj("scrollText");

  for(i=0;i<500;i++){
    // update the local string
    // with the value
    // in the message box
    distStr.cppvmReadMboxInfo();
    cout << '\r' << distStr;
    cout.flush();
  }

  for(i=0;i<500;i++){
    // update the local string
    // with the value
    // in the message box
    cout << '\r' << distStr.read();
    cout.flush();
  }

  // the distributed object can
  // still be used for
  // explicit message passing

  // receive descriptor
  // (blocking receive from
  // child process)
  cppvmReceiveStream recStrm(child,
    CPPvmRBchild);

  recStrm >> distStr;

  cout << endl
       << "received from child: "
       << distStr << endl;

  // halt the virtual machine
  pvmConn.halt();

  return 0;
}
```

diststr2.cpp

```
#include "cppvm.h"

int
main()
{
  // connect to pvm
  cppvmConnection pvmConn;

  int i;
  cppvmString distStr2("");

  // connect to message mailbox
  distStr2.cppvmDistObj("scrollText");
  // set message
  distStr2 = "I am a scrolling distributed
text. ";


  for(i=0; i< 500; i++){
    // use 'cppvmWriteMboxInfo' to
    // update distributed object
    distStr2.cycleShiftLeft();
    distStr2.cppvmWriteMboxInfo();
  }

  cppvmString rotateString("Look at me!
Another great scroll text! ");
  for(i=0; i<500; i++){
    // use write() to
    // update distributed object
    rotateString.cycleShiftLeft();
    distStr2.write(rotateString.getPtr());
  }

  // send string to parent process
  distStr2 = "The End";
  // send descriptor
  // blocking send to
  // parent process
  cppvmSendStream sendStrm(pvmConn,
    CPPvmSBparent);

  // send message to
  // parent process
  sendStrm << distStr2;

  return 0;
}
```

The process `diststr` spawns the child process `diststr2`. The process `diststr` registers the string `distStr` as distributed string using the key name `scrollText`. `diststr2` connects to PVM and registers a string `distStr2` using the same key `scrollText`. From now on the local strings are connected to each other via the data base.

In the example `diststr2` updates the data base object. This is done by calling the methods `cppvmWriteMboxInfo()` or `write()`. The process `diststr` reads the string from the data base with the methods `cppvmReadMboxInfo()` or `read()`. If the object was not registered as distributed object, the methods `cppvmReadMboxInfo` and `cppvmWriteMboxInfo` have no effect. In this case the methods `read` and `write` only access the local values of the object.

Distributed CPPVM objects are implemented as objects sent to and received from the PVM message mailbox. The PVM message mailbox is a global database for messages located on one host of the parallel virtual machine which can be accessed by all processes connected to PVM.

If an object is registered as distributed object with `cppvmDistObj("scrollText")`, CPPVM tries to find an object with the given key `scrollText` in the PVM message mailbox. If there is an object with `scrollText`, the object is send from the message mailbox to the local object. If there is no such object, a object with the key `scrollText` is created in the message mailbox and the local object is send to the mailbox. This is the reason why the string `distStr2` is set after the registration. If the string would be set before, it is possible that the process `diststr` has registered `scrollText` first and the local string `distStr2` is overwritten with the data base value.

Distributed objects are not bound to the context of a process or message tags (see sections 4.8 and 4.7). Processes can share information via distributed objects even when they are spawned in a different context or when they have a different message tag.

Using distributed objects is slower than using explicit message passing. The PVM message mailbox is dedicated to one host. This makes the access a potential bottleneck.

5.1 Standard Template Library Classes

The standard template library classes can not directly be used as distributed objects. To allow distributed STL objects, there are classes derived from cppvmObject and the STL classes: cppvmStlBitset, cppvmStlComplex, cppvmStlDeque, cppvmStlList, cppvmStlMap, cppvmStlMultimap, cppvmStlMultiset, cppvmStlPair, cppvmStlPriorityqueue, cppvmStlQueue, cppvmStlSet, cppvmStlSlist, cppvmStlStack, cppvmStlString, cppvmStlValarray, and cppvmStlVector.

This adds the functionality of cppvmObject to the STL functionality. These classes can be used like normal distributed objects:

stldeque.cpp

```
// stl deque class: cppvm
send/rec/distributed

#include "cppvm.h"

// stl namespace
using namespace std;

void printDeque(deque<double>& data)
{
  deque<double>::iterator i;
  for (i=data.begin(); i != data.end() ; i++) {
    cout << *i << " ";
  }
}

int
main()
{
  // connect to pvm
  cppvmConnection pvmConn;

  cppvmStlDeque<double> deqCppvm;
  deque<double> deq;
  // spawn child process,
  // redirect output
  cppvmCatchout catchOut(CPPvmStdout);
  cppvmSpawnConnection child("stldeque2");

  /////////////////////////
  // distributed object  //
  /////////////////////////

  // register the object
  // as a distributed object
  deqCppvm.cppvmDistObj("deque");

  int i;
  for(i=0;i<500;i++){
    // update the local value
    // with the value
    // in the message box
    deqCppvm.cppvmReadMboxInfo();
    cout << '\r';
    printDeque(deqCppvm);
    cout.flush();
  }
  cout << endl;

  /////////////////////////
  // send/receive object //
  /////////////////////////

  // receive descriptor
  // (blocking receive
  //  from child process)
  cppvmReceiveStream recStrm(child,
                             CPPvmRBchild);
...
```

stldeque2.cpp

```
#include "cppvm.h"

// stl namespace
using namespace std;

int
main()
{
  // connect to pvm
  cppvmConnection pvmConn;

  cppvmStlDeque<double> deqCppvm;
  deque<double> deq(3);

  cout << "distributed object..." << endl;

  /////////////////////////
  // distributed object  //
  /////////////////////////

  // register the object
  // as a distributed object
  deqCppvm.cppvmDistObj("deque");

  deqCppvm.resize(7);
  int i;
  for(i=0;i<500;i++){
    // update message mailbox
    deqCppvm[0] = i;
    deqCppvm.cppvmWriteMboxInfo();
  }

  cout << "send objects..." << endl;

  /////////////////////////
  // send/receive object //
  /////////////////////////

  // send descriptor
  // (blocking send
  //  to parent process)
  cppvmSendStream sendStrm(pvmConn,
                           CPPvmSBparent);

  // fill deques
  for(i=0;i<3;i++){
    deq[i] = i;
  }
  deqCppvm.resize(5);
  for(i=0;i<5;i++){
    deqCppvm[i] = i+1;
  }

  // send objects to
  // parent process
  sendStrm << deqCppvm << deq;

  return 0;
}
```

stldeque.cpp

```
...
  // receive data from child process
  recStrm >> deqCppvm >> deq;

  cout << "received objects: " ;
  printDeque(deqCppvm);
  cout << " and: ";
  printDeque(deq);
  cout << endl;

  // halt the virtual machine
  pvmConn.halt();

  return 0;
}
```

See subsection 11.17.1 for limitations of the CPPVM STL support.

5.2 Advanced Topics

The process which registers a distributed object at first can determine the persistence of the object with the second parameter of the `cppvmDistObj` method call. The parameter specifies whether the object is deleted from the mailbox when the process terminates (`false`) or if the object should remain in the mailbox until the complete parallel virtual machine is halted (`true`):

```
// distributed string object
cppvmString distStr("");
distStr.cppvmDistObj("scrollText", false);
```

All distributed objects are persistent by default (the parameter is `true`). Persistent objects may cause confusions because the virtual machine is not automatically halted when all processes of the machine are exited. So when new processes are started, the object is still in the database with its old values, and sends this values to all processes which register this distributed object again. To halt the virtual machine, use the `halt` method of the class cppvmConnection or the halt command on the PVM console (see sections 11.1 and 2.1.4).

On the other hand the programmer has to watch carefully for the life time of non-persistent objects. They can only be accessed until the first registration process is terminated.

A distributed object can also be deleted explicitly:

```
// distributed string object
cppvmString distStr("");
distStr.cppvmDistObj("scrollText");
```

```
  // init delete
  distStr.cppvmSetMboxDel("scrollText");
  // delete object
  distStr.cppvmDelMboxInfo();
```

The `operator=` of a distributed object does not hand over the key name from one object to another. The `cppvmReadMboxInfo` has no effect in the next example:

```
...
cppvmInt localInt;
cppvmInt distInt;
distInt.cppvmDistObj("distIntKey");
localInt = distInt;
localInt.cppvmReadMboxInfo();
```

The copy constructor does hand over the distributed key name. The `cppvmReadMboxInfo` in the example connects to the distributed object:

```
void read(cppvmInt intObj){
  intObj.cppvmReadMboxInfo();
}
int main(){
...
  cppvmInt distInt;
  distInt.cppvmDistObj("distIntKey");
  read(distInt);
...
}
```

For examples see sections 12.9 and 12.10.

Chapter 6

Message Mailbox Objects

Distributed objects are using a subset of the possibilities of message mailbox CPPVM objects. The basic properties of distributed objects are also valid for mailbox objects. That means that message mailbox objects can still be sent/received as described in chapter 4. On the other hand an object cannot be a message mailbox and a distributed object at the same time. Like for distributed objects, the `operator=` does not hand over the distributed key name but the copy constructor does. Message Mailbox objects are also not bound to the context of a process or to the message tags.

To use the message mailbox, an object has to initialize its mailbox functionality with the cppvmObject methods:

- `cppvmSetMbox`: Initialize the mailbox object for read, write, and delete.
- `cppvmSetMboxRead`: Initialize mailbox object for read access.
- `cppvmSetMboxWrite`: Initialize mailbox object for write access.
- `cppvmSetMboxDel`: Initialize mailbox object to delete it from the database.

There are several options for the mailbox initialization methods. It is also possible to use combinations of these options:

- Create mailbox object with modes:
  - Remove object from mailbox if the creating process exits (`PvmMboxDefault`).
  - Persistent object (`PvmMboxPersistent`).
  - Allow multiple instances of a mailbox object (`PvmMboxMultiInstance`).
  - Permit other processes to overwrite and delete the database object (`PvmMboxOverWritable`).

- Read object from mailbox with modes:
  - Read object specified by <name, index> (`PvmMboxDefault`).
  - Read first object specified by <name, index> with index greater than or equal to the specified index parameter (`PvmMboxFirstAvail`).
  - Read object and delete it from database (`PvmMboxReadAndDelete`).

After an object has initialized its mailbox functionality, it can access the message mailbox by using the cppvmObject methods:

- `cppvmWriteMboxInfo`: Send object to mailbox.
- `cppvmReadMboxInfo`: Receive object from mailbox.
- `cppvmDelMboxInfo`: Remove object from mailbox.
- `cppvmGetMboxInfo`: Get general informations about the mailbox entries.

Here is a small example:

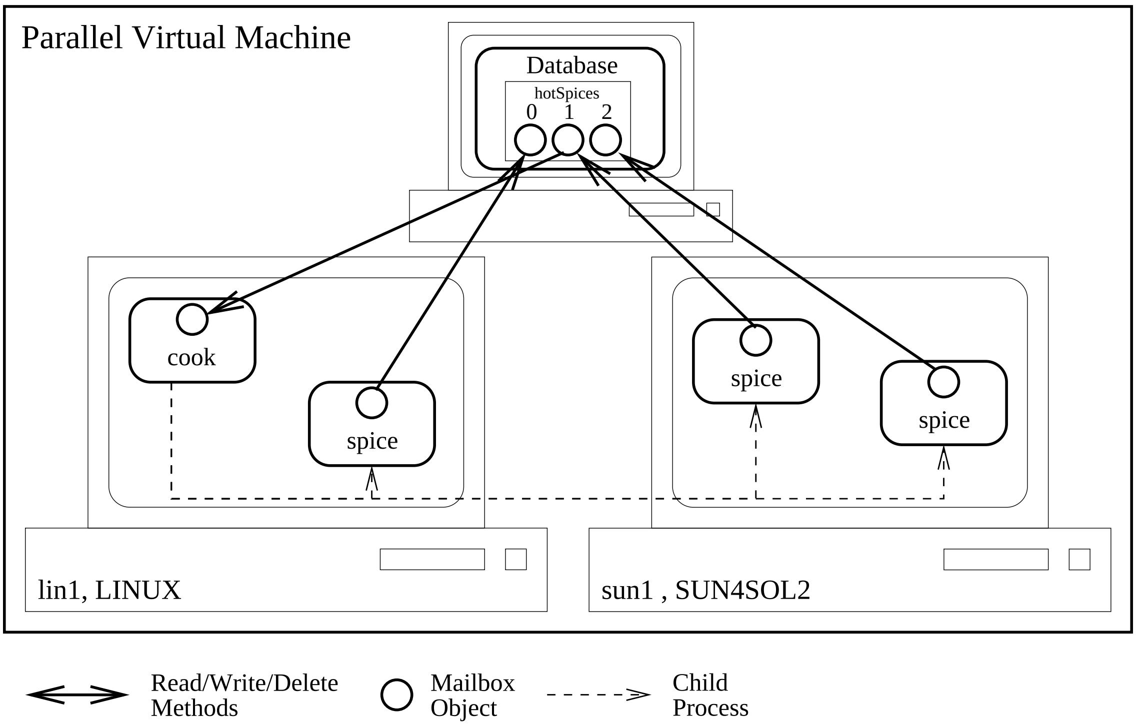


Figure 6.1: How about a tasty soup?

cook.cpp

```
#include "cppvm.h"

int
main()
{
  // connect process to pvm
  cppvmConnection pvmConn;

  // add host "sun1" to pvm
  //pvmConn.addHost("sun1");

  // spawn child processes
  // let pvm choose the host
  cppvmSpawnConnection spice1("spice",
                              "pepper");
  cppvmSpawnConnection spice2("spice",
                              "chili");
  cppvmSpawnConnection spice3("spice",
                              "curry");

  int spiceNumber=0;
  int nclasses;
  struct pvmmboxinfo *classes;

  // wait until all spices are
registered
  while(spiceNumber!=3){
    // read all message box entries
    pvmConn.cppvmGetMboxInfo("hotSpices",
                             &nclasses,
                             &classes);
    spiceNumber = classes[0].mi_nentries;
  }

  // empty string
  cppvmString favoriteSpice;

  // init read access
  favoriteSpice.cppvmSetMboxRead("hotSpices");

  // read object with
  // index 1 (range [0..2])
  favoriteSpice.cppvmReadMboxInfo(1);
  cout << "Well, for THIS soup I need some " <<
favoriteSpice
      << " (hehehe)." << endl;

  // halt the virtual machine
  pvmConn.halt();

  return 0;
}
```

spice.cpp

```
#include "cppvm.h"

int
main(int argc, char *argv[])
{
  // process must be called
  // with one argument
  if(argc!=2){
    cerr << "wrong number of arguments"
         << endl;
    exit(1);
  }

  // connect process to pvm
  cppvmConnection pvmConn;

  // empty string
  cppvmString spice;

  // connect to message mailbox
  // make object persistent;
  // allow multiple instances
  spice.cppvmSetMboxWrite("hotSpices",
                          PvmMboxPersistent

+PvmMboxMultiInstance);


  // copy first argument
  spice = argv[1];
  // write spice into message
mailbox
  spice.cppvmWriteMboxInfo();

  return 0;
}
```

The program `cook` is started on the host `lin1`. It spawns three `spice` child processes. The distribution of the processes is done by PVM and therefore can differ from figure 6.1 on real hosts.

All `spice` processes create a string as message mailbox object with the key name `hotSpices`. Due to the option `PvmMboxMultiInstance` more than one instance of the object can be created in the database. The option PvmMboxPersistent assures that the object is not deleted from the database when the process exits. After the initialization of the message mailbox object, each spice process writes its instance of the object into the database. An index counting up from zero is used in the database to differ between the instances.

The process which registers an object with a specific key name at first, decides about the access options. Consider for example that the spice process with pepper is the first process which registers an object with the name hotSpices. The option PvmMboxOverWritable would lead to only one instance of the object in the database because chili and curry would overwrite the value of the existing object in the database instead of creating new instances. This would happen even when the chili and curry processes would use the PvmMboxMultiInstance option.

6.1 Advanced Topics

It is possible to create multiple instances of objects with the same key name but with different object types in the database. For example, one spice process can write a cppvmString object, another spice a cppvmInt object, and the third spice process a cppvmArrayLong object. In the example the order of the instances in the database depends on how fast the spice processes writes their objects. It is not assured that the object with index 0 is the cppvmString object so in this case there will be a problem reading the objects out of the database.

The process cook gets the information from the database how many instances of hotSpices objects are registered in the database. For a detailed description of cppvmGetMboxInfo see section 11.12.

When the spice processes have registered all hotSpices, cook initializes a cppvmString to read an instance from the database. The method cppvmReadMboxInfo reads the instance with index 1 from the database into the local object. The instance is finally printed on cout.

By default only the process who has created a message mailbox instances has the permission to delete the instance from the database. Other processes can be explicitely permitted to overwrite and delete the instance by adding the option PvmMboxOverWritable when creating an instance. But the permission to overwrite and delete instances in the database has also another effect, as described above: When other processes want to create a new instance, the existing instance in the database is overwritten. That means no new instances are created even when the option PvmMboxMultiInstance is used.

When an process exits, all instances created in this process are removed from the database. This can be avoided by adding the option PvmMboxPersistent when creating instances. There are also two other ways to delete instances from the database. The first one is to read and delete an instance at the same time, the second to delete instances explicitely. Consider that each of the processes `spice1a` and `spice1b` create an instance of a `hotSpices` object with the index 1. This allows them to delete this instance. `spice1a.cpp` and `spice1b.cpp` describe to ways how to delete this instance:

spice1a.cpp

```
#include "cppvm.h"

int
main(int argc, char *argv[])
{
  // process must be called
  // with one argument
  if(argc!=2){
    cerr << "wrong number of arguments"
         << endl;
    exit(1);
  }

  // connect process to pvm
  cppvmConnection pvmConn;

  // empty string
  cppvmString spice;

  // connect to message mailbox
  // make object persistent;
  // allow multiple instances
  spice.cppvmSetMboxWrite("hotSpices",
                          PvmMboxPersistent
                          +PvmMboxMultiInstance);


  // copy first argument
  spice = argv[1];
  // write spice into message mailbox
  spice.cppvmWriteMboxInfo();

  // init read and delete access
  spice.cppvmSetMboxRead("hotSpices",
                         PvmMboxReadAndDelete);

  // read and delete object with
  // index 1 (range [0..2])
  spice.cppvmReadMboxInfo(1);

  return 0;
}
```

spice1b.cpp

```
#include "cppvm.h"

int
main(int argc, char *argv[])
{
  // process must be called
  // with one argument
  if(argc!=2){
    cerr << "wrong number of arguments"
         << endl;
    exit(1);
  }

  // connect process to pvm
  cppvmConnection pvmConn;

  // empty string
  cppvmString spice;

  // connect to message mailbox
  // make object persistent;
  // allow multiple instances
  spice.cppvmSetMboxWrite("hotSpices",
                          PvmMboxPersistent

+PvmMboxMultiInstance);


  // copy first argument
  spice = argv[1];
  // write spice into message
mailbox
  spice.cppvmWriteMboxInfo();

  // init delete access
  spice2.cppvmSetMboxDel("hotSpices");

  // delete object with
  // index 1 (range [0..2])
  spice2.cppvmDelMboxInfo(1);

  return 0;
}
```

These modifications would probably lead to a endless loop in `cook`. `cook` waits until 3 spices are registered in the database. Usually, the new `spice1a` or `spice1b` is fast enough to register and delete its instance before `cook` can get the information about 3 entries in the database. Otherwise `cook` would try to read the instance with index 1 which probably does not exist at this time any more.

For more examples see section 12.11.

Chapter 7

Process Synchronization

There are several possibilities to synchronize parallel processes. Blocking message passing for example can be used to define rendez-vous points in parallel programs. To make parallel programming more convenient CPPVM provides three special concepts for synchronization: semaphores, mutual exclusion (mutex) calls and barriers.

7.1 Semaphores

Semaphores were proposed by Dijkstra [Dijkstra 68]. The basic idea of semaphores is to have a shared counter for all parallel processes. The counter can be decreased with the P ("Passeer") operation and increased with the V ("Verlaat") operation. From now on the terms `enter` and `leave` are used instead of the Dutch terms P and V.

The operations are primitive, i.e. one operation can not interrupt another one. When the counter is 0, `enter` will block the calling process. The process can only be continued when another process calls the `leave` operation. The `leave` operation never blocks a process. Here is the pseudo-code for the semaphore implementation in CPPVM:

enter

```
void enter(){
  if (number==0) {
    blockCallingProcess();
  }
  else{
    number = number - 1;
  }
}
```

leave

```
void leave(){
  if(thereIsABlockedProcess){
    wakeup(oldestBlockedProcess);
  }
  else{
    number = number + 1;
  }
}
```

The semaphore counter can be initialized with values ≥ 0. The counter value defines how many `enter` operations are allowed until a calling process is blocked. To set the value, use the method `setNumber`. This should be done before using the first `enter` or `leave` operation. Otherwise the semaphore counter has the value 0.

The `setNumber` method can only be used once for a semaphore in the entire virtual machine. This means that only one process can call `setNumber` and this process is only allowed to set this value one time. This is because the initial value of a semaphore has great influence on what a program using the semaphore is doing. It is possible to set the counter again using the method `resetNumber`.

`resetNumber` sets the counter number, and deletes the waiting queue for this semaphore. If the semaphore is blocking some processes at that time, the processes are blocked forever and can not be waked up any more. But of course it is also possible to modify the counter number using the methods `getNumber`, `enter`, and `leave`.

Like distributed objects the CPPVM semaphores have key names to identify them in the virtual machine. The key name is set in the constructor. All semaphores are controlled by the semaphore server process `cppvms`. `cppvms` is started automatically when a cppvm-Semaphore object is created. By default the host for the semaphore server is chosen by CPPVM. But it is also possible to define this host.

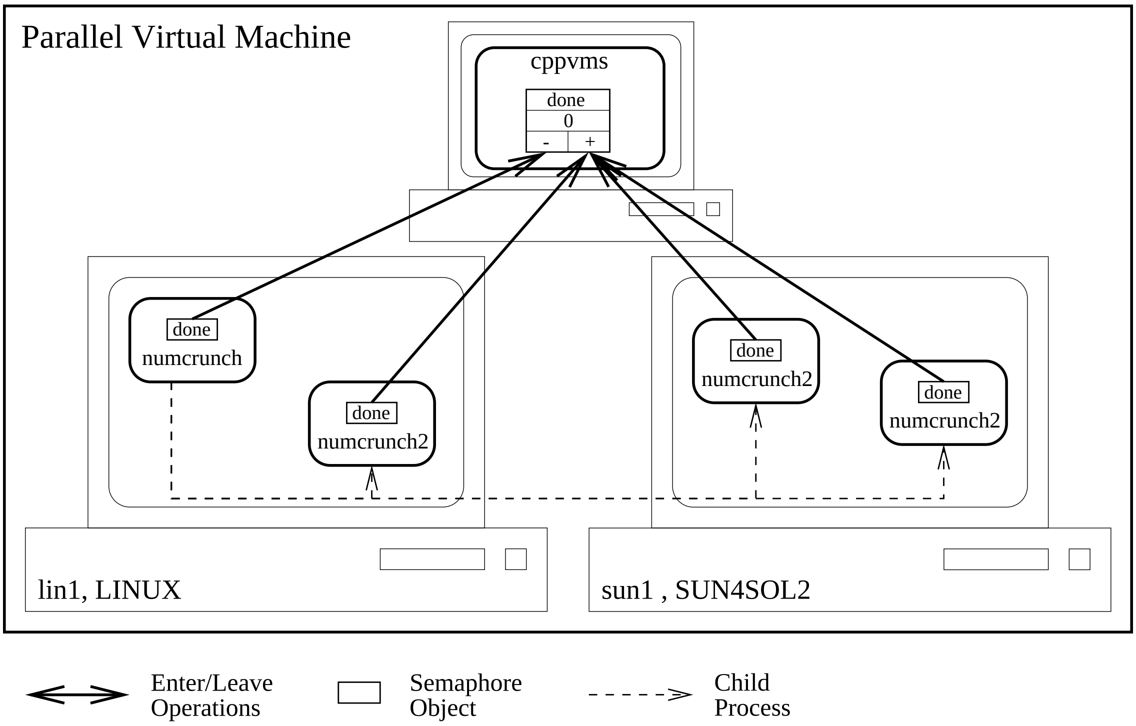


Figure 7.1: Process synchronization with semaphores.

Figure 7.1 describes a small synchronization example. The process `numchrunch` should be prevented to call `halt` until all of its children have finished their complex calculations. The semaphore with the key name `"done"` is used to synchronize the processes. `numcrunch` sets the semaphore counter to 0. Therefore the `enter` operation will block `numcrunch` until a `leave` operation is called. The loop ensures that all three `leave` operations of the children are necessary to get to the `halt` method.

numcruch

```
// cppvm semaphore
// synchronization example
#include "cppvm.h"

int
main()
{
  // connect to pvm
  cppvmConnection pvmConn;

  // add host "sun1" to pvm
  pvmConn.addHost("sun1");

  // semaphore
  cppvmSemaphore ready("done");
  // set number of allowed entries
  ready.setNumber(0);

  // catchout to stdout
  cppvmCatchout catchout(CPPvmStdout);
  // spawn 3 children 'numcrunch2'
  cppvmSpawnMultiConnection children
       ("numcrunch2",3);


  // wait for children
  int i;
  for(i=0; i<3; i++){
    ready.enter();
    cout << "Ready no.: " << i+1 << endl;
  }

  // halt the virtual machine
  pvmConn.halt();
  return 0;
}
```

numcrunch2

```
#include "cppvm.h"

int
main()
{
  // connect to pvm
  cppvmConnection pvmConn;

  // semaphore
  cppvmSemaphore ready("done");

  // number crunching
  int i=1;
  int j=1;
  int k;

  // puh!
  k = j+i;

  cout << "result: " << k << endl;

  ready.leave();
  return 0;
}
```

Semaphores can also be used for critical sections. A critical section is a set of instructions which shares resources with other parallel processes. The result of a critical section can change unpredictable when these processes are running at the same time.

Consider the seat reservation for an over-sea flight. It is not always funny when your seat was reserved for more than one person. So the seat reservation for a terminal A should be a critical section as well as the seat reservation for a terminal B.

terminalA

```
// seat reservation
#include "cppvm.h"

int
main()
{
  // connect to pvm
  cppvmConnection pvmConn;

  // catchout to stdout
  cppvmCatchout catchout(CPPvmStdout);

  // semaphores
  cppvmSemaphore reservation("seatReservation");
  cppvmSemaphore ready("synchronize");
  // set number of allowed entries
  reservation.setNumber(1);
  ready.setNumber(0);

  // spawn child process 'terminalB'
  cppvmSpawnConnection child("terminalB");

  // seat number object
  cppvmInt seats(0);
  // register the object
  // as distributed object
  seats.cppvmDistObj("seats");

  bool maxReached = false;

  while(maxReached==false){
    // enter critical section
    reservation.enter();

    // reserve seat
    seats.cppvmReadMboxInfo();
    if((int)seats==100){
      maxReached = true;
    }
    else{
      cout << "terminalA, reservation for seat no."
           << seats << endl;
      // set next available seat
      seats++;
      seats.cppvmWriteMboxInfo();
    }

    // leave critical section
    reservation.leave();
  }

  // wait for child process
  ready.enter();

  // halt the virtual machine
  pvmConn.halt();

  return 0;
}
```

terminalB

```
#include "cppvm.h"

int
main()
{
  // connect to pvm
  cppvmConnection pvmConn;

  // semaphores
  cppvmSemaphore reservation(
               "seatReservation");
  cppvmSemaphore ready("synchronize");

  // integer object
  cppvmInt seats(0);
  // register object
  seats.cppvmDistObj("seats");

  bool maxReached = false;

  while(maxReached==false){
    // enter critical section
    reservation.enter();

    // reserve seat
    seats.cppvmReadMboxInfo();
    if((int)seats==100){
      maxReached = true;
    }
    else{
      cout << "terminalB, reservation for
seat no."
           << seats << endl;
      // set next available seat
      seats++;
      seats.cppvmWriteMboxInfo();
    }

    // leave critical section
    reservation.leave();
  }

  // allow parent process
  // to continue
  ready.leave();

  return 0;
}
```

The distributed object with the key name `seats` is a shared resource of the processes `terminalA` and `terminalB`. The access to the distributed object is synchronized with the semaphore "`seatReservation`". In this case the semaphore is used for mutual exclusion.

Before every access to `seats` the `enter` method of `seatReservation` is called. The initial semaphore counter number of 1 ensures that only one `enter` call can continue. The other one is blocked until `leave` is called. With this synchronization it is not possible to reserve one seat twice[1].

Like distributed objects or message mailbox objects semaphores are not bound to the context of the process. Processes can use semaphores for synchronization even when they are spawned in a different context.

7.1.1 Producer/Consumer Problem

A classic synchronization problem is the producer/consumer example. Consider a warehouse with a certain amount of storage room. The producer should stop producing when the warehouse is full. The consumer should stop working when the warehouse is empty.

The warehouse is implemented as a class with the basic methods `put` and `get`. The producer/consumer problem can be solved using three semaphores: `access`, `notFull`, and `notEmpty`. `access` is used – like in the seat reservation program – to prevent concurrent access to the warehouse. In the beginning `access` is set to 1. This ensures that only one process can access to the warehouse. The other one is blocked.

The semaphore `notFull` indicates how many objects can be stored in the warehouse until the maximum amount is reached. In the example the producer sets this value to seven at first.

`notEmpty` describes how many objects can be taken from the warehouse. At the beginning, the warehouse is empty and therefore the value is set to 0.

Let's have a closer look at the `get` method. `notEmpty` has the value 0. By calling the `enter` operation, the method `get` is blocked.

The `put` method calls the `enter` operation of `notFull`. `notFull` is decreased to the value six and `produce` can continue. `access` is decreased to 0 and `put` can add a new object in the warehouse. After that, `leave` of `access` is called to enable `get` the ware access. Finally, the `leave` operation of `notEmpty` indicates `get` that there are objects in the warehouse.

This `leave` operation allows `get` to continue with the `enter` operation of `access`. When `put` currently does not block this operation, `get` takes an object from the warehouse, and ends the warehouse access with the `leave` operation of `access`. Finally, `notFull` is increased to indicate that there is one more storage room free in the warehouse.

On one hand `get` is blocked when there are no objects in the warehouse (`notEmpty` is 0). On the other hand `put` is blocked when the warehouse is full (`notFull` is 0).

[1] I bet some of you wish your favorite airline or railroad company would use this program.

The example uses the semaphore `ready` to synchronize the producer and consumer processes. When the consumer has taken all warehouse objects, the `leave` operation of `ready` releases the `enter` operation of the producer and the `leave` operation will block the process.

The producer resets the storage room of the warehouse to four using the method `resize` of `warehouse`. After that, the producer releases the consumer and the warehouse access starts once again.

warehouse.h

```
// class 'warehouse' specification
// producer/consumer semaphore example

#ifndef _WAREHOUSE_INCLUDE
#define _WAREHOUSE_INCLUDE

#include "cppvm.h"

class warehouse{
public:

// constructor/destructor
  warehouse();
  virtual ~warehouse();

  void get();
  void put();

  void init(unsigned int number);
  void resize(unsigned int number);

protected:
  // pvm access
  cppvmConnection pvmConn;

  // ware
  cppvmInt ware;

  // semaphores
  cppvmSemaphore notFull;
  cppvmSemaphore notEmpty;
  cppvmSemaphore access;
};

#endif
```

warehouse.cpp

```
// class 'warehouse' implementation

// include specification file
#include "warehouse.h"

// default constructor
warehouse::warehouse()
{
  // distributed object
  ware=0;
  ware.cppvmDistObj("ware");

  // set semaphore names
  notFull.setName("notFull");
  notEmpty.setName("notEmpty");
  access.setName("access");
}

// destructor
warehouse::~warehouse()
{
}
...
```

warehouse.cpp

```
...
// init
// can only be called one time
void
warehouse::init(unsigned int number)
{
  notFull.setNumber(number);
  notEmpty.setNumber(0);
  access.setNumber(1);
}

// resize
void
warehouse::resize(unsigned int number)
{
  notFull.resetNumber(number);
  notEmpty.resetNumber(0);
  access.resetNumber(1);
}

// get
void
warehouse::get()
{
  notEmpty.enter();
  access.enter();

  // remove ware
  ware.cppvmReadMboxInfo();
  ware--;
  ware.cppvmWriteMboxInfo();
  cout << "get: number of objects in
warehouse: "
       << ware << endl;

  access.leave();
  notFull.leave();
}

// put
void
warehouse::put()
{
  notFull.enter();
  access.enter();

  // add ware
  ware.cppvmReadMboxInfo();
  ware++;
  ware.cppvmWriteMboxInfo();
  cout << "put: number of objects in
warehouse: "
       << ware << endl;

  access.leave();
  notEmpty.leave();
}
```

producer.cpp

```
// cppvm producer/consumer example
#include "cppvm.h"
#include "warehouse.h"

int
main()
{
  int i;

  // connect to pvm
  cppvmConnection pvmConn;

  // semaphore
  cppvmSemaphore ready("ready");
  ready.setNumber(0);

  // warehouse
  warehouse room;
  // 7 object can be stored
  room.init(7);

  // catchout to stdout
  cppvmCatchout catchout(CPPvmStdout);
  // spawn child process 'consumer'
  cppvmSpawnConnection child("consumer");

  // produce 10 objects
  for(i=0; i<10; i++){
    room.put();
  }

  // wait for consumer
  ready.enter();
  // resize storage room
  room.resize(4);
  // release consumer
  ready.leave();

  // produce 10 objects
  for(i=0; i<10; i++){
    room.put();
  }

  // wait for consumer
  ready.enter();
  // halt the virtual machine
  pvmConn.halt();

  return 0;
}
```

consumer.cpp

```
#include "cppvm.h"
#include "warehouse.h"

int
main()
{
  int i;

  // connect to pvm
  cppvmConnection pvmConn;

  // semaphore
  cppvmSemaphore ready("ready");

  // warehouse
  warehouse room;

  // consume 10 objects
  for(i=0; i<10; i++){
    room.get();
  }

  // release producer
  ready.leave();
  // wait for resize
  ready.enter();

  // consume 10 objects
  for(i=0; i<10; i++){
    room.get();
  }

  // release producer
  ready.leave();

  return 0;
}
```

A good exercise would be to enhance the example:

- allow more than one warehouse.
- block all `put`, `get` or `resize` operations until one process has called the `init` method.

- block `resize` until the new size is greater or equal to the current size. The number of wares must stay consistent and no processes must be blocked forever.

7.1.2 Advanced Topics

The operations `enter` and `leave` have to be primitive (can not be interrupted) operations to ensure the functionality of semaphores. In CPPVM this is implemented with a semaphore server program `cppvms`.

The server program is executed when a cppvmSemaphore object is created. If there is already a running semaphore server, the creation of a cppvmSemaphore object connects to this server.

By default the host for the semaphore server is chosen by CPPVM. But it is also possible to define the host. There are three ways to do this:

```
// version 1:
cppvmSemaphore mySema("stop","sun1");
// version 2:
cppvmSemaphore dummy1(NULL, "win2");
// version 3:
cppvmSemaphore dummy2;
dummy2.startServer("lin3");
```

Version one creates creates a semaphore variable "stop" and starts the semaphore server on host "sun1". Version two and three start the server on the specified host but do not create a semaphore variable

There is another important thing to know: the first semaphore variable in the parallel virtual machine defines the location of the semaphore server. Imagine there are two processes 'A' and 'B', both defining semaphore variables. 'A' wants to start the semaphore server on "sun1", 'B' on "lin2". If the semaphore variable in 'A' is created faster than in 'B', the semaphore server is started on "sun1". In this case, the call for "lin2" in 'B' has no effect. If you want to assign the semaphore server to a specific host, ensure that this call is the first in the virtual machine.

The `cppvms` server process is not killed automatically when all application processes exit from PVM. When application processes are started again, they will connect to the already running semaphore server. This will cause problems when the `setNumber` method is called (forbidden) or if the application processes expect a different counter number for a semaphore. So it is recommended to call `halt` when leaving a semaphore program. All processes in the virtual machine are killed, i.e. `cppvms` as well.

If you don´t want to call the `halt` method, e.g. if you don´t want to kill all PVM daemons at the end of your program, you can also kill the server manually:

```
// semaphore
cppvmSemaphore ready("done");
...
ready.killServer();
```

Ensure not to access to semaphores after this call (this includes all processes currently running in the virtual machine)! Processes blocked by semaphores at this time will be blocked forever.

The `enter` operation will block the current process if the counter number is 0. But what to do if you just want to try to enter a critical section and do other things if this is not successful?

Take the seat reservation problem once again. How can you try to reserve a seat of another flight when the seat reservation of the first one is locked by another terminal?

The first idea can be to use the method `getNumber`. When the semaphore counter number is 0, you will check out other flights. When the counter number is 1, you can call the `enter` method. But this can also block the process. Consider another process calling the `enter` operation for this semaphore between the `getNumber` and `enter` operations of this process. The counter number will be 0 and the process is blocked.

The solution is to call the `enterNB` operation. `enterNB` is a non-blocking enter operation. If the counter number is $\geq$1, the counter is decreased and `enterNB` will return `true`. Otherwise the counter still has its old value and `false` is returned:

enter

```
void enter(){
  if (number==0) {
    blockCallingProcess();
  }
  else{
    number = number - 1;
  }
}
```

enterNB

```
bool enterNB(){
  if (number==0) {
    return false;
  }
  else{
    number = number - 1;
    return true;
  }
}
```

Please keep in mind not to enter a critical section if the `enterNB` returned `false`. You should also be careful not to call a `leave` operations in this case.

The `setNumber` method can only be used once for a semaphore in the entire virtual machine. If you are not sure whether this method was call before or not, you can use the method `tryToSetNumber` instead:

```
cppvmSemaphore sema("mySemaphore");
// try to set the number of allowed entries
bool result;
result = sema.tryToSetNumber(8);
```

If the number was set before, `tryToSetNumber` has no effect and returns `false`. Otherwise the number is set to `8` and `true` is returned.

7.2 Mutual Exclusion

Mutual Exclusions (mutex) are mainly used for critical sections. Only one process is allowed to enter the section, otherwise the result of this section can change unpredictable. Various problems can occur when more than one process enters a section:

lost update Process 1 reads data, process 2 writes data, process 1 writes data. In this case process 1 has not read the latest data written by process 2 and overwrites this data. The updated data from process 2 is lost.

dirty read Read access of process 1 during write access of process 2. The read access results in partly new and partly old data.

un-repeatable read Two read accesses of process 1 are interrupted by a write access of process 2. The second read is done on modified data.

As shown in the terminal example above, a semaphore with value 1 can be used for mutual exclusion. Since a mutex variable can be simulated by semaphores this way, the mutex concept of CPPVM is more coarse grained: critical section calls are encapsulated by `cppvmMutex` wrapper functions. The idea is that a user can concentrate on developing critical sections without calling `enter` and `leave` operations in the same way again and again. For example a user encapsulates a section where a dirty read can occur into a function. This function can now be protected by the CPPVM mutex concept.

The mutex concept consists of two template functions[2]:

```
template <class T>
void cppvmMutex (T& callElement,
                 char *semaphoreName=CPPVM_STANDARD_MUTEX);
template <class T, class R>
void cppvmMutex (T& callElement, R& returnVal,
                 char *semaphoreName);
```

Internally a semaphore is used for process synchronization. The name of the semaphore is given in `semaphoreName`. Only `cppvmMutex` functions calls with the same semaphore name are protected against each other. This allows to define several critical sections which do

[2]Since the `cppvmMutex` functions use templates, they are only available if the compiler flag `-DCPPvmDoNotUseTemplates` in not set (see subsection 2.2.5.3). Some old compilers (e.g. gcc 2.8.1) cannot handle this type of templates. In this case `cppvmMutex` cannot be used.

not interfere among each other. If no semaphore name is given in the first `cppvmMutex` function, a default name (define `CPPVM_STANDARD_MUTEX`) is used.

The parameter `callElement` represents the critical section. The `callElement` can be a function without parameter or an object of a class which has defined the `operator()`. In contrast to the first `cppvmMutex` version, the second version additionally allows return values for `callElement`:

```
// version 1 (without return value)
// typical function declaration for critical section:
void criticalSection() { ... do something ... }
// class declaration with operator():
class criticalClass {
...
void operator() { ... do something ... }
};

// version 2 (with return value):
// function with return type:
int criticalSection2() { ... do something ... }
// operator() has now a return type:
class criticalClass2 {
...
float operator() { ... do something ... }
};
```

Now these critical sections can be wrapped by the `cppvmMutex` functions:

```
// version 1 (without return value)
// call wrapped function
cppvmMutex(criticalSection);
// wrapped object; call 'operator()'
criticalClass obj;
cppvmMutex(obj);

// version 2 (with return value):
// call wrapped function
int resultFunction2;
cppvmMutex(criticalSection2, resultFunction2, "fMutex");
// wrapped object; call 'operator()'
float resultObj2;
criticalClass2 obj2;
cppvmMutex(obj2, resultObj2, "oMutex");
```

Every `cppvmMutex` call can only be entered by one process. When a second process tries to enter the function at the same time, it is blocked until the first process leaves the function.

7.2.1 Mutex for Functions

The next example shows how `cppvmMutex` can be used together with functions. In the example two functions (`incCounter()` and `decCounter()`) access the same distributed counter object. Function `incCounter()` reads the object data from the global message mailbox, increments the value and writes the data back, `decCounter()` decrements the counter in the same way. Function `incCounter()` is called within process `mutexfunction`, `decCounter()` in process `mutexfunction2`.

This can cause lost updates: imagine process `mutexfunction` reading the value 5. At the same time process `mutexfunction2` reads also value 5. Process `mutexfunction` increments the value to 6 and writes the data to the global message mailbox. Process `mutexfunction2` decrements the value to 4 and also writes the data to the global message mailbox. The update of process `mutexfunction` is lost. As a result the distributed counter object has the value 4 instead of the desired value 5. This can also happen in the other direction.

To avoid these problems, the functions `incCounter()` and `decCounter()` are wrapped by `cppvmMutex` calls. Only one function can enter this call, the other one is now blocked until the first call is finished.

The process `mutexfunction` increments the value 3000 times, `mutexfunction2` decrements the value 1000 times. So the resulting value is 2000. Calling e.g. `intCounter()` without the `cppvmMutex` wrapper can cause the explained lost updates. The resulting value is unpredictable. Try this change in `mutexfunction` to see the difference.

mutexfunction.cpp

```
// mutex concept using functions
# include "cppvm.h"

// counter
cppvmInt countVar;

void
incCounter()
{
  countVar.cppvmReadMboxInfo();
  countVar++;
  countVar.cppvmWriteMboxInfo();
}

int
main()
{
  int i;

  // let pvm choose the host
  cppvmSpawnConnection child("mutexfunction2");

  // register the object
  // as distributed object
  countVar.cppvmDistObj("counter");

  for(i=0; i< 3000; i++){
    // call the function 'incCounter'
    // encapsulated by the standard
    // semaphore variable
    cppvmMutex(incCounter);

    // The version without "cppvmMutex"
    // can cause lost updates! Try the
    // next call instead of the
    // 'cppvmMutex' to see the
    // difference
    //incCounter();

    cout << '\r'
         << countVar;
    cout.flush();
  }

  // halt the virtual machine
  child.halt();

  return 0;
}
```

mutexfunction2.cpp

```
#include "cppvm.h"

// counter
cppvmInt countVar;

void
decCounter()
{
  countVar.cppvmReadMboxInfo();
  countVar--;
  countVar.cppvmWriteMboxInfo();
}

int
main()
{
  int i;

  // connect to PVM
  cppvmConnection pvmConn;

  // register the object
  // as distributed object
  countVar.cppvmDistObj("counter");

  for(i=0; i< 1000; i++){
    // call the function
    // 'decCounter'encapsulated
    // by the standard
    // semaphore variable
    cppvmMutex(decCounter);
  }

  return 0;
}
```

7.2.2 Mutex for Objects

The CPPvm mutex concept can also be used for objects. In this case the `operator()` is called. The next example solves the same problem as the `mutexfunction` example: a counter is protected to avoid lost updates.

mutexobject.h

```
#ifndef _MUTEXOBJECT_H
#define _MUTEXOBJECT_H

#include "cppvm.h"

class distObjWrapper {

public:
  // common constructors
  distObjWrapper (bool increase_) :
increase(increase_)
  {
    // register object
    c.cppvmDistObj("counter");
  }
  // destructor
  virtual ~distObjWrapper() {}

  int operator() ()
  {
    c.cppvmReadMboxInfo();
    if(increase){
      c++;
    }
    else{
      c--;
    }
    c.cppvmWriteMboxInfo();
    return (int)c;
  }

  cppvmInt c;
  bool increase;
};

#endif /* _MUTEXOBJECT_H */
```

mutexobject.cpp

```
// mutex concept using objects
#include "mutexobject.h"
#include <unistd.h>

int
main()
{
  int i;
  int value;

  // let pvm choose the host
  cppvmSpawnConnection child("mutexobject2");

  // wrapper object
  distObjWrapper dow(true);

...
```

mutexobject.cpp

```
...

  for(i=0; i< 3000; i++){
    // call the 'operator()'
    // for the object 'dow'
    // encapsulated by a
    // semaphore variable with
    // name 'dowMutex'
    cppvmMutex(dow, value, "dowMutex");

    // The version without
    // 'cppvmMutex' can cause
    // lost updates! Try the next
    // call instead of the
    // 'cppvmMutex' call to see
    // the difference
    //value = dow();

    cout << '\r'
         << value
    cout.flush();

  }

  // halt the virtual machine
  child.halt();

  return 0;
}
```

mutexobject2.cpp

```
#include "mutexobject.h"

int
main()
{
  int i;

  // connect to pvm
  cppvmConnection pvmConn;

  // wrapper object
  distObjWrapper dow(false);

  for(i=0; i< 1000; i++){
    // call the 'operator()'
    // for the object 'dow'
    // encapsulated by a
    // semaphore variable with
    // name 'dowMutex'
    cppvmMutex(dow, "dowMutex");
  }

  return 0;
}
```

In this example the counter is encapsulated by the class distObjWrapper. This class contains the distributed counter object as well as the `operator()` to increment or decrement the counter. Objects of this class can now be used as parameter of `cppvmMutex`.

The `mutexobject` example shows also how to use additional features of `cppvmMutex`. Instead of the standard semaphore `CPPVM_STANDARD_MUTEX` the semaphore `"dowMutex"` is used for mutual exclusion. This allows other processes to use the standard semaphore or different semaphore names for their critical sections without blocking the `cppvmMutex` calls of `mutexobject` and `mutexobject2`. The `mutexobject` program also uses the possibility to access return values. The `operator()` returns an integer which is stored in the variable `value`.

As in the `mutexfunction` example, calling `operator()` of the object `dow` without the `cppvmMutex` wrapper can cause lost updates.

7.2.3 Advanced Topics

The mutex concept uses internally semaphores to protect the `cppvmMutex` calls against each other. The names of these semaphores are handed over to `cppvmMutex` as parameter. In `cppvmMutex` a cppvmSemaphore object is create with the given name. After that the semaphore value is set to 1, since a semaphore with value 1 is the same as a mutex variable.

This mechanism can be used to allow more than one entries in `cppvmMutex`: the value of the semaphore defines, how many processes can enter `cppvmMutex` at the same time. To use this feature, create a semaphore, set the desired value and hand over the semaphore name to `cppvmMutex`:

```
void foo() { ... do something ... }
...
// create and set new semaphore
cppvmSemaphore mySema("multEntry");
mySema.setNumber(8);

// use the new semaphore as mutex
cppvmMutex(foo, "multEntry");
```

In this examples eight processes can enter the `cppvmMutex` call simultaneously, the ninth is blocked.

7.3 Barrier

A barrier can be used by a group of processes. Each of these processes are blocked at the barrier. The processes will continue, when all participating processes have reached the barrier. This means that barriers define a synchronization point for processes.

Barriers are represented by objects of the class cppvmBarrier. A barrier has two parameters: a group name and a count number. The group name specifies the participating processes. When creating a cppvmBarrier object the process joins the specified group. This is the same mechanism like joining a group for group broadcasts (see section 4.4). Since processes can be part of more than one group, several barrier objects can be defined at the same time.

The count number specifies the number of processes participating at the barrier. This number must be the same as the number of participating processes and it must be set in every process. If different count numbers are used in the processes, the processes can be blocked forever. Here is an example how to create and use a barrier:

```
// connect to pvm
cppvmConnection pvmConn;

// generate barrier
cppvmBarrier syncPoint("syncGroup", 20);

// wait here until all
// participating processes
// have reached the barrier
syncPoint.barrier();
...
```

In this example the group name is `syncGroup` and 20 processes must call the barrier until the processes can continue.

The next example shows why barriers are so important. They can help people to stay intact, in this case a paperboy and a milkman. Imagine a house with a front door and a back door. The paperboy delivers a newspaper each morning at the front door and the milkman places fresh milk at the back door. The family in this house has two lovely dogs: a pit bull terrier and a rottweiler. The first dog uses the front door, the second one the back door. A dog is released once the item at its door is delivered.

Now one can imagine the problem: if the paperboy delivers the newspaper first, the pit bull is releases. The dog can run around the house and bite the milkman. In the case the milkman delivers first, the rottweiler can do the same with the poor paperboy.

The solution is to set a barrier that prevents the dogs to be released until both items - the milk and the newspaper - are delivered. The processes `frontdoor` and `backdoor` implement our dog problem. The barrier group name is `do-not-bite-me` and the count number is 2, since two processes participating at the barrier:

frontdoor.cpp

```
#include "cppvm.h"
// sleep
#include "examplesdefines.h"

int
main()
{
  // connect to pvm
  cppvmConnection pvmConn;

  // spawn child process,
  // redirect output
  cppvmCatchout catchOut(CPPvmStdout);
  cppvmSpawnConnection child("backdoor");

  // generate barrier
  cppvmBarrier pitbull("do-not-bite-me", 2);

  cout << "paperboy: here "
       << "is your newspaper."
       << endl;

  // wait until the news-
  // paper is delivered
  // and the the paper-
  // boy is gone
  CPPVM_SLEEP(1);

  // call barrier
  pitbull.barrier();

  cout << "pitbull: "
       << "grr-wuff-grr."
       << endl;

  // wait to collect
  // 'cout' from child
  CPPVM_SLEEP(1);

  // halt the virtual machine
  child.halt();

  return 0;
}
```

backdoor.cpp

```
#include "cppvm.h"
// sleep
#include "examplesdefines.h"

int
main()
{
  // connect to pvm
  cppvmConnection pvmConn;

  // generate barrier
  cppvmBarrier rottweiler(
    "do-not-bite-me", 2);

  cout << "milkman: here "
       << "is your milk."
       << endl;

  // wait until the
  // milk is delivered
  // and the the milkman
  // is gone
  CPPVM_SLEEP(3);

  // call barrier
  rottweiler.barrier();

  cout << "rottweiler: "
       << "wuff-grr-wuff"
       << endl;

  return 0;
}
```

Both processes are synchronized at the barrier. So today its only dog food for the pit bull and the rottweiler. The group name and count number can also be changed for a barrier object. As the next example shows, this is done with the method `set`.

You may have heard about the "Oktoberfest" in Munich, the world's greatest fair. The Oktoberfest is very popular for its beer[3], pretzels and the atmosphere, which is described with the Bavarian word "Gaudi" (fun). Among the about 60 rides there is everything from the big wheel to the flea circus.

[3]The beer is served in several beer tents. There´s room for up to 10.000 people in a single one.

oktoberfest.cpp

```
#include "cppvm.h"
// sleep
#include "examplesdefines.h"

int
main()
{
  int i;
  cppvmSpawnConnection *visitor;

  // connect to pvm
  cppvmConnection pvmConn;

  // redirect output
  cppvmCatchout catchOut(CPPvmStdout);

  cout << "Welcome to the Oktoberfest!"
       << endl;

  // visitors of
  // type A arrive
  for(i=0; i<5; i++){
    visitor = new
cppvmSpawnConnection("visitor-a");
    CPPVM_SLEEP(1);
  }

  // visitors of
  // type B arrive
  for(i=0; i<5; i++){
    visitor = new
cppvmSpawnConnection("visitor-b");
    CPPVM_SLEEP(1);
  }

  // halt the virtual machine
  pvmConn.halt();

  return 0;
}
```

visitor-a.cpp

```
// visitor type A
// (enthusiastic)
#include "cppvm.h"

int
main()
{
  // connect to pvm
  cppvmConnection pvmConn;

  // generate barrier
  cppvmBarrier ride("merry-go-round", 5);

  cout << "Wow, a merry-go-round!"
       << endl;

  // wait until enough
  // visitors joined
  // the merry-go-round
  ride.barrier();

  cout << "Huuuurrrray!"
       << endl;

  // set new barrier
  ride.set("roller coaster", 10);

  cout << "Hey, a roller coaster!"
       << endl;

  // wait until enough
  // visitors joined
  // the roller coaster
  ride.barrier();

  cout << "Yiiiiipppppiiii!"
       << endl;

  return 0;
}
```

visitor-b.cpp

```
// visitor type B
// (fearful)
#include "cppvm.h"

int
main()
{
  // connect to pvm
  cppvmConnection pvmConn;

  // generate barrier
  cppvmBarrier ride("roller coaster", 10);

  cout << "Oh no, a roller coaster!"
       << endl;
...
```

visitor-a.cpp

```
...
  // wait until enough
  // visitors joined
  // the roller coaster
  ride.barrier();

  cout << "AAAAAAAHHHHHHHHH!"
       << endl;

  return 0;
}
```

For our example we need some visitors and - no, not the beer - some rides. There are two types of visitors. Type A is very enthusiastic and wants to visit the merry-go-round ride and the roller coaster. Visitors of type B are more fearful and want only to try the roller coaster.

The merry-go-round starts only, if five visitors join the ride. The roller coaster needs ten visitors to start. So in the example the first barrier group name is `merry-go-round` and the count value is set to 5. The process `oktoberfest` spawns 5 visitors of type A with an interval of one second. The visitor processes `visitor-a` are blocked at the first barrier until all five processes reached the `merry-go-round`.

After that, they can enjoy the ride. Now the same barrier object is reused for the roller coaster by calling the method `set`: the group name is set to `roller coaster` and the count number to 10. All 5 visitors of type A have to wait at this barrier until five other processes have called the same barrier.

The remaining 5 processes are visitors of type B and are spawned by `oktoberfest`, again with an interval of one second. All `visitor-b` processes call the barrier with group name `roller coaster` and count number 10. After the last of these 5 processes has called the barrier, the needed 10 barrier calls are complete and all processes - 5 of type A and 5 of type B - can continue with the roller coaster ride.

7.3.1 Advanced Topics

As shown in the Oktoberfest example, the method `set` can be used to change to group name and the count number of a barrier. These value can also be modified separately by using the methods `setGroup` and `setCount`:

```
...
// new barrier
cppvmBarrier syncPoint("syncGroup", 20);

// set group
syncPoint.setGroup("newSyncGroup");
// set count
syncPoint.setCount(40);
...
```

Be aware that a barrier enrolls the process in a group specified by the group name. This can cause conflicts when using the same group name for different barriers or for group broadcasts (see section 4.4). To avoid these problems use unique group names.

Some methods of cppvmBarrier affect the group status of the process. The methods `set` and `setGroup` have an optional boolean parameter. This `leaveFormerGroup` parameter specifies whether the process is unenrolled from the former group or not (the default value is `false`):

```
...
// new barrier
cppvmBarrier syncPoint("syncGroup", 20);

// set all
syncPoint.set("newSyncGroup",30, false);
// set group
syncPoint.setGroup("anotherNewSyncGroup", true);
...
```

At first the process is enrolled in the group `syncGroup`. The `set` method enrolls the process additionally in group `newSyncGroup`. Since the last parameter is `false`, the process is now part of the groups `syncGroup` and `newSyncGroup`. The `setGroup` methods enrolls the process in group `anotherNewSyncGroup` and unenrolls it (parameter `true`) from the former group `newSyncGroup`. So at the end of this example, the process is part of the groups `syncGroup` and `anotherNewSyncGroup`.

A process is part of the group specified by a barrier, even if the barrier object is deleted (destructor call). To unenroll the process from the group use the `leaveGroup` method:

```
...
// new barrier
cppvmBarrier syncPoint("syncGroup", 20);

...

// leave group
syncPoint.leaveGroup();
...
```

After calling the method `leaveGroup` it is not possible to use the barrier any more. It is necessary to enroll the process in a group to use the barrier again. This can be done e.g. by calling the methods `set` or `setGroup` with the `leaveFormerGroup` parameter set to `false`.

Barriers are always arbitrated by the group server, even if the group has been made static with the method `freezeGroup` of the class cppvmConnection. If a process leaves a static group while other process are waiting at a barrier, then a bad result exception "giving a non-existent group name" is thrown by all processes waiting at the barrier. Future barrier calls with the defunct static group, throw the same exception.

Chapter 8

New Message Classes

The standard CPPVM message classes should be sufficient for most kinds of applications. But a very interesting feature of CPPVM is the possibility to write your own message classes. The classes can be used for explicit message passing as well as for distributed and message mailbox objects. You can also enhance existing classes with this abilities.

To create a CPPVM message class you just have to derive the class from cppvmObject. After that, you have to do some modifications on this class:

- Modify/create the constructor and virtual destructor.
- Implement the virtual method `cppvmTransfer`: This method defines all message passing class variables.
- Add the macro `CPPvmMethodsDeclaration(msgtag)` to your class declaration.
- Add your own features, e.g. methods, variables or aggregated objects.

The constructor of the class must call the constructor of the class cppvmObject (directly or indirectly). The virtual destructor is needed e.g. for deleting a collection of the class.

The method `cppvmTransfer` defines all class variables for message passing. Only the variables defined in this method are transfered by the message passing. This method is called for explicit message passing as well as for distributed objects and message mailbox objects.

The message passing data defined in the method `cppvmTransfer` can be:

- Standard C/C++ types (single values and arrays).
- Other CPPVM message objects derived from cppvmObject (call the `cppvmTransfer` methods of these objects).

- The data of the base class (call the `cppvmTransfer` method of the base class).

The standard C/C++ types can be transfered using functions with syntax:

```
cppvmTransfer<Type>(<type> *ptr, int nitem);
```

The parameter `ptr` is the pointer to the variable/array, `nitem` (default value 1) indicates the number of variables to transfer. The functions with the supported types are described in detail in section 11.12.

If you use CPPvm objects as class variables, you just have to call the `cppvmTransfer` method of these objects within the `cppvmTransfer` method of your message class.

cppvmObject is an abstract class and there is no need to call its transfer method. But if you have a class hierarchy with cppvmObject as base class, you need to call the transfer method of the base classes. This can be done by simply calling `cppvmTransfer` of the base class in every derived class.

The transfer method is called when an object is sent as well as when an object is received. If you need to know which transfer mode is active, you can call the functions:

```
bool cppvmIsTransferSend();
bool cppvmIsTransferReceive();
```

The macro `CPPvmMethodsDeclaration` defines the public method `cppvmMsgTag` and the class method `cppvmClassMsgTag` . These methods return the same message tag (integer value). The message tag is used to determine the class type at runtime. The tags in range [99900-99999] are reserved for the standard CPPvm classes and must not be used! Please keep in mind that `CPPvmMethodsDeclaration` contains a `public:` so all following methods and variables are declared as public, too. Just switch to protected or private if you do not want them to be public.

You can of course add additional methods, variables and objects to the class. They will not harm the message passing.

Let´s see a small example:

minimal.h

```
#ifndef _MINIMAL_INCLUDE
#define _MINIMAL_INCLUDE

// base class 'cppvmObject'
#include "pvm_obj.h"

// unique(!) message tag for this class
#define CPPvmMsgTagClassMinimal 100

class minimal : public cppvmObject{
public:

  // constructor/destructor
  minimal();
  virtual ~minimal();

  // transfer method
  virtual void cppvmTransfer();

  // pvm message tag declaration
  CPPvmMethodsDeclaration(
    CPPvmMsgTagClassMinimal)

public:
  int integerValue;
};

#endif
```

minimal.cpp

```
// include specification file
#include "minimal.h"

// default constructor;
// call the base class
// constructor
minimal::minimal()
    : cppvmObject()
{
  integerValue = 0;
}

// destructor
minimal::~minimal()
{
}

// cppvmTransfer
void
minimal::cppvmTransfer()
{
  // transfer class variable
  cppvmTransferInt(&integerValue);
}
```

The class minimal is derived from cppvmObject. To do this, you have to include `pvm_obj.h`. The constructor of minimal calls the constructor of its base class cppvmObject to initialize the CPPvm facilities. The message tag of the class is set to 100. This tag should be unique for all of your CPPvm message classes to avoid miss-leaded messages. The method `cppvmTransfer` defines the class variable `integerValue` to be transfered during the message passing.

Here is a small message passing example for this class:

mini.cpp

```
#include "cppvm.h"
// include new class 'minimal'
#include "minimal.h"

int
main()
{
  // connect to pvm
  cppvmConnection pvmConn;

  int i;
  // object of new class
  minimal miniObj;

  // spawn child process 'mini2'
  cppvmSpawnConnection child("mini2");

  /////////////////////////
  // distributed object  //
  /////////////////////////

  // register the object
  // as a distributed object
  miniObj.cppvmDistObj("minimalObject");

  for(i=0;i<500;i++){
    // update local value
    miniObj.cppvmReadMboxInfo();
    cout << '\r'
      << miniObj.integerValue;
    cout.flush();
  }
  cout << endl;

  /////////////////////////
  // send/receive object //
  /////////////////////////

  // receive descriptor
  // (blocking receive
  //  from child process)
  cppvmReceiveStream recStrm(child,
                      CPPvmRBchild);

  // receive message
  // from child process
  recStrm >> miniObj;

  cout << "received value: "
       << miniObj.integerValue << endl;

  // halt the virtual machine
  pvmConn.halt();

  return 0;
}
```

mini2.cpp

```
#include "cppvm.h"
// include new class 'minimal'
#include "minimal.h"

int
main()
{
  // connect to pvm
  cppvmConnection pvmConn;

  int i;
  // object of new class
  minimal miniObj;

  /////////////////////////
  // distributed object  //
  /////////////////////////

  // register the object
  // as a distributed object
  miniObj.cppvmDistObj("minimalObject");

  for(i=0;i<500;i++){
    // update message mailbox
    miniObj.integerValue = i;
    miniObj.cppvmWriteMboxInfo();
  }

  /////////////////////////
  // send/receive object //
  /////////////////////////

  // send descriptor
  // (blocking send
  //  to parent process)
  cppvmSendStream sendStrm(pvmConn,
                      CPPvmSBparent);

  miniObj.integerValue=999;
  // send object to parent process
  sendStrm << miniObj;

  return 0;
}
```

Here is an example for a derived class. This class bigger transfers data of the base class, standard C/C++ types and aggregated CPPvm objects:

bigger.h

```
// class 'bigger' specification

#ifndef _BIGGER_INCLUDE
#define _BIGGER_INCLUDE

// base class 'minimal'
#include "minimal.h"
// include cppvm types
#include "cppvm.h"

// unique(!) message tag
// for this class
#define CPPvmMsgTagClassBigger 101

class bigger : public minimal{
public:

// constructor/destructor
  bigger();
  virtual ~bigger();

// transfer method
  virtual void cppvmTransfer();

// pvm message tag declaration
  CPPvmMethodsDeclaration(
          CPPvmMsgTagClassBigger)

public:
  float *cFloatArray;
  cppvmDouble cppvmDoubleVal;
  cppvmArrayLong cppvmLongArrayVal;
};

#endif
```

bigger.cpp

```
// class 'bigger' implementation

// include specification file
#include "bigger.h"

// default constructor
// call the base class
constructor
bigger::bigger()
   : minimal()
{
  cFloatArray = new float[100];
  int i;
  for(i=0;i<100;i++){
    cFloatArray[i] = (float) i;
  }
}

// destructor
bigger::~bigger()
{
  delete [] cFloatArray;
}

// cppvmTransfer
void
bigger::cppvmTransfer()
{
  // do I send or receive data?
  if(cppvmIsTransferSend()){
    cout << "cppvmTransfer send"
         << endl;
  }
  if(cppvmIsTransferReceive()){
    cout << "cppvmTransfer receive"
         << endl;
  }

  // transfer base
  // class variables
  minimal::cppvmTransfer();

  // basic C/C++ types
  // transfer 100 floats
  cppvmTransferFloat(cFloatArray, 100);

  // transfer cppvm variables
  cppvmDoubleVal.cppvmTransfer();
  cppvmLongArrayVal.cppvmTransfer();
}
```

The example how to use this message class is very similar to the program above:

big.cpp

```
#include "cppvm.h"
// include new class 'bigger'
#include "bigger.h"

int
main()
{
  // connect to pvm
  cppvmConnection pvmConn;

  int i;
  // object of new class
  bigger bigObj;

  // spawn child process 'big2'
  cppvmSpawnConnection child("big2");

  /////////////////////////
  // distributed object  //
  /////////////////////////

  // register the object
  // as a distributed object
  bigObj.cppvmDistObj("biggerObject");

  for(i=0;i<500;i++){
    // update local value
    bigObj.cppvmReadMboxInfo();
    cout << bigObj.cppvmDoubleVal
         << endl;
  }
  cout << endl;

  /////////////////////////
  // send/receive object //
  /////////////////////////

  // receive descriptor
  // (blocking receive
  //  from child process)
  cppvmReceiveStream recStrm(child,
                       CPPvmRBchild);

  // receive message
  // from child process
  recStrm >> bigObj;

  cout << "received value: "
       << bigObj.cppvmDoubleVal
       << endl;

  // halt the virtual machine
  pvmConn.halt();

  return 0;
}
```

big2.cpp

```
#include "cppvm.h"
// include new class 'bigger'
#include "bigger.h"

int
main()
{
  // connect to pvm
  cppvmConnection pvmConn;

  int i;
  // object of new class
  bigger bigObj;

  /////////////////////////
  // distributed object  //
  /////////////////////////

  // register the object
  // as a distributed object
  bigObj.cppvmDistObj("biggerObject");

  for(i=0;i<500;i++){
    // update message mailbox
    bigObj.cppvmDoubleVal = (double) i;
    bigObj.cppvmWriteMboxInfo();
  }

  /////////////////////////
  // send/receive object //
  /////////////////////////

  // send descriptor
  // (blocking send
  //  to parent process)
  cppvmSendStream sendStrm(pvmConn,
                     CPPvmSBparent);

  bigObj.cppvmDoubleVal=8888;
  // send object to parent process
  sendStrm << bigObj;

  return 0;
}
```

For another example of user written CPPVM classes see section 12.13.

8.1 Templates

CPPvm supports also message passing for template classes. The functions to transfer template data are similar to the functions for the standard C/C++ types:

```
// for arrays
template<class T>
void
cppvmTransferTemplate(T* data, int nitem, int stride);
// for single data
template<class T>
void
cppvmTransferTemplate(T& data, int stride);
```

The parameter `stride` is the stride to be used when packing the items[1]. Usually `stride` is 1. Here is an example how to use the transfer functions within a template class derived from cppvmObject:

[1] See the PVM manual for more information or just type `man pvm_pack`.

template.h

```
// template class example

#ifndef _MYTEMPLATE_INCLUDE
#define _MYTEMPLATE_INCLUDE

// include cppvm types
#include "cppvm.h"

// unique(!) message tag for this class
#define CPPvmMsgTagClassMyTemplate 333

// template array size
#define MyTemplateArraySize 4

/////////////////
// declaration //
/////////////////

template <class T>
class myTemplate : public cppvmObject{
public:

// constructor/destructor
  myTemplate();
  virtual ~myTemplate();

// transfer method
  virtual void cppvmTransfer();

// pvm message tag declaration
  CPPvmMethodsDeclaration(
    CPPvmMsgTagClassMyTemplate)

public:
  int integerValue;
  cppvmFloat floatValue;
  T templSingle;
  T* templArray;

};
#endif
```

template.h

```
////////////////
// definition //
////////////////

// default constructor
template <class T>
myTemplate<T>::myTemplate()
    : cppvmObject()
{
  integerValue = 0;
  floatValue = 0.0;
  templArray = new T
[MyTemplateArraySize];
}

// destructor
template <class T>
myTemplate<T>::~myTemplate()
{
  delete [] templArray;
}

// cppvmTransfer
template <class T>
void
myTemplate<T>::cppvmTransfer()
{
  // transfer standard types
  // (int, float, double, ...)
  // with the functions
  // 'cppvmTransfer*'
  cppvmTransferInt(&integerValue);
  // transfer classes derived
  // from cppvmObject call
  // the methods
  // 'cppvmTransfer'
  floatValue.cppvmTransfer();
  // transfer template types
  // with the functions
  // 'cppvmTransferTemplate<T>'
  cppvmTransferTemplate<T>(templSingle,
    1);
  cppvmTransferTemplate<T>(templArray,
    MyTemplateArraySize, 1);
}

#endif
```

templ.cpp

```
// cppvm template example

#include "cppvm.h"
// include new class 'minimal'
#include "minimal.h"
// include new template class
'myTemplate'
#include "template.h"

void
templExample()
{
  // connect to pvm
  cppvmConnection pvmConn;

  // use one of these object
  // ATTENTION: it must be the
  // same as in 'templ2.cpp'
  // standard C type
  myTemplate<int> templObj;
  // cppvm class
  //myTemplate<cppvmDouble> templObj;
  // user defined class
  // derived from 'cppvmObject'
  //myTemplate<minimal> templObj;

  // catchout to stdout
  cppvmCatchout catchout(CPPvmStdout);
  // spawn child process 'templ2'
  cppvmSpawnConnection child("templ2");

  //////////////////////////
  // distributed object  //
  //////////////////////////
```

templ.cpp

```
  // register the object
  // as a distributed object

templObj.cppvmDistObj("templateObject");

  int i;
  for(i=0;i<500;i++){
    // update the local value
    // with the value
    // in the message box
    templObj.cppvmReadMboxInfo();
    cout << '\r'
     << templObj.integerValue;
    cout.flush();
  }
  cout << endl;

  //////////////////////////
  // send/receive object //
  //////////////////////////

  // receive descriptor
  // (blocking receive
  //  from child process)
  cppvmReceiveStream recStrm(child,
                             CPPvmRBchild);

  // receive message from child
process
  recStrm >> templObj;

  cout << "received value: "
       << templObj.integerValue << endl;

  // halt the virtual machine
  pvmConn.halt();
}

int
main(void){
  CPPvmTryAll(templExample())
  return 0;
}
```

templ2.cpp

```
#include "cppvm.h"
// include new class 'minimal'
#include "minimal.h"
// include new template class
// 'myTemplate'
#include "template.h"

void
templExample2()
{
  // connect to pvm
  cppvmConnection pvmConn;

  // use one of these object
  // ATTENTION: it must be the
  // same as in 'templ.cpp'
  myTemplate<int> templObj;
  //myTemplate<cppvmDouble> templObj;
  //myTemplate<minimal> templObj;

  //////////////////////////
  // distributed object  //
  //////////////////////////
```

templ2.cpp

```
  // register the object
  // as a distributed object

templObj.cppvmDistObj("templateObject");

  int i;
  for(i=0;i<500;i++){
    // update message mailbox
    templObj.integerValue = i;
    templObj.cppvmWriteMboxInfo();
  }

  //////////////////////////
  // send/receive object //
  //////////////////////////

  // send descriptor
  // (blocking send
  //  to parent process)
  cppvmSendStream sendStrm(pvmConn,
                   CPPvmSBparent);

  templObj.integerValue=999;
  // send object to parent process
  sendStrm << templObj;
}

int
main(void){
  CPPvmTryAll(templExample2())
  return 0;
}
```

The template transfer functions can be used for the types:

single data types cppvmObject (including all user defined classes derived from cppvmObject), `bool`, `char`, `int`, `short`, `long`, `unsigned int`, `unsigned short`, `unsigned long`, `float`, `double`

array data types cppvmObject* (including all user defined classes derived from cppvmObject), `bool*`, `char*`, `int*`, `short*`, `long*`, `unsigned int*`, `unsigned short*`, `unsigned long*`, `float*`, `double*`

To allow only certain types for the template functions contradicts of course the meaning of templates[2]. This is because PVM has to know the exact data types to encode and decode the messages on heterogeneous machines. But you can of course add new types as template parameters (see chapter 9).

[2] If you try to use the template functions for other types you probably will get an compile error.

8.2 Standard Template Library Classes

There are also functions to transfer aggregated STL objects of the classes bitset, complex, deque, list, map, multimap, multiset, pair, priority_queue, queue, set, slist, stack, string, valarray, and vector within a new CPPVM message class (see subsection 11.17.1 for limitations):

```
// for arrays
template<class T>
void
cppvmTransferStlCLS(CLS<T> *data, int nitem, int stride);
// for single data
template<class T>
void
cppvmTransferStlCLS(CLS<T> &data, int stride);
```

Replace `CLS` by the STL classes, e.g. by `vector`:

```
cppvmTransferStlVector(vector<T> &data, int stride);
```

An example for transferring aggregated STL objects is `pavec`. The class paramVector transfers a vector and a string STL object. The files `pavec.cpp`, `pavec2.cpp`, and `paramvector.h` can be found in the `examples` directory.

8.3 Collections

User written CPPVM classes can be treated completely like standard CPPVM classes. They can perform every type of CPPVM message passing. There is only one restriction: a cppvmObjectCollection collection (see section 11.15) with objects of a user written class type as elements cannot be received.

Since user written CPPVM classes are subclasses of the class cppvmObject, objects can be added to a cppvmObjectCollection collection. When a collection is send, the collection has to call the `cppvmTransfer` methods of all elements. But when a collection wants to receive an element, the collection must be able to create an object of the element type, add this new element to the collection and call the transfer method of this new element.

The cppvmObjectCollection collection can only create objects for standard CPPVM classes. It does not know anything about user written classes. If you want the standard collection to receive your classes, you have to write a receive function (a function, not a method!) for your classes and hand it over to the collection.

The input parameter of the function is the message tag of the incoming object in the receive stream. If the message tag is matching the tag of the user written class, the function has to create a new object of the user written class type, receive the message by calling the `cppvmTransfer` method and return the object pointer casted to cppvmObject. Otherwise the incoming object does not match to the user written class and the function has to return a NULL pointer. Here is an example:

```
// receive function for minimal objects
cppvmObject*
minimalClassRecColl(int classId)
{
  // minimal object?
  if(classId == minimal::cppvmClassMsgTag()){
    // yes, create new object
    minimal *objPtr;
    objPtr = new minimal();
    // receive object
    objPtr->cppvmTransfer();
    // return object to the collection
    return objPtr;
  }
  else{
    // NULL indicates that the object was not a minimal object
    return NULL;
  }
}
```

A cppvmObjectCollection object can be enabled/disabled to receive objects of new class types by calling the methods:

```
  void addRecFct(pvmCollRecFctPtr fctPtr);
  void removeRecFct(pvmCollRecFctPtr fctPtr);
```

If you have more than one new class, just write several functions and add them, or write one big function which can receive all of your classes. From now on the collection object can also receive your classes (even in nested collections):

```
...
  // cppvmObject collection
  cppvmObjectCollection miniColl;
...
  // enable the collection to receive 'minimal' objects
  myColl.addRecFct(minimalClassRecColl);
...
```

```
  // receive a collection of 'minimal' objects
  recStrm >> miniColl;
...
  // disable the collection to receive 'minimal' objects
  myColl.removeRecFct(minimalClassRecColl);
...
```

An example for this technique can be found in section 12.13.

8.4 Exception Handling

CPPvm uses C++ exceptions for the error handling (see section 10.1). For a complete integration of your own classes into CPPvm you should use the CPPvm exception handling. If an error occurs within your class, throw an suitable cppvmException object:

```
void
myClass::setValue(int index)
{
  if(index > arrayLimit){
    // file, class, method, error message
    throw cppvmExceptionOutOfRange("myclass.cpp",
                                   "myClass",
                                   "setValue",
                                   "index is > arrayLimit");
  }
...
}
```

The available CPPvm exception classes are described in section 11.19.

8.5 Message Tags

Like described in section 4.7, new CPPvm classes can have object and class message tags. The macro `CPPvmMethodsDeclaration` defines the message tag methods `cppvmMsgTag` and `cppvmClassMsgTag`. The methods should return a unique message tag value which is different from all other CPPvm message tags. The message tag is used to identify whether an incoming object can be received by the local object or not. If the tag matches, the object tries to receive the incoming message. This can be fatal if the incoming object is an object of another class than the local object which has only the same message tag.

By using the macro CPPvmMethodsDeclaration, it is possible modify the object message tag during runtime. This can be done by using the methods setCPPvmObjectMsgTag and cppvmDeactivateObjectMsgTag. You can also set and activate an object message tag by calling the constructor cppvmObject with a message tag value:

```
// call the base class constructor
// with the new object message tag
chameleon::chameleon(int msgTag)
    : cppvmObject(msgTag)
{
 ...
}
```

Use the macro CPPvmMethodsDeclarationVariable instead of CPPvmMethodsDeclaration to allow the modification of the class message tag as well. Additionally, the macro CPPvmMethodsDefinitionVariable has to be inserted in the implementation file of the class.

Now the message tag of a class can be changed during runtime by calling the class method setCPPvmClassMsgTag. The call of this method will change the message tag of a hole class, i.e. of all objects with this class type in the current process. If you want to receive any object of this class with the old message tag value, you have to reset the old value first.

Examples how to use changeable message tags within user message classes can be found in section 12.6, 12.13, and 12.14.

Especially when you want to use CPPvm libraries developed by other people, you cannot ensure that the library classes do not use the same message tags as your classes. To avoid miss-leaded messages, it is possible to spawn processes with separate contexts (see section 4.6).

On the other hand, you may want to create different classes which are able to send messages to each other. In this case you have to give them the same message tags and ensure that the same variables are transfered in the same order in both cppvmTransfer methods.

8.6 Advanced Topics

There are two methods of cppvmObject which are not necessary for message passing but you might find them useful:

```
  virtual void cppvmScanFrom(istream& = cin);
  virtual void cppvmPrintOn(ostream& = cout);
```

The class cppvmObject overloads the operators « and » for the standard I/O stream classes `istream` and `ostream` and calls these methods. By implementing these methods you can use your classes with standard streams:

```
void
bigger::cppvmPrintOn(ostream& strm)
{
  // call printOn of base class
  minimal::cppvmPrintOn(strm);

  int i;
  for(i=0; i<100; i++){
    strm << cFloatArray[i];
  }
  strm << endl;
  strm << cppvmDoubleVal;
  strm << cppvmLongArrayVal;
}


...
int
main(){
  bigger obj;

  cout << obj << endl;
...
}
```

See section 12.13 for an example.

Chapter 9

Transfer External Classes

To transfer C++ objects you can use the standard CPPvm message classes or write your own message classes as described in the previous chapter 8. These classes all have in common that they are derived from the class cppvmObject. But what to do if you want to transfer an external, non-CPPvm class? For example you are using a framework for graphical objects. Since the framework is a third party product, you don't have the source code. But you want to transfer some of these objects. There are several ways to solve this problem:

Copy to CPPvm **classes** Copy all class variables to standard CPPvm objects (cppvmInt, cppvmArrayDouble, ...). Send/Receive these CPPvm objects.

Template functions Write specialized template functions for the graphical class. Now you can use the graphical class as template parameter for the STL classes (complex, cppvmStlDeque, ...).

Multiple inheritance Create a new class which is derived from cppvmObject as well as from the graphical class.

Wrapper class Encapsulate the graphical class as aggregated object in a new class derived from cppvmObject.

In the following the latter three solutions are described. The graphical class is point. The next source code also contains a transfer function for aggregated point objects and the specialized template functions:

point.h

```
// point
// NO CPPvm class!

#ifndef _POINT_INCLUDE
#define _POINT_INCLUDE

// include template functions
#include "cppvm.h"
#include <stdio.h>

class point {
public:

  int x;
  int y;

// constructor/destructor
  point();
  virtual ~point(){};
};

// output
inline ostream&
operator<<(ostream& strm, point& obj){
  strm << "(" << obj.x
       << ";" << obj.y << ")";
  return strm;
}

// transfer function for
// aggregated 'point' objects
void
cppvmTransferPoint(point* data,
  int nitem=1, int stride=1);

// specialized template functions
template<>
void
cppvmTransferTemplate(point* data,
  int nitem, int stride);
template<>
void
cppvmTransferTemplate(point& data,
  int stride);

#endif
```

point.cpp

```
// point
// NO CPPvm class!

// include specification file
#include "point.h"

// default constructor
point::point()
{
  x=0;
  y=0;
}

// transfer function for
// aggregated 'point' objects
void
cppvmTransferPoint(point* data,
  int nitem, int stride)
{
  int i;
  for(i=0;i<nitem;i++){
    cppvmTransferInt(&(data[i].x));
    cppvmTransferInt(&(data[i].y));
  }
}

// specialized template functions
template<>
void
cppvmTransferTemplate(point* data,
  int nitem, int stride)
{
  int i;
  for(i=0;i<nitem;i++){
    cppvmTransferInt(&(data[i].x));
    cppvmTransferInt(&(data[i].y));
  }
}
template<>
void
cppvmTransferTemplate(point& data,
  int stride)
{
  cppvmTransferInt(&(data.x));
  cppvmTransferInt(&(data.y));
}
```

Like described in section 8.1, specialized template functions must have the syntax:

```
// for arrays
template<class T>
void
cppvmTransferTemplate(T* data, int nitem, int stride);
// for single data
template<class T>
void
cppvmTransferTemplate(T& data, int stride);
```

Once the specialized template functions are defined for the class point, the class can be transfered as template parameter of the supported STL classes. To transfer aggregated point objects use the function `cppvmTransferPoint`. This function will be used in the following classes. As you can see, all transfer functions use internally the CPPVM transfer function `cppvmTransferInt`. This is the same like writing a `cppvmTransfer` method for new CPPVM message classes (see chapter 8). The class cppvmDerivedPoint shows how multiple inheritance can be used to transfer the point data. The class cppvmPoint is the implementation of a wrapper class with an aggregated point object:

cppvmderivedpoint.h

```
// multiple inheritance

#ifndef _CPPVMDERIVEDPOINT_INCLUDE
#define _CPPVMDERIVEDPOINT_INCLUDE

// base class 'cppvmObject'
#include "pvm_obj.h"
// point class
#include "point.h"

// unique(!) message tag for this class
#define
CPPvmMsgTagClassCppvmDerivedPoint 777

class cppvmDerivedPoint :
  public point, public cppvmObject{
public:

// constructor/destructor
  cppvmDerivedPoint();
  virtual ~cppvmDerivedPoint();

// transfer method
  virtual void cppvmTransfer();

// pvm message tag declaration
  CPPvmMethodsDeclaration(
    CPPvmMsgTagClassCppvmDerivedPoint)
};

// output
inline ostream&
operator<<(ostream& strm,
  cppvmDerivedPoint& obj){
  strm << (point&)(obj);
  return strm;
}

#endif
```

cppvmderivedpoint.cpp

```
// multiple inheritance example

// include specification file
#include "cppvmderivedpoint.h"

// default constructor
// call the base class
constructor
cppvmDerivedPoint::cppvmDerivedPoint()
    : point(), cppvmObject()
{
}

// destructor
cppvmDerivedPoint::~cppvmDerivedPoint()
{
}

// cppvmTransfer
void
cppvmDerivedPoint::cppvmTransfer()
{
  cppvmTransferPoint((point*)(this));
}
```

cppvmpoint.h

```
// wrapper class for point

#ifndef _CPPVMPOINT_INCLUDE
#define _CPPVMPOINT_INCLUDE

// base class 'cppvmObject'
#include "pvm_obj.h"
// point class
#include "point.h"

// unique(!) message tag for this class
#define CPPvmMsgTagClassCppvmPoint 444

class cppvmPoint : public cppvmObject{
public:

// constructor/destructor
  cppvmPoint();
  virtual ~cppvmPoint();

// transfer method
  virtual void cppvmTransfer();

// pvm message tag declaration
  CPPvmMethodsDeclaration(
    CPPvmMsgTagClassCppvmPoint)

public:
   point value;
};

// output
inline ostream&
operator<<(ostream& strm,
  cppvmPoint& obj){
  strm << obj.value;
  return strm;
}

#endif
```

cppvmpoint.cpp

```
// point wrapper class

// include specification file
#include "cppvmpoint.h"

// default constructor
// call the base class
constructor
cppvmPoint::cppvmPoint()
   : cppvmObject()
{
}

// destructor
cppvmPoint::~cppvmPoint()
{
}

// cppvmTransfer
void
cppvmPoint::cppvmTransfer()
{
  cppvmTransferPoint(&value);
}
```

All of these solutions are now used to transfer `point` objects. The `external` example contains explicit message passing as well as distributed objects:

external.cpp

```
// transfer external
// (non-CPPvm) classes

#include "cppvm.h"
// external class
#include "point.h"
// CPPvm wrapper class
#include "cppvmpoint.h"
// mulitple inheritance class
#include "cppvmderivedpoint.h"

// stl namespace
using namespace std;

void printVector(vector<point>& data)
{
  vector<point>::iterator i;
  for (i=data.begin(); i != data.end() ; i++) {
    cout << *i << " ";
  }
}

int
main()
{
  // connect to pvm
  cppvmConnection pvmConn;

  cppvmPoint p;
  cppvmDerivedPoint dp;
  vector<point> vectP;

  // spawn child process, redirect
output
  cppvmCatchout catchOut(CPPvmStdout);
  cppvmSpawnConnection child("external2");

  /////////////////////////
  // distributed object  //
  /////////////////////////
```

external.cpp

```
  // register the objects
  // as a distributed objects
  p.cppvmDistObj("point");
  dp.cppvmDistObj("derivedPoint");

  int i;
  for(i=0;i<500;i++){
    // update the local values
    // with the values
    // in the message box
    p.cppvmReadMboxInfo();
    dp.cppvmReadMboxInfo();
    cout << '\r'
         << p << " and: "
         << dp;
    cout.flush();
  }
  cout << endl;

  /////////////////////////
  // send/receive object //
  /////////////////////////

  // receive descriptor
  // (blocking receive
  //  from child process)
  cppvmReceiveStream recStrm(child,
                             CPPvmRBchild);

  // receive data from child
process
  recStrm >> p >> dp >> vectP;

  cout << "received objects: "
       << p << " and: "
       << dp << " and: ";
  printVector(vectP);
  cout << endl;

  // halt the virtual machine
  pvmConn.halt();

  return 0;
}
```

external2.cpp

```
#include "cppvm.h"
// external class
#include "point.h"
// CPPvm wrapper class
#include "cppvmpoint.h"
// mulitple inheritance class
#include "cppvmderivedpoint.h"

// stl namespace
using namespace std;

int
main()
{
  // connect to pvm
  cppvmConnection pvmConn;

  cppvmPoint p;
  cppvmDerivedPoint dp;
  vector<point> vectP;

  cout << "distributed object..."
       << endl;

  /////////////////////////
  // distributed object  //
  /////////////////////////

  // register the objects
  // as a distributed objects
  p.cppvmDistObj("point");
  dp.cppvmDistObj("derivedPoint");

  int i;
  for(i=0;i<500;i++){
    // update message mailbox
    p.value.x = i;
    p.cppvmWriteMboxInfo();
    dp.x = i;
    dp.cppvmWriteMboxInfo();
  }
```

external2.cpp

```
  cout << "send objects..." << endl;

  /////////////////////////
  // send/receive object //
  /////////////////////////

  // send descriptor
  // (blocking send
  //  to parent process)
  cppvmSendStream sendStrm(pvmConn,
                           CPPvmSBparent);
  // set new values
  p.value.x = 4;
  p.value.y = 5;
  dp.x = 6;
  dp.y = 7;
  vectP.resize(5);
  i=0;
  vector<point>::iterator j;
  for (j=vectP.begin(); j != vectP.end() ;
j++) {
    j->x=i;
    j->y=i+1;
    i=i+2;
  }


  // send objects to parent
process
  sendStrm << p << dp << vectP;

  return 0;
}
```

By using cppvmStlVector instead of vector, the specialized template functions can also be used for distributed objects.

9.1 `PvmDataInPlace` Sending

The send encoding scheme `PvmDataInPlace` can cause problems if you want to transfer data of external classes but you don't have direct access to this data. The external class hiddenPoint for example allows only access to its data via `get` and `set` methods. A transfer function for this class might look like this:

```
// transfer function for
// aggregated 'hiddenPoint' objects
void
cppvmTransferHiddenPoint(hiddenPoint* data,
  int nitem, int stride)
{
  int i;
  int x,y;
  for(i=0;i<nitem;i++){
    x=data[i].getX();
    y=data[i].getY();
    cppvmTransferInt(&x));
    cppvmTransferInt(&y));
  }
}
```

This works for the `PvmDataDefault` encoding scheme, but not for `PvmDataInPlace`! When using `PvmDataDefault` the data is copied in a buffer. `PvmDataInPlace` stores only the sizes and the data pointer. The data is left in place during packing. Since `x` and `y` are local variables of the function, they are not valid any more after the function is exited. This means, that the data pointer of `PvmDataInPlace` points to invalid memory when the send operation is done.

To make `x` and `y` static is no solution. When arrays of hiddenPoint objects are transfered, only the data of the last point will be valid. But there are other ways to allow `PvmDataInPlace` sending:

cppvmNewSendStackData Use the template class cppvmNewSendStackData to allocate new memory, copy the hiddenPoint data into this memory.

Object variables Copy the hiddenPoint data to variables of a wrapper class.

The template class cppvmNewSendStackData<T> allocates memory (array of the template type) and frees the memory after a send command, i.e. when calling `cppvmWriteMboxInfo` for a cppvmObject or when using a cppvmSendStream. After sending, the allocated memory is freed by CPPVM. It must not be freed by the user. The allocated memory can be accessed by the class member `data`. The variable `data` is a pointer to the allocated memory of the template type. Only arrays of the template type can be allocated. If you hand over user allocated memory in the constructor of cppvmNewSendStackData<T>, it must be an array pointer.

The next example explains how to use cppvmNewSendStackData<T>. The example contains specialized template transfer functions as well as the `cppvmTransferHiddenPoint` transfer function for aggregated hiddenPoint objects. Please notice that now there is a difference between sending and receiving data.

hiddenpoint.h

```
// hidden point
// NO CPPvm class!

#ifndef _HIDDENPOINT_INCLUDE
#define _HIDDENPOINT_INCLUDE

// include template functions
#include "cppvm.h"
#include <stdio.h>

class hiddenPoint {

private:
  int x;
  int y;

public:

// constructor/destructor
  hiddenPoint();
  virtual ~hiddenPoint(){};

  int getX(){return x;}
  int getY(){return y;}
  void setX(int newX){x=newX;}
  void setY(int newY){y=newY;}

};

// output
inline ostream&
operator<<(ostream& strm,
  hiddenPoint& obj){
  strm << "(" << obj.getX()
       << ";" << obj.getY() << ")";
  return strm;
}

// transfer function for
// aggregated 'hiddenPoint' objects
void
cppvmTransferHiddenPoint(hiddenPoint* data,
  int nitem=1, int stride=1);

// specialized template functions
template<>
void
cppvmTransferTemplate(hiddenPoint* data,
  int nitem, int stride);
template<>
void
cppvmTransferTemplate(hiddenPoint& data,
  int stride);

#endif
```

hiddenpoint.cpp

```
// hiddenPoint
// NO CPPvm class!

// include specification file
#include "hiddenpoint.h"

// default constructor
hiddenPoint::hiddenPoint()
{
  x=0;
  y=0;
}

// transfer function for
// aggregated 'hiddenPoint'
objects
void
cppvmTransferHiddenPoint(hiddenPoint*
data,
  int nitem, int stride)
{
  int i;
  for(i=0; i<nitem; i++){
    if(cppvmIsTransferSend()==true){
      cppvmNewSendStackData<cppvmInt>
array(2);
      array.data[0] = data[i].getX();
      array.data[1] = data[i].getY();
      array.data[0].cppvmTransfer();
      array.data[1].cppvmTransfer();
    }
    else{
      cppvmInt x;
      cppvmInt y;
      x.cppvmTransfer();
      y.cppvmTransfer();
      data[i].setX((int)x);
      data[i].setY((int)y);
    }
  }
}

// specialized template functions
template<>
void
cppvmTransferTemplate(hiddenPoint* data,
  int nitem, int stride)
{
  int i;
  for (i=0; i<nitem ; i++) {
    cppvmTransferTemplate( data[i],
stride);
  }
}
```

hiddenpoint.cpp

```
template<>
void
cppvmTransferTemplate(hiddenPoint& data,
  int stride)
{
  if(cppvmIsTransferSend()==true){
    cppvmNewSendStackData<int> tmpIntArr(2);
    tmpIntArr.data[0] = data.getX();
    tmpIntArr.data[1] = data.getY();
    cppvmTransferInt(tmpIntArr.data, 2, stride);
  }
  else{
    int tmpIntArr[2];
    cppvmTransferInt(tmpIntArr,2, stride);
    data.setX(tmpIntArr[0]);
    data.setY(tmpIntArr[1]);
  }
}
```

The class cppvmHiddenPoint is a wrapper class for hiddenPoint. The transfer method of cppvmHiddenPoint just has to call the `cppvmTransferHiddenPoint` function as defined above.

cppvmhiddenpoint.h

```
// wrapper class for hiddenPoint

#ifndef _CPPVMHIDDENPOINT_INCLUDE
#define _CPPVMHIDDENPOINT_INCLUDE

// base class 'cppvmObject'
#include "pvm_obj.h"
// hiddenPoint class
#include "hiddenpoint.h"

// unique(!) message tag for this class
#define
CPPvmMsgTagClassCppvmHiddenPoint 555

class cppvmHiddenPoint : public cppvmObject{
public:

// constructor/destructor
  cppvmHiddenPoint();
  virtual ~cppvmHiddenPoint();

// transfer method
  virtual void cppvmTransfer();

// pvm message tag declaration
  CPPvmMethodsDeclaration(
    CPPvmMsgTagClassCppvmHiddenPoint)

public:
   hiddenPoint value;
};

// output
inline ostream&
operator<<(ostream& strm,
  cppvmHiddenPoint& obj){
  strm << obj.value;
  return strm;
}

#endif
```

cppvmhiddenpoint.cpp

```
// hiddenPoint wrapper class

// include specification file
#include "cppvmhiddenpoint.h"

// default constructor
// call the base class
constructor
cppvmHiddenPoint::cppvmHiddenPoint()
    : cppvmObject()
{
}

// destructor
cppvmHiddenPoint::~cppvmHiddenPoint()
{
}

// cppvmTransfer
void
cppvmHiddenPoint::cppvmTransfer()
{
  cppvmTransferHiddenPoint(&value);
}
```

The class cppvmAnotherHiddenPoint is another wrapper class for hiddenPoint. This wrapper class shows how to use object variables to store the data during the data transfer.

cppvmanotherhiddenpoint.h

```
// wrapper class for hiddenPoint

#ifndef
_CPPVMANOTHERHIDDENPOINT_INCLUDE
#define
_CPPVMANOTHERHIDDENPOINT_INCLUDE

// base class 'cppvmObject'
#include "pvm_obj.h"
// hiddenPoint class
#include "hiddenpoint.h"

// unique(!) message tag for this class
#define CPPvmMsgTagClassCppvmAnother-
HiddenPoint
666

class cppvmAnotherHiddenPoint : public
cppvmObject{
public:

// constructor/destructor
  cppvmAnotherHiddenPoint();
  virtual ~cppvmAnotherHiddenPoint();

// transfer method
  virtual void cppvmTransfer();

// pvm message tag declaration
  CPPvmMethodsDeclaration(
    CPPvmMsgTagClassCppvmAnotherHiddenPoint)

public:
   hiddenPoint value;
private:
  // only used for CPPvm transfer
  int x;
  int y;
};

// output
inline ostream&
operator<<(ostream& strm,
  cppvmAnotherHiddenPoint& obj){
  strm << obj.value;
  return strm;
}

#endif
```

cppvmanotherhiddenpoint.cpp

```
// hiddenPoint wrapper class

// include specification file
#include
"cppvmanotherhiddenpoint.h"

// default constructor
// call the base class
constructor
cppvmAnotherHiddenPoint::cppvmAnotherHiddenPoint()
    : cppvmObject()
{
}

// destructor
cppvmAnotherHiddenPoint::~cppvmAnotherHiddenPoint()
{
}

// cppvmTransfer
void
cppvmAnotherHiddenPoint::cppvmTransfer()
{
  if(cppvmIsTransferSend()==true){
    // copy point values to object
variables
    x=value.getX();
    y=value.getY();
  }
  cppvmTransferInt(&x);
  cppvmTransferInt(&y);
  if(cppvmIsTransferReceive()==true){
    // copy object variables to
point values
    value.setX(x);
    value.setY(y);
  }
}
```

The programs `hiddenexample` and `hiddenexample2` show how the new wrapper classes can be used for explicit message passing and distributed objects. The specialized transfer functions are used for the `vector` object.

hiddenexternal.cpp

```
// transfer external (non-CPPvm)
classes

#include "cppvm.h"
// external class
#include "hiddenpoint.h"
// CPPvm wrapper classes
#include "cppvmhiddenpoint.h"
#include "cppvmanotherhiddenpoint.h"

// stl namespace
using namespace std;

void printVector(vector<hiddenPoint>& data)
{
  vector<hiddenPoint>::iterator i;
  for (i=data.begin(); i != data.end() ; i++) {
    cout << *i << " ";
  }
}

int
main()
{
  // connect to pvm
  cppvmConnection pvmConn;

  cppvmHiddenPoint p;
  cppvmAnotherHiddenPoint ap;
  vector<hiddenPoint> vectP;

  // spawn child process, redirect
output
  cppvmCatchout catchOut(CPPvmStdout);
  cppvmSpawnConnection child("hiddenexternal2");

  /////////////////////////
  // distributed object  //
  /////////////////////////

  // register the objects
  // as a distributed objects
  p.cppvmDistObj("p");
  ap.cppvmDistObj("ap");

  int i;
  for(i=0;i<500;i++){
    // update the local value with the
value
    // in the message box
    p.cppvmReadMboxInfo();
    ap.cppvmReadMboxInfo();
    cout << '\r'
         << p << " and: "
         << ap;
    cout.flush();
  }
  cout << endl;
```

hiddenexternal.cpp

```
  /////////////////////////
  // send/receive object //
  /////////////////////////

  // receive descriptor
  // (blocking receive
  //  from child process)
  cppvmReceiveStream recStrm(child,
                             CPPvmRBchild);

  // receive data from child
process
  recStrm >> p >> ap >> vectP;

  cout << "received objects: "
       << p << " and: "
       << ap << " and: ";
  printVector(vectP);
  cout << endl;

  // halt the virtual machine
  pvmConn.halt();

  return 0;
}
```

hiddenexternal2.cpp

```
#include "cppvm.h"
// external class
#include "hiddenpoint.h"
// CPPvm wrapper classes
#include "cppvmhiddenpoint.h"
#include "cppvmanotherhiddenpoint.h"

// stl namespace
using namespace std;

int
main()
{
  // connect to pvm
  cppvmConnection pvmConn;

  cppvmHiddenPoint p;
  cppvmAnotherHiddenPoint ap;
  vector<hiddenPoint> vectP;

  cout << "distributed object..." << endl;

  /////////////////////////
  // distributed object  //
  /////////////////////////

  // register the objects
  // as a distributed objects
  p.cppvmDistObj("p");
  ap.cppvmDistObj("ap");

  int i;
  for(i=0;i<500;i++){
    // update message mailbox
    p.value.setX(i);
    p.cppvmWriteMboxInfo();
    ap.value.setX(i);
    ap.cppvmWriteMboxInfo();
  }

  cout << "send objects..." << endl;
```

hiddenexternal2.cpp

```
  /////////////////////////
  // send/receive object //
  /////////////////////////

  // send descriptor
  // (blocking send
  //  to parent process)
  cppvmSendStream sendStrm(pvmConn,
                           CPPvmSBparent,
                           PvmDataInPlace);
  // set new values
  p.value.setX(4);
  p.value.setY(5);
  ap.value.setX(6);
  ap.value.setY(7);
  vectP.resize(5);
  i=0;
  vector<hiddenPoint>::iterator j;
  for (j=vectP.begin(); j != vectP.end() ;
j++) {
    j->setX(i);
    j->setY(i+1);
    i=i+2;
  }

  // send objects to parent
process
  sendStrm << p << ap << vectP;

  return 0;
}
```

Chapter 10

Status Reports

There are several ways to get status information of CPPVM programs. Within one process, CPPVM uses C++ exceptions for error handling. To care about other members of the virtual machine, CPPVM allows to observe the status of processes and hosts. Another way to get status informations about added or deleted hosts and exited processes are the notification objects.

10.1 Exception Handling

CPPVM is using C++ exceptions to indicate internal errors. Every error which occurs in CPPVM throws an exception object which can be caught by the user and handled individually. The exception objects can be of the classes:

cppvmException	base class for all exceptions
cppvmExceptionBadParameter	a parameter has a invalid value
cppvmExceptionForbiddenCall	(currently) illegal call
cppvmExceptionBadResult	a function/method called by CPPVM returned an error
cppvmExceptionOutOfRange	a parameter is out of range

Using this exception classes a specific exception can be caught as well as all possible exceptions of CPPVM:

```
try {
  // function which contains cppvm code
  cppvmDistributedCalc();
}
catch (cppvmExceptionBadResult& exception) {
  // catch cppvm bad result exceptions
  cerr << "Bad Result Exception occurred: ";
```

```
  cerr <<  exception.getErrMsg();
  errorHandling();
}
catch (cppvmException& exception) {
  // catch all cppvm exceptions
  cerr << exception;
  exit(1);
}
catch (...) {
  cerr << "exception occurred (not part of cppvm)" << endl;
  exit(1);
}
```

There are also some macros available for a standard exception handling:

`CPPvmTry(expression)`	catch all CPPvm exceptions
`CPPvmTryAll(expression)`	catch all exceptions and print a message to `cerr`
`CPPvmTryAndErr(` `check_expression,` `error_expression)`	catch all exception and call `error_expression` if an exception occurs
`CPPvmTryAndSpecificErr(` `check_expression,` `cppvm_error_expression,` `error_expression)`	catch all exceptions, call `cppvm_error_expression` if a CPPvm exception occurred, call `error_expression` if an other exception occurred; if the exception was a CPPvm exception, the `cppvmException` object `exception` can be accessed

Here are small examples for error handling with macros:

```
// function using CPPvm
void cppvmCalc(){
...
}

// write a CPPvm exception to cerr
void myCppvmError(cppvmException &excep)
{
  cerr << excep << endl;
}

int
main(){
  // catch CPPvm exceptions
  CPPvmTry(cppvmCalc())
  ...
```

```
  // catch all exceptions
  CPPvmTryAndSpecificErr(cppvmCalc(), myCppvmError(exception), exit(1))
  return 0;
}
```

The section 11.19 contains detailed information about the CPPVM exceptions. For problems with exceptions concerning the compilers see subsection 2.2.5.2.

10.2 Status of Processes and Hosts

Each PVM-process can inquire informations about the parallel virtual machine: the status of each connected process and the status of the hosts which are part of the virtual machine.

To know the status of processes and host can be useful to detect failed processes and hosts. Consider if you try to pass messages to a failed processes. This may block the calling process forever. Or think about your program is using a cluster of hosts. Especially when your program has a long execution time, some hosts might fail[1] or being shut down for maintenance in this time. You can also use the status information to know which hosts were added to the virtual machine by subprocesses et cetera.

Use the following methods of `cppvmConnection` to get the status of processes:

`getTaskStat(int tid)`	Get the status of a process specified by its task id.
`getTaskStat(cppvmTask& task)`	Get the status of a process specified by `task`.
`getTaskStat(char* taskname)`	get the status of a process specified by `taskname`. This method can only be used for spawned processes.PVM does not know the names of processes executed by the user in a shell. If more than one processes have the name `taskName`, the task id of the first process in the list delivered by the method `taskInfo` is given back.
`taskInfo(int where, int *ntask, struct pvmtaskinfo **taskp)`	Status for a set of tasks.
`printTaskInfo()`	Print informations about all running tasks to `cout`.
`printOneTaskInfo(struct pvmtaskinfo& t)`	Print informations about one task to `cout`.

For the class `cppvmSpawnConnection` use the method:

[1] I did not mention the word "Windows"!

`getChildStat()`	Get the status of the spawned child process.

The `getTaskStat` and `getChildStat` methods return (for the other methods see the reference manual sections 11.1 and 11.2):

`PvmOk`	task is running
`PvmBadParam`	bad parameter (most likely, an invalid task id value)
`PvmSysErr`	PVM-daemon not responding
`PvmNoTask`	task not running

For the status of hosts use the cppvmConnection methods:

`getHostStat(char *host)`	Get the status of a host specified by `host`.
`hostInfo(int *nhost, int *narch, struct pvmhostinfo **hostp)`	Status for a set of hosts.
`printHostInfo()`	Print informations about all connected hosts to `cout`.
`printOneHostInfo(struct pvmhostinfo& t)`	Print informations about one host to `cout`.

The `getHostStat` method returns (for the other methods see the reference manual section 11.1):

`PvmOk`	host is ok
`PvmNoHost`	host is not in the virtual machine
`PvmHostFail`	host is unreachable (and thus possibly failed)

The `status` example uses some of these methods:

status.cpp

```
// status of task/hosts
#include <unistd.h>
#include "cppvm.h"

void
printHosts()
{
  cppvmConnection pvmConn;
  // of course, you could
  // also use this:
  //pvmConn.printHostInfo();

  int i;
  int nhost;
  int narch;
  struct pvmhostinfo *hostp;

  pvmConn.hostInfo(&nhost, &narch, &hostp);

  cout << "Number of host in the virtual machine:
"
       << nhost << endl;
  for(i=0;i<nhost;i++){
    cout << "Host name: " << hostp[i].hi_name
         << " Architecture: " << hostp[i].hi_arch
         << " Speed: " << hostp[i].hi_speed
         << " Host daemon tid: " << hostp[i].hi_tid
         << endl;
  }
}

void
printTasks()
{
  cppvmConnection pvmConn;
  // of course, you could
  // also use this:
  // pvmConn.printTaskInfo();

  int i;
  int ntask;
  struct pvmtaskinfo *taskp;

  pvmConn.taskInfo(0, &ntask, &taskp);

  cout << "Number of task running on virtual
machine: "
       << ntask << endl;
  for(i=0;i<ntask;i++){
    cout << "Task name: " << taskp[i].ti_a_out
         << " Status: " << taskp[i].ti_flag
         << " Task id: " << taskp[i].ti_tid
         << " Task process id: " << taskp[i].ti_pid
         << " Parent tid: ";
    if(taskp[i].ti_ptid == PvmNoParent){
      cout << " No parent";
    }
    else{
      cout << taskp[i].ti_ptid;
    }
    cout << " This tasks daemon tid: "
         << taskp[i].ti_host << endl;
  }
}

...
```

status.cpp

```
...
int
main()
{
  // connect to pvm
  cppvmConnection pvmConn;
  // spawn child 'status2' on
'sun1'
  cppvmSpawnConnection child("status2",
"",
     PvmTaskHost, "sun1");

  // print host info
  printHosts();
  // print task info
  printTasks();

  // is the host 'sun1' alive?
  // (pvm daemon on 'sun1'
running?)

if(pvmConn.getHostStat("sun1")==PvmOk){
    cout << "host 'sun1' is in the virtual
machine"
         << endl ;
  }
  else{
    cout << "host 'sun1' is NOT in the
virtual machine"
         << endl;
  }

  // is the child process still
alive?
  if(child.getChildStat()==PvmOk){
    cout << "child 'status2' is alive"
         << endl ;
  }
  else{
    cout << "child 'status2' is NOT alive"
         << endl;
  }

  // well, game over...
  // send SIGTERM signal to child
  child.killChild();
  // wait until killChild()
  // is done
  sleep(1);

  // child process still alive?

if(pvmConn.getTaskStat("status2")==PvmOk){
    cout << "child 'status2' is alive"
         << endl ;
  }
  else{
    cout << "child 'status2' is NOT alive"
         << endl;
  }

  // halt the virtual machine
  child.halt();

  return 0;
}
```

status2.cpp

```
#include <unistd.h>
#include "cppvm.h"
int
main()
{
  // connect to pvm
  cppvmConnection pvmConn;

  // just wait for beeing
  // killed (sniff)
  sleep(1000);

  return 0;
}
```

By looking at the results of the `status` example you will see that PVM does not know the name of the `status` process, but of `status2`. Like explained above, PVM only knows the names of spawned processes, not of user called processes.

When a user executes a CPPVM-process, the process starts a PVM-daemon on the local host. This daemon is the master daemon. When processes are spawned on other hosts slave PVM-daemons are started on these hosts. A PVM-daemon can recover from the loss of any PVM-daemon except the master. If a slave loses the master, the slave shuts itself (and therefore all CPPVM-processes running on this host) down. This ensures that the virtual machine can not be partitioned and run as two partial machines. One the other hand, the master must not crash to keep the virtual machine alive. The master daemon can not hand off its master status to other daemons. So it remains always part of the configuration even when all CPPVM-processes on this host leave the virtual machine.

When you want to start a CPPVM-process which controls failed processes and hosts, it makes sense to start it on the host with the master PVM-daemon. If it is started on a slave daemon host, it will be shut down when the connection to the master daemon fails. In this case, the rest of the virtual machine can be still alive but the control process is not able to recover other failed processes and hosts any more. It is also recommended to start the master PVM-daemon on a very reliable host. If you have a cluster of similar workstations, start the master on a host which is under your control. This prevents the host for being shut down by other users.

Usually messages are routed from the processes to its daemon, from the daemon the daemon of the target process and finally from this daemon to the target process (see section A.1). When the PVM-daemon times out during the communication, it terminates any operations waiting on the down host.

10.3 Notifications

The PVM-daemon can send notification messages about modifications of the virtual machine to PVM processes. The available notification classes are:

- cppvmNotification: Base class for all notifications.
- cppvmNotificationHostAdd: New hosts are added.
- cppvmNotificationHostDelete: Hosts are deleted or crashed.
- cppvmNotificationTaskExit: A PVM task exits or is killed.

By creating a notification object the calling process is registered by PVM to get notification messages. The message tags of the incoming messages are set as a constructor parameter. Ensure not to use identical message tags for different notification objects. The notification messages can be received by the notification objects using receive streams. Since the messages are send by a PVM-daemon, receive-from-any streams must be used.

Notification objects can not be sent, but only received. By creating multiple notification objects for the same event (e.g. cppvmNotificationTaskExit for different spawned processes), multiple notification messages are sent to the calling process. The contents of the notification messages depend on the created notification object and can be obtained by get methods:

cppvmNotificationHostAdd: `getAddedHostDaemonTids()` A parameter in the constructor determines the number of notification messages. By specifying a number of -1 all added hosts are notified, a number greater than 0 specifies a certain number of notification and 0 disables the notification messages. The notification messages contain a list of PVM-daemon task id's of the daemons running on the added hosts.

cppvmNotificationHostDelete: `getDeletedHostDaemonTid()` The constructor specifies one or more hosts to get notified about. For each registered host one notification message is send to the calling process. The message contains the PVM-daemon task id of the deleted or crashed host.

cppvmNotificationTaskExit: `getExitTid()` The constructor is called with one or more process id's to get notified about. For each of the registered process id's one notification message is send to the calling process. A message contains the task id of one exited or killed process.

cppvmNotification: `getTids()` Task or host id's, depending on the constructor parameters. The base class can be used to receive any kind of notification messages. But it is recommended to use the derived classes for specialized messages instead.

The notification messages for new hosts are stopped by deleting a cppvmNotificationHostAdd object. By deleting cppvmNotificationTaskExit or cppvmNotificationHostDelete objects the PVM-daemon will still send notification messages to the process. This is no critical problem, since the number of messages is limited by the number of processes/host given in the constructors.

The `notify` example shows how to use notification objects:

notify.cpp

```
// cppvm 'notify'
#include "cppvm.h"
// sleep
#include "examplesdefines.h"

int
main()
{
  unsigned int i;
  // connect to pvm
  cppvmConnection pvmConn;

  // message passing streams
  // for notification messages
  cppvmReceiveStream recStrmB(CPPvmRBany);
  cppvmReceiveStream
recStrmNB(CPPvmRNBany);
  cppvmReceiveStream recStrmT(CPPvmRTany, 0,
500000);

  ////////////////////////////////////
  // notification for exited tasks //
  ////////////////////////////////////

  // spawn child processes 'notify2'
  cppvmSpawnMultiConnection
children("notify2",3);

  // notification for task exits
  // message tag: 111
  cppvmNotificationTaskExit tasksExitNotify(111,
children);

  // wait for notification messages
  for(i=0;i<children.numElems();i++){
    while((recStrmT >>
tasksExitNotify).received()==false){
      cout << "notify: receive timout..." << endl;
    }
    cout << "Task with id: " <<
tasksExitNotify.getExitTid()
         << " ended." << endl;
  }

  ////////////////////////////////////
  // notification for killed tasks //
  ////////////////////////////////////

  // spawn child process 'notify2'
  cppvmSpawnConnection child("notify2");

  // notification for killed task
  // message tag: 222
  cppvmNotificationTaskExit taskExitNotify(222,
child);

  child.killChild();

  // wait for notification messages
  recStrmB >> taskExitNotify;
  cout << "Task with id: " <<
taskExitNotify.getExitTid()
       << " killed." << endl;

...
```

notify.cpp

```
...
  ////////////////////////////
  // notifications for added
hosts
  ////////////////////////////

  // notify for new hosts
  // message tag: 333
  cppvmNotificationHostAdd
hostAddNotify(333, -1);

  // replace the next line by a
host
  // of your choice
  char *hostName = "lin1";
  pvmConn.addHost(hostName);
  CPPVM_SLEEP(3);

  recStrmNB >> hostAddNotify;
  if(recStrmNB.received()){
    cppvmArrayInt hostDaemonTids;
    hostDaemonTids =
hostAddNotify.getAddedHostDaemonTids();
    cout << "Hosts added:";

for(i=0;i<hostDaemonTids.numElems();i++){
      cout << " name: "
           << pvm-
Conn.getHostName(hostDaemonTids[i])
           << " daemon tid: "
           << hostDaemonTids[i];
    }
    cout << "." << endl;
  }

  ////////////////////////////
  // notifications for delted
hosts
  ////////////////////////////

  // notify for deleted host
  // message tag: 444
  cppvmNotificationHostDelete
hostDeleteNotify(444, hostName);

  pvmConn.delHost(hostName);
  CPPVM_SLEEP(3);

  recStrmNB >> hostDeleteNotify;
  if(recStrmNB.received()){
    cout << "Host with PVM daemon tid: "
         << hostDeleteNo-
tify.getDeletedHostDaemonTid()
         << " was deleted."
         << endl;
  }

  // halt the virtual machine
  pvmConn.halt();

  return 0;
}
```

notify2.cpp

```
#include "cppvm.h"
// CPPVM_SLEEP define
#include "examplesdefines.h"

int
main()
{
  // connect to pvm
  cppvmConnection pvmConn;

  // stay alive for a while...
  CPPVM_SLEEP(3);

  return 0;
}
```

Chapter 11

CPPVM Reference Manual

The CPPVM classes and functions can be divided into six main groups, as described in chapter 3:

- process handling classes of the parallel virtual machine
  - cppvmConnection: Connect process to PVM
  - cppvmSpawnConnection: Spawn a child process
  - cppvmSpawnMultiConnection: Spawn several child processes
  - cppvmCatchout: Redirect the output of spawned processes
- message passing stream classes
  - cppvmSendStream: Send data
  - cppvmReceiveStream: Receive data
  - cppvmForwardStream: Forward received data
  - cppvmNewSendStackData: Allocate memory for `PvmDataInPlace` sending of external (non-CPPVM) classes
- message classes
  - cppvmObject: Base class for all message classes
  - cppvmObjectCollection: Collection for message classes
  - cppvmSingleData: Standard basic types: cppvmBool, cppvmChar, cppvmInt, cppvmLong, cppvmFloat, cppvmDouble, cppvmString
  - cppvmArrayData: Standard array types: cppvmArrayBool, cppvmArrayChar, cppvmArrayInt, cppvmArrayLong, cppvmArrayFloat, cppvmArrayDouble
  - cppvmTask: PVM process description

  - cppvmStlBitset: Transferable STL bitset class (only available with a special compiler switch, see subsection 2.2.5.3)
  - cppvmStlComplex: Transferable STL complex class
  - cppvmStlDeque: Transferable STL deque class
  - cppvmStlList: Transferable STL list class
  - cppvmStlMap: Transferable STL map class
  - cppvmStlMultimap: Transferable STL multimap class
  - cppvmStlMultiset: Transferable STL multiset class
  - cppvmStlPair: Transferable STL pair class
  - cppvmStlPriorityqueue: Transferable STL priority_queue class
  - cppvmStlQueue: Transferable STL queue class
  - cppvmStlSet: Transferable STL set class
  - cppvmStlSlist: Transferable STL slist class (only available with a special compiler switch, see subsection 2.2.5.3)
  - cppvmStlStack: Transferable STL stack class
  - cppvmStlString: Transferable STL string class
  - cppvmStlValarray: Transferable STL valarray class (only available with a special compiler switch, see subsection 2.2.5.3)
  - cppvmStlVector: Transferable STL vector class
  - user defined classes

- synchronization classes/function:
  - cppvmSemaphore: Semaphore variables
  - cppvmMutex: Mutual exclusion calls
  - cppvmBarrier: Block calling processes

- notification classes
  - cppvmNotification: Base class for all notifications.
  - cppvmNotificationHostAdd: New hosts are added.
  - cppvmNotificationHostDelete: Hosts are deleted or crashed.
  - cppvmNotificationTaskExit: A PVM task exits or is killed.

- exception handling classes
  - cppvmException: Base class for all exceptions

- cppvmExceptionForbiddenCall: Un-allowed method call
- cppvmExceptionBadParameter: A wrong parameter was given to a method
- cppvmExceptionOutOfRange: A parameter is not in the allowed range
- cppvmExceptionBadResult: A method or function call within a CPPVM method returned an error

The complete class hierarchy is shown in figure 11.1. Since some classes are very similar to each other (for example the cppvmArray classes), not all CPPVM classes are described explicitely. For these classes use analog the description of the most related CPPVM class.

For every class the constructors are described at first. The rest of the methods, functions, and macros are sorted by alphabetical order.

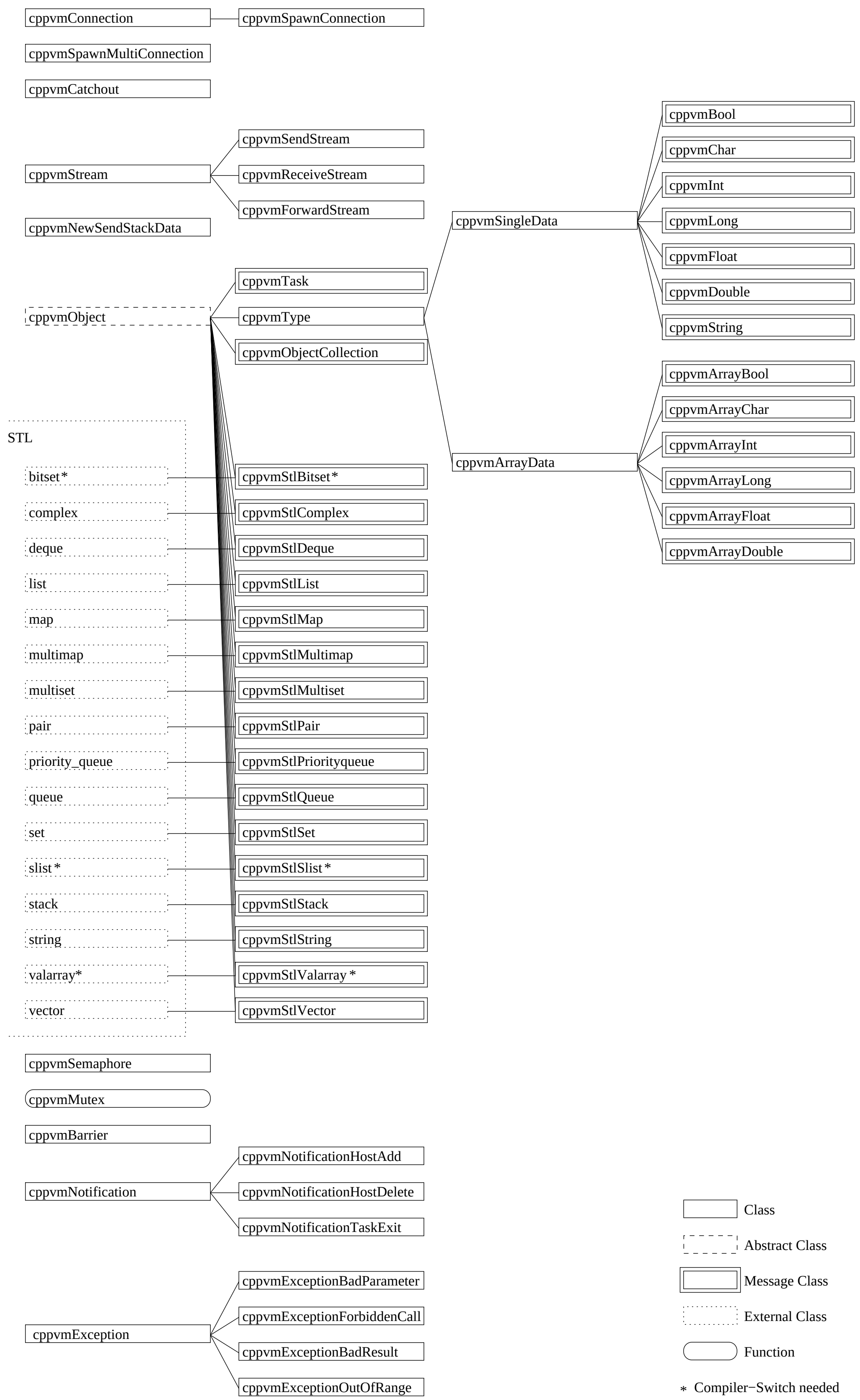


Figure 11.1: CPPvm class hierarchy.

11.1 cppvmConnection

The class cppvmConnection is a fundamental class of CPPVM. With this class a process can connect to a parallel virtual machine. This is simply done by generating a cppvmConnection object.

Each process connected to the parallel virtual machine can now send and receive messages from its parent process (if there is one). To disconnect the process from PVM just delete the cppvmConnection object.

The class offers also a complete interface to all standard PVM 3.4 functions even though they are usually not needed for CPPVM programs.

The objects of this class cannot be transfered to other processes because they contain information about the parent process. If a transfered object of this class would be used to initialize a send/receive stream, the information of the parent process might be false. Objects of the class `cppvmTask` can be used instead to transfer information about the process (see methods `getMyTask()` and `getParentTask()`).

Table 11.1: cppvmConnection: `constructor`

`constructor`	cppvmConnection
default constructor	
Synopsis	
`cppvmConnection();`	
Example	
`cppvmConnection myProcess();`	
Discussion	
Connect the process to PVM, start the PVM-daemon if it is not already running.	

Table 11.2: cppvmConnection: `constructor`

`constructor`			cppvmConnection
constructor with daemon arguments			
Synopsis			
`cppvmConnection(int argc, char **argv, int block);`			
Parameter			
Parameter	Type	Default	Description
`argc`	in		number of arguments in argv
`argv`	in		array of arguments to the PVM–daemon
			continued on next page...

. . . continued from previous page

constructor			cppvmConnection
block	in	0	If a host-file is passed as parameter, block determines: **!=0:** return when all hosts marked to start int the host-file have been added **==0:** return immediately

Exceptions	
cppvmExceptionBadResult	local pvmd is not responding tried to start the PVM daemon with arguments, but the daemon is already running cannot get the tid of this process can get no value from function clock()

Example

```
static char *pvmdArgv[] = {
"-d41",
"/home/smith/host-file.txt",
};
int pvmdArgc = 2;
cppvmConnection myProcess(pvmdArgc, pvmdArgv);
```

Discussion

start the PVM-daemon with the given arguments if it is not already running. Use this constructor to read a host-file within your program. See the man pages of the PVM-functions `pvm_start_pvmd` and `pvmd` for more informations about the PVM daemon arguments.

Table 11.3: cppvmConnection: `addHost`

addHost			cppvmConnection
add a host to the virtual parallel machine			

Synopsis

void addHost(char *host);

Parameter

Parameter	Type	Default	Description
host	in		name of the host to add to PVM

Exceptions

cppvmExceptionBadParameter	host is a NULL pointer
cppvmExceptionBadResult	several bad results (see error messages)

Example

continued on next page. . .

. . . continued from previous page	
addHost	cppvmConnection

```
cppvmConnection myProcess;
myProcess.addHost("sun11");
```

Discussion

For every `addHost` you should use `delHost` or `halt` when you don't need the host any more, or at the end of the program. Otherwise a pvmd is still running on the added host when your program has finished.
If a pvmd is already running on the host and the pvmd was not started from this program (or from a former run of this program), `addHost` does not work! The solution is to halt the already running pmvd, e.g.:

```
> rlogin host
> cd $PVM_ROOT/lib/$PVM_ARCH
> pvm
pvm> halt
```

To avoid leaving such pvmd zombies, use `halt` in your program.

Table 11.4: cppvmConnection: `barrier`

barrier	cppvmConnection
blocks the calling process until all processes in a group have called it	

Synopsis

void barrier(char *group, int count);

Parameter

Parameter	Type	Default	Description
`group`	in		group name; the group must exist and the calling process must be a member of the group
`count`	in		number of group members that must call barrier before they are released. Though not required, count is expected to be the total number of members of the specified group

Exceptions

cppvmExceptionBadResult	local pvmd is not responding count is < 1 a non-existent group name was given calling process is not in specified group

Example

```
cppvmConnection myProcess;
myProcess.barrier("server group",5);
```

Discussion

See the PVM-function `pvm_barrier` for more information.

Table 11.5: cppvmConnection: `copy constructor`

`copy constructor`	cppvmConnection
copy constructor	

Synopsis
`cppvmConnection(const cppvmConnection& obj);`

Parameter			
Parameter	Type	Default	Description
`obj`	in		object to copy

Example
`cppvmConnection myProcess = previousDefinedProcess;`

Table 11.6: cppvmConnection: `cppvmGetMboxInfo`

`cppvmGetMboxInfo`	cppvmConnection
Get information about message mailbox entries	

Synopsis

```
void cppvmGetMboxInfo(char *pattern,
int *nclasses,
struct pvmmboxinfo **classes);
```

Parameter			
Parameter	Type	Default	Description
`pattern`	in		GNU regular expression to match on names in mailbox database. Additionally, the singular '*' will match all names
`nclasses`	out		number of classes matching pattern
`classes`	out		array of pvmmboxinfo mailbox entries matching pattern

Example

```
cppvmConnection myTask;
int nclasses;
struct pvmmboxinfo *classes;
...
myTask.cppvmGetMboxInfo("*", &nclasses, &classes);
```

Discussion

continued on next page...

... continued from previous page	
`cppvmGetMboxInfo`	cppvmConnection

The memory for `classes` is allocated by this method and freed on the next call of this method

```
struct pvmmboxinfo{
char *mi_name; // class name
int mi_nentries; // # of entries for this class
int *mi_indices; // mbox entry indices
int *mi_owners; // mbox entry owner tids
int *mi_flags; // mbox entry flags
}
```

Table 11.7: cppvmConnection: `delHost`

`delHost`			cppvmConnection
delete a host from the virtual parallel machine			
Synopsis			
void delHost(char *host);			
Parameter			
Parameter	Type	Default	Description
`host`	in		name of the host to delete from PVM
Exceptions			
cppvmExceptionBadParameter	host is a NULL pointer		
cppvmExceptionBadResult	could not delete a host from the virtual machine		
Example			

```
cppvmConnection myProcess;
myProcess.addHost("sun11");
...
myProcess.delHost("sun11");
```

Discussion

All PVM processes and the PVM-daemon running on the specified host are killed.

Table 11.8: cppvmConnection: `exit`

`exit`	cppvmConnection
disconnect the process from the PVM system	
Synopsis	
void exit();	
Exceptions	
cppvmExceptionBadResult	local pvmd is not responding
Example	

continued on next page...

. . . continued from previous page	
`exit`	cppvmConnection
`cppvmConnection myProcess; myProcess.exit();`	
Discussion	
Disconnect the process from the PVM. After this method the process is not allowed to use any CPPVM/PVM call. In contrast to the method `halt`, no process or daemon in the PVM is killed. A process can connect to PVM again by creating a new cppvmConnection object.	

Table 11.9: cppvmConnection: `exportEnvVar`

`exportEnvVar`			cppvmConnection
mark environment variables to export through spawn			
Synopsis			
void exportEnvVar(char *environmentVar);			
Parameter			
Parameter	Type	Default	Description
`environmentVar`	in		name of an environment variable to add to the export list
Example			
`cppvmConnection myProcess;` `myProcess.exportEnvVar("DISPLAY");`			
Discussion			
The method edits the environment variable `PVM_EXPORT`. It checks to see if a name is already in `PVM_EXPORT` before including it. If you want to export environment variables to a process, you must edit `PVM_EXPORT` before spawning this process.			

Table 11.10: cppvmConnection: `freezeGroup`

`freezeGroup`			cppvmConnection
freezes dynamical group membership and caches info locally			
Synopsis			
void freezeGroup(char *group, int size);			
Parameter			
Parameter	Type	Default	Description
`group`	in		group name
`size`	in	-1	size of the group when it is frozen; -1 indicates that the current size of the group should be used
Exceptions			
			continued on next page. . .

...continued from previous page	
`freezeGroup`	cppvmConnection
cppvmExceptionBadResult	trying to freeze a group that is already frozen giving a NULL group name pvmd was not started or has crashed" trying to freeze a group that you are not in

Example

```
cppvmConnection myProcess;
myProcess.joinGroup("special group");
...
// call this method in every group member!
myProcess.freezeGroup("special group");
```

Discussion

This method makes a dynamic group static. The group information is then cached by all group members. `freezeGroup` is a synchronizing method and must be called by all group members to complete. All calling processes must be already members of the group (see method `joinGroup`) to be allowed to call `freezeGroup`. Otherwise the processes may be blocked forever when calling `freezeGroup`.
Once a dynamic group has been frozen, all subsequent group operations that can be satisfied with local data use the locally held information. For processes that are outside of the group, the first group call, e.d. a broadcast, will cause the static group information to be copied to the calling process. Subsequent operations then use the local information.
Barriers are always arbitrated by the group server, even if the group has been made static with the method `freezeGroup` of the class cppvmConnection. If a process leaves a static group while other process are waiting at a barrier, then a bad result exception "giving a non-existent group name" is thrown by all processes waiting at the barrier. Future barrier calls with the defunct static group, throw the same exception.

Table 11.11: cppvmConnection: `gather`

`gather`	cppvmConnection
a specified member of the group receives messages from each member of the group and gathers these messages into a single array	

Synopsis

void gather(void *result, void *data, int count, int datatype,
int msgtag, char *group, int rootginst)

Parameter

Parameter	Type	Default	Description
`result`	inout		on the root this is a pointer to the starting address of an array datatype of local values which are to be accumulated from the members of the group; if n is the number of members in the group, then this array of datatype should be of length at least n * count

continued on next page...

... continued from previous page

gather		cppvmConnection
`data`	inout	pointer to the starting address of an array of local datatype values to be send by each member of the group to the root
`count`	in	integer specifying the number of elements of datatype in the data array
`datatype`	in	type of the entries in the data array
`msgtag`	in	message tag for the gather operation
`group`	in	group name; the group must exist and the calling process must be a member of the group
`rootginst`	in	instance number of group member who performs the gather of the messages from the member of the group

Exceptions

cppvmExceptionBadResult	calling task is not in the group the datatype specified is not appropriate pvm system error

Example

```
cppvmConnection myProcess;
myProcess.gather(&getmatrix, &myrow, 10, PVM_INT,
msgtag, "server group", rootginst);
```

Discussion

See the PVM-function `pvm_gather` for more information.

Table 11.12: cppvmConnection: `getAllTaskIds`

`getAllTaskIds`	cppvmConnection
get all task ids on a specified host	

Synopsis

cppvmArrayInt getAllTaskIds(char* hostname);

Parameter

Parameter	Type	Default	Description
`hostname`	in		host name
`cppvmArrayInt`	return		task ids

Exceptions

cppvmExceptionBadResult	pvmd is not responding. no pvm daemon tid found for the given task/host name.

Example

continued on next page...

...continued from previous page	
getAllTaskIds	cppvmConnection

```
cppvmConnection myProcess;
cppvmArrayInt tids = myProcess.getAllTaskIds("linux1");
cppvmSendStream sendStrm;
cppvmString str("Hello");
int i;
// send string to all tasks on host "linux1"
for(i=0; i<tids.numElems; i++)
sendStrm.set(tids[i],CPPvmSNBtid);
sendStrm << str;
```

Table 11.13: cppvmConnection: getAllTaskIds

getAllTaskIds			cppvmConnection
get all task ids on a host specified by the PVM-daemon task id			
Synopsis			
cppvmArrayInt getAllTaskIds(int pvmDaemonTaskId);			
Parameter			
Parameter	Type	Default	Description
pvmDaemonTaskId	in		task id of the PVM-daemon
cppvmArrayInt	return		task ids
Exceptions			
cppvmExceptionBadResult	pvmd is not responding. no pvm daemon tid found for the given task/host name.		
Example			

```
cppvmConnection myProcess;
int dtid=myProcess.getHostDaemonTid("linux1");
cppvmArrayInt tids = myProcess.getAllTaskIds(dtid);
cppvmSendStream sendStrm;
cppvmString str("Hello");
int i;
// send string to all tasks on host "linux1"
for(i=0; i<tids.numElems; i++)
sendStrm.set(tids[i],CPPvmSNBtid);
sendStrm << str;
```

Table 11.14: cppvmConnection: getContextId

getContextId	cppvmConnection
get the context id of the current process	

continued on next page...

. . . continued from previous page			
`getContextId`			cppvmConnection
Synopsis			
int getContextId();			
Parameter			
Parameter	Type	Default	Description
`int`	return		context id
Example			
`cppvmConnection myProcess;` `cout << "Context id: " << myProcess.getContextId();`			
Discussion			
The context of a spawned process can be specified by the parent process. A spawned process can ask for its context id with this method.			

Table 11.15: cppvmConnection: `getDebugLevel`

`getDebugLevel`			cppvmConnection
get global debug level			
Synopsis			
void getDebugLevel(int debugLevel);			
Parameter			
Parameter	Type	Default	Description
`int`	return		current global debug level
Exceptions			
cppvmExceptionBadResult	unable to allocate memory to pack data error reading database		
Example			

`cppvmConnection pvmProcess;`
`int debugLevel;`
`debugLevel = pvmProcess.getDebugLevel();`			

Discussion			
The CPPvm debug level can be used by every process connected to PVM. It might be useful for a small information exchange between modules written by different developers.			

Table 11.16: cppvmSpawnConnection: `getEnrolledUSec`

`getEnrolledUSec`	cppvmSpawnConnection
get the time in microseconds when this child was enrolled in PVM	
Synopsis	
unsigned long getEnrolledUSec();	
	continued on next page. . .

... continued from previous page			
`getEnrolledUSec`			cppvmSpawnConnection
Parameter			
Parameter	Type	Default	Description
`unsigned long`	return		time
Example			
`cppvmConnection myProcess;`			
`...`			
`cout << "Enrolled time: [usec]" << myProcess.getEnrolledUSec();`			

Table 11.17: cppvmConnection: `getEnrolledSecs`

`getEnrolledSecs`			cppvmConnection
get the time when this process was enrolled in PVM			
Synopsis			
double getEnrolledSecs();			
Parameter			
Parameter	Type	Default	Description
`double`	return		enrolling time as seconds
Example			
`cppvmConnection myProcess;`			
`cout << "Enrolled time: [sec]" << myProcess.getEnrolledSecs();`			

Table 11.18: cppvmConnection: `getGroupSize`

`getGroupSize`			cppvmConnection
get the size (number of instances) of a group			
Synopsis			
int getGroupSize(char *group);			
Parameter			
Parameter	Type	Default	Description
`group`	in		group name
`int`	return		number of instances; values < 0 indicate an error
Example			
`cppvmConnection myProcess;`			
`int size;`			
`size = myProcess.getGroupSize("server group");`			

Table 11.19: cppvmConnection: getHostArchitecture

`getHostArchitecture`			cppvmConnection
get the host architecture for this process			
Synopsis			
char *getHostArchitecture();			
Parameter			
Parameter	Type	Default	Description
`char*`	return		host architecture
Exceptions			
cppvmExceptionBadResult	an invalid tid value was specified. pvmd is not responding. cannot get host daemon tid no host architecture for the given tid found		
Example			
`cppvmConnection myProcess;` `cout << "Host architecture: " << myProcess.getHostArchitecture();`			
Discussion			
The memory of the returned string is reused for every call.			

Table 11.20: cppvmConnection: getHostArchitecture

`getHostArchitecture`			cppvmConnection
get the host architecture of a task specified by its task name			
Synopsis			
char *getHostArchitecture(char* taskname);			
Parameter			
Parameter	Type	Default	Description
`taskname`	in		task name
`char*`	return		host architecture
Exceptions			
cppvmExceptionBadResult	an invalid tid value was specified. pvmd is not responding. cannot get host daemon tid no host architecture for the given tid found		
Example			
`cppvmConnection myProcess;` `cout << "Host architecture: " << myProcess.getHostArchitecture("calculatePrg");`			
Discussion			
The memory of the returned string is reused for every call.			

Table 11.21: cppvmConnection: getHostArchitecture

getHostArchitecture			cppvmConnection
get the host architecture of a task specified by a task object			
Synopsis			
char *getHostArchitecture(cppvmTask& task);			
Parameter			
Parameter	Type	Default	Description
`task`	in		task object
`char*`	return		host architecture
Exceptions			
cppvmExceptionBadResult	an invalid tid value was specified. pvmd is not responding. cannot get host daemon tid no host architecture for the given tid found		
Example			

`cppvmConnection myProcess;`
`cppvmTask myTask;`
`...`
`cout << "Host architecture: " << myProcess.getHostArchitecture(myTask);`			

Discussion			
The memory of the returned string is reused for every call.			

Table 11.22: cppvmConnection: getHostArchitecture

getHostArchitecture			cppvmConnection
get the host architecture of a task specified by its task id			
Synopsis			
char *getHostArchitecture(int tid);			
Parameter			
Parameter	Type	Default	Description
`tid`	in		task id
`char*`	return		host architecture
Exceptions			
cppvmExceptionBadResult	an invalid tid value was specified. pvmd is not responding. cannot get host daemon tid no host architecture for the given tid found		
Example			
`cppvmConnection myProcess;` `cout << "Host architecture: " << myProcess.getHostArchitecture(11111);`			
Discussion			

continued on next page...

...continued from previous page	
`getHostArchitecture`	cppvmConnection
The memory of the returned string is reused for every call.	

Table 11.23: cppvmConnection: `getHostDaemonTid`

`getHostDaemonTid`			cppvmConnection
get the task id of the PVM-daemon for this process			
Synopsis			
int getHostDaemonTid();			
Parameter			
Parameter	Type	Default	Description
`int`	return		daemon task id for the daemon of this process
Example			

```
cppvmConnection myProcess;
cout << "Host daemon task id: " << myProcess.getHostDaemonTid();
```

Table 11.24: cppvmConnection: `getHostDaemonTid`

`getHostDaemonTid`			cppvmConnection
get the task id of the PVM-daemon for the process with the given task or host name			
Synopsis			
int getHostDaemonTid(char* taskOrHostName);			
Parameter			
Parameter	Type	Default	Description
`taskOrHostName`	in		task or host name
`int`	return		daemon task id
Exceptions			
cppvmExceptionBadResult	pvmd is not responding. no pvm daemon tid found for the given task/host name.		
Example			

```
cppvmConnection myProcess;
cout << "Host daemon task id: " << myProcess.getHostDaemonTid("calculatePrg");
cout << "Host daemon task id: " << myProcess.getHostDaemonTid("linux1");
```

Table 11.25: cppvmConnection: `getHostDaemonTid`

`getHostDaemonTid`	cppvmConnection
get the task id of the PVM-daemon for the process with the given task object	
Synopsis	
int getHostDaemonTid(cppvmTask& task);	
	continued on next page...

. . . continued from previous page

getHostDaemonTid			cppvmConnection
Parameter			
Parameter	Type	Default	Description
`task`	in		task object
`int`	return		daemon task id
Exceptions			
cppvmExceptionBadResult	an invalid tid value was specified. pvmd is not responding. cannot get host daemon tid		
Example			

```
cppvmConnection myProcess;
cppvmTask myTask;
...
cout << "Host daemon task id: " << myProcess.getHostDaemonTid(myTask);
```

Table 11.26: cppvmConnection: getHostDaemonTid

getHostDaemonTid			cppvmConnection
get the task id of the PVM-daemon for the process with the given task id			
Synopsis			
int getHostDaemonTid(int tid);			
Parameter			
Parameter	Type	Default	Description
`tid`	in		task id
`int`	return		daemon task id
Exceptions			
cppvmExceptionBadResult	an invalid tid value was specified. pvmd is not responding. cannot get host daemon tid		
Example			

```
cppvmConnection myProcess;
cout << "Host daemon task id: " << myProcess.getHostDaemonTid(11111);
```

Table 11.27: cppvmConnection: getHostName

getHostName	cppvmConnection
get the host name for this process	
Synopsis	
char *getHostName();	
Parameter	

continued on next page. . .

... continued from previous page			
getHostName			cppvmConnection
Parameter	Type	Default	Description
char*	return		host name
Exceptions			
cppvmExceptionBadResult	an invalid tid value was specified. pvmd is not responding. cannot get host daemon tid no host name for the given tid found.		
Example			
cppvmConnection myProcess; cout << "Host name: " << myProcess.getHostName();			
Discussion			
The memory of the returned string is reused for every call.			

Table 11.28: cppvmConnection: getHostName

getHostName			cppvmConnection
get the host name of a task specified by its task name			
Synopsis			
char *getHostName(char* taskname);			
Parameter			
Parameter	Type	Default	Description
taskname	in		task name
char*	return		host name
Exceptions			
cppvmExceptionBadResult	an invalid tid value was specified. pvmd is not responding. cannot get host daemon tid no host name for the given tid found.		
Example			
cppvmConnection myProcess; cout << "Host name: " << myProcess.getHostName("calculatePrg");			
Discussion			
The memory of the returned string is reused for every call.			

Table 11.29: cppvmConnection: getHostName

getHostName	cppvmConnection
get the host name of a task specified by a task object	
Synopsis	
	continued on next page...

...continued from previous page			
`getHostName`			cppvmConnection
char *getHostName(cppvmTask& task);			
Parameter			
Parameter	Type	Default	Description
`task`	in		task object
`char*`	return		host name
Exceptions			
cppvmExceptionBadResult	an invalid tid value was specified. pvmd is not responding. cannot get host daemon tid no host name for the given tid found.		
Example			

```
cppvmConnection myProcess;
cppvmTask myTask;
...
cout << "Host name: " << myProcess.getHostName(myTask);
```

Discussion
The memory of the returned string is reused for every call.

Table 11.30: cppvmConnection: `getHostName`

`getHostName`			cppvmConnection
get the host name of a task specified by its task id			
Synopsis			
char *getHostName(int tid);			
Parameter			
Parameter	Type	Default	Description
`tid`	in		task id
`char*`	return		host name
Exceptions			
cppvmExceptionBadResult	an invalid tid value was specified. pvmd is not responding. cannot get host daemon tid no host name for the given tid found.		
Example			

```
cppvmConnection myProcess;
cout << "Host name: " << myProcess.getHostName(11111);
```

Discussion
The memory of the returned string is reused for every call.

Table 11.31: cppvmConnection: getHostSpeed

`getHostSpeed`			cppvmConnection
get the host speed for this process			
Synopsis			
int getHostSpeed();			
Parameter			
Parameter	Type	Default	Description
`int`	return		host speed
Exceptions			
cppvmExceptionBadResult	an invalid tid value was specified. pvmd is not responding. cannot get host daemon tid no host speed for the given tid found.		
Example			
`cppvmConnection myProcess;` `cout << "Host speed: " << myProcess.getHostSpeed();`			

Table 11.32: cppvmConnection: getHostSpeed

`getHostSpeed`			cppvmConnection
get the host speed of a task specified by its task name			
Synopsis			
int getHostSpeed(char* taskname);			
Parameter			
Parameter	Type	Default	Description
`taskname`	in		task name
`int`	return		host speed
Exceptions			
cppvmExceptionBadResult	an invalid tid value was specified. pvmd is not responding. cannot get host daemon tid no host speed for the given tid found.		
Example			
`cppvmConnection myProcess;` `cout << "Host speed: " << myProcess.getHostSpeed("calculatePrg");`			

Table 11.33: cppvmConnection: getHostSpeed

`getHostSpeed`	cppvmConnection
get the host speed of a task specified by a task object	
Synopsis	
	continued on next page...

...continued from previous page			
getHostSpeed			cppvmConnection
int getHostSpeed(cppvmTask& task);			
Parameter			
Parameter	Type	Default	Description
`task`	in		task object
`int`	return		host speed
Exceptions			
cppvmExceptionBadResult	an invalid tid value was specified. pvmd is not responding. cannot get host daemon tid no host speed for the given tid found.		
Example			

```
cppvmConnection myProcess;
cppvmTask myTask;
...
cout << "Host speed: " << myProcess.getHostSpeed(myTask);
```

Table 11.34: cppvmConnection: `getHostSpeed`

getHostSpeed			cppvmConnection
get the host speed of a task specified by its task id			
Synopsis			
int getHostSpeed(int tid);			
Parameter			
Parameter	Type	Default	Description
`tid`	in		task id
`int`	return		host speed
Exceptions			
cppvmExceptionBadResult	an invalid tid value was specified. pvmd is not responding. cannot get host daemon tid no host speed for the given tid found.		
Example			

```
cppvmConnection myProcess;
cout << "Host speed: " << myProcess.getHostSpeed(11111);
```

Table 11.35: cppvmConnection: `getHostStat`

getHostStat	cppvmConnection
get status of a specified host	
	continued on next page...

... continued from previous page	
`getHostStat`	cppvmConnection

Synopsis

int getHostStat(char *host);

Parameter

Parameter	Type	Default	Description
`host`	in		name of the host to get status from
`int`	return		host status: `PvmOk`: host ok; `PvmNoHost`: host is not in virtual machine; `PvmHostFail`: host is unreachable (and thus possibly failed)

Example

```
cppvmConnection myProcess;
int status;
status = myProcess.getHostStat("sun11");
```

Table 11.36: cppvmConnection: `getInst`

`getInst`	cppvmConnection
get the instance number in a group of a PVM process	

Synopsis

int getInst(char *group, int tid);

Parameter

Parameter	Type	Default	Description
`group`	in		group name
`tid`	in		task id
`int`	return		instance number; starts at 0 and count up; values less than zero indicate an error

Example

```
cppvmConnection myProcess;
int inum;
inum = myProcess.getInst("server group", 123456);
```

Table 11.37: cppvmConnection: `getMsgBufInfo`

`getMsgBufInfo`	cppvmConnection
get informations about the last message	

Synopsis

void getMsgBufInfo(int bufid, int *bytes, int *msgtag, int *tid);

continued on next page...

...continued from previous page			
`getMsgBufInfo`			cppvmConnection
Parameter			
Parameter	Type	Default	Description
`bufid`	in		id of the message buffer
`bytes`	out		length of the message in bytes
`msgtag`	out		message label
`tid`	out		source of the message
Exceptions			
cppvmExceptionBadParameter	one of the parameter is a NULL pointer		
cppvmExceptionBadResult	could not get buffer informations		
Example			

```
cppvmConnection myProcess;
myProcess.getMsgBufInfo(100, &bytes, &msgTag, &tid);
```

Table 11.38: cppvmConnection: `getMyInst`

`getMyInst`			cppvmConnection
get the instance number of this process in a group			
Synopsis			
int getMyInst(char *group);			
Parameter			
Parameter	Type	Default	Description
`group`	in		group name
`int`	return		instance number; starts at 0 and count up; values less than zero indicate an error
Example			

```
cppvmConnection myProcess;
int inum;
inum = myProcess.getMyInst(server group");
```

Table 11.39: cppvmConnection: `getMyTask`

`getMyTask`			cppvmConnection
get the description of the current process as cppvmTask object			
Synopsis			
cppvmTask getMyTask();			
Parameter			
Parameter	Type	Default	Description
`cppvmTask`	return		task description
Example			

continued on next page...

...continued from previous page	
`getMyTask`	cppvmConnection

```
cppvmConnection myProcess;
cppvmTask taskDesc;
...
taskDesc = myProcess.getMyTask();
```

Discussion
A cppvmConnection object cannot be sent to other PVM processes. Use objects of the class cppvmTask (see section 11.16) instead.

Table 11.40: cppvmConnection: `getParentTask`

`getParentTask`			cppvmConnection
get the description of the parent process of this process as cppvmTask object			
Synopsis			
cppvmTask getParentTask();			
Parameter			
Parameter	Type	Default	Description
`cppvmTask`	return		task description
Example			
`cppvmConnection myProcess; cppvmTask taskDesc; ...taskDesc = myProcess.getParentTask();`			
Discussion			
A cppvmConnection object cannot be sent to other PVM processes. Use objects of the class cppvmTask instead.			

Table 11.41: cppvmConnection: `getParentTid`

`getParentTid`			cppvmConnection
get the task id of the parent process of this process			
Synopsis			
int getParentTid();			
Parameter			
Parameter	Type	Default	Description
`int`	return		task id of the parent process; `PvmNoParent` if this process has no parent process
Example			

```
cppvmConnection myProcess;
int parentTid;
parentTid = myProcess.getParentTid();
```

Table 11.42: cppvmConnection: getTaskStat

getTaskStat			cppvmConnection
get status of a specified PVM process			
Synopsis			
int getTaskStat(char *taskname);			
Parameter			
Parameter	Type	Default	Description
taskname	in		name of the task to get status from
int	return		task status: `PvmOk`: task is running `PvmBadParam`: bad parameter (most likely an invalid tid value) `PvmSysErr`: pvmd not responding `PvmNoTask`: task not running
Exceptions			
cppvmExceptionBadParameter	taskName is a NULL pointer		
Example			

`cppvmConnection myProcess;`
`int status;`
`status = myProcess.getTaskStat("childproc");`			

Discussion			
This method can only be used for spawned processes. PVM does not know the names of processes executed by the user in a shell. If more than one processes have the name `taskName`, the state of the first process in the list delivered by the method `taskInfo` is given back.			

Table 11.43: cppvmConnection: getTaskStat

getTaskStat			cppvmConnection
get status of a specified PVM process			
Synopsis			
int getTaskStat(cppvmTask& task);			
Parameter			
Parameter	Type	Default	Description
task	in		task description of the process to get status from
			continued on next page...

. . . continued from previous page

getTaskStat			cppvmConnection
int	return		task status: `PvmOk`: task is running `PvmBadParam`: bad parameter (most likely an invalid tid value) `PvmSysErr`: pvmd not responding `PvmNoTask`: task not running

Example

```
cppvmConnection myProcess;
cppvmTask myTask;
...
int status;
status = myProcess.getTaskStat(myTask);
```

Table 11.44: cppvmConnection: `getTaskStat`

getTaskStat			cppvmConnection
get status of a specified PVM process			

Synopsis

int getTaskStat(int tid);

Parameter

Parameter	Type	Default	Description
tid	in		task id of the process to get status from
int	return		task status: `PvmOk`: task is running `PvmBadParam`: bad parameter (most likely an invalid tid value) `PvmSysErr`: pvmd not responding `PvmNoTask`: task not running

Example

```
cppvmConnection myProcess;
int status;
status = myProcess.getTaskStat(123456);
```

Table 11.45: cppvmConnection: getTid

getTid			cppvmConnection
get the task id of the current process			
Synopsis			
int getTid();			
Parameter			
Parameter	Type	Default	Description
int	return		task id of the current process
Example			

```
cppvmConnection myProcess;
int myTid;
myTid = myProcess.getTid();
```

Table 11.46: cppvmConnection: getTid

getTid			cppvmConnection
get the task id of the process identified by a group name and instance number			
Synopsis			
int getTid(char *group, int inum);			
Parameter			
Parameter	Type	Default	Description
`group`	in		group name
`inum`	in		instance number of the process in the group
`int`	return		task id; values <0 indicate an error
Example			

```
cppvmConnection myProcess;
int tid;
tid = myProcess.getTid("server group",2);
```

Table 11.47: cppvmConnection: getTidOfTask

getTidOfTask			cppvmConnection
get the task id of a task running in pvm specified by its name			
Synopsis			
int getTidOfTask(char *taskName);			
Parameter			
Parameter	Type	Default	Description
`taskName`	in		name of the PVM task
			continued on next page...

...continued from previous page

getTidOfTask		cppvmConnection
`int`	return	`>0`: task id of the process `PvmTaskChild`: no process with this name enrolled in PVM

Exceptions	
cppvmExceptionBadParameter	taskName is a NULL pointer

Example

```
cppvmConnection myProcess;
int tid = myProcess.getTidOfTask("hello2");
```

Discussion

This method can only be used for spawned processes. PVM does not know the names of processes executed by the user in a shell. If more than one processes have the name `taskName`, the task id of the first process in the list delivered by the method `taskInfo` is given back.

Table 11.48: cppvmConnection: `getTMask`

`getTMask`			cppvmConnection
get trace mask of a task or its children			

Synopsis

void getTMask(int who, Pvmtmask mask);

Parameter

Parameter	Type	Default	Description
`who`	in		specifies which trace mask to get `PvmTaskSelf`: the current task `PvmTaskChild`: children tasks to be spawned
`mask`	out		trace mask

Exceptions	
cppvmExceptionBadResult	invalid value for argument 'who'

Example

```
cppvmConnection myProcess;
Pvmtmask m;
myProcess.getTMask(PvmTaskChild, m);
```

Discussion

See the PVM-function `pvm_gettmask` for more information.

Table 11.49: cppvmConnection: halt

`halt`	cppvmConnection
shut down the entire PVM system	
Synopsis	
void halt();	
Exceptions	
cppvmExceptionBadResult	local pvmd is not responding
Example	
`cppvmConnection myProcess; myProcess.halt();`	
Discussion	
Kill all processes and all PVM-daemons in the parallel virtual machine. It is recommended to use this method to shut down your CPPVM program. Otherwise PVM daemons or processes can still run after your main program has ended. This can cause errors if you start the program again. If you have this problem, start a pvm console and type `halt`. To disconnect from PVM without a PVM shutdown, use the method `exit`.	

Table 11.50: cppvmConnection: hostInfo

`hostInfo`			cppvmConnection
configuration of the virtual machine			
Synopsis			
void hostInfo(int *nhost, int *narch, struct pvmhostinfo **hostp);			
Parameter			
Parameter	Type	Default	Description
`nhost`	inout		number of hosts on the virtual machine
`narch`	inout		number of different data formats being used
`pvmhostinfo`	inout		pointer to an array of structures containing information about each host
Exceptions			
cppvmExceptionBadResult	could not get configuration informations of the virtual machine		
Example			

```
cppvmConnection myProcess;
struct pvmhostinfo *hostp;
int nhost, narch;
...
myProcess.hostInfo(&nhost, &narch, &hostp);
```

Discussion

continued on next page...

. . . continued from previous page	
`hostInfo`	cppvmConnection

The memory for `hostp` is allocated by this method and freed on the next call

```
struct pvmhostinfo{
int hi_tid; // host task id
char *hi_name; // host name
char *hi_arch; // host architecture
int hi_speed; // host speed
}
```

Table 11.51: cppvmConnection: `joinGroup`

`joinGroup`	cppvmConnection
enrolls the calling process in a named group	

Synopsis

int joinGroup(char *group);

Parameter

Parameter	Type	Default	Description
`group`	in		group name
`int`	return		instance number (starts at 0 and count up)

Exceptions

cppvmExceptionBadResult	giving a NULL group name
	pvmd was not started or has crashed

Example

```
cppvmConnection myProcess;
myProcess.joinGroup("server group");
```

Table 11.52: cppvmConnection: `leaveGroup`

`leaveGroup`	cppvmConnection
unenrolls the calling process from a named group	

Synopsis

void leaveGroup(char *group);

Parameter

Parameter	Type	Default	Description
`group`	in		group name

Exceptions

cppvmExceptionBadResult	giving a non existent group name
	giving a NULL group name
	asking to leave a group you are not a member of
	pvmd was not started or has crashed

continued on next page. . .

. . . continued from previous page	
`leaveGroup`	cppvmConnection

Example
`cppvmConnection myProcess;` `myProcess.leaveGroup("server group");`

Table 11.53: cppvmConnection: `msgIsObjOfClass`

`msgIsObjOfClass`			cppvmConnection
checks whether the actual message has the given message tag; this message tag determines also the class of the incoming object so you can check which object is in the message buffer! If the message buffer is empty, false is returned			
Synopsis			
bool msgIsObjOfClass(int classMsgTag);			
Parameter			
Parameter	Type	Default	Description
`classMsgTag`	in		class message tag
`bool`	return		**`true:`** object of the given type is in message buffer **`false:`** there are other data in the message buffer or the message buffer is empty
Example			
`cppvmConnection myProcess;` `if(myProcess.msgIsObjOfClass(200)){ ...`			

Table 11.54: cppvmConnection: `msgIsObjOfClass`

`msgIsObjOfClass`			cppvmConnection
checks whether the actual message has the given message tag; this message tag determines also the class of the incoming object so you can check which object is in the message buffer! If the message buffer is empty, false is returned			
Synopsis			
bool msgIsObjOfClass(cppvmObject& obj);			
Parameter			
Parameter	Type	Default	Description
`obj`	in		object, for which should be checked whether there is a message for it or not
`bool`	return		**`true:`** object of the given type is in message buffer **`false:`** there are other data in the message buffer or the message buffer is empty
Example			
			continued on next page. . .

... continued from previous page

`msgIsObjOfClass`	cppvmConnection

```
cppvmConnection myProcess;
cppvmInt intObj;
if(myProcess.msgIsObjOfClass(intObj)){ ...
```

Table 11.55: cppvmConnection: `operator=`

`operator=`			cppvmConnection
assign object			
Synopsis			
`cppvmConnection& operator= (const cppvmConnection& obj)`			
Parameter			
Parameter	Type	Default	Description
`obj`	in		object to assign
`cppvmConnection`	return		this object
Example			

```
cppvmConnection c1, c2;
...
c1 = c2;
```

Table 11.56: cppvmConnection: `operator==`

`operator==`			cppvmConnection
test if objects are equal			
Synopsis			
`bool operator== (const cppvmConnection& obj)`			
Parameter			
Parameter	Type	Default	Description
`obj`	in		object to compare with
`bool`	return		compare result
Example			

```
cppvmConnection c1, c2;
...
if(c1==c2){ ...
```

Table 11.57: cppvmConnection: `operator!=`

`operator!=`	cppvmConnection
test if objects are not equal	
Synopsis	

continued on next page...

. . . continued from previous page	
`operator!=`	cppvmConnection

`bool operator!= (const cppvmConnection& obj)`

Parameter			
Parameter	Type	Default	Description
`obj`	in		object to compare with
`bool`	return		compare result

Example

```
cppvmConnection c1, c2;
...
if(c1!=c2){ ...
```

Table 11.58: cppvmConnection: `operator<<`

`operator<<`	cppvmConnection
write object on ostream	

Synopsis

```
ostream& operator<<(ostream& strm, cppvmConnection& obj);
```

Parameter			
Parameter	Type	Default	Description
`obj`	in		write this object on ostream
`strm`	in		ostream
`ostream`	return		ostream after writing the object

Example

```
cppvmConnection c;
cout << c;
```

Table 11.59: cppvmConnection: `printHostInfo`

`printHostInfo`	cppvmConnection
print information about the configuration of the virtual machine to cout	

Synopsis

void printHostInfo();

Exceptions	
cppvmExceptionBadResult	could not get configuration informations of the virtual machine

Example

```
cppvmConnection myProcess;
myProcess.printHostInfo();
```

Table 11.60: cppvmConnection: printTaskInfo

`printTaskInfo`	cppvmConnection
print information about all tasks running in the PVM–system	
Synopsis	
void printTaskInfo();	
Exceptions	
cppvmExceptionBadParameter	one of the parameter is a NULL pointer
cppvmExceptionBadResult	could not get task informations of the virtual machine
Example	

```
cppvmConnection myProcess;
myProcess.printTaskInfo();
```

Table 11.61: cppvmConnection: printOneHostInfo

`printOneHostInfo`			cppvmConnection
print information about one host			
Synopsis			
void printOneHostInfo(struct pvmhostinfo& hStruct);			
Parameter			
Parameter	Type	Default	Description
`hStruct`	in		host struct
Example			

```
cppvmConnection myProcess;
struct pvmhostinfo *hostp;
...
myProcess.printOneHostInfo(hostp[0]);
```

Discussion

```
struct pvmhostinfo{
int hi_tid; // host task id
char *hi_name; // host name
char *hi_arch; // host architecture
int hi_speed; // host speed
}
```

Table 11.62: cppvmConnection: printOneTaskInfo

`printOneTaskInfo`	cppvmConnection
print information about one task	
Synopsis	
void printOneTaskInfo(struct pvmtaskinfo& tStruct);	
Parameter	

continued on next page...

. . . continued from previous page			
`printOneTaskInfo`			cppvmConnection
Parameter	Type	Default	Description
`tStruct`	in		task struct

Example

```
cppvmConnection myProcess;
struct pvmtaskinfo *taskp;
...
myProcess.printOneTaskInfo(taskp[0]);
```

Discussion

```
struct pvmtaskinfo{
int ti_tid; // task id
int ti_ptid; // parent task id
int ti_host; // host task id
int ti_flag; // task status
char *ti_a_out; // name of spawned task. Manually started
// tasks return blank
}
```

Table 11.63: cppvmConnection: `reduce`

`reduce`	cppvmConnection
performs a reduction operation over members of the specified group	

Synopsis

void reduce(void (*func)(int*, void*, void*, int*, int*),
void *data, int count, int datatype,
int msgtag, char *group, int rootginst)

Parameter

Parameter	Type	Default	Description
`func`	in		function which defines the operation performed on the global data
`data`	inout		pointer to the starting address of an array of local values. On return, the data array on the root will be overwritten with the result of the reduce operation. For the other members of the group the values after the operation are not defined
`count`	in		integer specifying the number of elements of datatype in the data array
`datatype`	in		type of the entries in the data array
`msgtag`	in		message tag for the reduce operation
`group`	in		group name; the group must exist and the calling process must be a member of the group
`rootginst`	in		instance number of group member who gets the result

continued on next page. . .

... continued from previous page	
`reduce`	cppvmConnection

Exceptions	
cppvmExceptionBadResult	calling task is not in the group the datatype specified is not appropriate pvm system error

Example

```
cppvmConnection myProcess;
myProcess.reduce(PvmMax, MYVALS, COUNT, REAL4, MTAG,
"server group", rootginst);
```

Discussion

See the PVM-function `pvm_reduce` for more information.

Table 11.64: cppvmConnection: `scatter`

`scatter`	cppvmConnection
send from a specified member of the group to each member of a group a section of an array	

Synopsis

void scatter(void *result, void *data, int count, int datatype,
int msgtag, char *group, int rootginst)

Parameter

Parameter	Type	Default	Description
`result`	inout		pointer to the starting address of an array datatype which will be overwritten by the message from the specified root member of the group
`data`	inout		on the root this is a pointer to the starting address of an array datatype of local values which are to be distributed to the members of the group if n is the number of members in the group, then this array of datatype should be of length at least n * count
`count`	in		integer specifying the number of elements of datatype to be sent to each member of the group from the root
`datatype`	in		type of the entries in the data array
`msgtag`	in		message tag for the scatter operation
`group`	in		group name; the group must exist and the calling process must be a member of the group
`rootginst`	in		instance number of group member who performs the scatter of its array to the members of the group

Exceptions	
cppvmExceptionBadResult	calling task is not in the group the datatype specified is not appropriate pvm system error

continued on next page...

. . . continued from previous page	
`scatter`	cppvmConnection

Example

`cppvmConnection myProcess;`
`myProcess.scatter(&getmyrow, &matrix, 10, PVM_INT,`
`msgtag, "server group", rootginst);`

Discussion
See the PVM-function `pvm_scatter` for more information.

Table 11.65: cppvmConnection: `setDebugLevel`

`setDebugLevel`			cppvmConnection
set global debug level			
Synopsis			
void setDebugLevel(int debugLevel);			
Parameter			
Parameter	Type	Default	Description
debugLevel	in		new global debug level
Exceptions			
cppvmExceptionBadResult	cannot create new PVM send buffer unable to allocate memory to pack data error writing in database		
Example			
`cppvmConnection pvmProcess;` `pvmProcess.setDebugLevel(3);`			
Discussion			
The CPPVM debug level can be used by every process connected to PVM. It might be useful for a small information exchange between modules written by different developers.			

Table 11.66: cppvmConnection: `setOpt`

`setOpt`			cppvmConnection
set various libpvm option			
Synopsis			
void setOpt(int what, int val);			
Parameter			
Parameter	Type	Default	Description
			continued on next page. . .

. . . continued from previous page		
setOpt		cppvmConnection
what	in	option what is being set: CPPvmMsgEncoding: set local/global message encoding For more informations see the pvm_setopt manpages.
val	in	values for CPPvmMsgEncoding: CPPvmAllPvmDataLocal: use the local settings of the send streams CPPvmAllPvmDataDefault: global XDR encoding (works everytime) CPPvmAllPvmDataRaw: do not encode information (only for send/receive between task on homogeneous architectures); global setting CPPvmAllPvmDataInPlace: do not copy data in sending buffer, use the data memory instead. Global setting. ATTENTION: the data memory MUST not deleted between packing and sending! For more informations see the pvm_setopt manpages.

Exceptions

cppvmExceptionBadResult	could not set option
cppvmExceptionBadParameter	unknown 'val' for 'what'=CPPvmMsgEncoding

Example

```
cppvmConnection myProcess;
myProcess.setOpt(PvmRoute, PvmRouteDirect);
```

Discussion

See the PVM-function pvm_setopt for more information.

Table 11.67: cppvmConnection: setTMask

setTMask	cppvmConnection
set trace mask of a task or its children	

Synopsis

void setTMask(int who, Pvmtmask mask);

Parameter

Parameter	Type	Default	Description

continued on next page. . .

. . . continued from previous page		
setTMask		cppvmConnection
who	in	specifies which trace mask to get PvmTaskSelf: the current task PvmTaskChild: children tasks to be spawned
mask	in	trace mask
Exceptions		
cppvmExceptionBadResult	invalid value for argument 'who'	
Example		

```
cppvmConnection myProcess;
Pvmtmask m;
...
myProcess.setTMask(PvmTaskChild, m);
```

Discussion
See the PVM-function pvm_settmask for more information.

Table 11.68: cppvmConnection: taskInfo

taskInfo			cppvmConnection
information about tasks running on the virtual machine			
Synopsis			
void taskInfo(int where,int *ntask, struct pvmtaskinfo **taskp);			
Parameter			
Parameter	Type	Default	Description
where	in		information specifier: 0: all tasks on the virtual machine pvmd tid: all task on this host (tid of the pvm daemon, use getHostDaemonTid to get this id) tid: specific task
ntask	inout		number of task (depending on where)
taskp	inout		pointer to an array of structures which contain information about each task
Exceptions			
cppvmExceptionBadParameter	one of the parameter is a NULL pointer		
cppvmExceptionBadResult	could not get task informations of the virtual machine		
Example			

continued on next page. . .

...continued from previous page	
taskInfo	cppvmConnection

```
cppvmConnection myProcess;
int ntask;
struct pvmtaskinfo *taskp;
...
myProcess.taskInfo(0, &ntask, &taskp);
```

Discussion

The memory for `hostp` is allocated by this method and freed on the next call

```
struct pvmtaskinfo{
int ti_tid; // task id
int ti_ptid; // parent task id
int ti_host; // host task id
int ti_flag; // task status
char *ti_a_out; // name of spawned task. Manually started
// tasks return blank
}
```

Table 11.69: cppvmConnection: `unexportEnvVar`

unexportEnvVar	cppvmConnection
unmark environment variables to export through spawn	

Synopsis

void unexportEnvVar(char *environmentVar);

Parameter

Parameter	Type	Default	Description
`environmentVar`	in		name of an environment variable to remove from the export list

Example

```
cppvmConnection myProcess;
myProcess.unexportEnvVar("TERM");
```

Discussion

The method edits the environment variable `PVM_EXPORT`. If you want to export environment variables to a process, you must edit `PVM_EXPORT` before spawning this process.

There are also interface methods to several other PVM functions. They are only of interest for advanced PVM programming and not necessary for normal data transfer. Please refer the PVM documentation for further information about the equivalent pvm functions:

cppvmConnection **method**	**equivalent** PVM **function**
`archcode`	`pvm_archcode`
`freebuf`	`pvm_freebuf`
`getfds`	`pvm_getfds`
`getopt`	`pvm_getopt`
`getrbuf`	`pvm_getrbuf`
`getsbuf`	`pvm_getsbuf`
`hostsync`	`pvm_hostsync`
`siblings`	`pvm_siblings`
`addMhf`	`pvm_addmhf`
`delMhf`	`pvm_delmhf`
`pkMesg`	`pvm_pkmesg`
`pkMesgBody`	`pvm_pkmesgbody`
`upkMesg`	`pvm_upkmesg`
`mkbuf`	`pvm_mkbuf`
`notify`	`pvm_notify`
`perror`	`pvm_perror`
`probe`	`pvm_probe`
`recvf`	`pvm_recvf`
`regHoster`	`pvm_reg_hoster`
`regRm`	`pvm_reg_rm`
`regTasker`	`pvm_reg_tasker`
`sendsig`	`pvm_sendsig`
`setrbuf`	`pvm_setrbuf`
`setsbuf`	`pvm_setsbuf`
`tidtohost`	`pvm_tidtohost`

11.2 cppvmSpawnConnection

The class cppvmSpawnConnection is derived from the class cppvmConnection. It inherits the methods described in section 11.1 and adds the facility to spawn processes on hosts of the parallel virtual machine.

The processes can be spawned on the same host as the parent process or on any other host which has PVM and CPPVM installed and is connected to the host of the parent process. The user can specify a host or an architecture for the new process. Otherwise PVM can choose any machine to start the task on. The machine must be part of the current parallel virtual machine and is chosen by an easy load balancing algorithm. Processes can also be spawned into new contexts. This helps to avoid miss-leaded messages.

The objects of this class cannot be transfered to other processes because they contain informations about the parent and child process. If a transfered object of this class would be used to initialize a send/receive stream, this informations are false. Objects of the class `cppvmTask` can be used instead to transfer information about the process (see methods `getMyTask()` and `getParentTask()`).

Table 11.70: cppvmSpawnConnection: `constructor`

`constructor`			cppvmSpawnConnection
constructor			
Synopsis			

`cppvmSpawnConnection(char *task, char *args,`
`int optionFlag, char *where,`
`cppvmContextMode contextFlag, int contextId);`			

Parameter			
Parameter	Type	Default	Description
`task`	in		task name (program) to start
`args`	in	""	command line arguments for the task
			continued on next page...

. . . continued from previous page

`constructor`			cppvmSpawnConnection
`optionFlag`	in	PvmTaskDefault	spawn option: `PvmTaskDefault:` PVM can choose any machine `PvmTaskHost:` `where` specifies a particular host `PvmTaskArch:` `where` specifies a type of architecture `PvmTaskDebug:` start up processes under debugger; the file $PVM_ROOT/lib/debugger specifies the called debugger `PvmTaskTrace:` process will generate PVM trace data `PvmMppFront:` start process on MPP front-end `PvmHostCompl:` use complement host set Some combinations of the values are possible as well. For example `PvmTaskHost+PvmTaskDebug` will spawn the process on a host specified by `where` within a debugger.
`where`	in	""	specifies where to start the task: `where` depends on the value of `optionFlag` if `optionFlag` is 0, then `where` is ignored and PVM will select the most appropriate host
`contextFlag`	in	CPPvmCurrentContext	specifies the context of the child process: `CPPvmCurrentContext:` spawn within the context of the parent process `CPPvmNewContext:` create a new context and spawn the child process within this new context `CPPvmGivenContext:` spawn the child process in the context given by `contextId`
`contextId`	in	0	specifies a context; only used if `contextFlag` is set to `CPPvmGivenContext`

Exceptions	
cppvmExceptionBadParameter	task name is NULL argv is NULL `PvmTaskHost` is set and the host specifier `where` is NULL or an empty string

continued on next page. . .

. . . continued from previous page	
`constructor`	cppvmSpawnConnection
cppvmExceptionBadResult	invalid argument value specified host is not in the virtual machine specified executable cannot be found. The default location is in $PVM_ROOT/bin/$PVM_ARCH malloc failed. Not enough memory on host pvmd not responding out of resources cannot spawn process

Example

```
// Spawn the process "childprg2" with no arguments on a host
// chosen by PVM. Spawn the process within the current context.
cppvmSpawnConnection myChild1("childprg2");

// Spawn the process 'childprg' with the arguments '-d -h'
// on the host "sun2" within a debugger. Spawn the process
// within the context "100".
cppvmSpawnConnection myChild2("childprg", "-d -h",
PvmTaskHost+PvmTaskDebug,
"sun2", CPPvmGivenContext, 100);
```

Discussion

Stderr and stdout of all spawned tasks are logged in a file (`/tmp/pvml.<uid>`). Use cppvmCatchout to redirect this output. The child process is enrolled in the current context.

Table 11.71: cppvmSpawnConnection: `constructor`

`constructor`	cppvmSpawnConnection
constructor	

Synopsis

```
cppvmSpawnConnection(char *task, char **argv,
int optionFlag, char *where,
cppvmContextMode contextFlag, int contextId);
```

Parameter

Parameter	Type	Default	Description
`task`	in		task name (program) to start
`argv`	in		vector of command line arguments for the task; the end of the array is indicated by a 'NULL' pointer

continued on next page. . .

. . . continued from previous page

constructor			cppvmSpawnConnection
`optionFlag`	in	PvmTaskDefault	spawn option: `PvmTaskDefault:` PVM can choose any machine `PvmTaskHost:` `where` specifies a particular host `PvmTaskArch:` `where` specifies a type of architecture `PvmTaskDebug:` start up processes under debugger; the file $PVM_ROOT/lib/debugger specifies the called debugger `PvmTaskTrace:` process will generate PVM trace data `PvmMppFront:` start process on MPP front-end `PvmHostCompl:` use complement host set Some combinations of the values are possible as well. For example `PvmTaskHost+PvmTaskDebug` will spawn the process on a host specified by `where` within a debugger.
`where`	in	""	specifies where to start the task: `where` depends on the value of `optionFlag` if `optionFlag` is 0, then `where` is ignored and PVM will select the most appropriate host
`contextFlag`	in	CPPvmCurrentContext	specifies the context of the child process: `CPPvmCurrentContext:` spawn within the context of the parent process `CPPvmNewContext:` create a new context and spawn the child process within this new context `CPPvmGivenContext:` spawn the child process in the context given by `contextId`
`contextId`	in	0	specifies a context; only used if `contextFlag` is set to `CPPvmGivenContext`

Exceptions	
cppvmExceptionBadParameter	task name is NULL argv is NULL `PvmTaskHost` is set and the host specifier `where` is NULL or an empty string

continued on next page. . .

. . . continued from previous page	
`constructor`	cppvmSpawnConnection

cppvmExceptionBadResult	invalid argument value specified host is not in the virtual machine specified executable cannot be found. The default location is in $PVM_ROOT/bin/$PVM_ARCH malloc failed. Not enough memory on host pvmd not responding out of resources cannot spawn process

Example

```
// command line arguments
char *argsArray[3];

// initialize argument
// the end of the array is indicated by a 'NULL'
argsArray[0] = "-v";
argsArray[1] = "-c waitForRequests";
argsArray[2] = NULL;

// Spawn the process "childprg3" with given arguments on a host
// chosen by PVM. Spawn the process within the current context.
cppvmSpawnConnection myChild3("childprg3", argsArray);
```

Discussion

Stderr and stdout of all spawned tasks are logged in a file (`/tmp/pvml.<uid>`). Use cppvmCatchout to redirect this output. The child process is enrolled in the current context.

Table 11.72: cppvmSpawnConnection: `getChildHostArchitecture`

`getChildHostArchitecture`	cppvmSpawnConnection
get the host architecture of the child process	

Synopsis

cppvmString getChildHostArchitecture();

Parameter

Parameter	Type	Default	Description
`cppvmString`	return		host architecture

Exceptions

cppvmExceptionForbiddenCall	spawned process was killed, it is not allowed to use it

Example

continued on next page. . .

...continued from previous page	
`getChildHostArchitecture`	cppvmSpawnConnection

```
cppvmSpawnConnection myChild("server");
cout << "Host architecture of the child process: "
<< myChild.getChildHostArchitecture();
```

Table 11.73: cppvmSpawnConnection: `getChildHostDaemonTid`

`getChildHostDaemonTid`			cppvmSpawnConnection
get the task id of the PVM-daemon for the child process			
Synopsis			
int getChildHostDaemonTid();			
Parameter			
Parameter	Type	Default	Description
`int`	return		daemon task id for the daemon of the child process
Exceptions			
cppvmExceptionForbiddenCall	spawned process was killed, it is not allowed to use it		
Example			

```
cppvmSpawnConnection myChild("server");
cout << "Host daemon task id of child process: "
<< myChild.getChildHostDaemonTid();
```

Table 11.74: cppvmSpawnConnection: `getChildHostName`

`getChildHostName`			cppvmSpawnConnection
get the host name of the child process			
Synopsis			
cppvmString getChildHostName();			
Parameter			
Parameter	Type	Default	Description
`cppvmString`	return		host name
Exceptions			
cppvmExceptionForbiddenCall	spawned process was killed, it is not allowed to use it		
Example			

```
cppvmSpawnConnection myChild("server");
cout << "Host name of the child process: "
<< myChild.getChildHostName();
```

Table 11.75: cppvmSpawnConnection: getChildHostSpeed

getChildHostSpeed			cppvmSpawnConnection
get the host speed of the child process			
Synopsis			
int getChildHostSpeed();			
Parameter			
Parameter	Type	Default	Description
`int`	return		host speed
Exceptions			
cppvmExceptionForbiddenCall	spawned process was killed, it is not allowed to use it		
Example			

```
cppvmSpawnConnection myChild("server");
cout << "Host speed of the child process: "
<< myChild.getChildHostSpeed();
```

Table 11.76: cppvmSpawnConnection: getChildName

getChildName			cppvmSpawnConnection
get the name of the child process as string			
Synopsis			
cppvmString getChildName();			
Parameter			
Parameter	Type	Default	Description
cppvmString	return		name of the child process
Example			

```
cppvmSpawnConnection myChild("hello");
...
cout << myChild.getChildName();
```

Table 11.77: cppvmSpawnConnection: getChildParams

getChildParams			cppvmSpawnConnection
get the command line parameter of the child process as strings			
Synopsis			
cppvmString getChildParams(unsigned int index);			
Parameter			
Parameter	Type	Default	Description
index	in	0	index of the command line
cppvmString	return		command line parameter of the child process

continued on next page...

... continued from previous page	
getChildParams	cppvmSpawnConnection

Example

`cppvmSpawnConnection myChild("hello");`
`...`
`cout << myChild.getChildParams(0);`

Table 11.78: cppvmSpawnConnection: getChildStat

getChildStat			cppvmSpawnConnection
get status of the child process			
Synopsis			
int getChildStat();			
Parameter			
Parameter	Type	Default	Description
int	return		task status: **PvmOk:** task is running **PvmBadParam:** bad parameter (most likely an invalid tid value) **PvmSysErr:** pvmd not responding **PvmNoTask:** task not running
Example			

`cppvmSpawnConnection myChild("counter");`
`int status;`
`status = myChild.getChildStat();`			

Table 11.79: cppvmSpawnConnection: getChildTask

getChildTask			cppvmSpawnConnection
get the description of the child process of this process as cppvmTask object			
Synopsis			
cppvmTask getChildTask();			
Parameter			
Parameter	Type	Default	Description
cppvmTask	return		task description
Example			
			continued on next page...

. . . continued from previous page	
`getChildTask`	cppvmSpawnConnection

```
cppvmSpawnConnection myChild("hello");
cppvmTask taskDesc;
...
taskDesc = myChild.getChildTask();
```

Discussion
A cppvmSpawnConnection object cannot be sent to other PVM processes. Use objects of the class cppvmTask instead.

Table 11.80: cppvmSpawnConnection: `getChildTid`

`getChildTid`			cppvmSpawnConnection
get the PVM task id of the child process			
Synopsis			
int getChildTid();			
Parameter			
Parameter	Type	Default	Description
int	return		child task id
Exceptions			
cppvmExceptionForbiddenCall	spawned process was killed, it is not allowed to use it		
Example			

```
cppvmSpawnConnection myChild("hello");
...
cout << myChild.getChildTid();
```

Table 11.81: cppvmSpawnConnection: `getEnrolledChildUSec`

`getEnrolledChildUSec`			cppvmSpawnConnection
get the time in microseconds when the child was enrolled in PVM			
Synopsis			
unsigned long getEnrolledChildUSec();			
Parameter			
Parameter	Type	Default	Description
unsigned long	return		time
Exceptions			
cppvmExceptionForbiddenCall	spawned process was killed, it is not allowed to use it		
Example			

continued on next page. . .

. . . continued from previous page	
getEnrolledChildUSec	cppvmSpawnConnection

```
cppvmSpawnConnection myChild("hello");
...
cout << myChild.getEnrolledChildUSec();
```

Table 11.82: cppvmSpawnConnection: getEnrolledChildSecs

getEnrolledChildSecs			cppvmSpawnConnection
get the time in seconds when the child was enrolled in PVM			
Synopsis			
unsigned long getEnrolledChildSecs();			
Parameter			
Parameter	Type	Default	Description
unsigned long	return		time
Exceptions			
cppvmExceptionForbiddenCall	spawned process was killed, it is not allowed to use it		
Example			

```
cppvmSpawnConnection myChild("hello");
...
cout << myChild.getEnrolledChildSecs();
```

Table 11.83: cppvmSpawnConnection: getNumChildParams

getNumChildParams			cppvmSpawnConnection
get the parameter lines number of the child process			
Synopsis			
unsigned int getNumChildParams();			
Parameter			
Parameter	Type	Default	Description
unsigned int	return		parameter lines number
Example			

continued on next page. . .

...continued from previous page	
getNumChildParams	cppvmSpawnConnection

```
// command line arguments
char *argsArray[3];

// initialize argument
// the end of the array is indicated by a 'NULL'
argsArray[0] = "-l";
argsArray[1] = "-run fast";
argsArray[2] = NULL;

cppvmSpawnConnection myChild("calc", argsArray);
unsigned int i;
for(i=0;i<myChild.getNumChildParams();i++){
cout << myChild.getChildParams(i);
}
```

Table 11.84: cppvmSpawnConnection: getChildParentTid

getChildParentTid			cppvmSpawnConnection
get the task id of the child's parent process			
Synopsis			
int getChildParentTid();			
Parameter			
Parameter	Type	Default	Description
int	return		task id of the parent process
Example			

```
cppvmSpawnConnection child("hello2");
int parentTid;
parentTid = child.getChildParentTid();
```

Table 11.85: cppvmSpawnConnection: killChild

killChild	cppvmSpawnConnection
kill the child of this parent process	
Synopsis	
void killChild();	
Exceptions	
cppvmExceptionForbiddenCall	child process was already killed
cppvmExceptionBadResult	cannot kill child process
Example	

continued on next page...

...continued from previous page	
`killChild`	cppvmSpawnConnection

```
cppvmSpawnConnection myChild("hello");
...
myChild.killChild();
```

Table 11.86: cppvmSpawnConnection: `operator==`

`operator==`			cppvmSpawnConnection
test if objects are equal			
Synopsis			
`bool operator== (const cppvmSpawnConnection& obj)`			
Parameter			
Parameter	Type	Default	Description
obj	in		object to compare with
bool	return		compare result
Example			

```
cppvmSpawnConnection s1("myProggy");
cppvmSpawnConnection s2("myProggy");
...
if(s1==s2){ ...
```

Table 11.87: cppvmSpawnConnection: `operator!=`

`operator!=`			cppvmSpawnConnection
test if objects are not equal			
Synopsis			
`bool operator!= (const cppvmSpawnConnection& obj)`			
Parameter			
Parameter	Type	Default	Description
`obj`	in		object to compare with
`bool`	return		compare result
Example			

```
cppvmSpawnConnection s1("myProggy");
cppvmSpawnConnection s2("myProggy2");
...
if(s1!=s2){ ...
```

Table 11.88: cppvmSpawnConnection: `operator<<`

`operator<<`	cppvmSpawnConnection
write object on ostream	

Synopsis
`ostream& operator<<(ostream& strm, cppvmSpawnConnection& obj);`

Parameter	Type	Default	Description
`obj`	in		write this object on ostream
`strm`	in		ostream
`ostream`	return		ostream after writing the object

Example

```
cppvmSpawnConnection s1("myProgg");
cout << s2;
```

11.3 cppvmSpawnMultiConnection

With cppvmSpawnMultiConnection more than one child process can be spawned at one time. The class cppvmSpawnMultiConnection contains a collection of cppvmSpawnConnection objects.

The constructor of cppvmSpawnMultiConnection allows only to spawn multiple children on one host. If you want to add children spawned on other hosts to this collection use the `add` method for cppvmSpawnConnection objects and `append` for cppvmSpawnMultiConnection objects.

Each cppvmSpawnConnection member in the collection can be tagged or un-tagged (methods `tagElem()`, `untagElem()`, `tagAllElems()`, and `untagAllElems()`). The message passing operations of this class concern only the tagged elements. At the beginning, all members are un-tagged.

Like cppvmSpawnConnection objects, it is not possible to transfer objects of this class. A collection of `cppvmTask` objects can be used instead to transfer information about the processes (see methods `getTaggedChildTasks()` and `getAllChildTasks()`).

Table 11.89: cppvmSpawnMultiConnection: `constructor`

`constructor`			cppvmSpawnMultiConnection
constructor			
Synopsis			
`cppvmSpawnMultiConnection(char *task, unsigned int numTasks, char *args, int optionFlag, char *where, cppvmContextMode contextFlag, int contextId, cppvmMultiContextMode multiContextFlag);`			
Parameter			
Parameter	Type	Default	Description
`task`	in		task name (program) to start
`numTasks`	in	1	number of tasks to spawn
`args`	in	""	command line arguments for the tasks (the same for all spawned tasks)
			continued on next page...

. . . continued from previous page			
`constructor`			cppvmSpawnMultiConnection
`optionFlag`	in	PvmTaskDefault	spawn option: `PvmTaskDefault:` PVM can choose any machine `PvmTaskHost:` `where` specifies a particular host `PvmTaskArch:` `where` specifies a type of architecture `PvmTaskDebug:` start up processes under debugger; the file $PVM_ROOT/lib/debugger specifies the called debugger `PvmTaskTrace:` process will generate PVM trace data `PvmMppFront:` start process on MPP front-end `PvmHostCompl:` use complement host set Some combinations of the values are possible as well. For example `PvmTaskHost+PvmTaskDebug` will spawn the process on a host specified by `where` within a debugger.
`where`	in	""	specifies where to start the task: `where` depends on the value of `optionFlag` if `optionFlag` is 0, then `where` is ignored and PVM will select the most appropriate host
`contextFlag`	in	CPPvmCurrentContext	specifies the context of one child process: `CPPvmCurrentContext:` spawn within the context of the parent process `CPPvmNewContext:` create a new context and spawn the child within this new context `CPPvmGivenContext:` spawn the children in the context given by `contextId`
`contextId`	in	0	specifies a context; only used if `contextFlag` is set to `CPPvmGivenContext`
			continued on next page. . .

. . . continued from previous page

`constructor`	cppvmSpawnMultiConnection
`multiContextFlag` in	CPPvmMultiAllNewContext specifies the context of all child processes. this parameter is only used if `contextFlag` is set to `CPPvmNewContext`: `CPPvmOneNewContext`: create one new context and spawn all children within this new context `CPPvmAllNewContext`: create a new context for every child and spawn the children within these contexts

Exceptions

cppvmExceptionBadParameter	task name is NULL argv is NULL `PvmTaskHost` is set and the host specifier `where` is NULL or an empty string cannot add NULL pointer
cppvmExceptionBadResult	invalid argument value specified host is not in the virtual machine specified executable cannot be found. The default location is in $PVM_ROOT/bin/$PVM_ARCH malloc failed. Not enough memory on host pvmd not responding out of resources cannot spawn process

Example

```
// Start eight "childprg" on sun3 with the parameter "-o".
// Generate PVM trace data and spawn all processes
// within one new context
cppvmSpawnMultiConnection myChildren("childprg", 8, "-o",
PvmTaskHost+PvmTaskTrace, "sun3",
CPPvmNewContext, 0, CPPvmOneNewContext);
// Start four "child2" programs with no arguments on a host
// chosen by PVM. Spawn the process within the current context.
cppvmSpawnMultiConnection myOtherChildren("child2", 4);
```

Discussion

All stderr and stdout of spawned tasks are logged in a file (`/tmp/pvml.<uid>`). Use cppvmCatchout to redirect this output. The child process is enrolled in the current context.

Table 11.90: cppvmSpawnMultiConnection: `constructor`

`constructor`	cppvmSpawnMultiConnection
constructor	

continued on next page. . .

...continued from previous page

constructor	cppvmSpawnMultiConnection

Synopsis

```
cppvmSpawnMultiConnection(char *task, unsigned int numTasks, char
**argv, int optionFlag, char *where, cppvmContextMode contextFlag, int
contextId, cppvmMultiContextMode multiContextFlag);
```

Parameter

Parameter	Type	Default	Description
`task`	in		task name (program) to start
`numTasks`	in		number of tasks to spawn
`argv`	in		vector of command line arguments for the tasks (the same for all spawned tasks); the end of the array is indicated by a 'NULL' pointer
`optionFlag`	in	PvmTaskDefault	spawn option: `PvmTaskDefault:` PVM can choose any machine `PvmTaskHost:` `where` specifies a particular host `PvmTaskArch:` `where` specifies a type of architecture `PvmTaskDebug:` start up processes under debugger; the file $PVM_ROOT/lib/debugger specifies the called debugger `PvmTaskTrace:` process will generate PVM trace data `PvmMppFront:` start process on MPP front-end `PvmHostCompl:` use complement host set Some combinations of the values are possible as well. For example `PvmTaskHost+PvmTaskDebug` will spawn the process on a host specified by `where` within a debugger.
`where`	in	""	specifies where to start the task: `where` depends on the value of `optionFlag` if `optionFlag` is 0, then `where` is ignored and PVM will select the most appropriate host

continued on next page...

...continued from previous page

constructor			cppvmSpawnMultiConnection
`contextFlag`	in	CPPvmCurrentContext	specifies the context of one child process: **`CPPvmCurrentContext:`** spawn within the context of the parent process **`CPPvmNewContext:`** create a new context and spawn the child within this new context **`CPPvmGivenContext:`** spawn the children in the context given by `contextId`
`contextId`	in	0	specifies a context; only used if `contextFlag` is set to `CPPvmGivenContext`
`multiContextFlag`	in	CPPvmMultiAllNewContext	specifies the context of all child processes. this parameter is only used if `contextFlag` is set to `CPPvmNewContext`: **`CPPvmOneNewContext:`** create one new context and spawn all children within this new context **`CPPvmAllNewContext:`** create a new context for every child and spawn the children within these contexts

Exceptions

cppvmExceptionBadParameter	task name is NULL argv is NULL `PvmTaskHost` is set and the host specifier `where` is NULL or an empty string cannot add NULL pointer
cppvmExceptionBadResult	invalid argument value specified host is not in the virtual machine specified executable cannot be found. The default location is in $PVM_ROOT/bin/$PVM_ARCH malloc failed. Not enough memory on host pvmd not responding out of resources cannot spawn process

Example

continued on next page...

... continued from previous page	
constructor	cppvmSpawnMultiConnection

```
// command line arguments
char *argsArray[3];

// initialize argument
// the end of the array is indicated by a 'NULL'
argsArray[0] = "-h";
argsArray[1] = "now";
argsArray[2] = NULL;

// Start five "childprg" with given arguments
// on a host chosen by PVM.
// Spawn the process within the current context.
cppvmSpawnMultiConnection myChildren("childprg", 5, argsArray);
```

Discussion

All stderr and stdout of spawned tasks are logged in a file (/tmp/pvml.<uid>). Use cppvmCatchout to redirect this output. The child process is enrolled in the current context.

Table 11.91: cppvmSpawnMultiConnection: add

add	cppvmSpawnMultiConnection
add the pointer to a cppvmSpawnConnection element to the collection	

Synopsis

```
void add(cppvmSpawnConnection* objPtr);
```

Parameter

Parameter	Type	Default	Description
objPtr	in		pointer to a cppvmSpawnConnection object

Exceptions

cppvmExceptionBadParameter	cannot add NULL parameter

Example

```
cppvmSpawnMultiConnection myChildren("childprg", 5);
cppvmSpawnConnection* newChildPtr;
newChildPtr = new cppvmSpawnConnection("otherChildPrg");
myChildren.add(newChildPtr);
```

Discussion

With add several child programs can be combined to one children collection.

Table 11.92: cppvmSpawnMultiConnection: append

append			cppvmSpawnMultiConnection
append another cppvmSpawnMultiConnection collection to this collection			
Synopsis			
void append(cppvmSpawnMultiConnection& obj);			
Parameter			
Parameter	Type	Default	Description
obj	in		cppvmSpawnMultiConnection object
Example			

cppvmSpawnMultiConnection myChildren1("childprg1", 5);
cppvmSpawnMultiConnection myChildren2("childprg2", 3);
myChildren1.append(myChildren2);			

Discussion			
There is no deep copy available for cppvmSpawnMultiConnection. To avoid multiple pointers to one object the appended collection is deleted.			

Table 11.93: cppvmSpawnMultiConnection: getAllChildTasks

getAllChildTasks			cppvmSpawnMultiConnection
get all child tasks of the elements in the spawnCollection			
Synopsis			
cppvmObjectCollection * getAllChildTasks();			
Parameter			
Parameter	Type	Default	Description
cppvmObjectCollection	return		collection filled with cppvmTask objects
Example			

cppvmSpawnMultiConnection myChildren("childprg");
cppvmObjectCollection *taskDescCollectionPtr;
...
taskDescCollectionPtr = myChildren.getAllChildTasks();			

Discussion			
A cppvmSpawnMultiConnection object cannot be sent to other PVM processes. Use objects of the class cppvmTask instead. The collection is allocated by this method.			

Table 11.94: cppvmSpawnMultiConnection: getTaggedChildTasks

getTaggedChildTasks	cppvmSpawnMultiConnection
get all tagged elements elements in the spawnCollection	
Synopsis	

continued on next page...

... continued from previous page			
`getTaggedChildTasks`			cppvmSpawnMultiConnection
cppvmObjectCollection * getTaggedChildTasks();			
Parameter			
Parameter	Type	Default	Description
`cppvmObjectCollection`	return		collection filled with cppvmTask objects

Example

```
cppvmSpawnMultiConnection myChildren("childprg", 4);
cppvmObjectCollection *taskDescCollectionPtr;
...
myChildren.tagElem(0);
myChildren.tagElem(3);
taskDescCollectionPtr = myChildren.getTaggedChildTasks();
```

Discussion

A cppvmSpawnMultiConnection object cannot be sent to other PVM processes. Use objects of the class cppvmTask instead. The collection is allocated by this method.

Table 11.95: cppvmSpawnMultiConnection: `halt`

`halt`	cppvmSpawnMultiConnection
shut down the entire PVM system	

Synopsis

void halt();

Exceptions

cppvmExceptionBadResult	local pvmd is not responding

Example

```
cppvmSpawnMultiConnection myProcesses("childprg", 11);
myProcesses.halt();
```

Discussion

It is recommended to use this method to shut down your CPPVM program. Otherwise PVM daemons or processes can still run after your main program has ended. This can cause errors if you start the program again. If you have this problem, start a pvm console and type `halt`.

Table 11.96: cppvmSpawnMultiConnection: `isElemTagged`

`isElemTagged`	cppvmSpawnMultiConnection
is a specific element tagged?	

Synopsis

bool isElemTagged(unsigned int index);

Parameter

continued on next page...

. . . continued from previous page			
`isElemTagged`			cppvmSpawnMultiConnection
Parameter	Type	Default	Description
`index`	in		child index to test
`bool`	return		tag status
Exceptions			
cppvmExceptionOutOfRange	Bad index		
Example			

```
cppvmSpawnMultiConnection myChildren("childprg", 4);
myChildren.tagElem(3);
if(myChildren.isElemTagged(3)){ ...
```

Table 11.97: cppvmSpawnMultiConnection: `killAllChildren`

`killAllChildren`	cppvmSpawnMultiConnection
kill all children processes	
Synopsis	
void killAllChildren();	
Exceptions	
cppvmExceptionForbiddenCall	child process was already killed
cppvmExceptionBadResult	cannot kill child process
Example	

```
cppvmSpawnMultiConnection myChildren("childprg", 4);
...
myChildren.killAllChildren();
```

Table 11.98: cppvmSpawnMultiConnection: `killTaggedChildren`

`killTaggedChildren`	cppvmSpawnMultiConnection
kill all tagged children processes	
Synopsis	
void killTaggedChildren();	
Exceptions	
cppvmExceptionForbiddenCall	child process was already killed
cppvmExceptionBadResult	cannot kill child process
Example	

```
cppvmSpawnMultiConnection myChildren("childprg", 4);
...
myChildren.killTaggedChildren();
```

Table 11.99: cppvmSpawnMultiConnection: numElems

numElems			cppvmSpawnMultiConnection
number of cppvmSpawnConnection elements in the collection			
Synopsis			
unsigned int numElems();			
Parameter			
Parameter	Type	Default	Description
unsigned int	in		number of cppvmSpawnConnection elements in the collection
Example			
cppvmSpawnMultiConnection myChildren("childprg", 5); cout << "Number of elements in list: " << myChildren.numElems(); ...			

Table 11.100: cppvmSpawnMultiConnection: operator[]

operator[]			cppvmSpawnMultiConnection
get a cppvmSpawnConnection element from the collection			
Synopsis			
cppvmSpawnConnection& operator[] (unsigned int index);			
Parameter			
Parameter	Type	Default	Description
index	in		index of the cppvmSpawnConnection element
cppvmSpawnConn&	return		cppvmSpawnConnection element
Exceptions			
cppvmExceptionOutOfRange	bad index		
Example			
cppvmSpawnMultiConnection myChildren("childprg", 5); cout << myChildren[3].getChildName();			

Table 11.101: cppvmSpawnMultiConnection: operator==

operator==			cppvmSpawnMultiConnection
test if objects are equal			
Synopsis			
bool operator== (const cppvmSpawnMultiConnection& obj)			
Parameter			
Parameter	Type	Default	Description
obj	in		object to compare with
bool	return		compare result

Table 11.102: cppvmSpawnMultiConnection: operator!=

operator!=			cppvmSpawnMultiConnection
test if objects are not equal			
Synopsis			
`bool operator!= (const cppvmSpawnMultiConnection& obj)`			
Parameter			
Parameter	Type	Default	Description
`obj`	in		object to compare with
`bool`	return		compare result

Table 11.103: cppvmSpawnMultiConnection: operator<<

operator<<			cppvmSpawnMultiConnection
write object on ostream			
Synopsis			
`ostream& operator<<(ostream& strm, cppvmSpawnMultiConnection& obj);`			
Parameter			
Parameter	Type	Default	Description
`obj`	in		write this object on ostream
`strm`	in		ostream
`ostream`	return		ostream after writing the object
Example			

```
cppvmSpawnMultiConnection myChildren("childprg", 5, "", PvmTaskHost,
"sun11");
cout << myChildren;
```

Table 11.104: cppvmSpawnMultiConnection: remove

remove			cppvmSpawnMultiConnection
remove (delete) a `cppvmSpawnConnection` element from the collection			
Synopsis			
`void remove(cppvmSpawnConnection& obj);`			
Parameter			
Parameter	Type	Default	Description
`obj`	in		`cppvmSpawnConnection` object
Exceptions			
cppvmExceptionOutOfRange	Object not in spawn collection		
Example			

continued on next page...

...continued from previous page	
`remove`	cppvmSpawnMultiConnection

```
cppvmSpawnMultiConnection myChildren("childprg", 5);
cppvmSpawnConnection* newChildPtr;
newChildPtr = new cppvmSpawnConnection("otherChildPrg");
myChildren.add(newChildPtr);
...
myChildren.remove(*newChildPtr);
```

Table 11.105: cppvmSpawnMultiConnection: `remove`

`remove`			cppvmSpawnMultiConnection
remove (delete) a `cppvmSpawnConnection` element from the collection			
Synopsis			
`void remove(cppvmSpawnConnection* objPtr);`			
Parameter			
Parameter	Type	Default	Description
`objPtr`	in		pointer to a `cppvmSpawnConnection` object
Exceptions			
cppvmExceptionBadParameter	cannot remove NULL pointer		
cppvmExceptionOutOfRange	Object not in spawn collection		
Example			

```
cppvmSpawnMultiConnection myChildren("childprg", 5);
cppvmSpawnConnection* newChildPtr;
newChildPtr = new cppvmSpawnConnection("otherChildPrg");
myChildren.add(newChildPtr);
...
myChildren.remove(newChildPtr);
```

Table 11.106: cppvmSpawnMultiConnection: `remove`

`remove`			cppvmSpawnMultiConnection
remove a `cppvmSpawnConnection` element from the collection			
Synopsis			
`void remove(unsigned int index);`			
Parameter			
Parameter	Type	Default	Description
`index`	in		index of a `cppvmSpawnConnection` object in the collection
Exceptions			
cppvmExceptionOutOfRange	Bad index		

continued on next page...

. . . continued from previous page	
`remove`	cppvmSpawnMultiConnection
Example	
`cppvmSpawnMultiConnection myChildren("childprg", 5);` `myChildren.remove(3);`	

Table 11.107: cppvmSpawnMultiConnection: `clean`

`clean`	cppvmSpawnMultiConnection
remove all `cppvmSpawnConnection` elements from the collection	
Synopsis	
`void clean();`	
Example	
`cppvmSpawnMultiConnection myChildren("childprg", 5);` `myChildren.clean();`	

Table 11.108: cppvmSpawnMultiConnection: `tagAllElems`

`tagAllElems`	cppvmSpawnMultiConnection
tag all children in the collection	
Synopsis	
void tagAllElems();	
Example	
`cppvmSpawnMultiConnection myChildren("childprg", 8);` `myChildren.tagAllElems();`	

Table 11.109: cppvmSpawnMultiConnection: `tagElem`

`tagElem`			cppvmSpawnMultiConnection
tag a child in the collection			
Synopsis			
void tagElem(unsigned int index);			
Parameter			
Parameter	Type	Default	Description
`index`	in		child index to tag
Exceptions			
cppvmExceptionOutOfRange	Bad index		
Example			
`cppvmSpawnMultiConnection myChildren("childprg", 4);` `myChildren.tagElem(2);`			

Table 11.110: cppvmSpawnMultiConnection: `untagAllElems`

`untagAllElems`	cppvmSpawnMultiConnection
untag all children in the collection	
Synopsis	
void untagAllElems();	
Example	

```
cppvmSpawnMultiConnection myChildren("childprg", 100);
...
myChildren.untagAllElems();
```

Table 11.111: cppvmSpawnMultiConnection: `untagElem`

`untagElem`			cppvmSpawnMultiConnection
untag a child in the collection			
Synopsis			
void untagElem(unsigned int index);			
Parameter			
Parameter	Type	Default	Description
`index`	in		child index to untag
Exceptions			
cppvmExceptionOutOfRange	Bad index		
Example			

```
cppvmSpawnMultiConnection myChildren("childprg", 4);
...
myChildren.untagElem(3);
```

11.4 cppvmCatchout

All stderr and stdout of spawned tasks are logged by the master PVM daemon in a file (/tmp/pvml.<uid>). The class cppvmCatchout can be used to redirect this output.

Catch out causes the calling task (the parent) to catch output from tasks spawned after creating a cppvmCatchout object. Characters printed on stdout or stderr in children tasks are collected by the pvmd´s and sent in control messages to the parent task which tags each line and appends it to the specified stream. Output from grandchildren (spawned by children) tasks is also collected, provided the children don't reset PvmOutputTid. Each line of output has one of the following forms:

```
[txxxxx] BEGIN
[txxxxx] (text from child task)
[txxxxx] END
```

The output from each task includes one BEGIN line and one END line with whatever the task prints in between. If the parent process exits while output collection is in effect, it will block until all tasks sending their output have exited in order to print all their output. To avoid this, output collection can be turned off by calling cppvmCatchout(CPPvmStop) before exiting the parent process.

With PVM it is not possible to have more than one catch out stream. For example, it is not possible to catch the output of process myprocess1 into the file myprocess1.out and the output of process myprocess2 into the file myprocess2.out. Every caught output is redirected to the stream specified by the last cppvmCatchout object. For example:

```
cppvmCatchout catchout(CPPvmStdout);
// spawn child process 'myprocess1'
cppvmSpawnConnection firstChild("myprocess1");

// every new cppvmCatchout object will
// also redirect ALL previously
// caught children output

// catch output to file
cppvmCatchout catchout2(CPPvmFile, "myprocess2.out");
// spawn child process 'myprocess2'
cppvmSpawnConnection secondChild("myprocess2");
```

At first, the output of the process myprocess1 is redirected to standard out. By creating the object catchout2, the output of the process myprocess1 *and* the process myprocess1 is redirected to the file myprocess2.out.

You have to create a cppvmCatchout object before creating a cppvmSpawnConnection or cppvmSpawnMultiConnection object to redirect their output. It is not possible to redirect the output of processes once they are spawned without catchout.

The redirected output of the spawned processes is buffered to reduce the frequency of messages to the catchout process. To flush the output you can call fflush(stdout) in a spawned process. The buffering mode of stdout can be changed to line-oriented for the entire process by calling setlinebuf(stdout). When a spawned process disconnects from PVM (process exits, calls the CPPvm exit method, deletes all cppvmConnection, cppvmSpawnConnection or cppvmSpawnMultiConnection) objects) stdout is flushed as well.

It is not possible to redirect the command line input (prompt) of spawned processes. Each spawned task has /den/null opened as stdin. Only processes started by the user can use a command line input. Of course spawned processes can read files or use a graphical user interface (GUI) for its input.

When opening files remember that the working directory of spawned processes is $HOME by default. You can change the working directory by using the host-file option wd= (see subsection 2.1.5 and 2.2.5.1). When a processes uses a GUI and is spawned on another host, do not forget to enable the X-Window access on your local host (xhost <hostname>) and to export the DISPLAY variable. There are to ways to export variables:

- set the PVM_EXPORT environment variable before starting the program, e.g.:

  ```
  setenv PVM_EXPORT DISPLAY
  ```

- use the exportEnvVar method of the class cppvmConnection within your program:

  ```
  cppvmConnection pvmConn;
  pvmConn.exportEnvVar("DISPLAY");
  // redirect output
  cppvmCatchout catchout(CPPvmStdout);
  // spawn child 'gui-app' on 'sun1'
  cppvmSpawnConnection child("gui-app", "", PvmTaskHost, "sun1");
  ```

Ensure that the environment variable DISPLAY is set correctly.

Table 11.112: cppvmCatchout: constructor

constructor	cppvmCatchout
constructor with task description	
Synopsis	
cppvmCatchout(cppvmCatchoutMode catchoutFlag, char *catchoutFileName);	

continued on next page...

...continued from previous page	
`constructor`	cppvmCatchout

Parameter			
Parameter	Type	Default	Description
`catchoutFlag`	in	CPPvmStdout	specifies the catch out stream: `CPPvmStdout:` write the output on stdout `CPPvmStderr:` write the output on stderr `CPPvmFile:` write output on file specified by `catchoutFileName`
`catchoutFileName`	in	NULL	specifies a catch out file name; only used if `catchoutFlag` is set to `CPPvmFile`

Exceptions	
cppvmExceptionBadParameter	`CPPvmFile` set and no log file name given.
cppvmExceptionBadResult	cannot open catch out file
cppvmExceptionBadResult	cannot redirect output
cppvmExceptionBadParameter	unknown catchout flag

Example

```
// catch to standard out
cppvmCatchout catch1();
cppvmSpawnConnection child1("myprocess");
...
// catch to file 'my.out'
cppvmCatchout catch2(CPPvmFile, "my.out");
cppvmSpawnConnection child2("myprocess2");
```

Discussion

There can be only one active catch out object

Table 11.113: cppvmCatchout: `operator==`

`operator==`	cppvmCatchout
test if objects are equal	

Synopsis

```
bool operator== (const cppvmCatchout& obj)
```

Parameter			
Parameter	Type	Default	Description
obj	in		object to compare with
bool	return		compare result

Example

continued on next page...

... continued from previous page

operator==	cppvmCatchout

```
cppvmCatchout c1();
cppvmCatchout c2();
...
if(c1==c2){ ...
```

Table 11.114: cppvmCatchout: operator!=

operator!=			cppvmCatchout
test if objects are not equal			
Synopsis			
`bool operator!= (const cppvmCatchout& obj)`			
Parameter			
Parameter	Type	Default	Description
`obj`	in		object to compare with
`bool`	return		compare result

Example

```
cppvmCatchout c1();
cppvmCatchout c2();
...
if(c1!=c2){ ...
```

Table 11.115: cppvmCatchout: operator<<

operator<<			cppvmCatchout
write object on ostream			
Synopsis			
`ostream& operator<<(ostream& strm, cppvmCatchout& obj);`			
Parameter			
Parameter	Type	Default	Description
`obj`	in		write this object on ostream
`strm`	in		ostream
`ostream`	return		ostream after writing the object

Example

```
cppvmCatchout c();
cout << c;
```

Table 11.116: cppvmCatchout: reset

`reset`	cppvmCatchout
set the child catch out to the former given values	

Synopsis

void reset();

Parameter

Parameter	Type	Default	Description

Example

```
// catch to file 'my.out'
cppvmCatchout catch1(CPPvmFile, "my.out");
cppvmSpawnConnection child1("myprocess");
...
// catch to standard out
cppvmCatchout catch2();
cppvmSpawnConnection child2("myprocess2");

// catch again to file 'my.out'
catch1.reset();
```

Table 11.117: cppvmCatchout: set

`set`	cppvmCatchout
set the child catch out	

Synopsis

void set(cppvmCatchoutMode catchoutFlag, char *catchoutFileName);

Parameter

Parameter	Type	Default	Description
`catchoutFlag`	in	CPPvmStdout	specifies the catch out stream: `CPPvmStdout:` write the output on stdout `CPPvmStderr:` write the output on stderr `CPPvmFile:` write output on file specified by `catchoutFileName`
`catchoutFileName`	in	NULL	specifies a catch out file name; only used if `catchoutFlag` is set to `CPPvmFile`

Exceptions

cppvmExceptionBadParameter	`CPPvmFile` set and no log file name given.
cppvmExceptionBadResult	cannot open catch out file
cppvmExceptionBadResult	cannot redirect output
cppvmExceptionBadParameter	unknown catchout flag

continued on next page...

. . . continued from previous page	
`set`	cppvmCatchout

Example

```
// catch to standard out
cppvmCatchout catch();
cppvmSpawnConnection child("myprocess");
...
// catch to file 'my.out'
catch.set(CPPvmFile, "my.out");
```

Table 11.118: cppvmCatchout: `unset`

`unset`	cppvmCatchout
unset the current catch out redirection; flush all buffered data and close file if necessary	

Synopsis

void unset();

Parameter

Parameter	Type	Default	Description

Example

```
// catch to standard out
cppvmCatchout catch();
cppvmSpawnConnection child("myprocess");
...
// close this catch out
catch.unset();
```

11.5 cppvmSendStream

The class cppvmSendStream specifies a message passing send stream between PVM processes. Send streams are used like standard C++ out streams:

```
// spawn child process 'mychild'
cppvmSpawnConnection myprocess("mychild");
// initialize blocking send stream to child process
// use XDR (default) encoding scheme
cppvmSendStream sendStrm(myprocess, CPPvmSBchild, PvmDataDefault);
cppvmString string("test message");
// send object twice
sendStrm << string << string;
```

The send stream is specified by the process handling classes of the parallel virtual machine (e.g. cppvmConnection, cppvmSpawnConnection, cppvmTask ...), the send mode and the PVM encoding scheme.

There are two send modes:

`CPPvmSB<connectionSpecifier>` blocking send: the sending process is blocked until the message was received from the other process

`CPPvmSNB<connectionSpecifier>` nonblocking send: the sending process continues without waiting for the other process(es) to receive the message

The connection specifier depends on the involved process handling classes:

class	**connection specifier**
cppvmTask	`task`
cppvmConnection	`parent`
cppvmSpawnConnection	`parent`, `child`
cppvmSpawnMultiConnection	`taggedChildren`, `allChildren`
group multicasting with current context	`group`
sending to a specified process	`tid`
sending back to a process	`back`

There available PVM encoding schemes are:

`PvmDataDefault` XDR encoding. This encoding scheme is used by default and works everytime.

`PvmDataRaw` Do not encode information (only for send/receive between task on homogeneous architectures).

`PvmDataInPlace` Do not copy data in sending buffer. Use the data memory instead. *Attention*: the data memory must not been deleted between packing and sending!

The default encoding scheme `PvmDataDefault` works everytime on every architecture. The other schemes are only useful for optimizing the message passing speed and must be handled with care.

Here are some send stream examples:

```
cppvmString string("and now for a completely different test message");

// spawn child process 'hello2'
cppvmSpawnConnection myprocess("hello2");
// blocking send to child process
// use XDR (default) encoding scheme
cppvmSendStream sendStrm(myprocess, CPPvmSBchild);
sendStrm << string;

// uninitialized send descriptor
cppvmSendStream sendStrm2;
// spawn five children processes 'mcast2'
cppvmSpawnMultiConnection children("mcast2", 5);
// non-blocking send to all tagged child processes
// use raw encoding scheme
sendStrm2.set(children, CPPvmSNBtaggedChildren, PvmDataRaw);
sendStrm2 << string << string;

// non-blocking send to all group members
// use XDR encoding scheme
cppvmSendStream sendStrm3("my group name", CPPvmSNBgroup, PvmDataDefault);
sendStrm3 << string;

// blocking send to process
// specified by its task id
// use raw encoding scheme
cppvmSendStream sendStrm4(1234, CPPvmSBtid, PvmDataRaw);
sendStrm4 << string;

// blocking receive from any process
cppvmReceiveStream recStrm(CPPvmRBany);
cppvmString recStr("");
recStrm >> recStr;
// send a message back to the sender
// of the received message
// do not copy data in sending buffer
cppvmSendStream sendStrm5(recStrm, CPPvmSNBback, PvmDataInPlace);
sendStrm5 << string;
```

Table 11.119: cppvmSendStream: `constructor`

`constructor`	cppvmSendStream
default constructor	
Synopsis	
`cppvmSendStream();`	
Example	
`cppvmSendStream myStream;`	
Discussion	
Use the default constructor only if the stream is initialized later on with one of the `set` methods.	

Table 11.120: cppvmSendStream: `constructor`

`constructor`			cppvmSendStream
constructor for cppvmTask objects			
Synopsis			
`cppvmSendStream(cppvmTask& task, cppvmSendTask mode, int initSend);`			
Parameter			
Parameter	Type	Default	Description
`task`	in		description of the process to send to
`mode`	in		send mode: `CPPvmSBtask:` send blocking `CPPvmSNBtask:` send nonblocking
`initSend`	in	PvmDataDefault	PVM send scheme: `PvmDataDefault:` XDR encoding (works everytime) `PvmDataRaw:` do NOT encode information (only for send/receive between task on homogeneous architectures) `PvmDataInPlace:` do not copy data in sending buffer, use the data memory instead. ATTENTION: the data memory MUST not deleted between packing and sending!
Exceptions			
cppvmExceptionBadParameter	send mode does not exist invalid sendEncoding argument		

continued on next page...

...continued from previous page	
`constructor`	cppvmSendStream
cppvmExceptionOutOfRange	the given encoding scheme does not exist. the given connection does not exist. tid cannot be negative tid cannot be < 0 the given mode does not exist.
Example	

`cppvmTask task();`
`...`
`cppvmSendStream myStream(task, CPPvmSNBtask, PvmDataDefault);`	

Table 11.121: cppvmSendStream: `constructor`

`constructor`			cppvmSendStream
constructor for cppvmConnection objects			
Synopsis			
`cppvmSendStream(cppvmConnection& conn, cppvmSendConnection mode, int initSend);`			
Parameter			
Parameter	Type	Default	Description
`conn`	in		description of the parent process to send to
`mode`	in		send mode: `CPPvmSBparent:` send blocking `CPPvmSNBparent:` send nonblocking
`initSend`	in	PvmDataDefault	PVM send scheme: `PvmDataDefault:` XDR encoding (works everytime) `PvmDataRaw:` do NOT encode information (only for send/receive between task on homogeneous architectures) `PvmDataInPlace:` do not copy data in sending buffer, use the data memory instead. ATTENTION: the data memory MUST not deleted between packing and sending!
Exceptions			
cppvmExceptionBadParameter	send mode does not exist invalid sendEncoding argument		
			continued on next page...

. . . continued from previous page	
`constructor`	cppvmSendStream
cppvmExceptionOutOfRange	the given encoding scheme does not exist. the given connection does not exist. tid cannot be negative tid cannot be < 0 the given mode does not exist. cannot send to parent tid - No parent process
Example	

```
cppvmConnection pvmConn();
...
cppvmSendStream myStream(pvmConn, CPPvmSNBparent, PvmDataRaw);
```

Table 11.122: cppvmSendStream: `constructor`

`constructor`			cppvmSendStream
constructor for cppvmSpawnConnection objects			
Synopsis			
`cppvmSendStream(cppvmSpawnConnection& sconn, cppvmSendSpawnConnection mode, int initSend);`			
Parameter			
Parameter	Type	Default	Description
`sconn`	in		description of the child process to send to
`mode`	in		send mode: `CPPvmSBchild:` send blocking `CPPvmSNBchild:` send nonblocking
`initSend`	in	PvmDataDefault	PVM send scheme: `PvmDataDefault:` XDR encoding (works everytime) `PvmDataRaw:` do NOT encode information (only for send/receive between task on homogeneous architectures) `PvmDataInPlace:` do not copy data in sending buffer, use the data memory instead. ATTENTION: the data memory MUST not deleted between packing and sending!
Exceptions			
cppvmExceptionBadParameter	send mode does not exist invalid sendEncoding argument		

continued on next page. . .

. . . continued from previous page

`constructor`	cppvmSendStream
cppvmExceptionOutOfRange	the given encoding scheme does not exist. the given connection does not exist. tid cannot be negative tid cannot be < 0 the given mode does not exist.

Example

```
cppvmSpawnConnection child("hello2");
...
cppvmSendStream myStream(child, CPPvmSBchild, PvmDataInPlace);
```

Table 11.123: cppvmSendStream: `constructor`

`constructor`	cppvmSendStream
constructor for cppvmSpawnMultiConnection objects	

Synopsis

```
cppvmSendStream(cppvmSpawnMultiConnection& smconn,
cppvmSendSpawnMultiConnection mode, int initSend);
```

Parameter

Parameter	Type	Default	Description
`sconn`	in		description of the child processes to send to
`mode`	in		send mode: `CPPvmSBtaggedChildren:` send blocking to tagged children `CPPvmSNBtaggedChildren:` send nonblocking to tagged children `CPPvmSBallChildren:` send blocking to all children `CPPvmSNBallChildren:` send nonblocking to all children
`initSend`	in	PvmDataDefault	PVM send scheme: `PvmDataDefault:` XDR encoding (works everytime) `PvmDataRaw:` do NOT encode information (only for send/receive between task on homogeneous architectures) `PvmDataInPlace:` do not copy data in sending buffer, use the data memory instead. ATTENTION: the data memory MUST not deleted between packing and sending!

continued on next page. . .

. . . continued from previous page

constructor	cppvmSendStream

Exceptions

cppvmExceptionBadParameter	send mode does not exist invalid sendEncoding argument
cppvmExceptionOutOfRange	the given encoding scheme does not exist. the given connection does not exist. tid cannot be negative tid cannot be < 0 the given mode does not exist.

Example

```
cppvmSpawnMultiConnection children("hello2",5);
...
cppvmSendStream myStream(children, CPPvmSBallChildren,
PvmDataDefault);
```

Table 11.124: cppvmSendStream: `constructor`

`constructor`	cppvmSendStream
constructor for group broadcasting with current context	

Synopsis

```
cppvmSendStream(char *group, cppvmSendGroup mode, int initSend);
```

Parameter

Parameter	Type	Default	Description
`group`	in		group name
`mode`	in		send mode: `CPPvmSBgroup`: send blocking `CPPvmSNBgroup`: send nonblocking
`initSend`	in	PvmDataDefault	PVM send scheme: `PvmDataDefault`: XDR encoding (works everytime) `PvmDataRaw`: do NOT encode information (only for send/receive between task on homogeneous architectures) `PvmDataInPlace`: do not copy data in sending buffer, use the data memory instead. ATTENTION: the data memory MUST not deleted between packing and sending!

Exceptions

continued on next page. . .

...continued from previous page	
`constructor`	cppvmSendStream
cppvmExceptionBadParameter	send mode does not exist invalid sendEncoding argument
cppvmExceptionOutOfRange	the given encoding scheme does not exist. the given connection does not exist. tid cannot be negative tid cannot be < 0 the given mode does not exist.
Example	
`cppvmSendStream myStream("my group name", CPPvmSBgroup, PvmDataRaw);`	

Table 11.125: cppvmSendStream: `constructor`

`constructor`			cppvmSendStream
constructor for group broadcasting with given context			
Synopsis			
`cppvmSendStream(char *group, int context, cppvmSendGroup mode, int initSend);`			
Parameter			
Parameter	Type	Default	Description
`group`	in		group name
`context`	in		context to send with
`mode`	in		send mode: `CPPvmSBgroup:` send blocking `CPPvmSNBgroup:` send nonblocking
`initSend`	in	PvmDataDefault	PVM send scheme: `PvmDataDefault:` XDR encoding (works everytime) `PvmDataRaw:` do NOT encode information (only for send/receive between task on homogeneous architectures) `PvmDataInPlace:` do not copy data in sending buffer, use the data memory instead. ATTENTION: the data memory MUST not deleted between packing and sending!
Exceptions			
cppvmExceptionBadParameter	send mode does not exist invalid sendEncoding argument		

continued on next page...

. . . continued from previous page	
`constructor`	cppvmSendStream
cppvmExceptionOutOfRange	the given encoding scheme does not exist. the given connection does not exist. tid cannot be negative tid cannot be < 0 the given mode does not exist.
Example	
`cppvmSendStream myStream("my group name", 33, CPPvmSBgroup, PvmDataRaw);`	

Table 11.126: cppvmSendStream: `constructor`

`constructor`			cppvmSendStream
constructor for send to a specific task with current context			
Synopsis			
`cppvmSendStream(int taskId, cppvmSendTid mode, int initSend);`			
Parameter			
Parameter	Type	Default	Description
`taskId`	in		task id of the process to send to
`mode`	in		send mode: `CPPvmSBtid:` send blocking `CPPvmSNBtid:` send nonblocking
`initSend`	in	PvmDataDefault	PVM send scheme: `PvmDataDefault:` XDR encoding (works everytime) `PvmDataRaw:` do NOT encode information (only for send/receive between task on homogeneous architectures) `PvmDataInPlace:` do not copy data in sending buffer, use the data memory instead. ATTENTION: the data memory MUST not deleted between packing and sending!
Exceptions			
cppvmExceptionBadParameter	send mode does not exist invalid sendEncoding argument		

continued on next page. . .

...continued from previous page

constructor	cppvmSendStream
cppvmExceptionOutOfRange	the given encoding scheme does not exist. the given connection does not exist. tid cannot be negative tid cannot be < 0 the given mode does not exist.

Example

```
cppvmSendStream myStream(1201, CPPvmSNBtid, PvmDataDefault);
```

Table 11.127: cppvmSendStream: constructor

constructor	cppvmSendStream
constructor for send to a specific task with given context	

Synopsis

```
cppvmSendStream(int taskId, int context, cppvmSendTid mode, int
initSend);
```

Parameter

Parameter	Type	Default	Description
`taskId`	in		task id of the process to send to
`context`	in		context id
`mode`	in		send mode: `CPPvmSBtid:` send blocking `CPPvmSNBtid:` send nonblocking
`initSend`	in	PvmDataDefault	PVM send scheme: `PvmDataDefault:` XDR encoding (works everytime) `PvmDataRaw:` do NOT encode information (only for send/receive between task on homogeneous architectures) `PvmDataInPlace:` do not copy data in sending buffer, use the data memory instead. ATTENTION: the data memory MUST not deleted between packing and sending!

Exceptions

cppvmExceptionBadParameter	send mode does not exist invalid sendEncoding argument

continued on next page...

. . . continued from previous page	
`constructor`	cppvmSendStream
cppvmExceptionOutOfRange	the given encoding scheme does not exist. the given connection does not exist. tid cannot be negative tid cannot be < 0 the given mode does not exist.
Example	
`cppvmSendStream myStream(1201, 44, CPPvmSNBtid, PvmDataDefault);`	

Table 11.128: cppvmSendStream: `constructor`

`constructor`			cppvmSendStream
constructor for send to specific tasks with current context			
Synopsis			
`cppvmSendStream(int* taskIds, unsigned int numTasks, cppvmSendTids mode, int initSend);`			
Parameter			
Parameter	Type	Default	Description
`taskIds`	in		tasks array of the processes to send to
`numTasks`	in		number of task ids in the array
`mode`	in		send mode: `CPPvmSBtids:` send blocking `CPPvmSNBtids:` send nonblocking
`initSend`	in	PvmDataDefault	PVM send scheme: `PvmDataDefault:` XDR encoding (works everytime) `PvmDataRaw:` do NOT encode information (only for send/receive between task on homogeneous architectures) `PvmDataInPlace:` do not copy data in sending buffer, use the data memory instead. ATTENTION: the data memory MUST not deleted between packing and sending!
Exceptions			
cppvmExceptionBadParameter	taskIds pointer is NULL send mode does not exist invalid sendEncoding argument tid is already in the sending array		
continued on next page. . .			

...continued from previous page

constructor		cppvmSendStream
cppvmExceptionOutOfRange	the given encoding scheme does not exist. the given connection does not exist. tid cannot be negative tid cannot be < 0 the given mode does not exist.	

Example

```
int ids[2];
cppvmSpawnConnection child1("hi");
cppvmSpawnConnection child2("ho");
ids[0]=child1.getChildTid();
ids[1]=child2.getChildTid();
cppvmSendStream myStream(ids, 2, CPPvmSNBtids, PvmDataDefault);
```

Discussion

The task ids in the array must differ from each other. If a id occures twice or more often in the array, an exception is thrown.

Table 11.129: cppvmSendStream: constructor

constructor	cppvmSendStream
constructor for send to specific tasks with given context	

Synopsis

```
cppvmSendStream(int* taskIds, unsigned int numTasks, int context,
cppvmSendTids mode, int initSend);
```

Parameter

Parameter	Type	Default	Description
`taskIds`	in		tasks array of the processes to send to
`numTasks`	in		number of task ids in the array
`context`	in		context id
`mode`	in		send mode: `CPPvmSBtids:` send blocking `CPPvmSNBtids:` send nonblocking

continued on next page...

. . . continued from previous page			
`constructor`			cppvmSendStream
`initSend`	in	PvmDataDefault	PVM send scheme: `PvmDataDefault:` XDR encoding (works everytime) `PvmDataRaw:` do NOT encode information (only for send/receive between task on homogeneous architectures) `PvmDataInPlace:` do not copy data in sending buffer, use the data memory instead. ATTENTION: the data memory MUST not deleted between packing and sending!

Exceptions	
cppvmExceptionBadParameter	taskIds pointer is NULL send mode does not exist invalid sendEncoding argument tid is already in the sending array
cppvmExceptionOutOfRange	the given encoding scheme does not exist. the given connection does not exist. tid cannot be negative tid cannot be < 0 the given mode does not exist.

Example

```
int ids[2];
cppvmSpawnConnection child1("hi");
cppvmSpawnConnection child2("ho");
ids[0]=child1.getChildTid();
ids[1]=child2.getChildTid();
cppvmSendStream myStream(ids, 2, 1001, CPPvmSNBtids, PvmDataDefault);
```

Discussion

The task ids in the array must differ from each other. If a id occures twice or more often in the array, an exception is thrown.

Table 11.130: cppvmSendStream: `constructor`

`constructor`	cppvmSendStream
constructor for send to specific tasks with given context array	

Synopsis

```
cppvmSendStream(int* taskIds, unsigned int numTasks, int* contextArr,
cppvmSendTids mode, int initSend);
```

Parameter

continued on next page. . .

. . . continued from previous page

constructor	cppvmSendStream

Parameter	Type	Default	Description
`taskIds`	in		tasks array of the processes to send to
`numTasks`	in		number of task ids in the array
`contextArr`	in		context id array
`mode`	in		send mode: `CPPvmSBtids:` send blocking `CPPvmSNBtids:` send nonblocking
`initSend`	in	PvmDataDefault	PVM send scheme: `PvmDataDefault:` XDR encoding (works everytime) `PvmDataRaw:` do NOT encode information (only for send/receive between task on homogeneous architectures) `PvmDataInPlace:` do not copy data in sending buffer, use the data memory instead. ATTENTION: the data memory MUST not deleted between packing and sending!

Exceptions	
cppvmExceptionBadParameter	taskIds pointer is NULL contextArr pointer is NULL. send mode does not exist invalid sendEncoding argument tid is already in the sending array
cppvmExceptionOutOfRange	the given encoding scheme does not exist. the given connection does not exist. tid cannot be negative tid cannot be < 0 the given mode does not exist.

Example

```
int ids[2];
int contextArr[2];
cppvmSpawnConnection child1("hi");
cppvmSpawnConnection child2("ho");
ids[0]=child1.getChildTid();
contextArr[0]=child1.getContextId();
ids[1]=child2.getChildTid();
contextArr[1]=child2.getContextId();
cppvmSendStream myStream(ids, 2, contextArr, CPPvmSNBtids,
PvmDataDefault);
```

continued on next page. . .

... continued from previous page	
`constructor`	cppvmSendStream

Discussion
The task ids in the array must differ from each other. If a id occures twice or more often in the array, an exception is thrown.

Table 11.131: cppvmSendStream: `constructor`

`constructor`	cppvmSendStream
constructor to send back to a process, from which a message was received with `recStrm` with current context	

Synopsis

`cppvmSendStream(cppvmReceiveStream& recStrm,`
`cppvmSendBack mode,`
`int initSend);`

Parameter

Parameter	Type	Default	Description
`recStrm`	in		send back to the process from which a message was received with this stream
`mode`	in		send mode: `CPPvmSBback:` send blocking `CPPvmSNBback:` send nonblocking
`initSend`	in	PvmDataDefault	PVM send scheme: `PvmDataDefault:` XDR encoding (works everytime) `PvmDataRaw:` do NOT encode information (only for send/receive between task on homogeneous architectures) `PvmDataInPlace:` do not copy data in sending buffer, use the data memory instead. ATTENTION: the data memory MUST not deleted between packing and sending!

Exceptions

cppvmExceptionBadParameter	send mode does not exist invalid sendEncoding argument
cppvmExceptionOutOfRange	the given encoding scheme does not exist. the given connection does not exist. tid cannot be negative tid cannot be < 0 the given mode does not exist.

Example

continued on next page...

...continued from previous page	
`constructor`	cppvmSendStream

```
cppvmReceiveStream recStrm();
...
cppvmString recStr("");
recStrm >> recStr;
cppvmSendStream myStream(recStrm, CPPvmSNBback, PvmDataRaw);
```

Table 11.132: cppvmSendStream: `constructor`

`constructor`	cppvmSendStream
constructor to send back to a process, from which a message was received with `recStrm` with given context	

Synopsis

```
cppvmSendStream(cppvmReceiveStream& recStrm, int context,
cppvmSendBack mode, int initSend);
```

Parameter

Parameter	Type	Default	Description
`recStrm`	in		send back to the process from which a message was received with this stream
`context`	in		context id
`mode`	in		send mode: `CPPvmSBback:` send blocking `CPPvmSNBback:` send nonblocking
`initSend`	in	PvmDataDefault	PVM send scheme: `PvmDataDefault:` XDR encoding (works everytime) `PvmDataRaw:` do NOT encode information (only for send/receive between task on homogeneous architectures) `PvmDataInPlace:` do not copy data in sending buffer, use the data memory instead. ATTENTION: the data memory MUST not deleted between packing and sending!

Exceptions

cppvmExceptionBadParameter	send mode does not exist invalid sendEncoding argument

continued on next page...

. . . continued from previous page	
`constructor`	cppvmSendStream

cppvmExceptionOutOfRange	the given encoding scheme does not exist. the given connection does not exist. tid cannot be negative tid cannot be < 0 the given mode does not exist.

Example

```
cppvmReceiveStream recStrm();
...
cppvmString recStr("");
recStrm >> recStr;
cppvmSendStream myStream(recStrm, 55, CPPvmSNBback, PvmDataRaw);
```

Table 11.133: cppvmSendStream: `copy constructor`

`copy constructor`	cppvmSendStream
copy constructor	

Synopsis

```
cppvmSendStream(const cppvmSendStream& obj);
```

Parameter

Parameter	Type	Default	Description
`obj`	in		object to copy

Exceptions

cppvmExceptionBadParameter	group name is NULL
cppvmExceptionOutOfRange	obj.numTid cannot be negative

Example

```
cppvmSendStream myStream = previousDefinedStream;
```

Table 11.134: cppvmSendStream: `getContext`

`getContext`	cppvmSendStream
get the context id of a process to send to	

Synopsis

```
int getContext(int indexContext);
```

Parameter

Parameter	Type	Default	Description
`indexContext`	in		index of the send-to-process to get the context id from
int	return		context id

Exceptions

cppvmExceptionOutOfRange	Index is not in the array limits

continued on next page. . .

. . . continued from previous page	
getContext	cppvmSendStream

Example
`cppvmSpawnMultiConnection children("hello2",5);`
`cppvmSendStream myStream(children, CPPvmSBallChildren,`
`PvmDataDefault);`
`cout << myStream.getContext(3);`

Table 11.135: cppvmSendStream: getGroup

getGroup			cppvmSendStream
get the group name			
Synopsis			
`char *getGroup();`			
Parameter			
Parameter	Type	Default	Description
char*	return		group name
Exceptions			
cppvmExceptionForbiddenCall	object not initialized, it is not allowed to use it		
Example			
`cppvmSendStream myStream("my group", CPPvmSBgroup, PvmDataDefault);`			
`cout << myStream.getGroup();`			

Table 11.136: cppvmSendStream: getInitSend

getInitSend			cppvmSendStream
get the `pvm_initsend` encoding scheme			
Synopsis			
`int getInitSend();`			
Parameter			
Parameter	Type	Default	Description
int	return		mode: `PvmDataDefault`: XDR encoding (works everytime) `PvmDataRaw`: do NOT encode information (only for send/receive between task on homogeneous architectures) `PvmDataInPlace`: do not copy data in sending buffer, use the data memory instead. ATTENTION: the data memory MUST not deleted between packing and sending!
			continued on next page. . .

. . . continued from previous page	
`getInitSend`	cppvmSendStream
Exceptions	
cppvmExceptionForbiddenCall	object not initialized, it is not allowed to use it
Example	

```
cppvmSendStream myStream("my group", CPPvmSBgroup, PvmDataDefault);
cout << myStream.getInitSend();
```

Table 11.137: cppvmSendStream: `getMsgBuf`

`getMsgBuf`			cppvmSendStream
get the information if a new message buffer is used			
Synopsis			
`bool getMsgBuf();`			
Parameter			
Parameter	Type	Default	Description
bool	return		new message buffer used
Exceptions			
cppvmExceptionForbiddenCall	object not initialized, it is not allowed to use it		
Example			

```
cppvmSendStream myStream("my group", CPPvmSBgroup, PvmDataDefault);
cout << myStream.getMsgBuf();
```

Table 11.138: cppvmSendStream: `getMsgBufEncoding`

`getMsgBufEncoding`			cppvmSendStream
get the encoding scheme of a new message buffer			
Synopsis			
`int getMsgBufEncoding();`			
Parameter			
Parameter	Type	Default	Description
			continued on next page. . .

. . . continued from previous page		
`getMsgBufEncoding`		cppvmSendStream
int	return	PVM encoding scheme: `PvmDataDefault:` XDR encoding (works everytime) `PvmDataRaw:` do NOT encode information (only for send/receive between task on homogeneous architectures) `PvmDataInPlace:` do not copy data in sending buffer, use the data memory instead. ATTENTION: the data memory MUST not deleted between packing and sending!

Exceptions

cppvmExceptionForbiddenCall	object not initialized, it is not allowed to use it

Example

```
cppvmSendStream myStream("my group", CPPvmSBgroup, PvmDataDefault);
cout << myStream.getMsgBufEncoding();
```

Table 11.139: cppvmSendStream: `getNumTid`

`getNumTid`	cppvmSendStream
get the number of send targets	

Synopsis

```
int getNumTid();
```

Parameter

Parameter	Type	Default	Description
`indexContext`	return		number of processes to send to

Example

```
cppvmSpawnMultiConnection children("hello2",5);
cppvmSendStream myStream(children, CPPvmSBallChildren,
PvmDataDefault);
int i;
for(i=0; i<myStream.getNumTid(); i++)
cout << myStream.getTid(i);
```

Table 11.140: cppvmSendStream: `getSendConnection`

`getSendConnection`	cppvmSendStream
get the send connection type	

Synopsis

continued on next page. . .

...continued from previous page

getSendConnection	cppvmSendStream

`cppvmSendConnectionType getSendConnection();`

Parameter

Parameter	Type	Default	Description
int	return		connection type: `CPPvmOneToOne`: one to one send connection `CPPvmMulticast`: send to more than one process `CPPvmGroupBroadcast`: send to all processes of a group

Exceptions

cppvmExceptionForbiddenCall	object not initialized, it is not allowed to use it

Example

```
cppvmSendStream myStream("my group", CPPvmSBgroup, PvmDataDefault);
cout << myStream.getSendConnection();
```

Table 11.141: cppvmSendStream: getSendMode

getSendMode	cppvmSendStream
get the send mode	

Synopsis

`cppvmSendMode getSendMode();`

Parameter

Parameter	Type	Default	Description
int	return		mode: `CPPvmBlockingSend`: blocking send `CPPvmNonblockingSend`: nonblocking send

Exceptions

cppvmExceptionForbiddenCall	object not initialized, it is not allowed to use it

Example

```
cppvmSendStream myStream("my group", CPPvmSBgroup, PvmDataDefault);
cout << myStream.getSendMode();
```

Table 11.142: cppvmSendStream: getTid

getTid	cppvmSendStream
get task id of a process to send to	

Synopsis

continued on next page...

. . . continued from previous page

getTid			cppvmSendStream
`int getTid(int indexTid);`			
Parameter			
Parameter	Type	Default	Description
`indexTid`	in		index of the send-to-process to get the task id from
int	return		task id
Exceptions			
cppvmExceptionOutOfRange	Index is not in the array limits		
Example			

```
cppvmSpawnMultiConnection children("hello2",5);
cppvmSendStream myStream(children, CPPvmSBallChildren,
PvmDataDefault);
int i;
for(i=0; i<myStream.getNumTid(); i++)
cout << myStream.getTid(i);
```

Table 11.143: cppvmSendStream: operator<<

operator<<			cppvmSendStream
send stream operator			
Synopsis			
`cppvmSendStream& operator<<(cppvmSendStream& sendStrm, cppvmObject& obj);`			
Parameter			
Parameter	Type	Default	Description
`cppvmObject`	in		objects derived from `cppvmObject`
Exceptions			
cppvmExceptionBadResult	giving an invalid encoding value not enough memory to create the buffer cannot create new buffer cannot send data properly cannot send data to group: no such group cannot send data to group: no other group members got the wrong handshake unknown send mode		
cppvmExceptionBadParameter	the 'sendMode' in 'send' parameter has a bad value unknown send connection type		
Example			

continued on next page. . .

. . . continued from previous page	
`operator<<`	cppvmSendStream

```
cppvmSpawnConnection myprocess("mychild");
cppvmSendStream sendStrm(myprocess, CPPvmSBchild);
cppvmString string("test message");
// send object twice
sendStrm << string << string;
```

Table 11.144: cppvmSendStream: `operator[]`

`operator[]`			cppvmSendStream
get task id of a process to send to			
Synopsis			
`int& operator[](int indexTid);`			
Parameter			
Parameter	Type	Default	Description
`indexTid`	in		index of the send-to-process to get the task id from
int&	return		task id
Exceptions			
cppvmExceptionOutOfRange	Index is not in the array limits		
Example			

```
cppvmSpawnMultiConnection children("hello2",5);
cppvmSendStream myStream(children, CPPvmSBallChildren,
PvmDataDefault);
int i;
for(i=0; i<myStream.getNumTid(); i++)
cout << myStream[i];
```

Table 11.145: cppvmSendStream: `operator=`

`operator=`			cppvmSendStream
assign object			
Synopsis			
`cppvmSendStream& operator= (const cppvmSendStream& obj)`			
Parameter			
Parameter	Type	Default	Description
`obj`	in		object to assign
`cppvmSendStream`	return		this object
Exceptions			
cppvmExceptionForbiddenCall	`obj` was not initialized		

continued on next page. . .

...continued from previous page	
`operator=`	cppvmSendStream

Example
`cppvmSendStream s1, s2;`
`...`
`s1 = s2;`

Table 11.146: cppvmSendStream: `operator==`

`operator==`			cppvmSendStream
test if objects are equal			
Synopsis			
`bool operator== (const cppvmSendStream& obj)`			
Parameter			
Parameter	Type	Default	Description
obj	in		object to compare with
bool	return		compare result
Exceptions			
cppvmExceptionForbiddenCall	`obj` was not initialized		
Example			
`cppvmSendStream s1, s2;`			
`...`			
`if(s1==s2){ ...`			

Table 11.147: cppvmSendStream: `operator!=`

`operator!=`			cppvmSendStream
test if objects are not equal			
Synopsis			
`bool operator!= (const cppvmSendStream& obj)`			
Parameter			
Parameter	Type	Default	Description
`obj`	in		object to compare with
`bool`	return		compare result
Exceptions			
cppvmExceptionForbiddenCall	`obj` was not initialized		
Example			
`cppvmSendStream s1, s2;`			
`...`			
`if(s1!=s2){ ...`			

Table 11.148: cppvmSendStream: operator<<

operator<<			cppvmSendStream
write object on out ostream			
Synopsis			
`ostream& operator<<(ostream& strm, cppvmSendStream& obj);`			
Parameter			
Parameter	Type	Default	Description
`obj`	in		write this object on ostream
`strm`	in		ostream
`ostream`	return		ostream after writing the object
Exceptions			
cppvmExceptionForbiddenCall	`obj` was not initialized		
Example			
`cppvmSendStream s;` `cout << s;`			

Table 11.149: cppvmSendStream: set

set			cppvmSendStream
set method for cppvmTask objects			
Synopsis			
`set(cppvmTask& task, cppvmSendTask mode, int initSend);`			
Parameter			
Parameter	Type	Default	Description
`task`	in		description of the process to send to
`mode`	in		send mode: `CPPvmSBtask:` send blocking `CPPvmSNBtask:` send nonblocking
`initSend`	in	PvmDataDefault	PVM send scheme: `PvmDataDefault:` XDR encoding (works everytime) `PvmDataRaw:` do NOT encode information (only for send/receive between task on homogeneous architectures) `PvmDataInPlace:` do not copy data in sending buffer, use the data memory instead. ATTENTION: the data memory MUST not deleted between packing and sending!
Exceptions			
continued on next page...			

. . . *continued from previous page*	
`set`	cppvmSendStream

cppvmExceptionBadParameter	send mode does not exist invalid sendEncoding argument
cppvmExceptionOutOfRange	the given encoding scheme does not exist. the given connection does not exist. tid cannot be negative tid cannot be < 0 the given mode does not exist.

Example

```
cppvmTask task();
...
cppvmSendStream myStream():
myStream.set(task, CPPvmSNBtask, PvmDataDefault);
```

Table 11.150: cppvmSendStream: `set`

`set`	cppvmSendStream
set method for cppvmConnection objects	

Synopsis

```
set(cppvmConnection& conn, cppvmSendConnection mode, int initSend);
```

Parameter

Parameter	Type	Default	Description
`conn`	in		description of the parent process to send to
`mode`	in		send mode: `CPPvmSBparent:` send blocking `CPPvmSNBparent:` send nonblocking
`initSend`	in	PvmDataDefault	PVM send scheme: `PvmDataDefault:` XDR encoding (works everytime) `PvmDataRaw:` do NOT encode information (only for send/receive between task on homogeneous architectures) `PvmDataInPlace:` do not copy data in sending buffer, use the data memory instead. ATTENTION: the data memory MUST not deleted between packing and sending!

Exceptions

cppvmExceptionBadParameter	send mode does not exist invalid sendEncoding argument

continued on next page. . .

. . . continued from previous page	
`set`	cppvmSendStream
cppvmExceptionOutOfRange	the given encoding scheme does not exist. the given connection does not exist. tid cannot be negative tid cannot be < 0 the given mode does not exist. cannot send to parent tid - No parent process
Example	

```
cppvmConnection pvmConn();
...
cppvmSendStream myStream();
myStream.set(pvmConn, CPPvmSNBparent, PvmDataRaw);
```

Table 11.151: cppvmSendStream: `set`

`set`	cppvmSendStream
set method for cppvmSpawnConnection objects	

Synopsis

```
set(cppvmSpawnConnection& sconn, cppvmSendSpawnConnection mode, int
initSend);
```

Parameter

Parameter	Type	Default	Description
`sconn`	in		description of the child process to send to
`mode`	in		send mode: `CPPvmSBchild:` send blocking `CPPvmSNBchild:` send nonblocking
`initSend`	in	PvmDataDefault	PVM send scheme: `PvmDataDefault:` XDR encoding (works everytime) `PvmDataRaw:` do NOT encode information (only for send/receive between task on homogeneous architectures) `PvmDataInPlace:` do not copy data in sending buffer, use the data memory instead. ATTENTION: the data memory MUST not deleted between packing and sending!

Exceptions

cppvmExceptionBadParameter	send mode does not exist invalid sendEncoding argument

continued on next page. . .

... continued from previous page

set	cppvmSendStream
cppvmExceptionOutOfRange	the given encoding scheme does not exist. the given connection does not exist. tid cannot be negative tid cannot be < 0 the given mode does not exist.

Example

```
cppvmSpawnConnection child("hello2");
...
cppvmSendStream myStream();
myStream.set(child, CPPvmSBchild, PvmDataInPlace);
```

Table 11.152: cppvmSendStream: `set`

set	cppvmSendStream
set method for cppvmSpawnMultiConnection objects	

Synopsis

```
set(cppvmSpawnMultiConnection& smconn, cppvmSendSpawnMultiConnection
mode, int initSend);
```

Parameter

Parameter	Type	Default	Description
`sconn`	in		description of the child processes to send to
`mode`	in		send mode: **`CPPvmSBtaggedChildren:`** send blocking to tagged children **`CPPvmSNBtaggedChildren:`** send nonblocking to tagged children **`CPPvmSBallChildren:`** send blocking to all children **`CPPvmSNBallChildren:`** send nonblocking to all children

continued on next page...

...continued from previous page			
`set`			cppvmSendStream
`initSend`	in	PvmDataDefault	PVM send scheme: `PvmDataDefault:` XDR encoding (works everytime) `PvmDataRaw:` do NOT encode information (only for send/receive between task on homogeneous architectures) `PvmDataInPlace:` do not copy data in sending buffer, use the data memory instead. ATTENTION: the data memory MUST not deleted between packing and sending!

Exceptions	
cppvmExceptionBadParameter	send mode does not exist invalid sendEncoding argument
cppvmExceptionOutOfRange	the given encoding scheme does not exist. the given connection does not exist. tid cannot be negative tid cannot be < 0 the given mode does not exist.

Example

```
cppvmSpawnMultiConnection children("hello2",5);
...
cppvmSendStream myStream();
myStream.set(children, CPPvmSBallChildren, PvmDataDefault);
```

Table 11.153: cppvmSendStream: `set`

`set`			cppvmSendStream
set method for group broadcasting with current context			

Synopsis

```
set(char *group, cppvmSendGroup mode, int initSend);
```

Parameter			
Parameter	Type	Default	Description
`group`	in		group name
`mode`	in		send mode: `CPPvmSBgroup:` send blocking `CPPvmSNBgroup:` send nonblocking

continued on next page...

... continued from previous page			
`set`			cppvmSendStream
`initSend`	in	PvmDataDefault	PVM send scheme: `PvmDataDefault`: XDR encoding (works everytime) `PvmDataRaw`: do NOT encode information (only for send/receive between task on homogeneous architectures) `PvmDataInPlace`: do not copy data in sending buffer, use the data memory instead. ATTENTION: the data memory MUST not deleted between packing and sending!

Exceptions	
cppvmExceptionBadParameter	send mode does not exist invalid sendEncoding argument
cppvmExceptionOutOfRange	the given encoding scheme does not exist. the given connection does not exist. tid cannot be negative tid cannot be < 0 the given mode does not exist.

Example

```
cppvmSendStream myStream();
myStream.set("my group name", CPPvmSBgroup, PvmDataRaw);
```

Table 11.154: cppvmSendStream: `set`

`set`			cppvmSendStream
set method for group broadcasting with given context			
Synopsis			
`set(char *group, int context, cppvmSendGroup mode, int initSend);`			
Parameter			
Parameter	Type	Default	Description
`group`	in		group name
`context`	in		context to send with
`mode`	in		send mode: `CPPvmSBgroup`: send blocking `CPPvmSNBgroup`: send nonblocking

continued on next page...

. . . continued from previous page			
`set`			cppvmSendStream
`initSend`	in	PvmDataDefault	PVM send scheme: `PvmDataDefault:` XDR encoding (works everytime) `PvmDataRaw:` do NOT encode information (only for send/receive between task on homogeneous architectures) `PvmDataInPlace:` do not copy data in sending buffer, use the data memory instead. ATTENTION: the data memory MUST not deleted between packing and sending!

Exceptions	
cppvmExceptionBadParameter	send mode does not exist invalid sendEncoding argument
cppvmExceptionOutOfRange	the given encoding scheme does not exist. the given connection does not exist. tid cannot be negative tid cannot be < 0 the given mode does not exist.

Example

```
cppvmSendStream myStream();
myStream.set("my group name", 33, CPPvmSBgroup, PvmDataRaw);
```

Table 11.155: cppvmSendStream: `set`

`set`			cppvmSendStream
set method for send to a specific task with current context			
Synopsis			
`set(int taskId, cppvmSendGroup mode, int initSend);`			
Parameter			
Parameter	Type	Default	Description
`taskId`	in		task id of the process to send to
`mode`	in		send mode: `CPPvmSBtid:` send blocking `CPPvmSNBtid:` send nonblocking
			continued on next page. . .

. . . continued from previous page

set			cppvmSendStream
`initSend`	in	PvmDataDefault	PVM send scheme: `PvmDataDefault:` XDR encoding (works everytime) `PvmDataRaw:` do NOT encode information (only for send/receive between task on homogeneous architectures) `PvmDataInPlace:` do not copy data in sending buffer, use the data memory instead. ATTENTION: the data memory MUST not deleted between packing and sending!

Exceptions	
cppvmExceptionBadParameter	send mode does not exist invalid sendEncoding argument
cppvmExceptionOutOfRange	the given encoding scheme does not exist. the given connection does not exist. tid cannot be negative tid cannot be < 0 the given mode does not exist.

Example

```
cppvmSendStream myStream();
myStream.set(1201, CPPvmSNBtid, PvmDataDefault);
```

Table 11.156: cppvmSendStream: set

set	cppvmSendStream
set method for send to a specific task with given context	

Synopsis

```
set(int taskId, int context, cppvmSendGroup mode, int initSend);
```

Parameter

Parameter	Type	Default	Description
`taskId`	in		task id of the process to send to
`context`	in		context id
`mode`	in		send mode: `CPPvmSBtid:` send blocking `CPPvmSNBtid:` send nonblocking

continued on next page. . .

. . . continued from previous page			
`set`			cppvmSendStream
`initSend`	in	PvmDataDefault	PVM send scheme: `PvmDataDefault:` XDR encoding (works everytime) `PvmDataRaw:` do NOT encode information (only for send/receive between task on homogeneous architectures) `PvmDataInPlace:` do not copy data in sending buffer, use the data memory instead. ATTENTION: the data memory MUST not deleted between packing and sending!

Exceptions	
cppvmExceptionBadParameter	send mode does not exist invalid sendEncoding argument
cppvmExceptionOutOfRange	the given encoding scheme does not exist. the given connection does not exist. tid cannot be negative tid cannot be < 0 the given mode does not exist.

Example

```
cppvmSendStream myStream();
myStream.set(1201, 44, CPPvmSNBtid, PvmDataDefault);
```

Table 11.157: cppvmSendStream: `set`

`set`	cppvmSendStream
set method for send to specific tasks with current context	

Synopsis

```
void set(int* taskIds, unsigned int numTasks, cppvmSendTids mode, int
initSend);
```

Parameter

Parameter	Type	Default	Description
`taskIds`	in		tasks array of the processes to send to
`numTasks`	in		number of task ids in the array
`mode`	in		send mode: `CPPvmSBtids:` send blocking `CPPvmSNBtids:` send nonblocking

continued on next page. . .

...continued from previous page			
set			cppvmSendStream
initSend	in	PvmDataDefault	PVM send scheme: **PvmDataDefault:** XDR encoding (works everytime) **PvmDataRaw:** do NOT encode information (only for send/receive between task on homogeneous architectures) **PvmDataInPlace:** do not copy data in sending buffer, use the data memory instead. ATTENTION: the data memory MUST not deleted between packing and sending!

Exceptions

cppvmExceptionBadParameter	taskIds pointer is NULL send mode does not exist invalid sendEncoding argument tid is already in the sending array
cppvmExceptionOutOfRange	the given encoding scheme does not exist. the given connection does not exist. tid cannot be negative tid cannot be <0 the given mode does not exist.

Example

```
int ids[2];
cppvmSpawnConnection child1("hi");
cppvmSpawnConnection child2("ho");
ids[0]=child1.getChildTid();
ids[1]=child2.getChildTid();
cppvmSendStream myStream();
myStream.set(ids, 2, CPPvmSNBtids, PvmDataDefault);
```

Discussion

The task ids in the array must differ from each other. If a id occures twice or more often in the array, an exception is thrown.

Table 11.158: cppvmSendStream: set

set	cppvmSendStream
set method for send to specific tasks with given context	

Synopsis

```
void set(int* taskIds, unsigned int numTasks, int context,
cppvmSendTids mode, int initSend);
```

continued on next page...

...continued from previous page	
set	cppvmSendStream

Parameter

Parameter	Type	Default	Description
taskIds	in		tasks array of the processes to send to
numTasks	in		number of task ids in the array
context	in		context id
mode	in		send mode: `CPPvmSBtids:` send blocking `CPPvmSNBtids:` send nonblocking
initSend	in	PvmDataDefault	PVM send scheme: `PvmDataDefault:` XDR encoding (works everytime) `PvmDataRaw:` do NOT encode information (only for send/receive between task on homogeneous architectures) `PvmDataInPlace:` do not copy data in sending buffer, use the data memory instead. ATTENTION: the data memory MUST not deleted between packing and sending!

Exceptions

cppvmExceptionBadParameter	taskIds pointer is NULL send mode does not exist invalid sendEncoding argument tid is already in the sending array
cppvmExceptionOutOfRange	the given encoding scheme does not exist. the given connection does not exist. tid cannot be negative tid cannot be < 0 the given mode does not exist.

Example

```
int ids[2];
cppvmSpawnConnection child1("hi");
cppvmSpawnConnection child2("ho");
ids[0]=child1.getChildTid();
ids[1]=child2.getChildTid();
cppvmSendStream myStream();
myStream.set(ids, 2, 1001, CPPvmSNBtids, PvmDataDefault);
```

Discussion

The task ids in the array must differ from each other. If a id occures twice or more often in the array, an exception is thrown.

Table 11.159: cppvmSendStream: set

set	cppvmSendStream
set method for send to specific tasks with given context array	

Synopsis

```
void set(int* taskIds, unsigned int numTasks, int* contextArr,
cppvmSendTids mode, int initSend);
```

Parameter

Parameter	Type	Default	Description
`taskIds`	in		tasks array of the processes to send to
`numTasks`	in		number of task ids in the array
`contextArr`	in		context id array
`mode`	in		send mode: `CPPvmSBtids`: send blocking `CPPvmSNBtids`: send nonblocking
`initSend`	in	PvmDataDefault	PVM send scheme: `PvmDataDefault`: XDR encoding (works everytime) `PvmDataRaw`: do NOT encode information (only for send/receive between task on homogeneous architectures) `PvmDataInPlace`: do not copy data in sending buffer, use the data memory instead. ATTENTION: the data memory MUST not deleted between packing and sending!

Exceptions

cppvmExceptionBadParameter	taskIds pointer is NULL contextArr pointer is NULL. send mode does not exist invalid sendEncoding argument tid is already in the sending array
cppvmExceptionOutOfRange	the given encoding scheme does not exist. the given connection does not exist. tid cannot be negative tid cannot be < 0 the given mode does not exist.

Example

continued on next page...

...continued from previous page

set	cppvmSendStream

```
int ids[2];
int contextArr[2];
cppvmSpawnConnection child1("hi");
cppvmSpawnConnection child2("ho");
ids[0]=child1.getChildTid();
contextArr[0]=child1.getContextId();
ids[1]=child2.getChildTid();
contextArr[1]=child2.getContextId();
cppvmSendStream myStream();
myStream.set(ids, 2, contextArr, CPPvmSNBtids, PvmDataDefault);
```

Discussion

The task ids in the array must differ from each other. If a id occures twice or more often in the array, an exception is thrown.

Table 11.160: cppvmSendStream: set

set	cppvmSendStream

set to send back to a process, from which a message was received with `recStrm` with current context

Synopsis

`set(cppvmReceiveStream& recStrm, cppvmSendBack mode, int initSend);`

Parameter

Parameter	Type	Default	Description
`recStrm`	in		send back to the process from which a message was received with this stream
`mode`	in		send mode: `CPPvmSBback:` send blocking `CPPvmSNBback:` send nonblocking
`initSend`	in	PvmDataDefault	PVM send scheme: `PvmDataDefault:` XDR encoding (works everytime) `PvmDataRaw:` do NOT encode information (only for send/receive between task on homogeneous architectures) `PvmDataInPlace:` do not copy data in sending buffer, use the data memory instead. ATTENTION: the data memory MUST not deleted between packing and sending!

Exceptions

continued on next page...

. . . continued from previous page

set	cppvmSendStream
cppvmExceptionBadParameter	send mode does not exist invalid sendEncoding argument
cppvmExceptionOutOfRange	the given encoding scheme does not exist. the given connection does not exist. tid cannot be negative tid cannot be < 0 the given mode does not exist.

Example

```
cppvmReceiveStream recStrm();
...
cppvmString recStr("");
recStrm >> recStr;
cppvmSendStream myStream();
myStream.set(recStrm, CPPvmSNBback, PvmDataRaw);
```

Table 11.161: cppvmSendStream: set

set	cppvmSendStream
set to send back to a process, from which a message was received with `recStrm` with given context	

Synopsis

```
set(cppvmReceiveStream& recStrm, int context, cppvmSendBack mode, int
initSend);
```

Parameter

Parameter	Type	Default	Description
`recStrm`	in		send back to the process from which a message was received with this stream
`context`	in		context id
`mode`	in		send mode: `CPPvmSBback:` send blocking `CPPvmSNBback:` send nonblocking

continued on next page. . .

. . . continued from previous page			
`set`			cppvmSendStream
`initSend`	in	PvmDataDefault	PVM send scheme: `PvmDataDefault:` XDR encoding (works everytime) `PvmDataRaw:` do NOT encode information (only for send/receive between task on homogeneous architectures) `PvmDataInPlace:` do not copy data in sending buffer, use the data memory instead. ATTENTION: the data memory MUST not deleted between packing and sending!

Exceptions	
cppvmExceptionBadParameter	send mode does not exist invalid sendEncoding argument
cppvmExceptionOutOfRange	the given encoding scheme does not exist. the given connection does not exist. tid cannot be negative tid cannot be < 0 the given mode does not exist.

Example

```
cppvmReceiveStream recStrm();
...
cppvmString recStr("");
recStrm >> recStr;
cppvmSendStream myStream();
myStream.set(recStrm, 55, CPPvmSNBback, PvmDataRaw);
```

Table 11.162: cppvmSendStream: `setMsgBuf`

`setMsgBuf`			cppvmSendStream
set if a new send message buffer is wanted or not, and set also the encoding of this message buffer			
Synopsis			
`void setMsgBuf(bool ownMsgBuffer, int encoding);`			
Parameter			
Parameter	Type	Default	Description
`ownMsgBuffer`	in		use own message buffer `true:` new message buffer for each sent message `false:` use default message buffer
			continued on next page. . .

... continued from previous page

setMsgBuf			cppvmSendStream
`encoding`	in	PvmDataDefault	PVM send scheme: `PvmDataDefault:` XDR encoding (works everytime) `PvmDataRaw:` do NOT encode information (only for send/receive between task on homogeneous architectures) `PvmDataInPlace:` do not copy data in sending buffer, use the data memory instead. ATTENTION: the data memory MUST not deleted between packing and sending!

Exceptions	
cppvmExceptionBadParameter	invalid sendEncoding argument

Example

```
cppvmSendStream myStream(pvmConn, CPPvmSNBparent, PvmDataRaw);
myStream.setMsgBuf(true, PvmDataDefault);
```

Discussion

By using the constructors or the set methods no own message buffer is used for sending the messages

11.6 cppvmReceiveStream

The class cppvmReceiveStream specifies a message passing receive stream between PVM processes. Receive streams are used like standard C++ in streams:

```
// spawn child process 'mychild'
cppvmSpawnConnection myprocess("mychild");
// initialize blocking receive stream from child process
cppvmReceiveStream recStrm(myprocess, CPPvmRBchild);
cppvmString string1, string2;
// receive objects
recStrm >> string1 >> string2;
```

The receive stream is specified by the process handling classes of the parallel virtual machine (e.g. cppvmConnection, cppvmSpawnConnection, cppvmTask ...), the send mode and time limits.

There are three receive modes:

`CPPvmRB<connectionSpecifier>` blocking receive: the receiving process is blocked until the message was sent

`CPPvmRNB<connectionSpecifier>` nonblocking receive: the process tries to receive a message; if no message was sent, the process continues without having the message received

`CPPvmRT<connectionSpecifier>` timeout receive: the process tries to receive a message for a specified time; if no message was send within that time, the process continues without having the message received

The connection specifier depends on the involved process handling classes:

class	**connection specifier**
cppvmTask	`task`
cppvmConnection	`parent`
cppvmSpawnConnection	`parent`, `child`
receive from a process specified by a task id	`tid`
receive from any process	`any`
receive from the same process again	`same`

To check if there is a message or not use the `probe` methods. The `probe` methods can give several informations about an incoming message, for example the message tag, message size, buffer id and the task id of the sending process. By using a `probe` method the messages are not received. You still have to use the operator `>>` to receive messages.

The method `received` returns the result of the last receive operation. This is useful for the nonblocking and timeout receive modes. You can apply these receive operations and determine afterwards if the receive was successful of not. Do not use the `received` method to look for incoming messages. This has to be done with the `probe` methods.

Here are some receive stream examples:

```
cppvmString string("and now for completely the same test message");

// spawn child process 'hello2'
cppvmSpawnConnection myprocess("hello2");
// blocking receive from child process
cppvmReceiveStream recStrm(myprocess, CPPvmRBchild);
recStrm >> string;

// receive descriptor
cppvmReceiveStream recStrm2;
// non-blocking receive from any process
// use raw encoding scheme
recStrm2.set(myprocess, CPPvmRNBany);
recStrm2 >> string;
// object received?
bool msgReceived;
msgReceived = recStrm2.received();
if(msgReceived)
 cout << "Object: " << string << " received." << endl;


// timeout receive from parent process
cppvmConnection parent;
// timeout 1 sec, 200 usec
cppvmReceiveStream recStrm3(parent, CPPvmRTparent,
                            1, 200);
recStrm3 >> string;

// receive blocking from a process
// specified by its task id
cppvmReceiveStream recStrm4(1234, CPPvmRBtid);
recStrm4 >> string

// receive non-blocking from the
// same process
cppvmReceiveStream recStrm5(recStrm4, CPPvmRBsame);
recStrm5 >> string
```

Table 11.163: cppvmReceiveStream: constructor

constructor	cppvmReceiveStream
default constructor	
Synopsis	
`cppvmReceiveStream();`	
Example	
`cppvmReceiveStream myStream;`	
Discussion	
Use the default constructor only if the stream is initialized later on with one of the `set` methods.	

Table 11.164: cppvmReceiveStream: constructor

constructor			cppvmReceiveStream
constructor for cppvmTask objects			
Synopsis			
`cppvmReceiveStream(cppvmTask& task, cppvmRecTask mode,` `int seconds, int useconds);`			
Parameter			
Parameter	Type	Default	Description
`task`	in		description of the process to receive from
`mode`	in		receive mode: `CPPvmRBtask`: blocking receive `CPPvmRNBtask`: nonblocking receive `CPPvmRTtask`: timeout receive
`seconds`	in	0	timeout seconds (only used for mode = CPPvmRT*)
`useconds`	in	0	timeout microseconds (only used for mode = CP-PvmRT*)
Exceptions			
cppvmExceptionBadParameter	receive mode does not exist.		
cppvmExceptionOutOfRange	cannot set timeout, cppvmReceiveStream object is not in timeout mode (use setRecMode(CPPvmTimeoutRec) fist). the given 'newTimeout' contains negative times. 'newTimeout.tv_usec' > 1000000. Seconds must be set with the 'newTimeout.tv_sec' variable		
Example			

`cppvmTask task();`
`...`
`cppvmReceiveStream myStream(task, CPPvmRNBtask);`			

Table 11.165: cppvmReceiveStream: constructor

constructor	cppvmReceiveStream
constructor for cppvmConnection objects	

Synopsis

```
cppvmReceiveStream(cppvmConnection& conn, cppvmRecConnection mode,
int seconds, int useconds);
```

Parameter

Parameter	Type	Default	Description
conn	in		description of the process to receive from
mode	in		receive mode: CPPvmRBparent: blocking receive CPPvmRNBparent: nonblocking receive CPPvmRTparent: timeout receive
seconds	in	0	timeout seconds (only used for mode = CPPvmRT*)
useconds	in	0	timeout microseconds (only used for mode = CPPvmRT*)

Exceptions

cppvmExceptionBadParameter	receive mode does not exist.
cppvmExceptionOutOfRange	cannot set timeout, cppvmReceiveStream object is not in timeout mode (use setRecMode(CPPvmTimeoutRec) fist). the given 'newTimeout' contains negative times. 'newTimeout.tv_usec' > 1000000. Seconds must be set with the 'newTimeout.tv_sec' variable

Example

```
cppvmConnection conn();
...
cppvmReceiveStream myStream(conn, CPPvmRBparent);
```

Table 11.166: cppvmReceiveStream: constructor

constructor	cppvmReceiveStream
constructor for cppvmSpawnConnection objects	

Synopsis

```
cppvmReceiveStream(cppvmSpawnConnection& sconn,
cppvmRecSpawnConnection mode, int seconds, int useconds);
```

Parameter

Parameter	Type	Default	Description
sconn	in		description of the process to receive from

continued on next page...

. . . continued from previous page			
`constructor`			cppvmReceiveStream
`mode`	in		receive mode: `CPPvmRBparent:` blocking receive `CPPvmRNBparent:` nonblocking receive `CPPvmRTparent:` timeout receive
`seconds`	in	0	timeout seconds (only used for mode = CPPvmRT*)
`useconds`	in	0	timeout microseconds (only used for mode = CP-PvmRT*)

Exceptions	
cppvmExceptionBadParameter	receive mode does not exist.
cppvmExceptionOutOfRange	cannot set timeout, cppvmReceiveStream object is not in timeout mode (use setRecMode(CPPvmTimeoutRec) fist). the given 'newTimeout' contains negative times. 'newTimeout.tv_usec' > 1000000. Seconds must be set with the 'newTimeout.tv_sec' variable

Example

```
cppvmSpawnConnection child("hello");
...
cppvmReceiveStream myStream(child, CPPvmRTchild, 1, 333);
```

Table 11.167: cppvmReceiveStream: `constructor`

`constructor`	cppvmReceiveStream
constructor for processes specified by a task id	

Synopsis

```
cppvmReceiveStream(int taskId, cppvmRecTid mode,
int seconds, int useconds);
```

Parameter	Type	Default	Description
`taskId`	in		description of the process to receive from
`mode`	in		receive mode: `CPPvmRBtid:` blocking receive `CPPvmRNBtid:` nonblocking receive `CPPvmRTtid:` timeout receive
`seconds`	in	0	timeout seconds (only used for mode = CPPvmRT*)
`useconds`	in	0	timeout microseconds (only used for mode = CP-PvmRT*)

continued on next page. . .

... continued from previous page	
`constructor`	cppvmReceiveStream

Exceptions	
cppvmExceptionBadParameter	receive mode does not exist.
cppvmExceptionOutOfRange	cannot set timeout, cppvmReceiveStream object is not in timeout mode (use setRecMode(CPPvmTimeoutRec) fist). the given 'newTimeout' contains negative times. 'newTimeout.tv_usec' > 1000000. Seconds must be set with the 'newTimeout.tv_sec' variable

Example
`cppvmReceiveStream myStream(111, CPPvmRNBtid);`

Table 11.168: cppvmReceiveStream: `constructor`

`constructor`	cppvmReceiveStream
constructor for processes specified by a task id with given context	

Synopsis
`cppvmReceiveStream(int taskId, int context,` `cppvmRecTid mode, int seconds, int useconds);`

Parameter	Type	Default	Description
`taskId`	in		description of the process to receive from
`context`	in		context id
`mode`	in		receive mode: `CPPvmRBtid`: blocking receive `CPPvmRNBtid`: nonblocking receive `CPPvmRTtid`: timeout receive
`seconds`	in	0	timeout seconds (only used for mode = CPPvmRT*)
`useconds`	in	0	timeout microseconds (only used for mode = CPPvmRT*)

Exceptions	
cppvmExceptionBadParameter	receive mode does not exist.
cppvmExceptionOutOfRange	cannot set timeout, cppvmReceiveStream object is not in timeout mode (use setRecMode(CPPvmTimeoutRec) fist). the given 'newTimeout' contains negative times. 'newTimeout.tv_usec' > 1000000. Seconds must be set with the 'newTimeout.tv_sec' variable

Example
`cppvmReceiveStream myStream(111, 222, CPPvmRBtid);`

Table 11.169: cppvmReceiveStream: constructor

constructor			cppvmReceiveStream
constructor to receive from any process			
Synopsis			
`cppvmReceiveStream(cppvmRecAny mode, int seconds, int useconds);`			
Parameter			
Parameter	Type	Default	Description
`mode`	in		receive mode: `CPPvmRBany:` blocking receive `CPPvmRNBany:` nonblocking receive `CPPvmRTany:` timeout receive
`seconds`	in	0	timeout seconds (only used for mode = CPPvmRT*)
`useconds`	in	0	timeout microseconds (only used for mode = CPPvmRT*)
Exceptions			
cppvmExceptionBadParameter	receive mode does not exist.		
cppvmExceptionOutOfRange	cannot set timeout, cppvmReceiveStream object is not in timeout mode (use setRecMode(CPPvmTimeoutRec) fist). the given 'newTimeout' contains negative times. 'newTimeout.tv_usec' > 1000000. Seconds must be set with the 'newTimeout.tv_sec' variable		
Example			
`cppvmReceiveStream myStream(CPPvmRTany, 2, 90);`			

Table 11.170: cppvmReceiveStream: constructor

constructor			cppvmReceiveStream
receive from any process			
Synopsis			
`cppvmReceiveStream(int context, cppvmRecAny mode,` `int seconds, int useconds);`			
Parameter			
Parameter	Type	Default	Description
`context`	in		context id
			continued on next page...

. . . continued from previous page			
`constructor`			cppvmReceiveStream
`mode`	in		receive mode: `CPPvmRBany:` blocking receive `CPPvmRNBany:` nonblocking receive `CPPvmRTany:` timeout receive
`seconds`	in	0	timeout seconds (only used for mode = CPPvmRT*)
`useconds`	in	0	timeout microseconds (only used for mode = CPPvmRT*)
Exceptions			
cppvmExceptionBadParameter	receive mode does not exist.		
cppvmExceptionOutOfRange	cannot set timeout, cppvmReceiveStream object is not in timeout mode (use setRecMode(CPPvmTimeoutRec) fist). the given 'newTimeout' contains negative times. 'newTimeout.tv_usec' > 1000000. Seconds must be set with the 'newTimeout.tv_sec' variable		
Example			
`cppvmReceiveStream myStream(111, CPPvmRBany);`			

Table 11.171: cppvmReceiveStream: `constructor`

`constructor`			cppvmReceiveStream
receive from the same processes, from which a receive was done by 'oldStrm'			
Synopsis			
`cppvmReceiveStream(cppvmReceiveStream& oldStrm,` `cppvmRecSame mode, int seconds, int useconds);`			
Parameter			
Parameter	Type	Default	Description
`oldStrm`	in		last receive was done in this stream
`mode`	in		receive mode: `CPPvmRBsame:` blocking receive `CPPvmRNBsame:` nonblocking receive `CPPvmRTsame:` timeout receive
`seconds`	in	0	timeout seconds (only used for mode = CPPvmRT*)
`useconds`	in	0	timeout microseconds (only used for mode = CPPvmRT*)
Exceptions			
cppvmExceptionBadParameter	receive mode does not exist.		
			continued on next page. . .

. . . continued from previous page

constructor	cppvmReceiveStream
cppvmExceptionOutOfRange	cannot set timeout, cppvmReceiveStream object is not in timeout mode (use setRecMode(CPPvmTimeoutRec) fist). the given 'newTimeout' contains negative times. 'newTimeout.tv_usec' > 1000000. Seconds must be set with the 'newTimeout.tv_sec' variable

Example

```
cppvmReceiveStream myStream1(111, CPPvmRBany);
myStream1 >> string;
cppvmReceiveStream myStream2(myStream1, CPPvmRNBsame);
```

Table 11.172: cppvmReceiveStream: constructor

constructor	cppvmReceiveStream

receive from the same processes, from which a receive was done by 'oldStrm' with given context

Synopsis

```
cppvmReceiveStream(cppvmReceiveStream& oldStrm, int context,
cppvmRecSame mode, int seconds, int useconds);
```

Parameter

Parameter	Type	Default	Description
`oldStrm`	in		last receive was done in this stream
`context`	in		context id
`mode`	in		receive mode: `CPPvmRBsame:` blocking receive `CPPvmRNBsame:` nonblocking receive `CPPvmRTsame:` timeout receive
`seconds`	in	0	timeout seconds (only used for mode = CPPvmRT*)
`useconds`	in	0	timeout microseconds (only used for mode = CP-PvmRT*)

Exceptions

cppvmExceptionBadParameter	receive mode does not exist.
cppvmExceptionOutOfRange	cannot set timeout, cppvmReceiveStream object is not in timeout mode (use setRecMode(CPPvmTimeoutRec) fist). the given 'newTimeout' contains negative times. 'newTimeout.tv_usec' > 1000000. Seconds must be set with the 'newTimeout.tv_sec' variable

Example

continued on next page. . .

... continued from previous page	
`constructor`	cppvmReceiveStream

```
cppvmReceiveStream myStream1(111, CPPvmRBany);
myStream1 >> string;
cppvmReceiveStream myStream2(myStream1, 22, CPPvmRBsame);
```

Table 11.173: cppvmReceiveStream: `copy constructor`

`copy constructor`			cppvmReceiveStream
copy constructor			
Synopsis			
`cppvmReceiveStream(const cppvmReceiveStream& obj);`			
Parameter			
Parameter	Type	Default	Description
`obj`	in		object to copy
Exceptions			
cppvmExceptionForbiddenCall	object not initialized, it is not allowed to use it		
Example			
`cppvmReceiveStream myStream = previousDefinedStream;`			

Table 11.174: cppvmReceiveStream: `getContext`

`getContext`			cppvmReceiveStream
get the context id of a process to receive from			
Synopsis			
`int getContext();`			
Parameter			
Parameter	Type	Default	Description
int	return		context id
Example			
`cppvmReceiveStream myStream(CPPvmRBany);` `cout << myStream.getContext();`			

Table 11.175: cppvmReceiveStream: `getLastRecFromTid`

`getLastRecFromTid`	cppvmReceiveStream
the last message was received from a process with this task id	
Synopsis	
`int getLastRecFromTid();`	
Parameter	
	continued on next page...

...continued from previous page			
`getLastRecFromTid`			cppvmReceiveStream
Parameter	Type	Default	Description
int	return		task id
Exceptions			
cppvmExceptionForbiddenCall	object not initialized, it is not allowed to use it		
Example			

```
cppvmReceiveStream myStream(CPPvmRBany);
...
cout << myStream.getLastRecFromTid();
```

Table 11.176: cppvmReceiveStream: `getRecFromTid`

`getRecFromTid`			cppvmReceiveStream
the last message will be received from a process with this task id			
Synopsis			
`int getRecFromTid();`			
Parameter			
Parameter	Type	Default	Description
int	return		task id
Exceptions			
cppvmExceptionForbiddenCall	object not initialized, it is not allowed to use it		
Example			

```
cppvmReceiveStream myStream(CPPvmRNBany);
...
cout << myStream.getRecFromTid();
```

Table 11.177: cppvmReceiveStream: `getRecMode`

`getRecMode`			cppvmReceiveStream
get the current receive mode			
Synopsis			
`int getRecMode();`			
Parameter			
Parameter	Type	Default	Description
int	return		current receive mode: `CPPvmBlockingRec`: blocking receive `CPPvmNonblockingRec`: nonblocking receive `CPPvmTimeoutRec`: timeout receive
			continued on next page...

. . . continued from previous page	
`getRecMode`	cppvmReceiveStream

Exceptions	
cppvmExceptionForbiddenCall	object not initialized, it is not allowed to use it

Example

```
cppvmReceiveStream myStream(CPPvmRNBany);
...
cout << myStream.getRecMode();
```

Table 11.178: cppvmReceiveStream: `getTimeoutDirect`

`getTimeoutDirect`	cppvmReceiveStream
get the current timeout values	

Synopsis

```
struct timeval* getTimeoutDirect();
```

Parameter

Parameter	Type	Default	Description
struct timeval*	return		seconds and microseconds information

Exceptions	
cppvmExceptionForbiddenCall	object not initialized, it is not allowed to use it

Example

```
cppvmReceiveStream myStream(CPPvmRNBany);
struct timeval *timeout;
timout = myStream.getTimeoutDirect();
```

Table 11.179: cppvmReceiveStream: `operator>>`

`operator>>`	cppvmReceiveStream
receive stream operator	

Synopsis

```
cppvmReceiveStream& operator>>(cppvmReceiveStream& recStrm,
cppvmObject& obj);
```

Parameter

Parameter	Type	Default	Description
`cppvmObject`	in		objects derived from `cppvmObject`

Exceptions

continued on next page. . .

...continued from previous page	
`operator>>`	cppvmReceiveStream

cppvmExceptionBadResult	cannot receive a message cannot get my tid (pvm enrolled?) giving an invalid encoding value not enough memory to create the buffer cannot create new buffer cannot send data properly unknown send handshake

Example

```
cppvmSpawnConnection myprocess("mychild");
cppvmReceiveStream recStrm(myprocess, CPPvmRBchild);
cppvmString string("test message");
recStrm >> string;
```

Table 11.180: cppvmReceiveStream: `operator=`

`operator=`	cppvmReceiveStream
assign object	

Synopsis

```
cppvmReceiveStream& operator= (const cppvmReceiveStream& obj)
```

Parameter

Parameter	Type	Default	Description
`obj`	in		object to assign
`cppvmReceiveStream`	return		this object

Exceptions

cppvmExceptionForbiddenCall	`obj` was not initialized

Example

```
cppvmReceiveStream s1, s2;
...
s1 = s2;
```

Table 11.181: cppvmReceiveStream: `operator==`

`operator==`	cppvmReceiveStream
test if objects are equal	

Synopsis

```
bool operator== (const cppvmReceiveStream& obj)
```

Parameter

Parameter	Type	Default	Description
obj	in		object to compare with

continued on next page...

...continued from previous page			
`operator==`			cppvmReceiveStream
bool	return		compare result
Exceptions			
cppvmExceptionForbiddenCall	`obj` was not initialized		
Example			

`cppvmReceiveStream s1, s2;`
`...`
`if(s1==s2){ ...`			

Table 11.182: cppvmReceiveStream: `operator!=`

`operator!=`			cppvmReceiveStream
test if objects are not equal			
Synopsis			
`bool operator!= (const cppvmReceiveStream& obj)`			
Parameter			
Parameter	Type	Default	Description
`obj`	in		object to compare with
`bool`	return		compare result
Exceptions			
cppvmExceptionForbiddenCall	`obj` was not initialized		
Example			

`cppvmReceiveStream s1, s2;`
`...`
`if(s1!=s2){ ...`			

Table 11.183: cppvmReceiveStream: `operator<<`

`operator<<`			cppvmReceiveStream
write object on ostream			
Synopsis			
`ostream& operator<<(ostream& strm, cppvmReceiveStream& obj);`			
Parameter			
Parameter	Type	Default	Description
`obj`	in		write this object on ostream
`strm`	in		ostream
`ostream`	return		ostream after writing the object
Exceptions			
cppvmExceptionForbiddenCall	`obj` was not initialized		
Example			
			continued on next page...

...continued from previous page	
operator<<	cppvmReceiveStream

```
cppvmReceiveStream s;
cout << s;
```

Table 11.184: cppvmReceiveStream: probe

probe			cppvmReceiveStream
test if a message of a certain tag has arrived			
Synopsis			
`bool probe(int msgtag);`			
Parameter			
Parameter	Type	Default	Description
`msgTag`	in		message tag
`bool`	return		message with this message tag has arrived
Exceptions			
cppvmExceptionForbiddenCall	object not initialized, it is not allowed to use it		
Example			

```
cppvmReceiveStream myStream(CPPvmRBany);
if(myStream.probe(cppvmFloat::cppvmClassMsgTag()){
...
```

Table 11.185: cppvmReceiveStream: probe

probe			cppvmReceiveStream
test if a message of a certain tag has arrived, return some informations about the arrived message			
Synopsis			
`bool probe(int msgtag, int& bufid, int& bytes, int& tid);`			
Parameter			
Parameter	Type	Default	Description
`msgTag`	in		message tag
`bufid`	out		message buffer, in which the message has arrived
`bytes`	out		message body length in bytes
`tid`	out		source of the message
`bool`	return		message with this message tag has arrived
Exceptions			
cppvmExceptionForbiddenCall	object not initialized, it is not allowed to use it		
cppvmExceptionBadResult	could not get buffer informations.		
Example			

continued on next page...

...continued from previous page	
`probe`	cppvmReceiveStream

```
cppvmReceiveStream myStream(CPPvmRBany);
int bufid, bytes, tid;
if(myStream.probe(cppvmString::cppvmClassMsgTag(),
&bufid, &bytes, &tid)){
cout << bufid << bytes << tid << endl;
...
```

Table 11.186: cppvmReceiveStream: `probeAny`

`probeAny`			cppvmReceiveStream
test if any message has arrived			
Synopsis			
`bool probeAny();`			
Parameter			
Parameter	Type	Default	Description
`bool`	return		a message has arrived
Exceptions			
cppvmExceptionBadResult	could not get buffer informations.		
cppvmExceptionForbiddenCall	object not initialized, it is not allowed to use it		
Example			

```
cppvmReceiveStream myStream(CPPvmRBany);
if(myStream.probeAny()){
...
```

Table 11.187: cppvmReceiveStream: `probeAny`

`probeAny`			cppvmReceiveStream
test if any message has arrived			
Synopsis			
`bool probeAny(int &msgtag);`			
Parameter			
Parameter	Type	Default	Description
`msgTag`	out		message tag of the received message
`bool`	return		a message has arrived
Exceptions			
cppvmExceptionBadResult	could not get buffer informations.		
cppvmExceptionForbiddenCall	object not initialized, it is not allowed to use it		
Example			

continued on next page...

...continued from previous page	
probeAny	cppvmReceiveStream

```
cppvmReceiveStream myStream(CPPvmRBany);
int msgTag;
if(myStream.probeAny(&msgTag)){
...
```

Table 11.188: cppvmReceiveStream: probeAny

probeAny	cppvmReceiveStream
test if any message has arrived, return some informations about the arrived message	

Synopsis

```
bool probeAny(int &msgtag, int& bufid, int& bytes, int& tid);
```

Parameter

Parameter	Type	Default	Description
msgTag	out		message tag
bufid	out		message buffer, in which the message has arrived
bytes	out		message body length in bytes
tid	out		source of the message
bool	return		a message has arrived

Exceptions

cppvmExceptionBadParameter	receive mode does not exist.
cppvmExceptionForbiddenCall	object not initialized, it is not allowed to use it
cppvmExceptionBadResult	could not get buffer informations.

Example

```
cppvmReceiveStream myStream(CPPvmRBany);
int msgTag, bufid, bytes, tid;
if(myStream.probeAny(&msgTag, &bufid, &bytes, &tid)){
cout << msgTag << bufid << bytes << tid << endl;
...
```

Table 11.189: cppvmReceiveStream: received

received	cppvmReceiveStream
Result of the last receive.	

Synopsis

```
bool received();
```

Parameter

Parameter	Type	Default	Description
bool	return		result of the last receive

Example

continued on next page...

. . . continued from previous page

received	cppvmReceiveStream

```
cppvmReceiveStream myStream;
...
myStream >> cppvmObj;
if(myStream.received()){ ...
```

Discussion

The value is set to the last receive result. For example, the `received()` value of the receive call
`myStream ≫ o1 ≫ o2;`
is the receive result of `o2`. The result of the `received()` method is only modified by a receive operation, e.g.:
`myStream ≫ obj;`
The method itself can not receive data. To wait for an incoming message do not use `received()` like this:

```
cppvmReceiveStream myStream;
...
myStream >> cppvmObj;
bool msgArrived=myStream.received();
while(msgArrived==false){
msgArrived=myStream.received();
}
...
```

If the message does not arrive the first time, this is an endless loop. You have to try the receive operation in every cycle:

```
cppvmReceiveStream myStream;
...
bool msgArrived=false;
while(msgArrived==false){
myStream >> cppvmObj;
msgArrived=myStream.received();
}
...
```

To check for incoming messages without trying to receive them use the `probe` methods.

Table 11.190: cppvmReceiveStream: `set`

set	cppvmReceiveStream
set method for cppvmTask objects	

Synopsis

continued on next page. . .

...continued from previous page	
set	cppvmReceiveStream

```
set(cppvmTask& task, cppvmRecTask mode,
int seconds, int useconds);
```

Parameter

Parameter	Type	Default	Description
task	in		description of the process to receive from
mode	in		receive mode: CPPvmRBtask: blocking receive CPPvmRNBtask: nonblocking receive CPPvmRTtask: timeout receive
seconds	in	0	timeout seconds (only used for mode = CPPvmRT*)
useconds	in	0	timeout microseconds (only used for mode = CPPvmRT*)

Exceptions

cppvmExceptionBadParameter	receive mode does not exist.
cppvmExceptionOutOfRange	cannot set timeout, cppvmReceiveStream object is not in timeout mode (use setRecMode(CPPvmTimeoutRec) fist). the given 'newTimeout' contains negative times. 'newTimeout.tv_usec' > 1000000. Seconds must be set with the 'newTimeout.tv_sec' variable

Example

```
cppvmTask task();
cppvmReceiveStream myStream();
...
myStream.set(task, CPPvmRNBtask);
```

Table 11.191: cppvmReceiveStream: set

set	cppvmReceiveStream
set method for cppvmConnection objects	

Synopsis

```
set(cppvmConnection& conn, cppvmRecConnection mode,
int seconds, int useconds);
```

Parameter

Parameter	Type	Default	Description
conn	in		description of the process to receive from

continued on next page...

...continued from previous page			
`set`			cppvmReceiveStream
`mode`	in		receive mode: `CPPvmRBparent:` blocking receive `CPPvmRNBparent:` nonblocking receive `CPPvmRTparent:` timeout receive
`seconds`	in	0	timeout seconds (only used for mode = CPPvmRT*)
`useconds`	in	0	timeout microseconds (only used for mode = CPPvmRT*)

Exceptions

cppvmExceptionBadParameter	receive mode does not exist.
cppvmExceptionOutOfRange	cannot set timeout, cppvmReceiveStream object is not in timeout mode (use setRecMode(CPPvmTimeoutRec) fist). the given 'newTimeout' contains negative times. 'newTimeout.tv_usec' > 1000000. Seconds must be set with the 'newTimeout.tv_sec' variable

Example

```
cppvmConnection conn();
cppvmReceiveStream myStream();
...
myStream.set(conn, CPPvmRBparent);
```

Table 11.192: cppvmReceiveStream: `set`

`set`	cppvmReceiveStream
set method for cppvmSpawnConnection objects	

Synopsis

```
set(cppvmSpawnConnection& sconn, cppvmRecSpawnConnection mode,
int seconds, int useconds);
```

Parameter

Parameter	Type	Default	Description
`sconn`	in		description of the process to receive from
`mode`	in		receive mode: `CPPvmRBparent:` blocking receive `CPPvmRNBparent:` nonblocking receive `CPPvmRTparent:` timeout receive
`seconds`	in	0	timeout seconds (only used for mode = CPPvmRT*)
`useconds`	in	0	timeout microseconds (only used for mode = CPPvmRT*)

continued on next page...

. . . continued from previous page

set	cppvmReceiveStream

Exceptions

cppvmExceptionBadParameter	receive mode does not exist.
cppvmExceptionOutOfRange	cannot set timeout, cppvmReceiveStream object is not in timeout mode (use setRecMode(CPPvmTimeoutRec) fist). the given 'newTimeout' contains negative times. 'newTimeout.tv_usec' > 1000000. Seconds must be set with the 'newTimeout.tv_sec' variable

Example

```
cppvmSpawnConnection child("hello");
cppvmReceiveStream myStream();
...
myStream.set(child, CPPvmRTchild, 1, 333);
```

Table 11.193: cppvmReceiveStream: set

set	cppvmReceiveStream
set method for processes specified by a task id	

Synopsis

```
set(int taskId, cppvmRecTid mode,
int seconds, int useconds);
```

Parameter

Parameter	Type	Default	Description
`taskId`	in		description of the process to receive from
`mode`	in		receive mode: `CPPvmRBtid`: blocking receive `CPPvmRNBtid`: nonblocking receive `CPPvmRTtid`: timeout receive
`seconds`	in	0	timeout seconds (only used for mode = CPPvmRT*)
`useconds`	in	0	timeout microseconds (only used for mode = CPPvmRT*)

Exceptions

cppvmExceptionBadParameter	receive mode does not exist.
cppvmExceptionOutOfRange	cannot set timeout, cppvmReceiveStream object is not in timeout mode (use setRecMode(CPPvmTimeoutRec) fist). the given 'newTimeout' contains negative times. 'newTimeout.tv_usec' > 1000000. Seconds must be set with the 'newTimeout.tv_sec' variable

continued on next page. . .

...continued from previous page	
`set`	cppvmReceiveStream

Example
`cppvmReceiveStream myStream();` `myStream.set(111, CPPvmRNBtid);`

Table 11.194: cppvmReceiveStream: `set`

`set`	cppvmReceiveStream
set method for processes specified by a task id with given context	

Synopsis
`set(int taskId, int context,` `cppvmRecTid mode, int seconds, int useconds);`

Parameter	Type	Default	Description
`taskId`	in		description of the process to receive from
`context`	in		context id
`mode`	in		receive mode: `CPPvmRBtid`: blocking receive `CPPvmRNBtid`: nonblocking receive `CPPvmRTtid`: timeout receive
`seconds`	in	0	timeout seconds (only used for mode = CPPvmRT*)
`useconds`	in	0	timeout microseconds (only used for mode = CPPvmRT*)

Exceptions	
cppvmExceptionBadParameter	receive mode does not exist.
cppvmExceptionOutOfRange	cannot set timeout, cppvmReceiveStream object is not in timeout mode (use setRecMode(CPPvmTimeoutRec) fist). the given 'newTimeout' contains negative times. 'newTimeout.tv_usec' > 1000000. Seconds must be set with the 'newTimeout.tv_sec' variable

Example
`cppvmReceiveStream myStream();` `myStream.set(111, 222, CPPvmRBtid);`

Table 11.195: cppvmReceiveStream: `set`

`set`	cppvmReceiveStream
set method to receive from any process	

Synopsis
continued on next page...

. . . continued from previous page	
`set`	cppvmReceiveStream

`set(cppvmRecAny mode, int seconds, int useconds);`

Parameter

Parameter	Type	Default	Description
`mode`	in		receive mode: `CPPvmRBany:` blocking receive `CPPvmRNBany:` nonblocking receive `CPPvmRTany:` timeout receive
`seconds`	in	0	timeout seconds (only used for mode = CPPvmRT*)
`useconds`	in	0	timeout microseconds (only used for mode = CPPvmRT*)

Exceptions

cppvmExceptionBadParameter	receive mode does not exist.
cppvmExceptionOutOfRange	cannot set timeout, cppvmReceiveStream object is not in timeout mode (use setRecMode(CPPvmTimeoutRec) fist). the given 'newTimeout' contains negative times. 'newTimeout.tv_usec' > 1000000. Seconds must be set with the 'newTimeout.tv_sec' variable

Example

```
cppvmReceiveStream myStream();
myStream.set(CPPvmRTany, 2, 90);
```

Table 11.196: cppvmReceiveStream: `set`

`set`	cppvmReceiveStream
receive from any process	

Synopsis

`set(int context, cppvmRecAny mode, int seconds, int useconds);`

Parameter

Parameter	Type	Default	Description
`context`	in		context id
`mode`	in		receive mode: `CPPvmRBany:` blocking receive `CPPvmRNBany:` nonblocking receive `CPPvmRTany:` timeout receive
`seconds`	in	0	timeout seconds (only used for mode = CPPvmRT*)

continued on next page. . .

...continued from previous page

set			cppvmReceiveStream
useseconds	in	0	timeout microseconds (only used for mode = CPPvmRT*)

Exceptions

cppvmExceptionBadParameter	receive mode does not exist.
cppvmExceptionOutOfRange	cannot set timeout, cppvmReceiveStream object is not in timeout mode (use setRecMode(CPPvmTimeoutRec) fist). the given 'newTimeout' contains negative times. 'newTimeout.tv_usec' > 1000000. Seconds must be set with the 'newTimeout.tv_sec' variable

Example

```
cppvmReceiveStream myStream();
myStream.set(111, CPPvmRBany);
```

Table 11.197: cppvmReceiveStream: set

set	cppvmReceiveStream
receive from the same processes, from which a receive was done by 'oldStrm'	

Synopsis

```
set(cppvmReceiveStream& oldStrm, cppvmRecSame mode,
int seconds, int useconds);
```

Parameter

Parameter	Type	Default	Description
oldStrm	in		last receive was done in this stream
mode	in		receive mode: CPPvmRBsame: blocking receive CPPvmRNBsame: nonblocking receive CPPvmRTsame: timeout receive
seconds	in	0	timeout seconds (only used for mode = CPPvmRT*)
useconds	in	0	timeout microseconds (only used for mode = CPPvmRT*)

Exceptions

cppvmExceptionBadParameter	receive mode does not exist.
cppvmExceptionOutOfRange	cannot set timeout, cppvmReceiveStream object is not in timeout mode (use setRecMode(CPPvmTimeoutRec) fist). the given 'newTimeout' contains negative times. 'newTimeout.tv_usec' > 1000000. Seconds must be set with the 'newTimeout.tv_sec' variable

continued on next page...

. . . continued from previous page	
`set`	cppvmReceiveStream

Example

`cppvmReceiveStream myStream1(111, CPPvmRBany);`
`myStream1 >> string;`
`cppvmReceiveStream myStream2();`
`myStream2.set(myStream1, CPPvmRNBsame);`

Table 11.198: cppvmReceiveStream: `set`

`set`	cppvmReceiveStream
receive from the same processes, from which a receive was done by 'oldStrm' with given context	

Synopsis
`set(cppvmReceiveStream& oldStrm, int context,` `cppvmRecSame mode, int seconds, int useconds);`

Parameter	Type	Default	Description
`oldStrm`	in		last receive was done in this stream
`context`	in		context id
`mode`	in		receive mode: `CPPvmRBsame:` blocking receive `CPPvmRNBsame:` nonblocking receive `CPPvmRTsame:` timeout receive
`seconds`	in	0	timeout seconds (only used for mode = CPPvmRT*)
`useconds`	in	0	timeout microseconds (only used for mode = CPPvmRT*)

Exceptions	
cppvmExceptionBadParameter	receive mode does not exist.
cppvmExceptionOutOfRange	cannot set timeout, cppvmReceiveStream object is not in timeout mode (use setRecMode(CPPvmTimeoutRec) fist). the given 'newTimeout' contains negative times. 'newTimeout.tv_usec' > 1000000. Seconds must be set with the 'newTimeout.tv_sec' variable

Example

`cppvmReceiveStream myStream1(111, CPPvmRBany);`
`myStream1 >> string;`
`cppvmReceiveStream myStream2();`
`myStream2.set(myStream1, 22, CPPvmRBsame);`

11.7 cppvmForwardStream

The class cppvmForwardStream specifies a message passing forward stream between PVM processes. To forward messages the "|" symbol is used. Forward streams must be initialized with a send and a receive stream.

There are some restrictions for forwarding objects:

- You can use either receive or forward; it is not possible to receive objects at first and then forward them to another node.
- The forwarded message must be send nonblocking to the forward process.
- The forward process must forward the object with nonblocking send.

Blocking send operations are waiting for a handshake message from the destination process. If a process A sends an object blocking to a process B, A will wait for a handshake from B. But when B forwards the message to a process C and C receives the message finally, then C will send a handshake to B. So A will wait for its handshake from B forever.

On the other hand the process B must not use a blocking send to C, because the message from A must be a nonblocking send. When process C receives the message, it identifies (the original) message as a nonblocking send message and therefore sends no handshake to B. If B has used a blocking send, B will wait for his handshake forever.

In the next small example process A sends a integer and a double object nonblocking to process B. Process B does not receive these objects but forwards them directly to process C. Process C finally receives the two objects.

Process A:

```
// spawn child process 'B'
cppvmSpawnConnection child("B");

// nonblocking send to child process
cppvmSendStream sendStrm(child, CPPvmSNBchild);

// message objects
cppvmInt intObjA(1);
cppvmDouble doubleObjA(2.2);

// send objects
sendStrm << intObjA << doubleObjA;

...
```

Process B:

```
// spawn child process 'C'
cppvmSpawnConnection child("C");

// connect to pvm
cppvmConnection pvmConn;

// message passing streams
cppvmSendStream sendStrm;
cppvmReceiveStream recStrm;
cppvmForwardStream forwStrm;

// an object can only be forwarded if it
// * was sent to this process
//   with a nonblocking send
// * is sent to another process
//   with nonblocking send
sendStrm.set(child, CPPvmSNBchild);
recStrm.set(pvmConn, CPPvmRBparent);
forwStrm.set(recStrm, sendStrm);

// forward a integer and a double object
forwStrm | cppvmInt::cppvmClassMsgTag()
  | cppvmDouble::cppvmClassMsgTag();

...
```

Process C:

```
// connect to pvm
cppvmConnection pvmConn;

// message objects
cppvmInt intObjC();
cppvmDouble doubleObjC();

// blocking receive from parent process
cppvmReceiveStream recStrm(pvmConn, CPPvmRBparent);
recStrm >> intObjC >> doubleObjC;

cout << "object values: int: " << intObjC
     << " double: " << doubleObjC << endl;
```

Table 11.199: cppvmForwardStream: constructor

constructor	cppvmForwardStream
default constructor	
Synopsis	
`cppvmForwardStream();`	
Example	
`cppvmForwardStream myStream;`	
Discussion	
Using this constructor, the stream must be initialized later on with the `set` method.	

Table 11.200: cppvmForwardStream: constructor

constructor			cppvmForwardStream
constructor with stream initialization			
Synopsis			
`cppvmForwardStream(cppvmReceiveStream& recStrm,` `cppvmSendStream& sendStrm);`			
Parameter			
Parameter	Type	Default	Description
`recStrm`	in		receive descriptor (receive from)
`sendStrm`	in		send descriptor (forward to)
Example			

`cppvmSendStream mySendStrm(pvmConn, CPPvmSNBparent);`
`cppvmReceiveStream myRecStrm(pvmConn, CPPvmRBparent);`
`...`
`cppvmForwardStream myFStream(myRecStrm, mySendStrm);`			

Discussion			
the send/receive streams must be initialized			

Table 11.201: cppvmForwardStream: copy constructor

copy constructor			cppvmForwardStream
copy constructor			
Synopsis			
`cppvmForwardStream(const cppvmForwardStream& obj);`			
Parameter			
Parameter	Type	Default	Description
`obj`	in		object to copy
Example			
`cppvmForwardStream myStream = previousDefinedStream;`			

Table 11.202: cppvmForwardStream: `forward`

`forward`			cppvmForwardStream
receive a pvm object and forward it without unpacking the object			
Synopsis			
`bool forward(int msgTag);`			
Parameter			
Parameter	Type	Default	Description
`msgTag`	in		class message tag of the object to forward
`bool`	return		result of the last forward
Exceptions			
cppvmExceptionForbiddenCall	forward stream was not initialized forward cannot be called within a forward command!		
cppvmExceptionBadParameter	can only forward with nonblocking send unknown send connection type		
cppvmExceptionBadResult	cannot receive a message cannot send data properly cannot send data to group: no such group cannot send data to group: no other group members		

Example

```
cppvmForwardStream myStream;
...
bool result;
result = myStream.forward(cppvmInt::cppvmClassMsgTag());
```

Discussion

The `forward` method is also called implicitly by the `|` operator:

```
forwStrm | cppvmDouble::cppvmClassMsgTag();
```

is the same as:

```
{
forwardStrm.forward(cppvmDouble::cppvmClassMsgTag());
return forwardStrm;
}
```

Table 11.203: cppvmForwardStream: `forwarded`

`forwarded`			cppvmForwardStream
Result of the last forward.			
Synopsis			
`bool forwarded();`			
Parameter			
Parameter	Type	Default	Description

continued on next page...

... continued from previous page

forwarded		cppvmForwardStream
bool	return	result of the last forward

Example

```
cppvmForwardStream myStream;
...
myStream | cppvmInt::cppvmClassMsgTag();
if(myStream.forwarded()){ ...
```

Discussion

The value is set to the last forward result. For example, the `forwarded()` value of the forward call
`forwStrm | cppvmInt::cppvmClassMsgTag() | cppvmDouble::cppvmClassMsgTag();`
is the forward result of the double object.

Table 11.204: cppvmForwardStream: `getRecStrm`

getRecStrm			cppvmForwardStream
get the receive steam which is part of the forward stream			

Synopsis

```
cppvmReceiveStream& getRecStrm();
```

Parameter

Parameter	Type	Default	Description
cppvmReceiveStream	return		receive stream

Exceptions

cppvmExceptionForbiddenCall	forward stream was not initialized

Example

```
cppvmForwardStream myStream;
...
cout << myStream.getRecStrm();
```

Table 11.205: cppvmForwardStream: `getSendStrm`

getSendStrm			cppvmForwardStream
get the send steam which is part of the forward stream			

Synopsis

```
cppvmSendStream& getSendStrm();
```

Parameter

Parameter	Type	Default	Description
cppvmSendStream	return		send stream

Exceptions

cppvmExceptionForbiddenCall	forward stream was not initialized

continued on next page...

...continued from previous page	
getSendStrm	cppvmForwardStream

Example

```
cppvmForwardStream myStream;
...
cout << myStream.getSendStrm();
```

Table 11.206: cppvmForwardStream: operator|

operator|	cppvmForwardStream
forward stream operator	

Synopsis

```
cppvmForwardStream&
operator|(cppvmForwardStream& forwardStrm, int msgTag);
```

Parameter

Parameter	Type	Default	Description
forwardStrm	in		forward stream before forwarding
msgTag	in		message tag
cppvmForwardStream	in		forward stream after forwarding

Exceptions

cppvmExceptionForbiddenCall	forward stream was not initialized forward cannot be called within a forward command!
cppvmExceptionBadParameter	can only forward with nonblocking send unknown send connection type
cppvmExceptionBadResult	cannot receive a message cannot send data properly

Example

```
cppvmForwardStream forwStrm(recStrm, sendStrm);
...
// forward a long object
forwStrm | cppvmLong::cppvmClassMsgTag();
```

Table 11.207: cppvmForwardStream: operator=

operator=	cppvmForwardStream
assign object	

Synopsis

```
cppvmForwardStream& operator= (const cppvmForwardStream& obj)
```

Parameter

Parameter	Type	Default	Description

continued on next page...

. . . continued from previous page		
`operator=`		cppvmForwardStream
`obj`	in	object to assign
`cppvmForwardStream&`	return	this object

Example

```
cppvmForwardStream s1, s2;
...
s1 = s2;
```

Table 11.208: cppvmForwardStream: `operator==`

`operator==`			cppvmForwardStream
test if objects are equal			

Synopsis

```
bool operator== (const cppvmForwardStream& obj)
```

Parameter

Parameter	Type	Default	Description
obj	in		object to compare with
bool	return		compare result

Example

```
cppvmForwardStream s1, s2;
...
if(s1==s2){ ...
```

Table 11.209: cppvmForwardStream: `operator!=`

`operator!=`			cppvmForwardStream
test if objects are not equal			

Synopsis

```
bool operator!= (const cppvmForwardStream& obj)
```

Parameter

Parameter	Type	Default	Description
`obj`	in		object to compare with
`bool`	return		compare result

Example

```
cppvmForwardStream s1, s2;
...
if(s1!=s2){ ...
```

Table 11.210: cppvmForwardStream: `operator<<`

`operator<<`	cppvmForwardStream
write object on ostream	

Synopsis

`ostream& operator<<(ostream& strm, cppvmForwardStream& obj);`

Parameter

Parameter	Type	Default	Description
`obj`	in		write this object on ostream
`strm`	in		ostream
`ostream`	return		ostream after writing the object

Example

```
cppvmForwardStream s;
cout << s;
```

Table 11.211: cppvmForwardStream: `set`

`set`	cppvmForwardStream
set the receive and send descriptors	

Synopsis

```
void set(cppvmReceiveStream& recStrm,
cppvmSendStream& sendStrm)
```

Parameter

Parameter	Type	Default	Description
`recStrm`	in		receive descriptor (receive from)
`sendStrm`	in		send descriptor (forward to)

Example

```
cppvmSendStream mySendStrm;
cppvmReceiveStream myRecStrm;
cppvmForwardStream myFStream; ...
myFStream.set(myRecStrm, mySendStrm);
```

Discussion

the send/receive streams must be initialized

11.8 cppvmNewSendStackData

The template class cppvmNewSendStackData<T> should only be used for PvmDataInPlace sending of external classes (see section 9.1). The class allocates memory (array of the template type) and frees the memory after a send command, i.e. when calling the method cppvmWriteMboxInfo for an cppvmObject object or when using a cppvmSendStream. The PvmDataInPlace does not copy the send data into a buffer. It only reminds the data pointer and sizes. When sending the data is copied directly out of the user's memory. When using temporal variables in a cppvmTransfer method, the data is not valid any more at the time the data is sent. To avoid this problem, the class cppvmNewSendStackData<T> can be used to allocate memory or to manage memory allocated by the user. After sending, the allocated memory is freed by CPPvm. It must not be freed by the user. The allocated memory can be accessed by the class member data. The variable data is a pointer to the allocated memory of the template type. Only arrays of the template type can be allocated. If you hand over user allocated memory, it must be an array pointer. Here is an example how to use cppvmNewSendStackData<T>:

```
...
// allocate integer array
cppvmNewSendStackData<int> arr(2);
// the integer array can be accessed via 'data'
arr.data[0] = 5;
arr.data[1] = 3;

// you can also allocate memory:
float *fArr = new float[100];
// hand over allocated memory
cppvmNewSendStackData<float> nfArr(fArr);
nfArr.data[99] = 1.1;

// it is not allowed to allocate arrays of size 0
// WRONG: This will cause an exception:
cppvmNewSendStackData<cppvmDouble> dArr(0);

// do not allocate single variables:
// WRONG: This will cause problems while
// deleting the memory after the send
// operation (delete [])
short *s = new short;
cppvmNewSendStackData<short> sArr(s);

// do not free memory managed by
// cppvmNewSendStackData:
// WRONG: The memory is freed two times:
```

```
  // now and after sending:
  long *l = new long[20];
  cppvmNewSendStackData<long> lArr(l);
  delete [] l;
...
```

Table 11.212: cppvmNewSendStackData: constructor

constructor			cppvmNewSendStackData
constructor			
Synopsis			
`cppvmNewSendStackData<T>(unsigned int num);`			
Parameter			
Parameter	Type	Default	Description
`num`	in		number of allocated elements (must be > 0)
Exceptions			
cppvmExceptionBadParameter	array size (num) is 0		
Example			
`cppvmNewSendStackData<int> myIntArr(2);`			
Discussion			
The constructor allocates `num` elements of type `T`. The memory is deleted after sending data.			

Table 11.213: cppvmNewSendStackData: constructor

constructor			cppvmNewSendStackData
constructor			
Synopsis			
`cppvmNewSendStackData<T>(T* pointer);`			
Parameter			
Parameter	Type	Default	Description
`pointer`	in		pointer to externally allocated array with elements of type `T`
Exceptions			
cppvmExceptionBadParameter	parameter 'pointer' is NULL		
Example			
`float *fa = new float[100];` `cppvmNewSendStackData<float> nfa(fa);`			
Discussion			
Do not delete the memory externally. The memory is deleted by CPPVM after sending data.			

Table 11.214: cppvmNewSendStackData: `data`

`data`	cppvmNewSendStackData
Access the allocated array by using this public class member.	
Synopsis	
`T* data`	
Example	

```
// allocate double array of size 3
cppvmNewSendStackData<double> arr(3);
// the double values can be accessed via 'data'
arr.data[0] = 1.1;
arr.data[1] = 2.2;
arr.data[2] = 3.3;
```

Discussion
The class variable `data` of type T* is declared as public member of cppvmNewSendStackData. Access to the allocated memory by using this variable. Be aware that the memory is deleted after a send operation.

11.9 cppvmSemaphore

Semaphores are a concept to synchronize parallel task. The basic idea of semaphores is to have a shared counter for all parallel task. The counter can be decreased with the `enter` or `enterNB` method and increased with the `leave` operation. The operations are primitive, i.e. can only be performed by one process at one time and can not be interrupted.

When the counter has the value 0, a `enter` operation will block the calling process. The process can only be released with a `leave` call of another process. The operations can be described as following:

enter

```
void enter(){
  if (number==0) {
    blockCallingProcess();
  }
  else{
    number = number - 1;
  }
}
```

enterNB

```
bool enterNB(){
  if (number==0) {
    return false;
  }
  else{
    number = number - 1;
    return true;
  }
}
```

leave

```
void leave(){
  if(thereIsABlockedProcess){
    wakeup(oldestBlockedProcess);
  }
  else{
    number = number + 1;
  }
}
```

Semaphores need a initial number of allowed enter operations before processes are blocked. To initialize this value, use the `setNumber` method of cppvmSemaphore. Since the initial value is a important issue for semaphores, it is only allowed to set this value once. That means that only one process in the entire virtual machine can use the `setNumber` method for a specific semaphore, and it can call this method only one time.

Use the `resetNumber` method to set the value again. If there are waiting processes, `resetNumber` will not wake them up. They are blocked forever. To avoid this, you can use the methods `getNumber`, `enter`, `enterNB` and `leave` to modify the counter number.

Table 11.215: cppvmSemaphore: `constructor`

`constructor`	cppvmSemaphore
default constructor	
	continued on next page...

...continued from previous page

constructor	cppvmSemaphore

Synopsis

`cppvmSemaphore();`

Example

`cppvmSemaphore mySemaphore;`

Discussion

Use the default constructor only if the semaphore is initialized later on with the `setName` method.

Table 11.216: cppvmSemaphore: `constructor`

constructor	cppvmSemaphore
constructor	

Synopsis

`cppvmSemaphore(char *semaName, char *hostName);`

Parameter

Parameter	Type	Default	Description
`semaName`	in		semaphore key name
`hostName`	in	NULL	host of the semaphore server

Exceptions

cppvmExceptionBadParameter	semaName and hostName are NULL pointers
cppvmExceptionBadResult	no such host

Example

`cppvmSemaphore sema("writeAccess");`

Discussion

continued on next page...

...continued from previous page	
`constructor`	cppvmSemaphore

The semaphores of the different processes are matched by their name keys. By default (`hostName==NULL`) the host of the semaphore server is choosen by CPPvm. The server can be started on a specified host by setting the variable `hostname`:

```
cppvmSemaphore sema("readAccess", "sun1");
```

This version creates the variables `readAccess` and starts the semaphore server on host `sun1`. To start the semaphore server without creating a semaphore variable call the method `startServer` or type:

```
cppvmSemaphore sema(NULL, "lin1");
```

There is another important thing to know: the first semaphore variable in the parallel virtual machine defines the location of the semaphore server. Imagine there are two processes 'A' and 'B', both defining semaphore variables. 'A' wants to start the semaphore server on "sun1", 'B' on "lin2". If the semaphore variable in 'A' is created faster than in 'B', the semaphore server is started on "sun1". In this case, the call for "lin2" in 'B' has no effect. If you want to assign the semaphore server to a specific host, ensure that this call is the first in the virtual machine.

Table 11.217: cppvmSemaphore: `enter`

`enter`	cppvmSemaphore
Semaphore enter. Decrease the semaphore variable. Block the process if the semaphore variable is 0.	

Synopsis

void enter();

Exceptions

cppvmExceptionBadResult	got bad answer from semaphore server. Probably an old server is still alive from former runs or protocol error.
cppvmExceptionForbiddenCall	semaphore key name was not set. Must be set before using it.

Example

```
// create semaphore
cppvmSemaphore sema("semaName");
// set number of allowed entries
sema.setNumber(8);
// enter semaphore
sema.enter();
```

Discussion

The process is blocked if the semaphore variable has the value 0. To continue the process, call `leave` in another process.

Table 11.218: cppvmSemaphore: enterNB

enterNB			cppvmSemaphore
semaphore enter. Decrease the semaphore variable if possible. Otherwise return false. The process is not blocked.			
Synopsis			
bool enterNB();			
Parameter			
Parameter	Type	Default	Description
`bool`	return		enter result: `true:` enter operation successful message `false:` enter operation aborted
Exceptions			
cppvmExceptionForbiddenCall	semaphore key name was not set. Must be set before using it.		
Example			

```
// create semaphore
cppvmSemaphore sema("semaName");
// set number of allowed entries
sema.setNumber(8);
// try to enter semaphore
if(sema.enterNB()==true){;
...
```

Discussion
If the semaphore variable has the value 0, no enter is performed and the result `false` is returned. Otherwise the enter operation was successful.

Table 11.219: cppvmSemaphore: getNumber

getNumber			cppvmSemaphore
get the current number of entries			
Synopsis			
unsigned int getNumber();			
Parameter			
Parameter	Type	Default	Description
unsigned int	return		number of allowed entries
Exceptions			
cppvmExceptionForbiddenCall	semaphore key name was not set. Must be set before using it.		
Example			

continued on next page...

...continued from previous page

getNumber	cppvmSemaphore

```
// create semaphore
cppvmSemaphore sema("semaName");
// set number of allowed entries
sema.setNumber(8);
...
cout << sema.getNumber();
```

Table 11.220: cppvmSemaphore: killServer

killServer	cppvmSemaphore
kill the semaphore server cppvms.	

Synopsis

void killServer();

Exceptions

cppvmExceptionBadResult	could not get task informations of the virtual machine. cannot kill server process.

Example

```
// create semaphore
cppvmSemaphore sema("semaName");
// set number of allowed entries
sema.setNumber(8);
... // kill semaphore server
sema.killServer();
```

Discussion

The semaphore server is normally only killed when halting the parallel virtual machine. To kill the server `cppvms` manually use `killServer`. Ensure not to access to semaphores after this call (this includes all processes currently running in the virtual machine)! Processes blocked by semaphores at this time will be blocked forever.

Table 11.221: cppvmSemaphore: leave

leave	cppvmSemaphore
semaphore leave. Increase the semaphore variable. If there are blocked processes, continue one and do not increase semaphore variable.	

Synopsis

void leave();

Exceptions

cppvmExceptionBadResult	got bad answer from semaphore server. Probably an old server is still alive from former runs or protocol error.
cppvmExceptionForbiddenCall	semaphore key name was not set. Must be set before using it.

continued on next page...

...continued from previous page	
`leave`	cppvmSemaphore

Example

```
// create semaphore
cppvmSemaphore sema("semaName");
// leave semaphore
sema.leave();
```

Table 11.222: cppvmSemaphore: `operator==`

`operator==`	cppvmSemaphore
test if objects are equal	

Synopsis

```
bool operator== (const cppvmSemaphore& obj)
```

Parameter

Parameter	Type	Default	Description
obj	in		object to compare with
bool	return		compare result

Example

```
cppvmSemaphore s1("sema1");
cppvmSemaphore s2("sema2");
...
if(s1==s2){
...
```

Table 11.223: cppvmSemaphore: `operator!=`

`operator!=`	cppvmSemaphore
test if objects are not equal	

Synopsis

```
bool operator!= (const cppvmSemaphore& obj)
```

Parameter

Parameter	Type	Default	Description
`obj`	in		object to compare with
`bool`	return		compare result

Example

```
cppvmSemaphore s1("sema1");
cppvmSemaphore s2("sema2");
...
if(s1!=s2){
...
```

Table 11.224: cppvmSemaphore: `operator<<`

`operator<<`			cppvmSemaphore
write object on ostream			
Synopsis			
`ostream& operator<<(ostream& strm, cppvmSemaphore& obj);`			
Parameter			
Parameter	Type	Default	Description
`obj`	in		write this object on ostream
`strm`	in		ostream
`ostream`	return		ostream after writing the object
Example			

```
cppvmSemaphore s("semaName");
cout << s;
```

Table 11.225: cppvmSemaphore: `resetNumber`

`resetNumber`			cppvmSemaphore
reset the initial number of allowed entries			
Synopsis			
void resetNumber(unsigned int number);			
Parameter			
Parameter	Type	Default	Description
`number`	in		new number of allowed entries
Exceptions			
cppvmExceptionBadResult	got bad answer from semaphore server. Probably an old server is still alive from former runs or protocol error.		
cppvmExceptionForbiddenCall	semaphore key name was not set. Must be set before using it.		
Example			

```
// create semaphore
cppvmSemaphore sema("semaName");
sema.setNumber(8);
...
sema.resetNumber(4);
```

Discussion

It is only possible to initialize the value of allowed entries once (`setNumber`). The reason is to avoid unintentional modifications of an already used semaphore. The use of `resetNumber` tells the system that you know exactly what your are doing. `resetNumber` deletes all entries for the semaphore. If the semaphore has blocked some processes at that time, the processes are blocked forever.

Table 11.226: cppvmSemaphore: setName

setName			cppvmSemaphore
Initialize the key name for a semaphore.			
Synopsis			
void setName(char *semaName);			
Parameter			
Parameter	Type	Default	Description
`semaName`	in		new semaphore name
Exceptions			
cppvmExceptionBadParameter	semaName is a NULL pointer.		
Example			

```
// create semaphore
cppvmSemaphore sema();
// set name
sema.setName("mySemaphore");
```

Discussion

The semaphore name must be set if the semaphore was created with the default constructor.

Table 11.227: cppvmSemaphore: setNumber

setNumber			cppvmSemaphore
Initialize the number of allowed entries for a semaphore. The method can only be called once for a semaphore in the entire virtual machine.			
Synopsis			
void setNumber(unsigned int number);			
Parameter			
Parameter	Type	Default	Description
`number`	in		initial value of allowed entries
Exceptions			
cppvmExceptionForbiddenCall	the number can only set once from all processes. Maybe another process has called setNumber() for this semaphore. Use resetNumber() to modify the number.		
cppvmExceptionForbiddenCall	semaphore key name was not set. Must be set before using it.		
Example			

```
// create semaphore
cppvmSemaphore sema("semaName");
// set number of allowed entries
sema.setNumber(8);
```

Discussion

continued on next page...

. . . continued from previous page	
`setNumber`	cppvmSemaphore
It is only possible to initialize the value of allowed entries once (`setNumber`). The reason is to avoid unintentional modifications of an already used semaphore. The use of `resetNumber` tells the system that you know exactly what your are doing. `resetNumber` deletes all entries for the semaphore. If the semaphore has blocked some processes at that time, the processes are blocked forever.	

Table 11.228: cppvmSemaphore: `startServer`

`startServer`			cppvmSemaphore
start the semaphore server on a specified host			
Synopsis			
`void startServer(char *hostName);`			
Parameter			
Parameter	Type	Default	Description
`hostName`	in		host of the semaphore server
Exceptions			
cppvmExceptionBadResult	no such host		
Example			
`cppvmSemaphore dummy; dummy.startServer("win2");`			
Discussion			
If `hostName==NULL` the host of the semaphore server is choosen by CPPVM. The server can be started on a specified host by setting the variable `hostname` as in the example above. There is another important thing to know: the first semaphore variable in the parallel virtual machine defines the location of the semaphore server. Imagine there are two processes 'A' and 'B', both defining semaphore variables. 'A' wants to start the semaphore server on "sun1", 'B' on "lin2". If the semaphore variable in 'A' is created faster than in 'B', the semaphore server is started on "sun1". In this case, the call for "lin2" in 'B' has no effect. If you want to assign the semaphore server to a specific host, ensure that this call is the first in the virtual machine.			

Table 11.229: cppvmSemaphore: `tryToSetNumber`

`tryToSetNumber`	cppvmSemaphore
Initialize the number of allowed entries for a semaphore. In contrast to the method `setNumber` this method can be called more than once for a semaphore in the entire virtual machine. If the number of a semaphore is already set, nothing happens. Otherwise the number is set.	
Synopsis	
bool tryToSetNumber(unsigned int number);	
	continued on next page. . .

... continued from previous page

`tryToSetNumber`	cppvmSemaphore

Parameter

Parameter	Type	Default	Description
`number`	in		initial value of allowed entries
`bool`	return		set was successful `true` or not `false`

Exceptions

cppvmExceptionForbiddenCall	semaphore key name was not set. Must be set before using it.

Example

```
// create semaphore
cppvmSemaphore sema("semaName");
// try to set the number of allowed entries
bool result;
result = sema.tryToSetNumber(8);
```

Discussion

It is only possible to initialize the value of allowed entries once (see `setNumber`). By using `tryToSeNumber` you can avoid an exception if the number is already set for a semaphore in the virtual machine. The result indicates the success of the set operation. To force a new number of an existing semaphore use the method `resetNumber`. `resetNumber` deletes all entries for the semaphore. If the semaphore has blocked some processes at that time, the processes are blocked forever.

11.10 cppvmMutex

The `cppvmMutex` template functions[1] are wrapper for critical sections (see section 7.2). When is critical section is wrapped into a `cppvmMutex` call, only one process can enter this section. If a second process tries to enter `cppvmMutex` at the same time, it is blocked until the first process leaves the function.

Internally semaphores are used to synchronize the parallel processes. If no semaphore name is given, a default name (define `CPPVM_STANDARD_MUTEX`) is used. Otherwise a semaphore with the given name is created and set to value 1. Only mutex calls with the same semaphore name are blocked against each other.

`cppvmMutex` can wrap functions without parameter or an object of a class which has defined the `operator()`:

```
// typical function declaration for critical section:
void criticalSection() { ... do something ... }
// class declaration with operator():
class criticalClass {
...
void operator() { ... do something ... }
};

...

// Now these critical sections can be wrapped
// by cppvmMutex:

// call wrapped function
cppvmMutex(criticalSection);
// wrapped object; call 'operator()'
criticalClass obj;
cppvmMutex(obj);
```

Table 11.230: cppvmObject: cppvmMutex

cppvmMutex	**function**
wrapper function for mutual exclusion calls	
	continued on next page...

[1] Since the `cppvmMutex` functions use templates, they are only available if the compiler flag `-DCPPvmDoNotUseTemplates` in not set (see subsection 2.2.5.3). Some old compilers (e.g. gcc 2.8.1) cannot handle this type of templates. In this case `cppvmMutex` cannot be used.

...continued from previous page	
`cppvmMutex`	**function**

Synopsis
`template <class T>` `void cppvmMutex(T& callElement, char *semaphoreName);`

Parameter	Type	Default	Description
`callElement`	in		function or object to be called
`semaphoreName`	in	CPPVM_STANDARD_MUTEX	semaphore name

Exceptions	
cppvmExceptionBadParameter	semaName and hostName are NULL pointers
cppvmExceptionBadResult	got bad answer from semaphore server. Probably an old server is still alive from former runs or protocol error.

Discussion
Wrapper for mutual exclusion calls of `callElement` using a semaphore with name `semaphoreName`. The parameter `callElement` can be a function without parameter or an object of a class which has defined the `operator()`.

Table 11.231: cppvmObject: `cppvmMutex`

`cppvmMutex`	**function**
wrapper function for mutual exclusion calls with return values	

Synopsis
`template <class T, class R>` `void cppvmMutex(T& callElement, R& returnVal, char *semaphoreName);`

Parameter	Type	Default	Description
`callElement`	in		function or object to be called
`returnVal`	inout		return value of `callElement`
`semaphoreName`	in		semaphore name

Exceptions	
cppvmExceptionBadParameter	semaName and hostName are NULL pointers
cppvmExceptionBadResult	got bad answer from semaphore server. Probably an old server is still alive from former runs or protocol error.

Discussion
Wrapper for mutual exclusion calls of `callElement` using a semaphore with name `semaphoreName`. The parameter `callElement` can be a function without parameter or an object of a class which has defined the `operator()`. Both (function and `operator()`) must have a return value. This value is handed over in `returnVal`.

11.11 cppvmBarrier

Barriers are a concept to synchronize parallel tasks. A barrier can be used for a set of processes. The barrier blocks each process until all participating processes have reached the barrier.

Barriers are represented by objects of the class cppvmBarrier. A barrier has two parameters: a group name and a count number. The group name specifies the participating processes. When creating a cppvmBarrier object the process joins the specified group. This is the same mechanism like joining a group for group broadcasts (see section 4.4). The count number specifies the number of processes participating at the barrier. This number must be the same as the number of participating processes and it must be set in every process. If different count numbers are used in the processes, the processes can be blocked forever. Here is an example how to create and use a barrier:

```
// connect to pvm
cppvmConnection pvmConn;

// generate barrier
cppvmBarrier syncPoint("syncGroup", 20);

// wait here until all
// participating processes
// have reached the barrier
syncPoint.barrier();
...
```

In this example the group name is `syncGroup` and 20 processes must call the barrier until the processes can continue.

Table 11.232: cppvmBarrier: `constructor`

`constructor`	cppvmBarrier
default constructor	
Synopsis	
`cppvmBarrier();`	
Example	
`cppvmBarrier myBarrier;`	
Discussion	
Use the default constructor only if the barrier is initialized later on with the `set` method.	

Table 11.233: cppvmBarrier: `constructor`

`constructor`			cppvmBarrier
constructor			
Synopsis			
`cppvmBarrier(char *groupName, unsigned int count);`			
Parameter			
Parameter	Type	Default	Description
`groupName`	in		barrier group name
`count`	in		number of group members participating at the barrier
Exceptions			
cppvmExceptionBadParameter	groupName is a NULL pointer		
cppvmExceptionBadResult	pvmd was not started or has crashed		
Example			
`cppvmBarrier syncPoint("syncCalculations", 8);`			
Discussion			
Join the given group and set the barrier count. The count number must be the same as the number of participating processes and it must be set to this value in every process.			

Table 11.234: cppvmBarrier: `copy constructor`

`copy constructor`			cppvmBarrier
copy constructor			
Synopsis			
`cppvmBarrier(const cppvmBarrier& obj);`			
Parameter			
Parameter	Type	Default	Description
`obj`	in		object to copy
Example			
`cppvmBarrier syncPoint = previousDefinedSyncPoint;`			

Table 11.235: cppvmBarrier: `barrier`

`barrier`	cppvmBarrier
call the barrier	
Synopsis	
void barrier();	
Exceptions	
continued on next page...	

. . . continued from previous page	
`barrier`	cppvmBarrier
cppvmExceptionBadResult	pvm-daemon was not started or has crashed. count < 1 giving a non-existent group name calling process is not in specified group
cppvmExceptionForbiddenCall	Group name is not set. Probably you called the 'leaveGroup' method. The group name must be set with method 'setGroup' before calling 'barrier' again.
Example	

`// create barrier`
`cppvmBarrier syncPoint("mySyncGroup",10);`
`// call barrier`
`syncPoint.barrier();`	

Discussion	
The process is blocked until all participating processes have called their `barrier` method, too.	

Table 11.236: cppvmBarrier: `leaveGroup`

`leaveGroup`	cppvmBarrier
unenrolls the calling process from the current barrier group	
Synopsis	
void leaveGroup();	
Exceptions	
cppvmExceptionBadResult	pvmd was not started or has crashed giving a non existent group name asking to leave a group you are not a member of
cppvmExceptionForbiddenCall	Group name is not set. Probably you called the 'leaveGroup' method. The group name must be set with method 'setGroup' before calling 'barrier' again.
Example	

`// create barrier`
`cppvmBarrier syncPoint("mySyncGroup",10);`
`// leave current barrier group`
`syncPoint.leaveGroup();`	

Discussion	
Use this method with care. Be aware that you can not call the method `barrier` again once the process has left the barrier group. Use the method `set` or `setGroup` to join a group again.	

Table 11.237: cppvmBarrier: operator=

operator=			cppvmBarrier
assign object			
Synopsis			
`cppvmBarrier& operator= (const cppvmBarrier& obj)`			
Parameter			
Parameter	Type	Default	Description
`obj`	in		object to assign
`cppvmBarrier`	return		this object
Example			

```
cppvmBarrier b1("syncGroup1", 20);
cppvmBarrier b2("syncGroup2", 30);
...
b1 = b2;
```

Table 11.238: cppvmBarrier: operator==

operator==			cppvmBarrier
test if objects are equal			
Synopsis			
`bool operator== (const cppvmBarrier& obj)`			
Parameter			
Parameter	Type	Default	Description
obj	in		object to compare with
bool	return		compare result
Example			

```
cppvmBarrier b1("syncGroup1");
cppvmBarrier b2("syncGroup2");
...
if(b1==b2){
...
```

Table 11.239: cppvmBarrier: operator!=

operator!=			cppvmBarrier
test if objects are not equal			
Synopsis			
`bool operator!= (const cppvmBarrier& obj)`			
Parameter			
Parameter	Type	Default	Description
`obj`	in		object to compare with

continued on next page...

. . . continued from previous page			
`operator!=`			cppvmBarrier
`bool`	return		compare result
Example			

```
cppvmBarrier b1("syncGroup1");
cppvmBarrier b2("syncGroup2");
...
if(b1!=b2){
...
```

Table 11.240: cppvmBarrier: `operator<<`

`operator<<`			cppvmBarrier
write object on ostream			
Synopsis			
`ostream& operator<<(ostream& strm, cppvmBarrier& obj);`			
Parameter			
Parameter	Type	Default	Description
`obj`	in		write this object on ostream
`strm`	in		ostream
`ostream`	return		ostream after writing the object
Example			

```
cppvmBarrier b("syncGroup");
cout << b;
```

Table 11.241: cppvmBarrier: `set`

`set`			cppvmBarrier
set the barrier group name and count number. Join the given group.			
Synopsis			
void set(char* groupName, unsigned int count, bool leaveFormerGroup);			
Parameter			
Parameter	Type	Default	Description
`groupName`	in		barrier group name
`count`	in		number of group members participating at the barrier
`leaveFormerGroup`	in	false	leave the former barrier group
Exceptions			
cppvmExceptionBadParameter	groupName is a NULL pointer		
cppvmExceptionBadResult	pvmd was not started or has crashed		
	giving a non existent group name		
	asking to leave a group you are not a member of		
			continued on next page. . .

. . . continued from previous page

set	cppvmBarrier
cppvmExceptionForbiddenCall	Group name is not set. Probably you called the 'leaveGroup' method. The group name must be set with method 'setGroup' before calling 'barrier' again.

Example

```
// create barrier
cppvmBarrier syncPoint("mySyncGroup",10);
// set new group and count
syncPoint.set("newSyncGroup", 20);
```

Discussion

If `leaveFormerGroup` is set to `false`, the process will be still member of the former group (`mySyncGroup` in the example), otherwise the process leaves the former group. Ensure not to have left the group from the outside (e.g. by calling the method `leaveGroup`) when setting `leaveFormerGroup` to `true`.

Table 11.242: cppvmBarrier: `setCount`

`setCount`	cppvmBarrier
set the count number	

Synopsis

void setCount(unsigned int count);

Parameter

Parameter	Type	Default	Description
`count`	in		number of group members participating at the barrier

Example

```
// create barrier
cppvmBarrier syncPoint("mySyncGroup",10);
// set new count
syncPoint.setCount(33);
```

Table 11.243: cppvmBarrier: `setGroup`

`setGroup`	cppvmBarrier
set the barrier group name and join the group	

Synopsis

void setGroup(char* groupName, bool leaveFormerGroup);

Parameter

Parameter	Type	Default	Description
`groupName`	in		barrier group name
`leaveFormerGroup`	in	false	leave the former barrier group

continued on next page. . .

... continued from previous page	
setGroup	cppvmBarrier

Exceptions

cppvmExceptionBadParameter	groupName is a NULL pointer
cppvmExceptionBadResult	pvmd was not started or has crashed giving a non existent group name asking to leave a group you are not a member of
cppvmExceptionForbiddenCall	Group name is not set. Probably you called the 'leaveGroup' method. The group name must be set with method 'setGroup' before calling 'barrier' again.

Example

```
// create barrier
cppvmBarrier syncPoint("mySyncGroup",10);
// set new group
syncPoint.setGroup("newSyncGroup", true);
```

Discussion

If `leaveFormerGroup` is set to `false`, the process will be still member of the former group (`mySyncGroup` in the example), otherwise the process leaves the former group. Ensure not to have left the group from the outside (e.g. by calling the method `leaveGroup`) when setting `leaveFormerGroup` to `true`.

11.12 cppvmObject

The class cppvmObject is the base class for all transferable CPPvm message classes. It contains all basic macros and methods for message passing, distributed and message mailbox objects. User defined CPPvm classes have to be derived from this base class.

Table 11.244: cppvmObject: constructor

constructor			cppvmObject
constructor			
Synopsis			
`cppvmObject(int objectMsgTag);`			
Parameter			
Parameter	Type	Default	Description
`objectMsgTag`	in	CPPvmNoObjectMsgTag	object message tag
Exceptions			
cppvmExceptionOutOfRange	this message tag is reserved for CPPvm classes!		
Example			
`cppvmObject object;`			
Discussion			
The constructor should only be used indirectly by calling a constructor of a derived class. Using the constructor without parameter deactivates the object message tag, otherwise the parameter is used as object message tag. If the object message tag is active, the method `cppvmMsgTag` returns now the new message tag instead of the class message tag. To deactivate the object message tags use the method `cppvmDeactivateObjectMsgTag`.			

Table 11.245: cppvmObject: cppvmClassMsgTag

cppvmClassMsgTag	cppvmObject
get the message tag for this class (class method)	
Synopsis	
static int cppvmClassMsgTag();	

Table 11.246: cppvmObject: cppvmDeactivateObjectMsgTag

cppvmDeactivateObjectMsgTag	cppvmObject
change the message tag of an object	
Synopsis	
`cppvmDeactivateObjectMsgTag(msgtag);`	
Parameter	
	continued on next page...

... continued from previous page			
`cppvmDeactivateObjectMsgTag`			cppvmObject
Parameter	Type	Default	Description
`msgtag`	in		new message tag for this object

Example

```
cppvmObject obj;
obj.setCPPvmObjectMsgTag(400);
obj.cppvmDeactivateObjectMsgTag();
```

Discussion

The method `cppvmMsgTag` returns now the class message tag again instead of an object message tag. To activate the object message tags use the method `setCPPvmObjectMsgTag`.

Table 11.247: cppvmObject: `cppvmDelMboxInfo`

`cppvmDelMboxInfo`	cppvmObject
delete information from the database with the current `keyName`, `delFlags`	

Synopsis

```
void cppvmDelMboxInfo(int keyIndex);
```

Parameter

Parameter	Type	Default	Description
`keyIndex`	in	0	database key index

Exceptions

cppvmExceptionBadResult	requested key does not exist
cppvmExceptionBadResult	the key is locked by another task and cannot be replaced or deleted
cppvmExceptionBadResult	error deleting from database

Example

```
cppvmDouble d;
d.cppvmSetMbox("mboxDoubleObj");
d.cppvmDelMboxInfo();
```

Table 11.248: cppvmObject: `cppvmDistObj`

`cppvmDistObj`	cppvmObject
register the object as distributed object	

Synopsis

```
void cppvmDistObj(char *keyName, bool persistent,
int sendEncoding, bool useOwnSendBuffer);
```

Parameter

Parameter	Type	Default	Description
`keyName`	in		database key, any null-terminated string; must not contain regular expression chars (*, ?)

continued on next page...

. . . continued from previous page			
`cppvmDistObj`			cppvmObject
`persistent`	in	true	`true:` entry remains in the database when the owner task exits. Entries are removed from the database when PVM is halted or a reset is issued from the console `false:` distributed object is removed from database if the task has ended
`sendEncoding`	in	PvmDataDefault	PVM encoding: `PvmDataDefault:` XDR encoding `PvmDataRaw:` no encoding (only for homogenous architectures) `PvmDataInPlace:` data left in place during packing; not allowed for shared memory architectures
`useOwnSendBuffer`	in		`true:` a new send buffer is created and deleted after sending the message `false:` default send buffer is used

Exceptions	
cppvmExceptionBadParameter	invalid sendEncoding argument
cppvmExceptionBadParameter	keyName is NULL

Example

```
cppvmDouble d;
d.cppvmDistObj("distDoubleObj");
```

Discussion

Special kind of message box object. Set all message mailbox variables for read/write objects from/to the message mailbox. It is not possible that an object is a distributed object and a message mailbox object at one time.

Table 11.249: cppvmObject: `cppvmGetMboxInfo`

`cppvmGetMboxInfo`	cppvmObject
Get information about message mailbox entries	

Synopsis

```
void cppvmGetMboxInfo(char *pattern, int *nclasses, struct pvmmboxinfo
**classes);
```

Parameter			
Parameter	Type	Default	Description
`pattern`	in		GNU regular expression to match on names in mailbox database. Additionally, the singular '*' will match all names
`nclasses`	out		number of classes matching pattern

continued on next page. . .

. . . continued from previous page

cppvmGetMboxInfo			cppvmObject
classes	out		array of pvmmboxinfo mailbox entries matching pattern

Example

```
cppvmDouble d;
int nclasses;
struct pvmmboxinfo *classes;
...
d.cppvmGetMboxInfo("*", &nclasses, &classes);
```

Discussion

`classes` is allocated by this method and freed on the next call of this method or the call of the other method `cppvmGetMboxInfo`

```
struct pvmmboxinfo{
char *mi_name; // class name
int mi_nentries; // # of entries for this class
int *mi_indices; // mbox entry indices
int *mi_owners; // mbox entry owner tids
int *mi_flags; // mbox entry flags
}
```

Table 11.250: cppvmObject: cppvmGetMboxInfo

cppvmGetMboxInfo			cppvmObject
get information about the database entry with the current `keyName`			

Synopsis

```
void cppvmGetMboxInfo(struct pvmmboxinfo **classInfo);
```

Parameter

Parameter	Type	Default	Description
classInfo	out		pvmmboxinfo mailbox entry matching current `keyName`

Example

```
cppvmDouble d;
d.cppvmSetMbox("mboxDoubleObj"); struct pvmmboxinfo classInfo;
...
d.cppvmGetMboxInfo(&classInfo);
cout << "member name: "
<< classInfo.mi_name;
```

Discussion

continued on next page. . .

. . . continued from previous page	
`cppvmGetMboxInfo`	cppvmObject
`classes` is allocated by this method and freed on the next call of this method or the call of the other method `cppvmGetMboxInfo`	

```
struct pvmmboxinfo{
char *mi_name; // class name
int mi_nentries; // # of entries for this class
int *mi_indices; // mbox entry indices
int *mi_owners; // mbox entry owner tids
int *mi_flags; // mbox entry flags
}
```

Table 11.251: cppvmObject: `cppvmGetMboxKeyIndex`

`cppvmGetMboxKeyIndex`			cppvmObject
get the current message mailbox key index			
Synopsis			
`int cppvmGetMboxKeyIndex();`			
Parameter			
Parameter	Type	Default	Description
`int`	return		current database key index

Table 11.252: cppvmObject: `cppvmGetMboxKeyName`

`cppvmGetMboxKeyName`			cppvmObject
get the current message mailbox key name			
Synopsis			
`char *cppvmGetMboxKeyName();`			
Parameter			
Parameter	Type	Default	Description
`char*`	return		current database key name

Table 11.253: cppvmObject: `cppvmIsSame`

`cppvmIsSame`	cppvmObject
the constructor of cppvmObject gives every new object a unique id. This methods test whether two objects are identical or not.	
Synopsis	
`bool cppvmIsSame(const cppvmObject& obj) const;`	
Parameter	
	continued on next page. . .

. . . continued from previous page			
`cppvmIsSame`			cppvmObject
Parameter	Type	Default	Description
`obj`	in		object to compare with
`bool`	return		compare result
Example			
`cppvmInt o1;`			
`cppvmInt* o2=&o1;`			
`if(o2->cppvmIsSame(o1)){ ...`			

Table 11.254: cppvmObject: `cppvmIsTransferReceive`

`cppvmIsTransferReceive`			**function**
indicates whether the current transfer is a receive transfer or not			
Synopsis			
`bool cppvmIsTransferReceive();`			
Parameter			
Parameter	Type	Default	Description
`bool`	return		transfer mode
Example			
`if(cppvmIsTransferReceive()){`			
`...`			
Discussion			
This function can be used in the `cppvmTransfer` method of user written CPPVM classes to transfer the class data.			

Table 11.255: cppvmObject: `cppvmIsTransferSend`

`cppvmIsTransferSend`			**function**
indicates whether the current transfer is a send transfer or not			
Synopsis			
`bool cppvmIsTransferSend();`			
Parameter			
Parameter	Type	Default	Description
`bool`	return		transfer mode
Example			
`if(cppvmIsTransferSend()){`			
`...`			
Discussion			
This function can be used in the `cppvmTransfer` method of user written CPPVM classes to transfer the class data.			

Table 11.256: cppvmObject: `CPPvmMethodsDeclaration`

`CPPvmMethodsDeclaration`			cppvmObject
macro to generate the message tag methods of a class			
Synopsis			
`CPPvmMethodsDeclaration(msgtag)`			
Parameter			
Parameter	Type	Default	Description
`msgtag`	in		message tag for this class

Example

```
class myClass : public cppvmObject{
public:
// pvm message tag declaration
CPPvmMethodsDeclaration(200)
...
```

Discussion

This macro has to be used within the declaration of new user written CPPvm classes. The macro inserts public methods. So use 'protected' or 'private' before writing the next methods if you do not want them to be public. Alternatively you can use `CPPvmMethodsDeclarationVariable` and `CPPvmMethodsDefinitionVariable`.

Table 11.257: cppvmObject: `CPPvmMethodsDeclarationVariable`

`CPPvmMethodsDeclarationVariable`	cppvmObject
macro to generate modifiable message tags for a class.	
Synopsis	
`CPPvmMethodsDeclarationVariable`	

Example

```
class myClass : public cppvmObject{
public:
// pvm message tag declaration
CPPvmMethodsDeclarationVariable
...
```

Discussion

This macro has to be used within the declaration of new user written CPPvm classes. Unlike the macro `CPPvmMethodsDeclaration` this macro allows to modify the message tag during runtime with the method `setCPPvmClassMsgTag` and the object message tag with the method `setCPPvmObjectMsgTag`. By using `CPPvmMethodsDeclarationVariable` in the declaration file `CPPvmMethodsDefinitionVariable` must be used in the definition file. The macro inserts public methods. So use 'protected' or 'private' before writing the next methods if you do not want them to be public.

Table 11.258: cppvmObject: CPPvmMethodsDefinitionVariable

CPPvmMethodsDefinitionVariable			cppvmObject
macro to generate modifiable message tags for a class			
Synopsis			
CPPvmMethodsDefinitionVariable(msgtag)			
Parameter			
Parameter	Type	Default	Description
msgtag	in		message tag for this class

Example

```
// include specification file
#include "myclass.h"

// static variables
CPPvmMethodsDefinitionVariable(myClass, 300)
...
```

Discussion

This macro has to be used within the definition of new user written CPPvm classes. Use the macro in combination with `CPPvmMethodsDeclarationVariable`. This macro allows to modify the class message tag during runtime with the method `setCPPvmClassMsgTag` and the object message tag with the method `setCPPvmObjectMsgTag`.

Table 11.259: cppvmObject: cppvmMsgTag

cppvmMsgTag	cppvmObject
get the message tag for this class (instance method)	
Synopsis	
virtual int cppvmMsgTag();	

Table 11.260: cppvmObject: cppvmReadMboxInfo

cppvmReadMboxInfo			cppvmObject
read (receive + unpack) information from the database current `keyName` and `readFlags`			
Synopsis			
void cppvmReadMboxInfo(int keyIndex);			
Parameter			
Parameter	Type	Default	Description
keyIndex	in	0	database key index
Exceptions			

continued on next page...

...continued from previous page	
cppvmReadMboxInfo	cppvmObject
cppvmExceptionBadResult	the requested <name, index> entry does not exist

Example

```
cppvmDouble d;
d.cppvmSetMbox("mboxDoubleObj");
d.cppvmReadMboxInfo();
```

Table 11.261: cppvmObject: cppvmSetMbox

cppvmSetMbox	cppvmObject
Initialize the mailbox object for read, write and delete	

Synopsis

```
void cppvmSetMbox(char *keyName, int writeFlags, int readFlags, int
delFlags,
int sendEncoding, bool useOwnSendBuffer);
```

Parameter

Parameter	Type	Default	Description
keyName	in		database key, any null-terminated string; must not contain regular expression chars (*, ?)

continued on next page...

...continued from previous page

cppvmSetMbox			cppvmObject
`writeFlags`	in	PvmMboxDefault + PvmMbox-OverWritable	options for write into database, the options can be combined by adding them together (+) **`PvmMboxDefault:`** inserts entry as the only named instance for a given name. This entry may only be modified and deleted by its owner. It is automatically deleted when its owner exits **`PvmMboxPersistent:`** entry remains in the database when the owner task exits. Entries are removed from the database when PVM is halted or a reset is issued from the console **`PvmMboxMultiInstance:`** permits multiple entry instances of the same name. PVM will assign an index key to each instance. **`PvmMboxOverWritable`** permits other tasks to overwrite and delete this database entry
`readFlags`	in	PvmMboxDefault	options for read from the database, the options can be combined by adding them together (+) **`PvmMboxDefault:`** exact match on key <name, index> is returned. **`PvmMboxFirstAvail:`** the first entry in <name> with index greater than or equal to the specified index parameter is returned. PvmMboxFirstAvail with index=0 will produce the same results as using PvmMboxDefault **`PvmMboxReadAndDelete:`** return entry and delete from database. Task must be permitted to do both read and delete, otherwise an error will occur.
`delFlags`	in	0	options for delete entries from the database; no flags presently specified

continued on next page...

. . . continued from previous page			
`cppvmSetMbox`			cppvmObject
`sendEncoding`	in	PvmDataDefault	PVM encoding: `PvmDataDefault`: XDR encoding `PvmDataRaw`: no encoding (only for homogenous architectures) `PvmDataInPlace`: data left in place during packing; not allowed for shared memory architectures
`useOwnSendBuf`	in	false	`true`: a new send buffer is created and deleted after sending the message `false`: default send buffer is used

Exceptions

cppvmExceptionBadParameter	invalid sendEncoding argument
cppvmExceptionBadParameter	keyName is NULL

Example

```
cppvmDouble d;
d.cppvmSetMbox("mboxDoubleObj");
```

Discussion

Message box object. It is not possible that an object is a distributed object and a message mailbox object at the same time.

Table 11.262: cppvmObject: `cppvmSetMboxDel`

`cppvmSetMboxDel`	cppvmObject
Initialize mailbox object to delete it from the database	

Synopsis

```
void cppvmSetMboxDel(char *keyName, int delFlags);
```

Parameter

Parameter	Type	Default	Description
`keyName`	in		database key, any null-terminated string; must not contain regular expression chars (*, ?)
`delFlags`	in	0	options for delete entries from the database; no flags presently specified

Exceptions

cppvmExceptionBadParameter	keyName is NULL

Example

```
cppvmDouble d;
d.cppvmSetMboxDel("mboxDoubleObj");
```

Discussion

Message box object. It is not possible that an object is a distributed object and a message mailbox object at the same time.

Table 11.263: cppvmObject: cppvmSetMboxKeyIndex

cppvmSetMboxKeyIndex			cppvmObject
set the message mailbox key index			
Synopsis			
`void cppvmSetMboxKeyIndex(int keyIndex);`			
Parameter			
Parameter	Type	Default	Description
`keyIndex`	in		database key index

Table 11.264: cppvmObject: cppvmSetMboxKeyName

cppvmSetMboxKeyName			cppvmObject
set the message mailbox key name			
Synopsis			
`void cppvmSetMboxKeyName(char *keyName);`			
Parameter			
Parameter	Type	Default	Description
`keyName`	in		current database key name

Table 11.265: cppvmObject: cppvmSetMboxKey

cppvmSetMboxKey			cppvmObject
set the message mailbox key index and key name			
Synopsis			
`void cppvmSetMboxKey(char* keyName, int keyIndex);`			
Parameter			
Parameter	Type	Default	Description
`keyName`	in		database key name
`keyIndex`	in		database key index

Table 11.266: cppvmObject: cppvmSetMboxRead

cppvmSetMboxRead			cppvmObject
Initialize mailbox object for read access			
Synopsis			
`void cppvmSetMboxRead(char *keyName, int readFlags);`			
Parameter			
Parameter	Type	Default	Description
`keyName`	in		database key, any null-terminated string; must not contain regular expression chars (*, ?)
			continued on next page...

. . . continued from previous page			
`cppvmSetMboxRead`			cppvmObject
`readFlags`	in	PvmMboxDefault	options for read from the database, the options can be combined by adding them together (+) `PvmMboxDefault:` exact match on key <name, index> is returned. `PvmMboxFirstAvail:` the first entry in <name> with index greater than or equal to the specified index parameter is returned. PvmMboxFirstAvail with index=0 will produce the same results as using PvmMboxDefault `PvmMboxReadAndDelete:` return entry and delete from database. Task must be permitted to do both read and delete, otherwise an error will occur.
Exceptions			
cppvmExceptionBadParameter	keyName is NULL		
Example			
`cppvmDouble d;` `d.cppvmSetMboxRead("mboxDoubleObj");`			
Discussion			
Message box object. It is not possible that an object is a distributed object and a message mailbox object at the same time.			

Table 11.267: cppvmObject: `cppvmSetMboxSend`

`cppvmSetMboxSend`			cppvmObject
set message mailbox send encoding and message buffer			
Synopsis			
`void cppvmSetMboxSend(int sendEncoding, bool useOwnSendBuffer);`			
Parameter			
Parameter	Type	Default	Description
`sendEncoding`	in		PVM encoding: `PvmDataDefault:` XDR encoding `PvmDataRaw:` no encoding (only for homogenous architectures) `PvmDataInPlace:` data left in place during packing; not allowed for shared memory architectures
			continued on next page. . .

. . . continued from previous page			
`cppvmSetMboxSend`			cppvmObject
`useOwnSendBuf`	in		`true`: a new send buffer is created and deleted after sending the message `false`: default send buffer is used
Exceptions			
cppvmExceptionBadParameter	invalid sendEncoding argument		
Example			
`cppvmDouble d;` `d.cppvmSetMboxSend(PvmDataRaw, true);`			

Table 11.268: cppvmObject: `cppvmSetMboxWrite`

`cppvmSetMboxWrite`			cppvmObject
Initialize mailbox object for write access			
Synopsis			
`void cppvmSetMboxWrite(char *keyName, int writeFlags, int sendEncoding, bool useOwnSendBuffer);`			
Parameter			
Parameter	Type	Default	Description
`keyName`	in		database key, any null-terminated string; must not contain regular expression chars (*, ?)
			continued on next page. . .

... continued from previous page			
`cppvmSetMboxWrite`			cppvmObject
`writeFlags`	in	PvmMboxDefault + PvmMboxOverWritable	options for write into database, the options can be combined by adding them together (+) `PvmMboxDefault:` inserts entry as the only named instance for a given name. This entry may only be modified and deleted by its owner. It is automatically deleted when its owner exits `PvmMboxPersistent:` entry remains in the database when the owner task exits. Entries are removed from the database when PVM is halted or a reset is issued from the console `PvmMboxMultiInstance:` permits multiple entry instances of the same name. PVM will assign an index key to each instance. `PvmMboxOverWritable` permits other tasks to overwrite and delete this database entry
`sendEncoding`	in	PvmDataDefault	PVM encoding: `PvmDataDefault:` XDR encoding `PvmDataRaw:` no encoding (only for homogenous architectures) `PvmDataInPlace:` data left in place during packing; not allowed for shared memory architectures
`useOwnSendBuf`	in	false	`true:` a new send buffer is created and deleted after sending the message `false:` default send buffer is used

Exceptions	
cppvmExceptionBadParameter	invalid sendEncoding argument
cppvmExceptionBadParameter	keyName is NULL

Example

continued on next page...

. . . continued from previous page	
`cppvmSetMboxWrite`	cppvmObject
`cppvmDouble d;` `d.cppvmSetMboxWrite("mboxDoubleObj");`	
Discussion	
Message box object. It is not possible that an object is a distributed object and a message mailbox object at the same time.	

Table 11.269: cppvmObject: `cppvmTransferBool`

`cppvmTransferBool`			**function**
transfer bool data type(s) to/from send/receive buffer			
Synopsis			
`void cppvmTransferBool(bool *ptr, int nitem, int stride);`			
Parameter			
Parameter	Type	Default	Description
`ptr`	in		data pointer
`nitem`	in	1	number of data
`stride`	in	1	stride to be used when packing items
Exceptions			
cppvmExceptionBadResult	cannot pack data		
Discussion			
This function can be used in the `cppvmTransfer` method of user written CPPVM classes to transfer the class data. See the PVM functions `pvm_pack` and `pvm_pack` for detailed informations.			

Table 11.270: cppvmObject: `cppvmTransferByte`

`cppvmTransferByte`			**function**
transfer byte data type(s) to/from send/receive buffer			
Synopsis			
`void cppvmTransferByte(char *ptr, int nitem, int stride);`			
Parameter			
Parameter	Type	Default	Description
`ptr`	in		data pointer
`nitem`	in	1	number of data
`stride`	in	1	stride to be used when packing items
Exceptions			
cppvmExceptionBadResult	cannot pack data		
Discussion			
continued on next page. . .			

. . . continued from previous page	
`cppvmTransferByte`	**function**
This function can be used in the `cppvmTransfer` method of user written CPPVM classes to transfer the class data. See the PVM functions `pvm_pack` and `pvm_pack` for detailed informations.	

Table 11.271: cppvmObject: `cppvmTransferCplx`

`cppvmTransferCplx`			**function**
transfer complex data type(s) to/from send/receive buffer			
Synopsis			
`void cppvmTransferCplx(float *ptr, int nitem, int stride);`			
Parameter			
Parameter	Type	Default	Description
`ptr`	in		data pointer
`nitem`	in	1	number of data
`stride`	in	1	stride to be used when packing items
Exceptions			
cppvmExceptionBadResult	cannot pack data		
Discussion			
This function can be used in the `cppvmTransfer` method of user written CPPVM classes to transfer the class data. See the PVM functions `pvm_pack` and `pvm_pack` for detailed informations.			

Table 11.272: cppvmObject: `cppvmTransferDcplx`

`cppvmTransferDcplx`			**function**
transfer complex data type(s) to/from send/receive buffer			
Synopsis			
`void cppvmTransferDcplx(double *ptr, int nitem, int stride);`			
Parameter			
Parameter	Type	Default	Description
`ptr`	in		data pointer
`nitem`	in	1	number of data
`stride`	in	1	stride to be used when packing items
Exceptions			
cppvmExceptionBadResult	cannot pack data		
Discussion			
This function can be used in the `cppvmTransfer` method of user written CPPVM classes to transfer the class data. See the PVM functions `pvm_pack` and `pvm_pack` for detailed informations.			

Table 11.273: cppvmObject: cppvmTransferDouble

cppvmTransferDouble			**function**
transfer double data type(s) to/from send/receive buffer			
Synopsis			
`void cppvmTransferDouble(double *ptr, int nitem, int stride);`			
Parameter			
Parameter	Type	Default	Description
`ptr`	in		data pointer
`nitem`	in	1	number of data
`stride`	in	1	stride to be used when packing items
Exceptions			
cppvmExceptionBadResult	cannot pack data		
Discussion			
This function can be used in the `cppvmTransfer` method of user written CPPvm classes to transfer the class data. See the PVM functions `pvm_pack` and `pvm_pack` for detailed informations.			

Table 11.274: cppvmObject: cppvmTransferFloat

cppvmTransferFloat			**function**
transfer float data type(s) to/from send/receive buffer			
Synopsis			
`void cppvmTransferFloat(float *ptr, int nitem, int stride);`			
Parameter			
Parameter	Type	Default	Description
`ptr`	in		data pointer
`nitem`	in	1	number of data
`stride`	in	1	stride to be used when packing items
Exceptions			
cppvmExceptionBadResult	cannot pack data		
Discussion			
This function can be used in the `cppvmTransfer` method of user written CPPvm classes to transfer the class data. See the PVM functions `pvm_pack` and `pvm_pack` for detailed informations.			

Table 11.275: cppvmObject: cppvmTransferFReceive

cppvmTransferFReceive	**function**
unpack scanf-like data type(s) from receive buffer	
Synopsis	
	continued on next page...

. . . continued from previous page

`cppvmTransferFReceive`			**function**
`void cppvmTransferFReceive(char *ptr, ...);`			
Parameter			
Parameter	Type	Default	Description
`ptr`	in		pointer to printf-like expression
Exceptions			
cppvmExceptionBadResult	cannot unpack data		
Example			

```
void myclass:cppvmTransfer()
{
  if(cppvmIsTransferSend()==true){
   cppvmTransferFSend("%d %f", integerValue, floatValue);
  }
  else{
   cppvmTransferFReceive("%d %f", &integerValue, &floatValue);
  }
}
```

Discussion

Printf- and scanf-like arguments are different: in the first case variables are given as parameters, in the second case pointer to the variables are used. This prevents a symmetric `cppvmTransferF` function. So one must differ between sending (`cppvmTransferFSend`) and receiving `cppvmTransferFReceive` data.

Table 11.276: cppvmObject: `cppvmTransferFSend`

`cppvmTransferFSend`			**function**
pack printf-like data type(s) into send buffer			
Synopsis			
`void cppvmTransferFSend(char *ptr, ...);`			
Parameter			
Parameter	Type	Default	Description
`ptr`	in		pointer to printf-like expression
Exceptions			
cppvmExceptionBadResult	cannot pack data		
Example			

continued on next page. . .

. . . continued from previous page

cppvmTransferFSend	**function**

```
void myclass:cppvmTransfer()
{
  if(cppvmIsTransferSend()==true){
   cppvmTransferFSend("%d %f", integerValue, floatValue);
  }
  else{
   cppvmTransferFReceive("%d %f", &integerValue, &floatValue);
  }
}
```

Discussion

Printf- and scanf-like arguments are different: in the first case variables are given as parameters, in the second case pointer to the variables are used. This prevents a symmetric `cppvmTransferF` function. So one must differ between sending (`cppvmTransferFSend`) and receiving `cppvmTransferFReceive` data.

Table 11.277: cppvmObject: `cppvmTransferInt`

cppvmTransferInt	**function**
transfer int data type(s) to/from send/receive buffer	

Synopsis

`void cppvmTransferInt(int *ptr, int nitem, int stride);`

Parameter

Parameter	Type	Default	Description
`ptr`	in		data pointer
`nitem`	in	1	number of data
`stride`	in	1	stride to be used when packing items

Exceptions

cppvmExceptionBadResult	cannot pack data

Discussion

This function can be used in the `cppvmTransfer` method of user written CPPVM classes to transfer the class data. See the PVM functions `pvm_pack` and `pvm_pack` for detailed informations.

Table 11.278: cppvmObject: `cppvmTransferLong`

cppvmTransferLong	**function**
transfer long data type(s) to/from send/receive buffer	

Synopsis

`void cppvmTransferLong(long *ptr, int nitem, int stride);`

Parameter

continued on next page. . .

. . . continued from previous page			
`cppvmTransferLong`			**function**
Parameter	Type	Default	Description
`ptr`	in		data pointer
`nitem`	in	1	number of data
`stride`	in	1	stride to be used when packing items
Exceptions			
cppvmExceptionBadResult	cannot pack data		
Discussion			
This function can be used in the `cppvmTransfer` method of user written CPPVM classes to transfer the class data. See the PVM functions `pvm_pack` and `pvm_pack` for detailed informations.			

Table 11.279: cppvmObject: `cppvmTransferShort`

`cppvmTransferShort`			**function**
transfer short data type(s) to/from send/receive buffer			
Synopsis			
`void cppvmTransferShort(short *ptr, int nitem, int stride);`			
Parameter			
Parameter	Type	Default	Description
`ptr`	in		data pointer
`nitem`	in	1	number of data
`stride`	in	1	stride to be used when packing items
Exceptions			
cppvmExceptionBadResult	cannot pack data		
Discussion			
This function can be used in the `cppvmTransfer` method of user written CPPVM classes to transfer the class data. See the PVM functions `pvm_pack` and `pvm_pack` for detailed informations.			

Table 11.280: cppvmObject: `cppvmTransferStl*`

`cppvmTransferStl*`			**function**
transfer of the standard template library classes: bitset, complex, deque, list, map, multimap, multiset, pair, priority_queue, queue, set, slist, stack, string, valarray, and vector. Replace `CLS` by one of these STL classes.			
Synopsis			
`template<class T> void cppvmTransferStlCLS(CLS<T> & data, int stride);`			
Parameter			
Parameter	Type	Default	Description
			continued on next page. . .

... continued from previous page			
`cppvmTransferStl*`			**function**
`data`	in		STL data
`stride`	in		stride to be used when packing items
Exceptions			
cppvmExceptionBadResult	cannot pack data		
Example			

```
...
vector<int> v;
...
cppvmTransferStlVector(v, 1);
```

Discussion

These template functions can be used to transfer aggregated STL objects. See the section 8.2 for more information.

Table 11.281: cppvmObject: `cppvmTransferStl*`

`cppvmTransferStl*`			**function**
transfer arrays of the standard template library classes: bitset, complex, deque, list, map, multimap, multiset, pair, priority_queue, queue, set, slist, stack, string, valarray, and vector. Replace `CLS` by one of these STL classes.			
Synopsis			
`template<class T> void cppvmTransferStlCLS(CLS<T> * data, int, nitem, int stride);`			
Parameter			
Parameter	Type	Default	Description
`data`	in		STL array data pointer
`nitem`	in		number of data
`stride`	in		stride to be used when packing items
Exceptions			
cppvmExceptionBadResult	cannot pack data		
Example			

```
...
vp = new vector<int>[50];
...
cppvmTransferStlVector(vp, 50, 1);
```

Discussion

These template functions can be used to transfer arrays of aggregated STL objects. See the section 8.2 for more information.

Table 11.282: cppvmObject: `cppvmTransferStr`

`cppvmTransferStr`			**function**
transfer string data type(s) to/from send/receive buffer			
Synopsis			
`void cppvmTransferStr(char *ptr);`			
Parameter			
Parameter	Type	Default	Description
`ptr`	in		data pointer
Exceptions			
cppvmExceptionBadResult	cannot pack data		
Discussion			
This function can be used in the `cppvmTransfer` method of user written CPPVM classes to transfer the class data. See the PVM functions `pvm_pack` and `pvm_pack` for detailed informations.			

Table 11.283: cppvmObject: `cppvmTransferTemplate`

`cppvmTransferTemplate`			**function**
transfer a single template data type to/from send/receive buffer			
Synopsis			
`template<class T> void cppvmTransferTemplate(T& data, int stride);`			
Parameter			
Parameter	Type	Default	Description
`data`	in		template data
`stride`	in		stride to be used when packing items
Exceptions			
cppvmExceptionBadResult	cannot pack data		
Discussion			
This template function can be used in the `cppvmTransfer` method of user written CPPVM classes to transfer template data. See the section 8.1 for more information.			

Table 11.284: cppvmObject: `cppvmTransferTemplate`

`cppvmTransferTemplate`			**function**
transfer an array of template data types to/from send/receive buffer			
Synopsis			
`template<class T> void cppvmTransferTemplate(T* data, int nitem, int stride);`			
Parameter			
Parameter	Type	Default	Description
			continued on next page…

...continued from previous page			
`cppvmTransferTemplate`			**function**
`data`	in		template data pointer
`nitem`	in		number of data
`stride`	in		stride to be used when packing items
Exceptions			
cppvmExceptionBadResult	cannot pack data		
Discussion			
This template function can be used in the `cppvmTransfer` method of user written CPPVM classes to transfer template data. See the section 8.1 for more information.			

Table 11.285: cppvmObject: `cppvmTransferUint`

`cppvmTransferUint`			**function**
transfer unsigned int data type(s) to/from send/receive buffer			
Synopsis			
`void cppvmTransferUint(unsigned int *ptr, int nitem, int stride);`			
Parameter			
Parameter	Type	Default	Description
`ptr`	in		data pointer
`nitem`	in	1	number of data
`stride`	in	1	stride to be used when packing items
Exceptions			
cppvmExceptionBadResult	cannot pack data		
Discussion			
This function can be used in the `cppvmTransfer` method of user written CPPVM classes to transfer the class data. See the PVM functions `pvm_pack` and `pvm_pack` for detailed informations.			

Table 11.286: cppvmObject: `cppvmTransferUlong`

`cppvmTransferUlong`			**function**
transfer unsigned long data type(s) to/from send/receive buffer			
Synopsis			
`void cppvmTransferUlong(unsigned long *ptr, int nitem, int stride);`			
Parameter			
Parameter	Type	Default	Description
`ptr`	in		data pointer
`nitem`	in	1	number of data
`stride`	in	1	stride to be used when packing items
Exceptions			
continued on next page...			

. . . continued from previous page	
`cppvmTransferUlong`	**function**
cppvmExceptionBadResult	cannot pack data
Discussion	
This function can be used in the `cppvmTransfer` method of user written CPPVM classes to transfer the class data. See the PVM functions `pvm_pack` and `pvm_pack` for detailed informations.	

Table 11.287: cppvmObject: `cppvmTransferUshort`

`cppvmTransferUshort`			**function**
transfer unsigned short data type(s) to/from send/receive buffer			
Synopsis			
`void cppvmTransferUshort(unsigned short *ptr, int nitem, int stride);`			
Parameter			
Parameter	Type	Default	Description
`ptr`	in		data pointer
`nitem`	in	1	number of data
`stride`	in	1	stride to be used when packing items
Exceptions			
cppvmExceptionBadResult	cannot pack data		
Discussion			
This function can be used in the `cppvmTransfer` method of user written CPPVM classes to transfer the class data. See the PVM functions `pvm_pack` and `pvm_pack` for detailed informations.			

Table 11.288: cppvmObject: `cppvmWriteMboxInfo`

`cppvmWriteMboxInfo`	cppvmObject
write (pack+send) information in the database with the current `keyName` and `writeFlags`	
Synopsis	
`void cppvmWriteMboxInfo();`	
Exceptions	
cppvmExceptionBadResult	giving an invalid encoding value
cppvmExceptionBadResult	not enough memory to create the send buffer
cppvmExceptionBadResult	cannot create new send buffer
Example	

```
cppvmDouble d;
d.cppvmSetMbox("mboxDoubleObj");
d.cppvmWriteMboxInfo();
```

Table 11.289: cppvmObject: `getCPPvmObjId`

`getCPPvmObjId`			cppvmObject
the constructor of cppvmObject gives every new object a unique id. This methods returns this id.			
Synopsis			
`int getCPPvmObjId();`			
Parameter			
Parameter	Type	Default	Description
`int`	return		object id
Example			
`cppvmInt o1;` `cout << o1.getCPPvmObjId();`			

Table 11.290: cppvmObject: `operator<<`

`operator<<`			**cppvmObject**
out stream operator which calls the method `cppvmPrintOn`			
Synopsis			
`ostream& operator<<(ostream& strm, cppvmObject& obj);`			
Parameter			
Parameter	Type	Default	Description
`strm`	inout		out stream
`obj`	in		object to write on stream

Table 11.291: cppvmObject: `operator>>`

`operator>>`			**cppvmObject**
in stream operator which calls the method `cppvmScanFrom`			
Synopsis			
`istream& operator>>(istream& strm, cppvmObject& obj);`			
Parameter			
Parameter	Type	Default	Description
`strm`	inout		in stream
`obj`	in		object to read from stream

Table 11.292: cppvmObject: `operator<<`

`operator<<`			**cppvmObject**
out stream operator for PVM send			
Synopsis			
continued on next page...			

... continued from previous page			
`operator<<`			**cppvmObject**
`cppvmSendStream& operator<<(cppvmSendStream& sendStrm, cppvmObject& obj);`			
Parameter			
Parameter	Type	Default	Description
`sendStrm`	inout		PVM send stream
`obj`	in		object to write on stream

Table 11.293: cppvmObject: `operator>>`

`operator>>`			**cppvmObject**
in stream operator, for PVM receive			
Synopsis			
`cppvmReceiveStream& operator>>(cppvmReceiveStream& recStrm, cppvmObject& obj);`			
Parameter			
Parameter	Type	Default	Description
`recStrm`	inout		PVM receive stream
`obj`	in		object to read from stream

Table 11.294: cppvmObject: `operator>>`

`operator>>`			**cppvmObject**
in stream operator, for PVM forward			
Synopsis			
`cppvmForwardStream& operator>>(cppvmForwardStream& forwardStrm, cppvmObject& obj);`			
Parameter			
Parameter	Type	Default	Description
`forwardStrm`	inout		PVM forward stream
`obj`	in		object to forward

Table 11.295: cppvmObject: `setCPPvmClassMsgTag`

`setCPPvmClassMsgTag`			cppvmObject
change the message tag of a class			
Synopsis			
`setCPPvmClassMsgTag(msgtag);`			
Parameter			
Parameter	Type	Default	Description
			continued on next page...

... continued from previous page			
`setCPPvmClassMsgTag`			cppvmObject
`msgtag`	in		new message tag for this class
Example			
`myClass::setCPPvmClassMsgTag(400);`			
Discussion			
This methods only exists when the macros `CPPvmMethodsDeclarationVariable` and `CPPvmMethodsDefinitionVariable` are used.			

Table 11.296: cppvmObject: `setCPPvmObjectMsgTag`

`setCPPvmObjectMsgTag`			cppvmObject
change the message tag of an object			
Synopsis			
`setCPPvmObjectMsgTag(msgtag);`			
Parameter			
Parameter	Type	Default	Description
`msgtag`	in		new message tag for this object
Example			
`cppvmObject obj;` `obj.setCPPvmObjectMsgTag(400);`			
Discussion			
The method `cppvmMsgTag` returns now the new message tag instead of the class message tag. To deactivate the object message tags use the method `cppvmDeactivateObjectMsgTag`.			

11.13 cppvmSingleData

There are several standard message classes defined in CPPvm. With these classes most of the message passing problems should be covered. So the user can immediately start the message passing without having to implement new user defined CPPvm classes.

The base class for the single data classes is cppvmSingleData . The single data classes are:

- cppvmBool
- cppvmChar
- cppvmInt
- cppvmLong
- cppvmFloat
- cppvmDouble
- cppvmString

The single data classes are equivalent to the standard C/C++ data types. Since their methods are quite the same, only all methods of cppvmFloat are described. The classes cppvmBool and cppvmString are somehow different to the other single data classes. Therefore they are described separately.

The cppvmSingleData classes are derived from cppvmObject. They also contain additional information, e.g. to allow a blocking send within CPPvm. For this reason, it is not possible to use them together with standard PVM send/receive functions. The next example will fail:

```
Process A:
...
cppvmInt value;
sendStrm << value;
...

Process B:
...
int value;
pvm_recv(-1, -1);
pvm_upkint(&value, 1, 1);
...
```

Of course it is possible to use PVM functions together with CPPvm. But do not use them for the CPPvm message classes.

Table 11.297: cppvmSingleData: readDouble

readDouble			cppvmSingleData
get a single data value as double value; get value from the message mailbox, if the mailbox was initialized for this object			
Synopsis			
double readDouble();			
Parameter			
Parameter	Type	Default	Description
`double`	return		current single data value as double value
Exceptions			
cppvmExceptionBadParameter	cannot convert string into double		
Example			

`cppvmInt myInt(5);`
`cppvmString myString("8.8");`
`cout << "Double values: " << myInt.readDouble()`
`<< " and " << myString.readDouble() << endl;`			

Discussion			
The method `readDouble` is useful for example if you have a collection of cppvmSingleData objects			

Table 11.298: cppvmSingleData: writeDouble

writeDouble			cppvmSingleData
convert a double value into the type of the single data class; send value to message mailbox if the mailbox was initialized for this object			
Synopsis			
void writeDouble(double value);			
Parameter			
Parameter	Type	Default	Description
`double`	in		write this value converted to the class type into the current single data value
Exceptions			
cppvmExceptionBadParameter	cannot convert double into string		
Example			

`cppvmInt myInt(5);`
`cppvmString myString("8.8");`
`myInt.writeValDouble(3.3);`
`myString.writeValDouble(4.4);`			

Discussion			
The method `writeDouble` is useful for example if you have a collection of cppvmSingleData objects			

11.13.1 cppvmFloat

The class cppvmFloat is an example for a single data message class. The methods can be used equivalent for the classes:

- cppvmBool
- cppvmChar
- cppvmInt
- cppvmLong
- cppvmDouble

Table 11.299: cppvmFloat: `constructor`

`constructor`			cppvmFloat
constructor			
Synopsis			
`cppvmFloat(float value);`			
Parameter			
Parameter	Type	Default	Description
`value`	in	0.0	float value
Example			
`cppvmFloat myFloat(5.5);`			

Table 11.300: cppvmFloat: `copy constructor`

`copy constructor`			cppvmFloat
copy constructor			
Synopsis			
`cppvmFloat(const cppvmFloat& obj);`			
Parameter			
Parameter	Type	Default	Description
`obj`	in		object to copy
Example			
`cppvmFloat myFloat = previousDefinedFloat;`			

Table 11.301: cppvmFloat: cppvmClassMsgTag

cppvmClassMsgTag	cppvmFloat
get the message tag for this class (class method)	
Synopsis	
static int cppvmClassMsgTag();	

Table 11.302: cppvmFloat: cppvmMsgTag

cppvmMsgTag	cppvmFloat
get the message tag for this class (instance method)	
Synopsis	
virtual int cppvmMsgTag();	

Table 11.303: cppvmFloat: getPtr()

getPtr()			cppvmFloat
get the pointer to the float value			
Synopsis			
`float* getPtr();`			
Parameter			
Parameter	Type	Default	Description
`float*`	return		pointer to float value
Example			

```
cppvmFloat myFloat(8.8);
float *fPtr;
fPtr = myFloat.getPtr();
(*fPtr) = 1.1;
```

Table 11.304: cppvmFloat: operator+

operator+			cppvmFloat
add the value of another object			
Synopsis			
`float operator+(const cppvmFloat& obj);`			
Parameter			
Parameter	Type	Default	Description
`obj`	in		object to add
`float`	return		result
Example			

continued on next page...

. . . continued from previous page	
`operator+`	cppvmFloat

```
cppvmFloat myFloat1(8.8);
cppvmFloat myFloat2(1.1);
cppvmFloat result;
result = myFloat1 + myFloat2;
```

Table 11.305: cppvmFloat: `operator-`

`operator-`	cppvmFloat
subtract the value of another object	

Synopsis

```
float operator-(const cppvmFloat& obj);
```

Parameter

Parameter	Type	Default	Description
`obj`	in		object to subtract
`float`	return		result

Example

```
cppvmFloat myFloat1(8.8);
cppvmFloat myFloat2(1.1);
cppvmFloat result;
result = myFloat1 - myFloat2;
```

Table 11.306: cppvmFloat: `operator+`

`operator+`	**cppvmFloat**
add the value of this object to a float value	

Synopsis

```
float operator+(float value, cppvmFloat obj);
```

Parameter

Parameter	Type	Default	Description
`value`	in		float value to add
`obj`	in		object to add
`float`	return		result

Example

```
cppvmFloat myFloat(7.7);
cppvmFloat addVal;
float f = 1.1;
addVal = f + myFloat;
```

Table 11.307: cppvmFloat: operator-

operator-			**cppvmFloat**
subtract the value of this object from a float value			
Synopsis			
`float operator-(float value, cppvmFloat obj);`			
Parameter			
Parameter	Type	Default	Description
`value`	in		float value to subtract from
`obj`	in		object to subtract
`float`	return		result
Example			

```
cppvmFloat myFloat(4.4);
cppvmFloat subValue();
float f = 1.1;
subValue = f - myFloat;
```

Table 11.308: cppvmFloat: operator++

operator++			cppvmFloat
increment prefix operator			
Synopsis			
float operator++();			
Parameter			
Parameter	Type	Default	Description
`float`	return		result
Example			

```
cppvmFloat myFloat(5.5);
++myFloat;
```

Table 11.309: cppvmFloat: operator++

operator++			cppvmFloat
increment postfix operator			
Synopsis			
float operator++(int);			
Parameter			
Parameter	Type	Default	Description
`float`	return		result
Example			

continued on next page...

. . . continued from previous page

operator++			cppvmFloat

```
cppvmFloat myFloat(5.5);
myFloat++;
```

Table 11.310: cppvmFloat: operator--

operator--			cppvmFloat
decrement prefix operator			
Synopsis			
`float operator--();`			
Parameter			
Parameter	Type	Default	Description
`float`	return		result
Example			

```
cppvmFloat myFloat(5.5);
--myFloat;
```

Table 11.311: cppvmFloat: operator--

operator--			cppvmFloat
decrement postfix operator			
Synopsis			
`float operator--(int);`			
Parameter			
Parameter	Type	Default	Description
`float`	return		result
Example			

```
cppvmFloat myFloat(5.5);
myFloat--;
```

Table 11.312: cppvmFloat: operator<<

operator<<			**cppvmFloat**
write object on out-stream			
Synopsis			
`ostream& operator<<(ostream& strm, cppvmObject& obj);`			
Parameter			
Parameter	Type	Default	Description
`ostream`	inout		out stream to write object on
`obj`	in		write this object on stream

continued on next page. . .

...continued from previous page

operator<<	**cppvmFloat**

Example
`cppvmFloat myFloat(4.4);`
`cout << myFloat;`

Table 11.313: cppvmFloat: operator>>

operator>>	**cppvmFloat**
read object from in-stream	

Synopsis
`istream& operator>>(istream& strm, cppvmObject& obj);`

Parameter			
Parameter	Type	Default	Description
`istream`	inout		in stream to read object from
`obj`	in		read this object from stream

Example
`cppvmFloat myFloat();`
`cin >> myFloat;`

Table 11.314: cppvmFloat: operator=

operator=	cppvmFloat
assign object	

Synopsis
`cppvmFloat& operator= (const cppvmFloat& obj)`

Parameter			
Parameter	Type	Default	Description
`obj`	in		object to assign
`cppvmFloat`	return		this object

Example
`cppvmFloat f1, f2;`
`...`
`f1 = f2;`

Table 11.315: cppvmFloat: operator==

operator==	cppvmFloat
test if objects are equal	

Synopsis
`bool operator== (const cppvmFloat& obj)`

continued on next page...

... continued from previous page

operator==	cppvmFloat

Parameter

Parameter	Type	Default	Description
obj	in		object to compare with
bool	return		compare result

Example

```
cppvmFloat f1, f2;
...
if(f1==f2){ ...
```

Table 11.316: cppvmFloat: operator!=

operator!=	cppvmFloat
test if objects are not equal	

Synopsis

```
bool operator!= (const cppvmFloat& obj)
```

Parameter

Parameter	Type	Default	Description
obj	in		object to compare with
bool	return		compare result

Example

```
cppvmFloat f1, f2;
...
if(f1!=f2){ ...
```

Table 11.317: cppvmFloat: operator float()

operator float()	cppvmFloat
cast object to float type	

Synopsis

```
operator float();
```

Example

```
cppvmFloat myFloat(8.8);
float f;
f=(float) myFloat;
```

Table 11.318: cppvmFloat: read

read	cppvmFloat
get the float value; get value from the message mailbox if the mailbox was initialized	

continued on next page...

... continued from previous page			
`read`			cppvmFloat
Synopsis			
`float read();`			
Parameter			
Parameter	Type	Default	Description
`float`	return		current float value
Example			

`cppvmFloat myFloat(8.8);`
`float f;`
`f = myFloat.read();`			

Table 11.319: cppvmFloat: `write`

`write`			cppvmFloat
set the float value; send value to the message mailbox if the mailbox was initialized			
Synopsis			
`void write(float value);`			
Parameter			
Parameter	Type	Default	Description
`value`	in		new float value
Example			

`cppvmFloat myFloat(8.8);`
`float f = 5.5;`
`myFloat.write(f);`			

11.13.2 cppvmBool

The class cppvmBool is similar to the described class cppvmFloat. Additionally cppvmBool contains some bool operators. See the section 12.3 for an example.

Table 11.320: cppvmBool: operator!

operator!			cppvmBool
logical not operator			
Synopsis			
`bool operator!();`			
Parameter			
Parameter	Type	Default	Description
`bool`	return		inverted bool value
Example			

```
cppvmBool myBool=false;
cppvmBool result;
result = !myBool;
```

Table 11.321: cppvmBool: operator&&

operator&&			cppvmBool
logical and operator			
Synopsis			
`bool operator&&(cppvmBool obj);`			
Parameter			
Parameter	Type	Default	Description
`obj`	in		object for and operation
`bool`	return		result
Example			

```
cppvmBool myBool1 = true;
cppvmBool myBool2 = false;
cppvmBool result;
result = myBool1 && myBool2;
```

Table 11.322: cppvmBool: operator||

operator||			cppvmBool
logical or operator			
Synopsis			
`bool operator||(cppvmBool obj);`			
Parameter			

continued on next page...

. . . continued from previous page			
`operator||`			cppvmBool
Parameter	Type	Default	Description
`obj`	in		object for or operation
`bool`	return		result

Example

```
cppvmBool myBool1 = true;
cppvmBool myBool2 = false;
cppvmBool result;
result = myBool1 || myBool2;
```

Table 11.323: cppvmBool: `operator bool()`

`operator bool()`	cppvmBool
cast object to bool type	

Synopsis

```
operator bool();
```

Example

```
cppvmBool myBool=true;
bool b;
b=(bool) myBool;
```

Table 11.324: cppvmBool: `getPtr()`

`getPtr()`	cppvmBool
get the pointer to the bool value	

Synopsis

```
bool* getPtr();
```

Parameter

Parameter	Type	Default	Description
`bool*`	return		pointer to bool value

Example

```
cppvmBool myBool=true;
bool *bPtr;
bPtr = myBool.getPtr();
(*bPtr) = false;
```

11.13.3 cppvmString

The class cppvmString is a special kind of a single data class. Besides the normal single data class methods cppvmString has also a lot of methods which make working with strings more comfortable.

Table 11.325: cppvmString: `constructor`

`constructor`	cppvmString
constructor	
Synopsis	
`cppvmString();`	
Example	
`cppvmString myString();`	

Table 11.326: cppvmString: `constructor`

`constructor`			cppvmString
constructor			
Synopsis			
`cppvmString(char * value);`			
Parameter			
Parameter	Type	Default	Description
`value`	in	`""`	initialize string
Exceptions			
cppvmExceptionBadParameter	string is NULL		
Example			
`cppvmString myString("hello");`			

Table 11.327: cppvmString: `constructor`

`constructor`			cppvmString
constructor			
Synopsis			
`cppvmString(unsigned int length, bool cleanString);`			
Parameter			
Parameter	Type	Default	Description
`length`	in		array length
`cleanString`	in	true	clean array after allocation
Example			
`cppvmString myString(100); cppvmArrayDouble myString(50, false);`			

continued on next page...

. . . continued from previous page	
`constructor`	cppvmString
Discussion	
If the parameter `cleanString` is set to `false`, the string characters are undefined!	

Table 11.328: cppvmString: `copy constructor`

`copy constructor`			cppvmString
copy constructor			
Synopsis			
`cppvmString(const cppvmString& obj);`			
Parameter			
Parameter	Type	Default	Description
`obj`	in		object to copy
Example			
`cppvmString myString = previousDefinedString;`			

Table 11.329: cppvmString: `append`

`append`			**cppvmString**
append a char to the string			
Synopsis			
`void append(char c);`			
Parameter			
Parameter	Type	Default	Description
`c`	in		char to append
Example			
`cppvmString myString("hell");` `myString.append('o');`			

Table 11.330: cppvmString: `append`

`append`			**cppvmString**
append a string to this string			
Synopsis			
`void append(char *str);`			
Parameter			
Parameter	Type	Default	Description
`str`	in		string to append
Exceptions			
			continued on next page. . .

...continued from previous page	
`append`	**cppvmString**
cppvmExceptionBadParameter	str is NULL
Example	

```
cppvmString myString("hello");
myString.append(" world");
```

Table 11.331: cppvmString: append

`append`	**cppvmString**
append a cppvmString object to the string	

Synopsis

```
void append(cppvmString& obj);
```

Parameter

Parameter	Type	Default	Description
`obj`	in		cppvmString object

Example

```
cppvmString myString1("hello ");
cppvmString myString2("world!");
myString1.append(myString2);
```

Table 11.332: cppvmString: clean

`clean`	**cppvmString**
clean string (does not free memory)	

Synopsis

```
void clean();
```

Example

```
cppvmString myString("hello");
myString.clean();
```

Table 11.333: cppvmString: cleanElem

`cleanElem`	cppvmString
clean a specified character in string	

Synopsis

void cleanElem(unsigned int index);

Parameter

Parameter	Type	Default	Description
`index`	in		element index

continued on next page...

...continued from previous page

`cleanElem`	cppvmString
Exceptions	
cppvmExceptionOutOfRange	index >= string length
Example	
`cppvmString myString("ball");` `myString.cleanElem(0);`	

Table 11.334: cppvmString: `cleanRange`

`cleanRange`			cppvmString
clean a range in the string			
Synopsis			
void cleanRange(unsigned int fromIndex, unsigned in toIndex);			
Parameter			
Parameter	Type	Default	Description
`fromIndex`	in		starting index
`inIndex`	in		ending index
Exceptions			
cppvmExceptionOutOfRange	'fromIndex' is greater than 'toIndex'		
cppvmExceptionOutOfRange	'fromIndex' is greater than array length		
cppvmExceptionOutOfRange	'toIndex' is greater than array length		
Example			
`cppvmString myString("basketball");` `myString.cleanRange(0, 5);`			
Discussion			
The element with index `toIndex` is also cleaned.			

Table 11.335: cppvmString: `copyWord`

`copyWord`			**cppvmString**
append the word with `index` of this string to the `target` string; words are: every sequence of chars separated by whitespaces			
Synopsis			
`bool copyWord(unsigned int index, cppvmString& target);`			
Parameter			
Parameter	Type	Default	Description
`index`	in		word index
`target`	inout		append word to this string
`bool`	return		copy successful or not (wrong index?)
Example			

continued on next page...

... continued from previous page	
`copyWord`	**cppvmString**

```
cppvmString source("Hi out there!");
cppvmString target("Let me ");
source.copyWord(1, target);
cout << target;
```

Table 11.336: cppvmString: `countChars`

`countChars`			**cppvmString**
count the characters in the current string			
Synopsis			
`int countChars();`			
Parameter			
Parameter	Type	Default	Description
`int`	return		number of characters in string
Example			

```
cppvmString myString("hello");
cout << myString.countChars();
```

Table 11.337: cppvmString: `countWords`

`countWords`			**cppvmString**
count the number of words in the current string; words are: every sequence of chars separated by whitespaces			
Synopsis			
`int countWords();`			
Parameter			
Parameter	Type	Default	Description
`int`	return		number of words in string
Example			

```
cppvmString myString("Hello world! How are you?");
cout << myString.countWords();
```

Table 11.338: cppvmString: `cppvmClassMsgTag`

`cppvmClassMsgTag`	cppvmString
get the message tag for this class (class method)	
Synopsis	
static int cppvmClassMsgTag();	

Table 11.339: cppvmString: cppvmMsgTag

cppvmMsgTag	cppvmString
get the message tag for this class (instance method)	
Synopsis	
virtual int cppvmMsgTag();	

Table 11.340: cppvmString: getCPPvmTransferMode

getCPPvmTransferMode			**cppvmString**
get the current transfer mode			
Synopsis			
`int getCPPvmTransferMode();`			
Parameter			
Parameter	Type	Default	Description
`mode`	return		transfer mode: `CPPvmTransferModeClean:` clean old string, then receive new sting (default) `CPPvmTransferModeAppend:` append the received string to the current string
Example			
`cppvmString myString("12345");` `cout << myString.getCPPvmTransferMode();`			
Discussion			
The initial transfer mode value of an array is `CPPvmTransferModeClean`.			

Table 11.341: cppvmString: getFitSizeToLength

getFitSizeToLength			**cppvmString**
get the current state of the `fitSizeToLength` variable			
Synopsis			
`bool getFitSizeToLength();`			
Parameter			
Parameter	Type	Default	Description
`bool`	return		`true:` whenever a new or delete of a the string is needed, the string size is fitted to the string length `false:` the size can only get bigger, not smaller
Example			

continued on next page...

. . . continued from previous page	
`getFitSizeToLength`	**cppvmString**
`cppvmString myString("Hello");` `cout << myString.getFitSizeToLength();`	

Table 11.342: cppvmString: `getLength`

`getLength`			**cppvmString**
get the current length of the string, i.e. the number of string characters.			
Synopsis			
`unsigned int getLength();`			
Parameter			
Parameter	Type	Default	Description
`unsigned int`	return		current string length
Example			
`cppvmString myString("Hello");` `cout << myString.getLength();`			
Discussion			
There can be more allocated memory than the current string length. To get the current allocated memory size, use the method `getSize()`. To adapt the string length automatically to the string size use the method `setFitSizeToLength(true)`.			

Table 11.343: cppvmString: `getPtr()`

`getPtr()`	cppvmString
get string pointer	
Synopsis	
`char* getPtr();`	
Example	

`cppvmString myString("hi");`
`char* cPtr;`
`cPtr=myString.getPtr();`	

Table 11.344: cppvmString: `getSize`

`getSize`	**cppvmString**
get the current size of the string, i.e. the total length of the allocated memory for the string	
Synopsis	
`unsigned int getSize();`	
Parameter	
	continued on next page. . .

. . . continued from previous page			
`getSize`			**cppvmString**
Parameter	Type	Default	Description
`unsigned int`	return		current total string size
Example			
`cppvmString myString("Hello");` `cout << myString.getSize();`			
Discussion			
There can be more allocated memory than the current string length. To get the current string length, use the method `getLength()`. To adapt the string length automatically to the string size use the method `setFitSizeToLength(true)`.			

Table 11.345: cppvmString: operator[]

`operator[]`			**cppvmString**
access to specified character of the string			
Synopsis			
`char& operator[](unsigned int index);`			
Parameter			
Parameter	Type	Default	Description
`index`	in		character index
`char&`	return		character
Exceptions			
cppvmExceptionOutOfRange	wrong index		
Example			
`cppvmString myString("hello");` *// get value* *cout << myString[3];* *// set value* *myString[1] = 'a';*			

Table 11.346: cppvmString: operator<<

`operator<<`			**cppvmString**
write object on out-stream			
Synopsis			
`ostream& operator<<(ostream& strm, cppvmObject& obj);`			
Parameter			
Parameter	Type	Default	Description
`ostream`	inout		out stream to write object on
`obj`	in		write this object on stream
			continued on next page. . .

... continued from previous page

operator<<	**cppvmString**

Example
`cppvmString myString("hello");`
`cout << myString;`

Table 11.347: cppvmString: operator>>

operator>>	**cppvmString**
read object from in-stream	

Synopsis
`istream& operator>>(istream& strm, cppvmObject& obj);`

Parameter			
Parameter	Type	Default	Description
`istream`	inout		in stream to read object from
`obj`	in		read this object from stream

Example
`cppvmString myString();`
`cin >> myString;`

Table 11.348: cppvmString: operator=

operator=	cppvmString
assign object	

Synopsis
`cppvmString& operator= (const cppvmString& obj)`

Parameter			
Parameter	Type	Default	Description
`obj`	in		object to assign
`cppvmString`	return		this object

Example
`cppvmString s1(), s2();`
`...`
`s1 = s2;`

Table 11.349: cppvmString: operator==

operator==	cppvmString
test if objects are equal	

Synopsis
`bool operator== (const cppvmString& obj)`

continued on next page...

... continued from previous page			
`operator==`			cppvmString
Parameter			
Parameter	Type	Default	Description
obj	in		object to compare with
bool	return		compare result
Example			

```
cppvmString s1, s2;
...
if(s1==s2){ ...
```

Table 11.350: cppvmString: `operator!=`

`operator!=`			cppvmString
test if objects are not equal			
Synopsis			
`bool operator!= (const cppvmString& obj)`			
Parameter			
Parameter	Type	Default	Description
`obj`	in		object to compare with
`bool`	return		compare result
Example			

```
cppvmString s1, s2;
...
if(s1!=s2){ ...
```

Table 11.351: cppvmString: `printFormat`

`printFormat`			**cppvmString**
clean string; transform other data types into strings			
Synopsis			
`void printFormat(const char *format, ...);`			
Parameter			
Parameter	Type	Default	Description
`format`	in		format string
`...`	in		list of values defined in format string
Example			

```
cppvmString myString();
int i = 5;
float f = 8.8;
myString.printFormat("%d %f",i,f)
```

Table 11.352: cppvmString: read

read			cppvmString
Get value from the message mailbox if the mailbox was initialized. Otherwise the local string is returned.			
Synopsis			
`char *read();`			
Parameter			
Parameter	Type	Default	Description
`char *`	return		current string
Example			

`cppvmString myString("hello");`
`char *cPtr;`
`cPtr = myString.read();`			

Table 11.353: cppvmString: resize

resize			**cppvmString**
Resize the array to a new length. Copy old values if possible. Do not clean new elements.			
Synopsis			
`void resize(unsigned in newLen);`			
Parameter			
Parameter	Type	Default	Description
`newLen`	in		new length of the array
Example			
`cppvmString myString("Hello ");` `myString.resize(3);`			
Discussion			

The result of the `resize` method depends on the variable `fitSizeToLength` `fitSizeToLength`:
`fitSizeToLength = true`: allocate new array if `newLen!=size+1`
`fitSizeToLength = false`: allocate new array if `newLen+1>size`
`fitSizeToLength` can be manipulated with the method `setFitSizeToLength`.
If the new size is greater than the current size of the array, the additional elements have undefined values! Use the option `cleanOnlyNewElements` option of the method `resizeClean` to clean new elements.			

Table 11.354: cppvmString: resizeClean

resizeClean	**cppvmString**
resize the array to a new length and clean string	
	continued on next page...

... continued from previous page	
`resizeClean`	**cppvmString**

Synopsis

```
void resizeClean(unsigned in newLen, bool cleanOnlyNewElements);
```

Parameter

Parameter	Type	Default	Description
`newLen`	in		new length of the array
`cleanOnlyNewElements`		false	clean only new elements

Example

```
cppvmString myString("Hello ");
myString.resizeClean(15);
```

Discussion

The result of the `resizeClean` method depends on the variable `fitSizeToLength` `fitSizeToLength`:
`fitSizeToLength = true`: allocate new array if `newLen!=size+1`
`fitSizeToLength = false`: allocate new array if `newLen+1>size`
`fitSizeToLength` can be manipulated with the method `setFitSizeToLength`.
The option `cleanOnlyNewElements` can be used to clean new elements if the new size of the array is greater than the old size. If `cleanOnlyNewElements` is set `true` the old values are copied (if possible). In contrast to the `cppvmArrayData` classes cleaned elements are not visible:

```
cppvmString myString("123");
myString.resizeClean(5, true);
// the string is now "123"
myString.resizeClean(3);
// the string is now ""
```

Table 11.355: cppvmString: `setCPPvmTransferMode`

`setCPPvmTransferMode`	**cppvmString**
set the pvm transfer mode	

Synopsis

```
void setCPPvmTransferMode(int mode);
```

Parameter

Parameter	Type	Default	Description
`mode`	in		transfer mode: **`CPPvmTransferModeClean:`** clean old string, then receive new sting (default) **`CPPvmTransferModeAppend:`** append the received string to the current string

continued on next page...

. . . continued from previous page	
`setCPPvmTransferMode`	**cppvmString**

Exceptions	
cppvmExceptionBadParameter	unknown transfer mode

Example

```
cppvmReceiveStream recStrm(child, CPPvmRBchild);
cppvmString myString("Hello ");
myString.setCPPvmTransferMode(CPPvmTransferModeAppend);
recStrm >> myString;
```

Discussion

The initial transfer mode value of an array is `CPPvmTransferModeClean`.

Table 11.356: cppvmString: `setFitSizeToLength`

`setFitSizeToLength`	**cppvmString**
Fit the array size to the current length if `value = true`. The value is also used for the next calls of methods concerning a new or delete of the string.	

Synopsis

```
void setFitSizeToLength(bool value);
```

Parameter

Parameter	Type	Default	Description
`value`	in		`true:` whenever a new or delete of a the string is needed, the string size is fitted to the string length `false:` the size can only get bigger, not smaller

Example

```
cppvmString myString("Hello");
myString.setFitSizeToLength(false);
```

Table 11.357: cppvmString: `cycleShiftLeft`

`cycleShiftLeft`	**cppvmString**
cycle shift left	

Synopsis

```
void cycleShiftLeft();
```

Example

```
cppvmString myString("12345");
myString.cycleShiftLeft();
```

Table 11.358: cppvmString: cycleShiftRight

`cycleShiftRight`	**cppvmString**
cycle shift right	

Synopsis
`void cycleShiftRight();`

Example
`cppvmString myString("12345");` `myString.cycleShiftRight();`

Table 11.359: cppvmString: write

`write`	cppvmString
Set the string. Additionally send the string to the message mailbox if the mailbox was initialized.	

Synopsis
`void write(char *value);`

Parameter			
Parameter	Type	Default	Description
`value`	in		new string

Example
`cppvmString myString("hello");` `mySting.write("how do you do?");`

11.14 cppvmArrayData

There are also predefined array classes for message passing. The base class for the array data classes is cppvmArrayData . The array data classes are:

- cppvmArrayBool
- cppvmArrayChar
- cppvmArrayInt
- cppvmArrayLong
- cppvmArrayFloat
- cppvmArrayDouble

The array data classes are equivalent to arrays of the standard C/C++ data types. Since the methods of these classes are quite the same, only all methods cppvmArrayInt are described.

The cppvmArrayData classes are derived from cppvmObject. They also contain additional information, e.g. to allow a blocking send within CPPVM. For this reason, it is not possible to use them together with standard PVM send/receive functions. The next example will fail:

```
Process A:
...
cppvmArrayFloat floatArr(100);
sendStrm << floatArr;
...

Process B:
...
float floatArr[100];
pvm_recv(-1, -1);
pvm_upkfloat(floatArr, 100, 1);
...
```

Of course it is possible to use PVM functions together with CPPVM. But do not use them for the CPPVM message classes.

Table 11.360: cppvmArrayData: add

add			cppvmArrayData
add the elements of two arrays			
Synopsis			
cppvmArrayData& add(cppvmArrayData& obj);			
Parameter			
Parameter	Type	Default	Description
`obj`	in		array to add
Exceptions			
cppvmExceptionOutOfRange	dimensions (array lengths) are not the same		
Example			

`cppvmArrayLong myLongArray(10);`
`cppvmArrayDouble myDoubleArray(10);`
`myLongArray.add(myDoubleArray);`			

Discussion			
Only possible if the arrays have the same dimension.			

Table 11.361: cppvmArrayData: append

append			cppvmArrayData
append one array to another			
Synopsis			
void append(cppvmArrayData& obj);			
Parameter			
Parameter	Type	Default	Description
`obj`	in		array to append
Example			

`cppvmArrayLong myLongArray(30);`
`cppvmArrayDouble myDoubleArray(10);`
`myLongArray.append(myDoubleArray);`			

Discussion			
The element values of the added array are casted to the types of this array. In the example, the double values of `myDoubleArray` are casted to long values and added to the array `myLongArray`.			

Table 11.362: cppvmArrayData: clean

clean	cppvmArrayData
set all array elements to 0	
Synopsis	

continued on next page...

. . . continued from previous page	
`clean`	cppvmArrayData
void clean();	
Example	
`cppvmArrayLong myLongArray(300);` `myLongArray.clean();`	

Table 11.363: cppvmArrayData: `cleanElem`

`cleanElem`			cppvmArrayData
clean a specified array element			
Synopsis			
void cleanElem(unsigned int index);			
Parameter			
Parameter	Type	Default	Description
`index`	in		element index
Exceptions			
cppvmExceptionOutOfRange	index >= array length		
Example			
`cppvmArrayInt myIntArray(4);` `myIntArray.cleanElem(2);`			

Table 11.364: cppvmArrayData: `cleanRange`

`cleanRange`			cppvmArrayData
clean a range in the array			
Synopsis			
void cleanRange(unsigned int fromIndex, unsigned in toIndex);			
Parameter			
Parameter	Type	Default	Description
`fromIndex`	in		starting index
`inIndex`	in		ending index
Exceptions			
cppvmExceptionOutOfRange	'fromIndex' is greater than 'toIndex'		
cppvmExceptionOutOfRange	'fromIndex' is greater than array length		
cppvmExceptionOutOfRange	'toIndex' is greater than array length		
Example			
`cppvmArrayLong myLongArray(300);` `myLongArray.cleanRange(22, 99);`			
Discussion			
The element with index `toIndex` is also cleaned.			

Table 11.365: cppvmArrayData: `divide`

`divide`			cppvmArrayData
divide the elements of two arrays			
Synopsis			
cppvmArrayData& divide(cppvmArrayData& obj);			
Parameter			
Parameter	Type	Default	Description
`obj`	in		array to divide
Exceptions			
cppvmExceptionOutOfRange	dimensions (array lengths) are not the same		
Example			

`cppvmArrayInt myIntArray(10);`
`cppvmArrayFloat myFloatArray(10);`
`myIntArray.divide(myFloatArray);`			

Discussion			
Only possible if the arrays have the same dimension. The values are casted to the destination array element type.			

Table 11.366: cppvmArrayData: `getCPPvmTransferMode`

`getCPPvmTransferMode`			cppvmArrayData
get the current transfer mode of this object			
Synopsis			
int getCPPvmTransferMode();			
Parameter			
Parameter	Type	Default	Description
`int`	return		There are two transfer modes for arrays: `CPPvmTransferModeClean` clean own data, then receive new ones `CPPvmTransferModeAppend` append received data to the own ones
Example			
`cppvmArrayInt myIntArray(100);` `if(myIntArray.getCPPvmTransferMode()==CPPvmTransferModeClean){ ...`			
Discussion			
The initial transfer mode value of an array is `CPPvmTransferModeClean`.			

Table 11.367: cppvmArrayData: getElemDouble

getElemDouble			cppvmArrayData
Get an element as an double. This can be useful for a unique access to a collection of different array types.			
Synopsis			
double getElemDouble(unsigned int index);			
Parameter			
Parameter	Type	Default	Description
`index`	in		array element index
`double`	return		array value casted to double
Exceptions			
cppvmExceptionOutOfRange	index >= array length		
Example			
`cppvmArrayInt myIntArray(100);` `cout << myIntArray.getElemDouble(3);`			

Table 11.368: cppvmArrayData: getFitSizeToLength

getFitSizeToLength			**cppvmArrayData**
get the current state of the `fitSizeToLength` variable			
Synopsis			
`bool getFitSizeToLength();`			
Parameter			
Parameter	Type	Default	Description
`bool`	return		`true` whenever a new or delete of a the array is needed, the array size is fitted to the array length `false` the size can only get bigger, not smaller
Example			
`cppvmArrayDouble myDoubleArray(20);` `cout << myLongArray.getFitSizeToLength();`			

Table 11.369: cppvmArrayData: getLength

getLength			**cppvmArrayData**
get the current length of the array			
Synopsis			
`unsigned int getLength();`			
Parameter			
Parameter	Type	Default	Description

continued on next page...

. . . continued from previous page		
`getLength`		**cppvmArrayData**
`unsigned int`	return	current array length
Example		
`cppvmArrayInt myIntArray(20);` `cout << myIntArray.getLength();`		
Discussion		
There can be more allocated memory than the current array length. To get the current allocated memory size, use the method `getSize()`. To adapt the array length automatically to the array size use the method `setFitSizeToLength(true)`.		

Table 11.370: cppvmArrayData: `getSize`

`getSize`			**cppvmArrayData**
get the current size of the array, i.e. the total length of the allocated memory for the array			
Synopsis			
`unsigned int getSize();`			
Parameter			
Parameter	Type	Default	Description
`unsigned int`	return		current total array size
Example			
`cppvmArrayLong myLongArray(20);` `cout << myLongArray.getLength();`			
Discussion			
There can be more allocated memory than the current array length. To get the current array length, use the method `getLength()`. To adapt the array length automatically to the array size use the method `setFitSizeToLength(true)`.			

Table 11.371: cppvmArrayData: `multiply`

`multiply`			cppvmArrayData
multiply the elements of two arrays			
Synopsis			
cppvmArrayData& multiply(cppvmArrayData& obj);			
Parameter			
Parameter	Type	Default	Description
`obj`	in		array to multiply
Exceptions			
cppvmExceptionOutOfRange	dimensions (array lengths) are not the same		
Example			
			continued on next page. . .

. . . continued from previous page	
multiply	cppvmArrayData

```
cppvmArrayInt myIntArray(10);
cppvmArrayFloat myFloatArray(10);
myIntArray.multiply(myFloatArray);
```

Discussion

Only possible if the arrays have the same dimension. The values are casted to the destination array element type.

Table 11.372: cppvmArrayData: numElems

numElems	cppvmArrayData
get the current number of array elements	

Synopsis

unsigned int numElems();

Parameter

Parameter	Type	Default	Description
unsigned int	return		number of array elements

Example

```
cppvmArrayInt myIntArray(200);
cout << myIntArray.numElems();
```

Table 11.373: cppvmArrayData: operator++

operator++	cppvmArrayData
increment prefix operator; increment every array element	

Synopsis

void operator++();

Example

```
cppvmArrayInt myIntArray(5);
++myIntArray;
```

Table 11.374: cppvmArrayData: operator++

operator++	cppvmArrayData
increment postfix operator; increment every array element	

Synopsis

void operator++(int);

Example

```
cppvmArrayFloat myFloatArray(5);
myFloatArray++;
```

Table 11.375: cppvmArrayData: operator--

operator--	cppvmArrayData
decrement prefix operator; decrement every array element	
Synopsis	
`void operator--();`	
Example	
`cppvmArrayDouble myDoubleArray(5);` `--myDoubleArray;`	

Table 11.376: cppvmArrayData: operator--

operator--	cppvmArrayData
decrement postfix operator; decrement every array element	
Synopsis	
`void operator--(int);`	
Example	
`cppvmArrayLong myLongArray(8);` `myLongArray--;`	

Table 11.377: cppvmArrayData: resize

resize			**cppvmArrayData**
Resize the array to a new length. Copy old values if possible. Do not clean new elements.			
Synopsis			
`void resize(unsigned in newLen);`			
Parameter			
Parameter	Type	Default	Description
`newLen`	in		new length of the array
Example			
`cppvmArrayInt myIntArray(7);` `myIntArray.resize(3);`			
Discussion			

The result of the `resize` method depends on the variable `fitSizeToLength` `fitSizeToLength`:
`fitSizeToLength = true`: allocate new array if `newLen!=size+1`
`fitSizeToLength = false`: allocate new array if `newLen+1>size`
`fitSizeToLength` can be manipulated with the method `setFitSizeToLength`.
If the new size is greater than the current size of the array, the additional elements have undefined values! Use the option `cleanOnlyNewElements` option of the method `resizeClean` to clean new elements.			

Table 11.378: cppvmArrayData: resizeClean

`resizeClean`			**cppvmArrayData**
resize the array to a new length and clean array			
Synopsis			
`void resizeClean(unsigned in newLen, bool cleanOnlyNewElements);`			
Parameter			
Parameter	Type	Default	Description
`newLen`	in		new length of the array
`cleanOnlyNewElements`		false	clean only new elements
Example			
`cppvmArrayFloat myFloatArray(4);` `myFloatArray.resizeClean(22);`			
Discussion			

The result of the `resizeClean` method depends on the variable `fitSizeToLength` `fitSizeToLength`:
`fitSizeToLength = true`: allocate new array if `newLen!=size+1`
`fitSizeToLength = false`: allocate new array if `newLen+1>size`
`fitSizeToLength` can be manipulated with the method `setFitSizeToLength`.
The option `cleanOnlyNewElements` can be used to clean new elements if the new size of the array is greater than the old size. If `cleanOnlyNewElements` is set `true` the old values are copied (if possible).

```
cppvmArrayInt myIntArray(3);
...
// e.g. the array is here (1 2 3)
myIntArray.resizeClean(5, true);
// the array is now (1 2 3 0 0)
...
// e.g. the array is here (1 2 3) again
myIntArray.resizeClean(5);
// the array is now (0 0 0 0 0)
```

Table 11.379: cppvmArrayData: setCPPvmTransferMode

`setCPPvmTransferMode`			cppvmArrayData
set the transfer mode of this object			
Synopsis			
void setCPPvmTransferMode(int mode);			
Parameter			
Parameter	Type	Default	Description
			continued on next page...

...continued from previous page		
`setCPPvmTransferMode`		cppvmArrayData
`mode`	in	There are two transfer modes for arrays: `CPPvmTransferModeClean` clean own data, then receive new ones `CPPvmTransferModeAppend` append received data to the own ones

Exceptions	
cppvmExceptionBadParameter	unknown transfer mode

Example
`cppvmArrayFloat myFloatArray(30);` `myFloatArray.setCPPvmTransferMode(CPPvmTransferModeAppend);`

Discussion
The initial transfer mode value of an array is `CPPvmTransferModeClean`

Table 11.380: cppvmArrayData: `setFitSizeToLength`

`setFitSizeToLength`			**cppvmArrayData**
Fit the array size to the current length if `value = true`. The value is also used for the next calls of methods concerning a new or delete of the array.			
Synopsis			
`void setFitSizeToLength(bool value);`			
Parameter			
Parameter	Type	Default	Description
`value`	in		`true` whenever a new or delete of a the array is needed, the array size is fitted to the array length `false` the size can only get bigger, not smaller
Example			
`cppvmArrayInt myIntArray(30);` `myIntArray.setFitSizeToLength(false);`			

Table 11.381: cppvmArrayData: `subtract`

`subtract`			cppvmArrayData
subtract the elements of two arrays			
Synopsis			
cppvmArrayData& subtract(cppvmArrayData& obj);			
Parameter			
Parameter	Type	Default	Description

continued on next page...

. . . continued from previous page

`subtract`		cppvmArrayData
`obj`	in	array to add
Exceptions		
cppvmExceptionOutOfRange	dimensions (array lengths) are not the same	

Example

```
cppvmArrayInt myIntArray(15);
cppvmArrayFloat myFloatArray(15);
myIntArray.subtract(myFloatArray);
```

Discussion

Only possible if the arrays have the same dimension. The values are casted to the destination array element type.

11.14.1 cppvmArrayInt

The class cppvmArrayInt is an example for an array data class. The methods can be used equivalent for the classes:

- cppvmArrayBool
- cppvmArrayChar
- cppvmArrayLong
- cppvmArrayFloat
- cppvmArrayDouble

Table 11.382: cppvmArrayInt: `constructor`

`constructor`	cppvmArrayInt
default constructor; array length = 0	
Synopsis	
`cppvmArrayInt();`	
Example	
`cppvmArrayInt myIntArray();`	

Table 11.383: cppvmArrayInt: `constructor`

`constructor`			cppvmArrayInt
constructor			
Synopsis			
`cppvmArrayInt(unsigned int length, bool cleanArray);`			
Parameter			
Parameter	Type	Default	Description
`length`	in		array length
`cleanArray`	in	true	clean array after allocation
Example			
`cppvmArrayInt myIntArray(100); cppvmArrayDouble myDoubleArray(50, false);`			
Discussion			
If the parameter `cleanArray` is set to `false`, the values of the elements are undefined!			

Table 11.384: cppvmArrayInt: `copy constructor`

`copy constructor`	cppvmArrayInt
copy constructor	
Synopsis	
`cppvmArrayInt(const cppvmArrayInt& obj);`	

Parameter			
Parameter	Type	Default	Description
`obj`	in		object to copy

Example
`cppvmArrayInt myIntArray = previousDefinedIntArray;`

Table 11.385: cppvmArrayInt: `cppvmClassMsgTag`

`cppvmClassMsgTag`	cppvmArrayInt
get the message tag for this class (class method)	
Synopsis	
static int cppvmClassMsgTag();	

Table 11.386: cppvmArrayInt: `cppvmMsgTag`

`cppvmMsgTag`	cppvmArrayInt
get the message tag for this class (instance method)	
Synopsis	
virtual int cppvmMsgTag();	

Table 11.387: cppvmArrayInt: `getPtr()`

`getPtr()`	cppvmArrayInt
get the pointer to the integer array	
Synopsis	
`int* getPtr();`	

Parameter			
Parameter	Type	Default	Description
`int*`	return		pointer to the integer array

Example

```
cppvmArrayInt myIntArray(100);
int *iPtr;
iPtr = myIntArray.getPtr();
iPtr[0] = 333;
```

Table 11.388: cppvmArrayInt: operator=

operator=			cppvmArrayInt
assign object			
Synopsis			
`cppvmArrayInt& operator= (const cppvmArrayInt& obj)`			
Parameter			
Parameter	Type	Default	Description
`obj`	in		object to assign
`cppvmArrayInt`	return		this object
Example			

```
cppvmArrayInt f1(20);
cppvmArrayInt f2(30);
...
f1 = f2;
```

Table 11.389: cppvmArrayInt: operator==

operator==			cppvmArrayInt
test if objects are equal			
Synopsis			
`bool operator== (const cppvmArrayInt& obj)`			
Parameter			
Parameter	Type	Default	Description
obj	in		object to compare with
bool	return		compare result
Example			

```
cppvmArrayInt f1(10);
cppvmArrayInt f2(10);
...
if(f1==f2){ ...
```

Table 11.390: cppvmArrayInt: operator!=

operator!=			cppvmArrayInt
test if objects are not equal			
Synopsis			
`bool operator!= (const cppvmArrayInt& obj)`			
Parameter			
Parameter	Type	Default	Description
`obj`	in		object to compare with
`bool`	return		compare result

continued on next page...

. . . continued from previous page	
`operator!=`	cppvmArrayInt

Example
`cppvmArrayInt f1(20);`
`cppvmArrayInt f2(20);`
`...`
`if(f1!=f2){ ...`

Table 11.391: cppvmArrayInt: `operator[]`

`operator[]`	cppvmArrayInt
access to specified array entry	

Synopsis
`int& operator[](unsigned int index);`

Parameter			
Parameter	Type	Default	Description
`index`	in		array element index
`int&`	return		element value

Exceptions	
cppvmExceptionOutOfRange	index >= array length

Example
`cppvmArrayInt myIntArray(8);`
// get value
cout << myIntArray[6];
// set value
myIntArray[3] = 42;

Table 11.392: cppvmArrayInt: `operator<<`

`operator<<`	**cppvmArrayInt**
write object on out-stream	

Synopsis
`ostream& operator<<(ostream& strm, cppvmObject& obj);`

Parameter			
Parameter	Type	Default	Description
`ostream`	inout		out stream to write object on
`obj`	in		write this object on stream

Example
`cppvmArrayInt myIntArray(4);`
`cout << myIntArray;`

Table 11.393: cppvmArrayInt: operator>>

operator>>			**cppvmArrayInt**
read object from in-stream			
Synopsis			
`istream& operator>>(istream& strm, cppvmObject& obj);`			
Parameter			
Parameter	Type	Default	Description
`istream`	inout		in stream to read object from
`obj`	in		read this object from stream
Example			
`cppvmArrayInt myIntArray();` `cin >> myIntArray;`			
Discussion			
the syntax e.g. for a array of length 4 is: length [elem0 elem1 elem2 elem3] So for the example above type in when the prompt appears: 4 [12 77 123456 1]			

Table 11.394: cppvmArrayInt: read

read			cppvmArrayInt
get the array element value; read the array first from the message mailbox if the mailbox was initialized			
Synopsis			
`int read(unsigned int index);`			
Parameter			
Parameter	Type	Default	Description
`index`	in		element index
`int`	return		current element value
Exceptions			
cppvmExceptionOutOfRange	index > array length		
Example			

`cppvmArrayInt myIntArray(7);`
`int i;`
`i = myIntArray.read(3);`			

Table 11.395: cppvmArrayInt: cycleShiftLeft

cycleShiftLeft	**cppvmArrayInt**
cycle shift left; (1 2 3) -> (2 3 1)	
Synopsis	
	continued on next page...

...continued from previous page	
`cycleShiftLeft`	**cppvmArrayInt**
`void cycleShiftLeft();`	
Example	
`cppvmArrayInt myIntArray(7);` `myIntArray.cycleShiftLeft();`	

Table 11.396: cppvmArrayInt: `cycleShiftRight`

`cycleShiftRight`	**cppvmArrayInt**
cycle shift right; (1 2 3) -> (3 1 2)	
Synopsis	
`void cycleShiftRight();`	
Example	
`cppvmArrayInt myIntArray(10);` `myIntArray.cycleShiftRight();`	

Table 11.397: cppvmArrayInt: `write`

`write`			cppvmArrayInt
set an array element value; send array to the message mailbox if the mailbox was initialized			
Synopsis			
`void write(unsigned int index, int value);`			
Parameter			
Parameter	Type	Default	Description
`index`	in		element index
`value`	in		new element value
Exceptions			
cppvmExceptionOutOfRange	index > array length		
Example			

`cppvmArrayInt myIntArray(8);`
`int i = 5;`
`myIntArray.write(2,i);`			

11.15 cppvmObjectCollection

The class cppvmObjectCollection is an ordered collection for all classes derived from cppvmObject. The collection contains a set of pointer to cppvmObject objects. The objects must be allocated and passed to the collection. From now on the collection administrates and deletes the elements with its methods.

Since the elements are pointers to cppvmObject class objects, the objects must be downcasted to the correct type when they are accessed. For instance if you have added cppvmFloat objects to the collection, you have to downcast the collection element after the access:

`(cppvmFloat&)myPvmObjColl[0]`

The class cppvmObjectCollection is also derived from cppvmObject. Therefore nested collections (collections of collections) are possible and even more interesting, collections are capable for message passing. The message passing works for all standard CPPVM classes. It is also possible to make new user defined CPPVM classes send-/receivable within collections (see chapter 8).

There is no copy constructor or `operator=` for this class since there is no deep copy implemented for cppvmObject class objects.

Table 11.398: cppvmObjectCollection: `constructor`

`constructor`	cppvmObjectCollection
constructor	
Synopsis	
`cppvmObjectCollection();`	
Example	
`cppvmObjectCollection myColl();`	

Table 11.399: cppvmObjectCollection: `add`

`add`			cppvmObjectCollection
add an element to the collection			
Synopsis			
void add(cppvmObject* objPtr);			
Parameter			
Parameter	Type	Default	Description
`objPtr`	in		pointer to an cppvmObject object
Example			
`cppvmObjectCollection myColl();` `myColl.add(new cppvmFloat(8.8));`			
			continued on next page...

... continued from previous page	
add	cppvmObjectCollection

Discussion
The element must be allocated by the caller and is from now on administrated within the collection

Table 11.400: cppvmObjectCollection: addRecFct

addRecFct	cppvmObjectCollection
add a user defined receive function to the set of receive functions	

Synopsis
`void addRecFct(pvmCollRecFctPtr fctPtr);`

Parameter			
Parameter	Type	Default	Description
`fctPtr`	in		receive function for user defined classes derived from cppvmObject of type `cppvmObject* (*pvmCollRecFctPtr)(int objClassId)`.

Example

```
cppvmObjectCollection myColl();
myColl.addRecFct(recFunction);
```

Discussion

The receive function has to do:

- if the `objClassId` fits the object(s) which can be received by the function:
  - create a new object with new operator
  - receive the object
  - give the pointer to this object back (`recObj`)
- if the `objClassId` does not fit the object(s) which can be received by the function:
  - return NULL pointer

See chapter 8 for more information

Table 11.401: cppvmObjectCollection: append

append	cppvmObjectCollection
append another collection to this collection; the appended collection is deleted	

Synopsis
void append(cppvmObjectCollection *objColl);

Parameter
continued on next page...

. . . continued from previous page			
`append`			cppvmObjectCollection
Parameter	Type	Default	Description
`objColl`	in		pointer to appended collection
Example			

```
cppvmObjectCollection myColl1();
cppvmObjectCollection myColl2();
...
myColl1.append(&myColl2);
```

Discussion

The appended collection is deleted because there is no deep copy for cppvmObject objects

Table 11.402: cppvmObjectCollection: `cppvmClassMsgTag`

`cppvmClassMsgTag`	cppvmObjectCollection
get the message tag for this class (class method)	
Synopsis	
static int cppvmClassMsgTag();	

Table 11.403: cppvmObjectCollection: `cppvmMsgTag`

`cppvmMsgTag`	cppvmObjectCollection
get the message tag for this class (instance method)	
Synopsis	
virtual int cppvmMsgTag();	

Table 11.404: cppvmObjectCollection: `getConstObjPtr`

`getConstObjPtr`			cppvmObjectCollection
get the pointer to an collection element for a constant object			
Synopsis			
`cppvmObject* getConstObjPtr(unsigned int index) const;`			
Parameter			
Parameter	Type	Default	Description
`index`	in		element index
`cppvmObject*`	return		pointer to the collection element
Exceptions			
cppvmExceptionOutOfRange	index > collection length		
Example			

continued on next page. . .

. . . continued from previous page	
`getConstObjPtr`	cppvmObjectCollection

```
cppvmObjectCollection myColl();
myColl.add(new cppvmFloat(1.1));
...
cppvmObject* objPtr;
objPtr = myColl.getConstObjPtr(0);
```

Table 11.405: cppvmObjectCollection: `getCPPvmTransferMode`

`getCPPvmTransferMode`			cppvmObjectCollection
get the current transfer mode of this object			
Synopsis			
int getCPPvmTransferMode();			
Parameter			
Parameter	Type	Default	Description
`int`	return		There are two transfer modes for a collection: `CPPvmTransferModeClean` clean collection elements, then receive new elements `CPPvmTransferModeAppend` append received elements to the own ones
Example			
`cppvmObjectCollection myColl();` `if(myColl.getCPPvmTransferMode()==CPPvmTransferModeClean){ ...`			
Discussion			
The initial transfer mode value of an collection is `CPPvmTransferModeClean`			

Table 11.406: cppvmObjectCollection: `getObjPtr`

`getObjPtr`			cppvmObjectCollection
get the pointer to an collection element			
Synopsis			
`cppvmObject* getObjPtr(unsigned int index);`			
Parameter			
Parameter	Type	Default	Description
`index`	in		element index
`cppvmObject*`	return		pointer to the collection element
Exceptions			
cppvmExceptionOutOfRange	index > collection length		

continued on next page. . .

. . . continued from previous page	
`getObjPtr`	cppvmObjectCollection

Example

```
cppvmObjectCollection myColl();
myColl.add(new cppvmFloat(1.1));
...
cppvmObject* objPtr;
objPtr = myColl.getObjPtr(0);
```

Table 11.407: cppvmObjectCollection: `getTaskWithName`

`getTaskWithName`	cppvmObjectCollection
get the cppvmTask collection member with the given name; only useful if the collection contains cppvmTask objects	

Synopsis

```
cppvmTask& getTaskWithName(cppvmString taskName);
```

Parameter

Parameter	Type	Default	Description
`taskName`	in		task name
`cppvmTask&`	return		description of the task

Exceptions

cppvmExceptionBadParameter	there is no cppvmTask with the given name in the collection

Example

```
cppvmObjectCollection myColl();
...
cout << myColl.getTaskWithName("hello_world");
```

Table 11.408: cppvmObjectCollection: `numElems`

`numElems`	cppvmObjectCollection
get the current number of collection elements	

Synopsis

unsigned int numElems();

Parameter

Parameter	Type	Default	Description
`unsigned int`	return		number of collection elements

Example

```
cppvmObjectCollection myColl();
cout << myColl.numElems();
```

Table 11.409: cppvmObjectCollection: operator==

operator==			cppvmObjectCollection
test if objects are equal			
Synopsis			
`bool operator== (const cppvmObjectCollection& obj)`			
Parameter			
Parameter	Type	Default	Description
obj	in		object to compare with
bool	return		compare result
Example			

```
cppvmObjectCollection oColl1();
cppvmObjectCollection oColl2();
...
if(oColl1==oColl2){ ...
```

Table 11.410: cppvmObjectCollection: operator!=

operator!=			cppvmObjectCollection
test if objects are not equal			
Synopsis			
`bool operator!= (const cppvmObjectCollection& obj)`			
Parameter			
Parameter	Type	Default	Description
obj	in		object to compare with
bool	return		compare result
Example			

```
cppvmObjectCollection oColl1();
cppvmObjectCollection oColl2();
...
if(oColl1!=oColl2){ ...
```

Table 11.411: cppvmObjectCollection: operator[]

operator[]			cppvmObjectCollection
get a specified collection entry			
Synopsis			
`cppvmObject& operator[](unsigned int index);`			
Parameter			
Parameter	Type	Default	Description
index	in		collection element index
int	return		element value

continued on next page...

...continued from previous page	
`operator[]`	cppvmObjectCollection

Exceptions	
cppvmExceptionOutOfRange	index > collection length

Example

```
cppvmObjectCollection myColl();
myColl.add(new cppvmFloat(1.1));
...
(cppvmFloat&)myColl[0] = 4.4;
```

Table 11.412: cppvmObjectCollection: `operator<<`

`operator<<`	**cppvmObjectCollection**
write object on out-stream	

Synopsis

```
ostream& operator<<(ostream& strm, cppvmObject& obj);
```

Parameter

Parameter	Type	Default	Description
`ostream`	inout		out stream to write object on
`obj`	in		write this object on stream

Example

```
cppvmObjectCollection myColl();
cout << myColl;
```

Table 11.413: cppvmObjectCollection: `read`

`read`	cppvmObjectCollection
get a collection element ; get the collection before from the message mailbox if the mailbox was initialized	

Synopsis

```
cppvmObject& read(unsigned int index);
```

Parameter

Parameter	Type	Default	Description
`index`	in		element index
`int`	return		collection element

Exceptions	
cppvmExceptionOutOfRange	index > collection length

Example

```
myColl.add(new cppvmFloat(1.1));
...
cout << myColl.read(0);
```

Table 11.414: cppvmObjectCollection: `remove`

`remove`			cppvmObjectCollection
remove an (identical) element from the collection			
Synopsis			
void remove(cppvmObject& obj);			
Parameter			
Parameter	Type	Default	Description
`obj`	in		object to remove from collection
Exceptions			
cppvmExceptionBadParamter	Object not in collection		
Example			

```
cppvmObjectCollection myColl();
cppvmFloat *fPtr;
fPtr = new cppvmFloat(3.3);
myColl.add(fPtr);
...
myColl.remove(*fPtr);
```

Table 11.415: cppvmObjectCollection: `remove`

`remove`			cppvmObjectCollection
remove an (identical) element from the collection			
Synopsis			
void remove(cppvmObject* objPtr);			
Parameter			
Parameter	Type	Default	Description
`objPtr`	in		pointer to object to remove from collection
Exceptions			
cppvmExceptionBadParameter	Object not in collection		
Example			

```
cppvmObjectCollection myColl();
cppvmFloat *fPtr;
fPtr = new cppvmFloat(3.3);
myColl.add(fPtr);
...
myColl.remove(fPtr);
```

Table 11.416: cppvmObjectCollection: `remove`

`remove`	cppvmObjectCollection
remove an element from the collection	

continued on next page...

. . . continued from previous page	
`remove`	cppvmObjectCollection

Synopsis
void remove(unsigned int index);

Parameter			
Parameter	Type	Default	Description
`index`	in		element index in collection

Exceptions	
cppvmExceptionOutOfRange	Bad index

Example

```
cppvmObjectCollection myColl();
...
myColl.remove(3);
```

Table 11.417: cppvmObjectCollection: `clean`

`clean`	cppvmObjectCollection
remove all elements from the collection	

Synopsis
void clean();

Example

```
cppvmObjectCollection myColl();
...
myColl.clean();
```

Table 11.418: cppvmObjectCollection: `removeRecFct`

`removeRecFct`	cppvmObjectCollection
remove a user defined receive function from the set of receive functions	

Synopsis
`void removeRecFct(pvmCollRecFctPtr fctPtr);`

Parameter			
Parameter	Type	Default	Description
`fctPtr`	in		receive function for user defined classes derived from cppvmObject of type `cppvmObject* (*pvmCollRecFctPtr)(int objClassId)`.

Exceptions	
cppvmExceptionBadParameter	function is not in the collection

Example

```
cppvmObjectCollection myColl();
myColl.removeRecFct(recFunction);
```

Table 11.419: cppvmObjectCollection: `setCPPvmTransferMode`

`setCPPvmTransferMode`			cppvmObjectCollection
set the transfer mode of this object			
Synopsis			
void setCPPvmTransferMode(int mode);			
Parameter			
Parameter	Type	Default	Description
`mode`	in		There are two transfer modes for a collection: `CPPvmTransferModeClean` clean collection elements, then receive new elements `CPPvmTransferModeAppend` append received elements to the own ones
Exceptions			
cppvmExceptionBadParameter	unknown transfer mode		
Example			
`cppvmObjectCollection myColl();` `myColl.setCPPvmTransferMode(CPPvmTransferModeAppend);`			
Discussion			
The initial transfer mode value of an collection is `CPPvmTransferModeClean`			

Table 11.420: cppvmObjectCollection: `sort`

`sort`			cppvmObjectCollection
sorts the members of the collection by the given compare function with the direct insert method			
Synopsis			
`void sort(pvmCollCompFctPtr cmpPtr);`			
Parameter			
Parameter	Type	Default	Description
`cmpPtr`	in		compare function of type `bool (*pvmCollCompFctPtr) (cppvmObject* obj1, cppvmObject* obj2);`
Example			
			continued on next page...

. . . continued from previous page

sort	cppvmObjectCollection

```
bool
compareDescending(cppvmObject* obj1, cppvmObject* obj2){
double val1, val2;

val1 = ((cppvmSingleData*)obj1)->readDouble();
val2 = ((cppvmSingleData*)obj2)->readDouble();

if(val1<val2){
return true;
}
else{
return false;
}
}
...
cppvmObjectCollection myColl();
myColl.add(new cppvmFloat(4.1)); myColl.add(new cppvmInt(4));
myColl.add(new cppvmString("4.3")); ...
myColl.sort(compareDescending);
```

Discussion

The compare function must be usable for all objects in the collection, see section 12.7 for more information.

Table 11.421: cppvmObjectCollection: swap

swap	cppvmObjectCollection
swap two members of the collection	

Synopsis

```
void swap(unsigned int index1, unsigned int index2);
```

Parameter

Parameter	Type	Default	Description
index1	in		index of the first element
index2	in		index of the second element

Exceptions

cppvmExceptionOutOfRange	index1 or index2 > collection length

Example

```
cppvmObjectCollection myColl();
...
myColl.swap(2,5);
```

11.16 cppvmTask

Objects of the class cppvmTask can be used to transfer informations about PVM processes. Normally there is no need to create cppvmTask objects by your own. Just use the adequate methods of `cppvmConnection`, `cppvmSpawnConnection` and `cppvmSpawnMultiConnection` to get objects of this class.

Table 11.422: cppvmTask: `constructor`

`constructor`	cppvmTask
default constructor	
Synopsis	
`cppvmTask();`	
Example	
`cppvmTask myProcess;`	
Discussion	
Use this constructor only if you are setting the variables to proper values later on.	

Table 11.423: cppvmTask: `constructor`

`constructor`			cppvmTask
constructor with task description			
Synopsis			
`cppvmTask(int id, int contextId, cppvmString name);`			
Parameter			
Parameter	Type	Default	Description
`id`	in		PVM task id
`contextId`	in		context id of the task
`name`	in		task name
Example			
`cppvmTask myProcess(12345, 4711, "myproggy");`			
Discussion			
The created object describes a PVM task.			

Table 11.424: cppvmTask: `copy constructor`

`copy constructor`	cppvmTask
copy constructor	
Synopsis	
`cppvmTask(const cppvmTask& obj);`	
Parameter	
	continued on next page...

. . . continued from previous page			
`copy constructor`			cppvmTask
Parameter	Type	Default	Description
`obj`	in		object to copy
Example			
`cppvmTask myProcess = previousDefinedProcess;`			

Table 11.425: cppvmTask: `cppvmClassMsgTag`

`cppvmClassMsgTag`	cppvmTask
get the message tag for this class (class method)	
Synopsis	
static int cppvmClassMsgTag();	

Table 11.426: cppvmTask: `cppvmMsgTag`

`cppvmMsgTag`	cppvmTask
get the message tag for this class (instance method)	
Synopsis	
virtual int cppvmMsgTag();	

Table 11.427: cppvmTask: `getContextId`

`getContextId`			cppvmTask
get the context id			
Synopsis			
int getContextId();			
Parameter			
Parameter	Type	Default	Description
`int`	return		context id
Example			
`cppvmTask myProcess(1234, 4711, "myproggy");` `cout << myProcess.getContextId();`			

Table 11.428: cppvmTask: `getTid`

`getTid`	cppvmTask
get the task id	
Synopsis	
int getTid();	
	continued on next page. . .

... continued from previous page	
`getTid`	cppvmTask

Parameter			
Parameter	Type	Default	Description
`int`	return		task id

Example

```
cppvmTask myProcess(1234, 4711, "myproggy");
cout << myProcess.getTid();
```

Table 11.429: cppvmTask: `getName`

`getName`	cppvmTask
get the task name	

Synopsis

cppvmString getName();

Parameter			
Parameter	Type	Default	Description
`cppvmString`	return		task name

Example

```
cppvmTask myProcess(1234, 4711, "myproggy");
cout << myProcess.getName();
```

Table 11.430: cppvmTask: `halt`

`halt`	cppvmTask
shut down the entire PVM system	

Synopsis

void halt();

Exceptions

cppvmExceptionBadResult	local pvmd is not responding

Example

```
cppvmTask myProcesses(); myProcesses.halt();
```

Discussion

It is recommended to use this method to shut down your CPPVM program. Otherwise PVM daemons or processes can still run after your main program has ended. This can cause errors if you start the program again. If you have this problem, start a pvm console and type `halt`.

Table 11.431: cppvmTask: killTask

killTask	cppvmTask
kill the task	
Synopsis	
void killTask();	
Exceptions	
cppvmExceptionBadResult	cannot kill task process
Example	

```
cppvmTask myProcess(1234, 4711, "myproggy");
...
myProcess.killTask();
```

Table 11.432: cppvmTask: operator=

operator=			cppvmTask
assign object			
Synopsis			
cppvmTask& operator= (const cppvmTask& obj)			
Parameter			
Parameter	Type	Default	Description
obj	in		object to assign
cppvmTask	return		this object
Example			

```
cppvmTask c1, c2;
...
c1 = c2;
```

Table 11.433: cppvmTask: operator==

operator==			cppvmTask
test if objects are equal			
Synopsis			
bool operator== (const cppvmTask& obj)			
Parameter			
Parameter	Type	Default	Description
obj	in		object to compare with
bool	return		compare result
Example			

```
cppvmTask c1, c2;
...
if(c1==c2){ ...
```

Table 11.434: cppvmTask: operator!=

operator!=			cppvmTask
test if objects are not equal			
Synopsis			
bool operator!= (const cppvmTask& obj)			
Parameter			
Parameter	Type	Default	Description
obj	in		object to compare with
bool	return		compare result
Example			

```
cppvmTask c1, c2;
...
if(c1!=c2){ ...
```

Table 11.435: cppvmTask: operator<<

operator<<			cppvmTask
write object on ostream			
Synopsis			
ostream& operator<<(ostream& strm, cppvmTask& obj);			
Parameter			
Parameter	Type	Default	Description
obj	in		write this object on ostream
strm	in		ostream
ostream	return		ostream after writing the object
Example			

```
cppvmTask c;
cout << c;
```

Table 11.436: cppvmTask: setContextId

setContextId			cppvmTask
set the context id			
Synopsis			
void setContextId(int contextId);			
Parameter			
Parameter	Type	Default	Description
contextId	in		context id
Example			

```
cppvmTask myProcess(1234, 4711, "myproggy");
myProcess.setContextId(5678);
```

Table 11.437: cppvmTask: setCPPvmClassMsgTag

`setCPPvmClassMsgTag`			cppvmTask
set the message tag of this class			
Synopsis			
void setCPPvmClassMsgTag(int tag);			
Parameter			
Parameter	Type	Default	Description
`tag`	in		new message tag for this class
Example			
`cppvmTask myProcess();` `myProcess.setCPPvmClassMsgTag(33);`			
Discussion			
This method should be handled with care. The message tag ensures that sent messages can be received by the objects of the same class. If the message tag of a class is modified in the sending process but not in the receiving process, the message cannot be received. Use this method only for very specific applications, for example to receive standard CPPvm classes within user-written classes or for notification messages. The sending process can send for example cppvmInt objects with different message tags to indicate different notifications. The receiving process can check the messages for the different message tags.			

Table 11.438: cppvmTask: setName

`setName`			cppvmTask
set the task name			
Synopsis			
void setName(cppvmString name);			
Parameter			
Parameter	Type	Default	Description
`name`	in		task name
Example			

`cppvmTask myProcess(1234, 4711, "myproggy");`
`cppvmString name = "hello";`
`myProcess.setName(name);`			

Table 11.439: cppvmTask: setTid

`setTid`	cppvmTask
set the task id	
Synopsis	
void setTid(int id);	
	continued on next page…

... continued from previous page			
`setTid`			cppvmTask

Parameter			
Parameter	Type	Default	Description
`id`	in		task id

Example

```
cppvmTask myProcess(1234, 4711, "myproggy");
myProcess.setTid(5678);
```

11.17 cppvmStl*

The classes cppvmStl* are not described. They are derived from cppvmObject as well as the corresponding standard template library (STL) classes. Refer the description of cppvmObject and a STL manual for further information.

11.17.1 Limitations

The standard template library (STL) versions of the gcc (UNIX) and Visual C++ (Windows) packages were used as reference STL implementations. Whenever a decision had to be made between functionality and portability, portability was chosen.

The STL classes bitset, slist and valarray are not supported by default. This is due to they are not implemented in Visual C++ or in older versions of gcc. To enable these classes within CPPvm use the compiler switches `-DCPPvmUseStlBitset`, `-DCPPvmUseStlSlist`, and `-DCPPvmUseStlValarray`.

Some constructors cause problems with Visual C++ or gcc. Therefore all constructors in the class cppvmStlString using `size_type` and the constructors in cppvmStlValarray using `size_t` as parameter are missing.

The template parameters of the STL classes can be standard C++ types (`bool`, `char`, ...) as well as classes derived from cppvmObject. This list can be enhanced by writing new `cppvmTransferTemplate` functions (see chapter 9). It is not possible to send nested STL objects, e.g.:

```
...
  // not possible!!
  vector< deque<int> > vd;
  sendStrm << vd;
...
```

This is due to limitations of the Visual C++ compiler. The only exception to this rule is the STL class string. string can be used as template parameter for other STL classes. Visual C++ does not support partly instantiated templates. This prevents specialized transfer functions, e.g. for `map<T, bool>`.

For all compilers the following namespaces are used globally:

```
using std::bitset;
using std::complex;
using std::deque;
using std::list;
using std::map;
using std::multiset;
```

```
using std::multimap;
using std::pair;
using std::priority_queue;
using std::queue;
using std::set;
using std::slist;
using std::stack;
using std::string;
using std::valarray;
using std::vector;
```

Be aware that this may cause name conflicts.

11.18 cppvmNotification

The notification classes are used to get informations about modifications of the virtual machine. By creating a notification object the calling process is registered by PVM to get notification messages. The message tags of the incoming messages are set as a constructor parameter. The notification messages can be received by the notification objects using receive streams. The characteristics of notification objects are:

- Notification objects can only be received, not send.
- The incoming notification messages have the message tags given in the constructors of the notification objects. Since the messages can be of different type, ensure not to use identical message tags for different notification objects.
- The notification messages are send by a PVM-daemon. Use a receive-from-any stream to receive the messages.
- Only the notification messages for cppvmNotificationHostAdd can be disabled. Even when cppvmNotificationTaskExit or cppvmNotificationHostDelete objects are deleted, the PVM-daemon will still send notification messages to the process. This is no critical problem, since the number of messages is limited by the number of processes/host given in the constructors.
- By creating multiple notification objects for the same event (e.g. cppvmNotification-TaskExit for different spawned processes), multiple notification messages are sent to the calling process.
- Use the `set` method only to initialize an uninitialized object created by a default constructor. E.g. if you if you set a different message tag than in a previous initialization, two notification messages are send to the calling process. But the first one can not be received any more, since no notification object is responsible for it any more.

The base class cppvmNotification can be used for any PVM notification message. To make things easier you should use the specialized notification classes:

- cppvmNotificationHostAdd: New hosts are added.
- cppvmNotificationHostDelete: Hosts are deleted or crashed.
- cppvmNotificationTaskExit: A PVM task exits or is killed.

An example how to use the notification classes is given in section 10.3.

Table 11.440: cppvmNotification: `constructor`

`constructor`	cppvmNotification
default constructor	
Synopsis	
`cppvmNotification();`	
Example	
`cppvmNotification myNotification;`	
Discussion	
Use the default constructor only if the notification object is initialized later on with the `set` method.	

Table 11.441: cppvmNotification: `constructor`

`constructor`			cppvmNotification
constructor			
Synopsis			
`cppvmNotification(int event, int msgTag, int number, int* tids);`			
Parameter			
Parameter	Type	Default	Description
`event`	in		notification event: `PvmTaskExit:` task exits or is killed `PvmHostDelete:` Host is deleted or crashes `PvmHostAdd:` New host is added
`msgTag`	in		message tag to be used for the notification messages
`number`	in		The meaning of `number` depends on the `event` value: `PvmTaskExit:` Length of the `tids` array `PvmHostDelete:` Length of the `tids` array `PvmHostAdd:` Number of times to notify. Specifying a `number` of -1 turns on `PvmHostAdd` messages until a future call of the `set` method, a `number` of zero disables them.
`tids`	in		The meaning of `tids` depends on the `event` value: `PvmTaskExit:` Array of length `number` of task id's to be notified about `PvmHostDelete:` Array of length `number` of host id's to be notified about `PvmHostAdd:` Not used

continued on next page...

...continued from previous page	
`constructor`	cppvmNotification

Exceptions	
cppvmExceptionBadParameter	could not start notification request

Example
`cppvmNotification notify(PvmHostAdd, 111, -1, NULL);`

Discussion
You have to receive the notification messages and read the results with the `getReceivedTids` method later on.

Table 11.442: cppvmNotification: `getEvent`

`getEvent`			cppvmNotification
get the initialized value for `event`			
Synopsis			
int getEvent();			
Parameter			
Parameter	Type	Default	Description
int	return		initialized `event` value
Exceptions			
cppvmExceptionForbiddenCall	object not initialized, it is not allowed to use it		
Example			

```
int event = PvmHostAdd;
cppvmNotification notify(event, 111, -1, NULL);
cout << notify.getEvent() << endl;
```

Table 11.443: cppvmNotification: `getMsgTag`

`getMsgTag`			cppvmNotification
get the initialized value for `msgTag`			
Synopsis			
int getMsgTag();			
Parameter			
Parameter	Type	Default	Description
int	return		initialized `msgTag` value
Exceptions			
cppvmExceptionForbiddenCall	object not initialized, it is not allowed to use it		
Example			

```
int msgTag = 444;
cppvmNotification notify(PvmHostAdd, msgTag, -1, NULL);
cout << notify.getMsgTag() << endl;
```

Table 11.444: cppvmNotification: `getNumber`

`getNumber`			cppvmNotification
get the initialized value for `number`			
Synopsis			
int getNumber();			
Parameter			
Parameter	Type	Default	Description
int	return		initialized `number` value
Exceptions			
cppvmExceptionForbiddenCall	object not initialized, it is not allowed to use it		
Example			

```
int number = -1;
cppvmNotification notify(PvmHostAdd, 111, number, NULL);
cout << notify.getNumber() << endl;
```

Table 11.445: cppvmNotification: `getReceivedTids`

`getReceivedTids`			cppvmNotification
Get the notification task or host id's, which were received by the object.			
Synopsis			
cppvmArrayInt getReceivedTids();			
Parameter			
Parameter	Type	Default	Description
`cppvmArrayInt`	return		The meaning of the return values depends on the `event` value in the constructor or in the `set` method: `PvmTaskExit`: Task id of the exited task. Each notification message contains only one task id. If you have initialized the notification object with more than one task id's, you have to receive several messages. `PvmHostDelete`: PVM-daemon task id of the exited host. Each notification message contains only one id. If you have initialized the notification object with more than one daemon task id's, you have to receive several messages. `PvmHostAdd`: List of PVM-daemon task id's of new hosts. The `number` parameter in the constructor or `set` method specifies the number of messages send to the calling process.
Exceptions			

continued on next page...

...continued from previous page	
getReceivedTids	cppvmNotification
cppvmExceptionForbiddenCall	object not initialized, it is not allowed to use it

Example

```
// receive notification message steam (blocking)
cppvmReceiveStream recStrm(CPPvmRBany);
// spawn child process 'calcit'
cppvmSpawnConnection child("calcit");

// notification for killed task
int tids[1];
tids[0] = child.getChildTid();
cppvmNotification notify(PvmTaskExit, 333, 1, tids);

child.killChild();

recStrm >> notify;
cout << "Task with id: " << (notify.getReceivedTids())[0]
<< " killed." << endl;
```

Table 11.446: cppvmNotification: getTids

getTids	cppvmNotification
get the initialized value for tids	

Synopsis

int getTids();

Parameter

Parameter	Type	Default	Description
int	return		initialized tids value

Exceptions

cppvmExceptionForbiddenCall	object not initialized, it is not allowed to use it

Example

```
// notification for killed task
int tids[2];
tids[0]=1234;
tids[1]=5678;
cppvmNotification notify(PvmTaskExit, 555, 2, tids);
cout << notify.getTids() << endl;
```

Table 11.447: cppvmNotification: operator==

operator==			cppvmNotification
test if objects are equal			
Synopsis			
`bool operator== (const cppvmNotification& obj)`			
Parameter			
Parameter	Type	Default	Description
obj	in		object to compare with
bool	return		compare result

Example

```
int tids[1];
tids[0]=4711;
cppvmNotification n1(PvmHostAdd, 111, -1, NULL);
cppvmNotification n2(PvmTaskExit, 222, 1, tids);
...
if(n1==n2){
...
```

Table 11.448: cppvmNotification: operator!=

operator!=			cppvmNotification
test if objects are not equal			
Synopsis			
`bool operator!= (const cppvmNotification& obj)`			
Parameter			
Parameter	Type	Default	Description
`obj`	in		object to compare with
`bool`	return		compare result

Example

```
int tids[1];
tids[0]=4711;
cppvmNotification n1(PvmHostAdd, 111, -1, NULL);
cppvmNotification n2(PvmTaskExit, 222, 1, tids);
...
if(n1!=n2){
...
```

Table 11.449: cppvmNotification: operator<<

operator<<	cppvmNotification
write object on ostream	

continued on next page...

... continued from previous page	
`operator<<`	cppvmNotification

Synopsis
`ostream& operator<<(ostream& strm, cppvmNotification& obj);`

Parameter	Type	Default	Description
`obj`	in		write this object on ostream
`strm`	in		ostream
`ostream`	return		ostream after writing the object

Example

```
cppvmNotification n(PvmHostAdd, 111, -1, NULL);
cout << n;
```

Table 11.450: cppvmNotification: `set`

`set`	cppvmNotification
Initialize object for receiving notification events.	

Synopsis
void set(int event, int msgTag);

Parameter	Type	Default	Description
`event`	in		notification event: `PvmTaskExit:` task exits or is killed `PvmHostDelete:` Host is deleted or crashes `PvmHostAdd:` New host is added
`msgTag`	in		message tag to be used for the notification messages

Example

```
cppvmNotification notify;
notify.set(PvmHostDelete, 222);
```

Discussion

You have to receive the notification messages and read the results with the `getReceivedTids` method later on. In contrast to the other `set` method, this method does not start notification requests. This can be used to receive notifications which have been already requested by another cppvmNotification object.

Table 11.451: cppvmNotification: `set`

`set`	cppvmNotification
Initialize the notification event.	

Synopsis

continued on next page...

. . . continued from previous page	
`set`	cppvmNotification

void set(int event, int msgTag, int number, int* tids);

Parameter

Parameter	Type	Default	Description
`event`	in		notification event: `PvmTaskExit:` task exits or is killed `PvmHostDelete:` Host is deleted or crashes `PvmHostAdd:` New host is added
`msgTag`	in		message tag to be used for the notification messages
`number`	in		The meaning of `number` depends on the `event` value: `PvmTaskExit:` Length of the `tids` array `PvmHostDelete:` Length of the `tids` array `PvmHostAdd:` Number of times to notify. Specifying a `number` of -1 turns on `PvmHostAdd` messages until a future call of the `set` method, a `number` of zero disables them.
`tids`	in		The meaning of `tids` depends on the `event` value: `PvmTaskExit:` Array of length `number` of task id's to be notified about `PvmHostDelete:` Array of length `number` of host id's to be notified about `PvmHostAdd:` Not used

Exceptions

cppvmExceptionBadParameter	could not start notification request

Example

```
cppvmNotification notify;
cppvmSpawnConnection child("calcit");
int tids[1];
tids[0] = child.getChildTid();
notify.set(PvmTaskExit, 222, 1, tids);
```

Discussion

You have to receive the notification messages and read the results with the `getReceivedTids` method later on. By calling the `set` method, the notification object is initialized. You should not use the set method to modify a notification. E.g. if you set a different message tag than in a previous initialization, two notification messages are send to the calling process. The only exception is `PvmHostAdd`. You can modify the number of notification messages by using the set method. Ensure to use the same message tag for this.

11.18.1 cppvmNotificationHostAdd

The class cppvmNotificationHostAdd is used to get notification messages for added hosts. The number of messages is specified in the constructor. By specifying a number of -1 all added hosts are notified, a number greater than 0 specifies a certain number of notification and 0 disables the notification messages. The notification messages contain a list of PVM-daemon task id's of the daemons running on the added hosts.

Table 11.452: cppvmNotificationHostAdd: `constructor`

`constructor`	cppvmNotificationHostAdd
default constructor	
Synopsis	
`cppvmNotificationHostAdd();`	
Example	
`cppvmNotificationHostAdd myNotification;`	
Discussion	
Use the default constructor only if the notification object is initialized later on with the `set` method.	

Table 11.453: cppvmNotificationHostAdd: `constructor`

`constructor`			cppvmNotificationHostAdd
constructor for receiving notifications only			
Synopsis			
`cppvmNotificationHostAdd(int msgTag);`			
Parameter			
Parameter	Type	Default	Description
`msgTag`	in		message tag to be used for the notification messages
Example			

`cppvmNotificationHostAdd notify(111, 5);`
`...`
`cppvmNotificationHostAdd notify2(111);`
`recStream >> notify2;`			

Discussion			
This constructor if for receiving notifications only. It does not start notification requests. You have to receive the notification messages and read the results with the `getAddedHostDaemonTids` method later on.			

Table 11.454: cppvmNotificationHostAdd: constructor

constructor			cppvmNotificationHostAdd
constructor			
Synopsis			
`cppvmNotificationHostAdd(int msgTag, int number);`			
Parameter			
Parameter	Type	Default	Description
`msgTag`	in		message tag to be used for the notification messages
`number`	in		Number of times to notify. Specifying a `number` of -1 turns on `PvmHostAdd` messages until a future call of the `set` method, a `number` greater than 0 specifies a certain number of notification and zero disables them.
Exceptions			
cppvmExceptionBadParameter	could not start notification request		
Example			
`cppvmNotificationHostAdd notify(111, 5);`			
Discussion			
You have to receive the notification messages and read the results with the `getAddedHostDaemonTids` method later on.			

Table 11.455: cppvmNotificationHostAdd: getAddedHostDaemonTids

getAddedHostDaemonTids			cppvmNotificationHostAdd
Get the PVM-daemon task id's, which were received by the object.			
Synopsis			
cppvmArrayInt getAddedHostDaemonTids();			
Parameter			
Parameter	Type	Default	Description
`cppvmArrayInt`	return		List of PVM-daemon task id's of new hosts. The `number` parameter in the constructor or `set` method specifies the number of messages send to the calling process.
Exceptions			
cppvmExceptionForbiddenCall	object not initialized, it is not allowed to use it		
Example			
			continued on next page...

. . . continued from previous page	
`getAddedHostDaemonTids`	cppvmNotificationHostAdd

```
// receive notification message steam (blocking)
cppvmReceiveStream recStrm(CPPvmRBany);
cppvmNotificationHostAdd notify(111);
recStrm >> notify;
cppvmArrayInt hostDaemonTids;
hostDaemonTids = notify.getAddedHostDaemonTids();
cout << "Hosts added:";
for(int i=0;i<hostDaemonTids.numElems();i++){
cout << " name: "
<< pvmConn.getHostName(hostDaemonTids[i])
<< " daemon tid: "
<< hostDaemonTids[i];
}
```

Table 11.456: cppvmNotificationHostAdd: `operator==`

`operator==`			cppvmNotificationHostAdd
test if objects are equal			
Synopsis			
`bool operator== (const cppvmNotificationHostAdd& obj)`			
Parameter			
Parameter	Type	Default	Description
obj	in		object to compare with
bool	return		compare result

Example

```
cppvmNotificationHostAdd n1(111);
cppvmNotificationHostAdd n2(222, 5);
...
if(n1==n2){
...
```

Table 11.457: cppvmNotificationHostAdd: `operator!=`

`operator!=`			cppvmNotificationHostAdd
test if objects are not equal			
Synopsis			
`bool operator!= (const cppvmNotificationHostAdd& obj)`			
Parameter			
Parameter	Type	Default	Description

continued on next page. . .

. . . continued from previous page			
operator!=			cppvmNotificationHostAdd
obj	in		object to compare with
bool	return		compare result
Example			

```
cppvmNotificationHostAdd n1(111, 0);
cppvmNotificationHostAdd n2(222, -1);
...
if(n1!=n2){
...
```

Table 11.458: cppvmNotificationHostAdd: operator<<

operator<<			cppvmNotificationHostAdd
write object on ostream			
Synopsis			
ostream& operator<<(ostream& strm, cppvmNotificationHostAdd& obj);			
Parameter			
Parameter	Type	Default	Description
obj	in		write this object on ostream
strm	in		ostream
ostream	return		ostream after writing the object
Example			

```
cppvmNotificationHostAdd n(333);
cout << n;
```

Table 11.459: cppvmNotificationHostAdd: set

set			cppvmNotificationHostAdd
Initialize the notification event for receive only.			
Synopsis			
void set(int msgTag);			
Parameter			
Parameter	Type	Default	Description
msgTag	in		message tag to be used for the notification messages
Example			

```
cppvmNotificationHostAdd notify;
notify.set(222);
```

Discussion

continued on next page. . .

... continued from previous page	
set	cppvmNotificationHostAdd
In contrast to the other set method, this method does not start notification requests. This can be used to receive notifications which have been already requested by another cppvmNotificationHostAdd object. You have to receive the notification messages and read the results with the getAddedHostDaemonTids method later on.	

Table 11.460: cppvmNotificationHostAdd: set

set			cppvmNotificationHostAdd
Initialize the notification event.			
Synopsis			
void set(int msgTag, int number);			
Parameter			
Parameter	Type	Default	Description
msgTag	in		message tag to be used for the notification messages
number	in		Number of times to notify. Specifying a number of -1 turns on PvmHostAdd messages until a future call of the reset method, a number greater than 0 specifies a certain number of notification and zero disables them.
Exceptions			
cppvmExceptionBadParameter	could not start notification request		
Example			
cppvmNotificationHostAdd notify; notify.set(222, 5);			
Discussion			
You have to receive the notification messages and read the results with the getAddedHostDaemonTids method later on. By calling the set method, the notification object is initialized. You should not use the set method to modify a notification. E.g. if you set a different message tag than in a previous initialization, two notification messages are send to the calling process. To modify the notification use the method reset.			

Table 11.461: cppvmNotificationHostAdd: reset

reset			cppvmNotificationHostAdd
reset the number of notifications			
Synopsis			
void reset(int number);			
Parameter			
Parameter	Type	Default	Description
			continued on next page...

. . . continued from previous page		
`reset`		cppvmNotificationHostAdd
`number`	in	Number of times to notify. Specifying a `number` of -1 turns on `PvmHostAdd` messages until a future call of the `reset` method, a `number` greater than 0 specifies a certain number of notification and zero disables them.

Exceptions

cppvmExceptionBadParameter	could not start notification request

Example

```
cppvmNotificationHostAdd notify(444, 10);
notify.reset(-1);
```

Discussion

Use this method an not `set` to modify the number of notification messages.

11.18.2 cppvmNotificationHostDelete

The class cppvmNotificationHostDelete is used to get notification messages for deleted or crashed hosts. The constructor specifies one or more hosts to get notified about. For each registered host one notification message is send to the calling process. The message contains the PVM-daemon task id of the deleted or crashed host.

Table 11.462: cppvmNotificationHostDelete: `constructor`

`constructor`	cppvmNotificationHostDelete
default constructor	
Synopsis	
`cppvmNotificationHostDelete();`	
Example	
`cppvmNotificationHostDelete myNotification;`	
Discussion	
Use the default constructor only if the notification object is initialized later on with the `set` method.	

Table 11.463: cppvmNotificationHostDelete: `constructor`

`constructor`			cppvmNotificationHostDelete
constructor for receiving notifications only			
Synopsis			
`cppvmNotificationHostDelete(int msgTag);`			
Parameter			
Parameter	Type	Default	Description
`msgTag`	in		message tag to be used for the notification messages
Example			

`int tids[1];`
`tids[0]=1234;`
`cppvmNotificationHostDelete notify(111, 1, tids);`
`...`
`cppvmNotificationHostDelete notify2(111);`
`recStream >> notify2;`			

Discussion			
This constructor if for receiving notifications only. It does not start notification requests. You have to receive notification messages and read the results with the `getDeletedHostDaemonTid` method later on.			

Table 11.464: cppvmNotificationHostDelete: `constructor`

`constructor`			cppvmNotificationHostDelete
constructor			
Synopsis			
`cppvmNotificationHostDelete(int msgTag, int number, int* tids);`			
Parameter			
Parameter	Type	Default	Description
`msgTag`	in		message tag to be used for the notification messages
`number`	in		length of the `tids` array
`tids`	in		array of length `number` of host id's to be notified about
Exceptions			
cppvmExceptionBadParameter	could not start notification request		
Example			

`int tids[1];`
`tids[0]=1234;`
`cppvmNotificationHostDelete notify(111, 1, tids);`			

Discussion			
One notification message is send for each requested id. You have to receive notification messages and read the results with the `getDeletedHostDaemonTid` method later on.			

Table 11.465: cppvmNotificationHostDelete: `constructor`

`constructor`			cppvmNotificationHostDelete
constructor			
Synopsis			
`cppvmNotificationHostDelete(int msgTag, cppvmArrayInt& tids);`			
Parameter			
Parameter	Type	Default	Description
`msgTag`	in		message tag to be used for the notification messages
`tids`	in		array of host id's to be notified about
Exceptions			
cppvmExceptionBadParameter	could not start notification request		
Example			

`cppvmArrayInt tids(1);`
`tids[0]=5678;`
`cppvmNotificationHostDelete notify(111, tids);`			

Discussion			
One notification message is send for each requested id. You have to receive the notification messages and read the results with the `getDeletedHostDaemonTid` method later on.			

Table 11.466: cppvmNotificationHostDelete: `constructor`

`constructor`			cppvmNotificationHostDelete
constructor			
Synopsis			
`cppvmNotificationHostDelete(int msgTag, char *hostName);`			
Parameter			
Parameter	Type	Default	Description
`msgTag`	in		message tag to be used for the notification messages
`hostName`	in		name of the host to notify about
Exceptions			
cppvmExceptionBadParameter	could not start notification request		
cppvmExceptionBadResult	pvmd is not responding. name is a NULL pointer a parameter is a NULL pointer. could not get task informations of the virtual machine. could not get configuration informations of the virtual machine. no pvm daemon tid found for the given task/host name.		
Example			
`cppvmNotificationHostDelete notify(444, "sun1");`			
Discussion			
You have to receive the notification message and read the result with the `getDeletedHostDaemonTid` method later on.			

Table 11.467: cppvmNotificationHostDelete: `getDeletedHostDaemonTid`

`getDeletedHostDaemonTid`			cppvmNotificationHostDelete
get the PVM-daemon task id of the deleted host			
Synopsis			
int getDeletedHostDaemonTid();			
Parameter			
Parameter	Type	Default	Description
`int`	return		PVM-daemon task id of a exited host.
Exceptions			
cppvmExceptionForbiddenCall	object not initialized, it is not allowed to use it		
Example			

continued on next page...

...continued from previous page	
`getDeletedHostDaemonTid`	cppvmNotificationHostDelete

```
// receive notification message steam (blocking)
cppvmReceiveStream recStrm(CPPvmRBany);
cppvmNotificationHostDelete notify(333, "lin1");
recStrm >> notify;
cout << "Host with PVM daemon tid: "
<< notify.getDeletedHostDaemonTid()
<< " was deleted." << endl;
```

Discussion

Each notification message contains only one id. If you have initialized the notification object with more than one daemon task id's, you have to receive several messages.

Table 11.468: cppvmNotificationHostDelete: `operator==`

`operator==`	cppvmNotificationHostDelete
test if objects are equal	

Synopsis

`bool operator== (const cppvmNotificationHostDelete& obj)`

Parameter

Parameter	Type	Default	Description
obj	in		object to compare with
bool	return		compare result

Example

```
cppvmNotificationHostDelete n1(111, "lin1");
cppvmNotificationHostDelete n2(222, "lin2");
...
if(n1==n2){
...
```

Table 11.469: cppvmNotificationHostDelete: `operator!=`

`operator!=`	cppvmNotificationHostDelete
test if objects are not equal	

Synopsis

`bool operator!= (const cppvmNotificationHostDelete& obj)`

Parameter

Parameter	Type	Default	Description
`obj`	in		object to compare with
`bool`	return		compare result

Example

continued on next page...

...continued from previous page	
`operator!=`	cppvmNotificationHostDelete

```
cppvmNotificationHostDelete n1(111, "lin1");
cppvmNotificationHostDelete n2(222, "lin2");
...
if(n1!=n2){
...
```

Table 11.470: cppvmNotificationHostDelete: `operator<<`

`operator<<`	cppvmNotificationHostDelete
write object on ostream	

Synopsis

```
ostream& operator<<(ostream& strm, cppvmNotificationHostDelete& obj);
```

Parameter

Parameter	Type	Default	Description
`obj`	in		write this object on ostream
`strm`	in		ostream
`ostream`	return		ostream after writing the object

Example

```
cppvmNotificationHostDelete n(555, "sun1");
cout << n;
```

Table 11.471: cppvmNotificationHostDelete: `set`

`set`	cppvmNotificationHostDelete
Initialize the notification event for receive only.	

Synopsis

void set(int msgTag);

Parameter

Parameter	Type	Default	Description
`msgTag`	in		message tag to be used for the notification messages

Example

```
cppvmNotificationHostDelete notify;
notify.set(222);
```

Discussion

In contrast to the other `set` method, this method does not start notification requests. This can be used to receive notifications which have been already requested by another cppvmNotificationHostDelete object. You have to receive the notification messages and read the results with the `getDeletedHostDaemonTid` method later on.

Table 11.472: cppvmNotificationHostDelete: set

set			cppvmNotificationHostDelete
Initialize the notification event.			
Synopsis			
void set(int msgTag, int number, int* tids);			
Parameter			
Parameter	Type	Default	Description
`msgTag`	in		message tag to be used for the notification messages
`number`	in		length of the `tids` array
`tids`	in		array of length `number` of host id's to be notified about
Exceptions			
cppvmExceptionBadParameter	could not start notification request		
Example			

```
cppvmNotificationHostDelete notify;
int tids[1];
tids[0] = 5678;
notify.set(222, 1, tids);
```

Discussion

You have to receive the notification messages and read the results with the `getDeletedHostDaemonTid` method later on. By calling the `set` method, the notification object is initialized. You should not use the set method to modify a notification. E.g. if you set a different message tag than in a previous initialization, two notification messages are send to the calling process.

Table 11.473: cppvmNotificationHostDelete: set

set			cppvmNotificationHostDelete
Initialize the notification event.			
Synopsis			
void set(int msgTag, int number, cppvmArrayInt& tids);			
Parameter			
Parameter	Type	Default	Description
`msgTag`	in		message tag to be used for the notification messages
`tids`	in		array of host id's to be notified about
Exceptions			
cppvmExceptionBadParameter	could not start notification request		
Example			

```
cppvmNotificationHostDelete notify;
cppvmArrayInt tids(1);
tids[0]=1234;
notify.set(222, tids);
```

continued on next page...

...continued from previous page	
`set`	cppvmNotificationHostDelete

Discussion
You have to receive the notification messages and read the results with the `getDeletedHostDaemonTid` method later on. By calling the `set` method, the notification object is initialized. You should not use the set method to modify a notification. E.g. if you set a different message tag than in a previous initialization, two notification messages are send to the calling process.

Table 11.474: cppvmNotificationHostDelete: `set`

`set`	cppvmNotificationHostDelete
Initialize the notification event.	

Synopsis
void set(int msgTag, char *hostName);

Parameter			
Parameter	Type	Default	Description
`msgTag`	in		message tag to be used for the notification messages
`hostName`	in		name of the host to notify about

Exceptions	
cppvmExceptionBadParameter	could not start notification request
cppvmExceptionBadResult	pvmd is not responding.
	name is a NULL pointer
	a parameter is a NULL pointer.
	could not get task informations of the virtual machine.
	could not get configuration informations of the virtual machine.
	no pvm daemon tid found for the given task/host name.

Example

```
cppvmNotificationHostDelete notify;
notify.set(555, "sun1");
```

Discussion
You have to receive the notification message and read the result with the `getDeletedHostDaemonTid` method later on. By calling the `set` method, the notification object is initialized. You should not use the set method to modify a notification. E.g. if you set a different message tag than in a previous initialization, two notification messages are send to the calling process.

11.18.3 cppvmNotificationTaskExit

The class cppvmNotificationTaskExit is used to get notification messages for exited or killed PVM processes. The constructor is called with one or more process id's to get notified about. For each of the registered process id's one notification message is send to the calling process. A message contains the task id of one exited or killed process.

Table 11.475: cppvmNotificationTaskExit: `constructor`

`constructor`	cppvmNotificationTaskExit
default constructor	
Synopsis	
`cppvmNotificationTaskExit();`	
Example	
`cppvmNotificationTaskExit myNotification;`	
Discussion	
Use the default constructor only if the notification object is initialized later on with the `set` method.	

Table 11.476: cppvmNotificationTaskExit: `constructor`

`constructor`			cppvmNotificationTaskExit
constructor for receiving notifications only			
Synopsis			
`cppvmNotificationTaskExit(int msgTag);`			
Parameter			
Parameter	Type	Default	Description
`msgTag`	in		message tag to be used for the notification messages
Example			

`cppvmSpawnConnection child("slave");`
`cppvmNotificationTaskExit notify(666, child);`
`...`
`cppvmNotificationTaskExit notify2(666);`
`recStream » notify2;`			

Discussion			
This constructor if for receiving notifications only. It does not start notification requests. You have to receive the notification message and read the result with the `getExitTid` method later on.			

Table 11.477: cppvmNotificationTaskExit: `constructor`

`constructor`			cppvmNotificationTaskExit
constructor			
Synopsis			
`cppvmNotificationTaskExit(int msgTag, cppvmSpawnConnection& child);`			
Parameter			
Parameter	Type	Default	Description
`msgTag`	in		message tag to be used for the notification messages
`child`	in		get notification for this task
Exceptions			
cppvmExceptionBadParameter	could not start notification request		
Example			
`cppvmSpawnConnection child("slave");` `cppvmNotificationTaskExit notify(666, child);`			
Discussion			
You have to receive the notification message and read the result with the `getExitTid` method later on.			

Table 11.478: cppvmNotificationTaskExit: `constructor`

`constructor`			cppvmNotificationTaskExit
constructor			
Synopsis			
`cppvmNotificationTaskExit(int msgTag, cppvmSpawnMultiConnection& children);`			
Parameter			
Parameter	Type	Default	Description
`msgTag`	in		message tag to be used for the notification messages
`children`	in		get notifications for these tasks
Exceptions			
cppvmExceptionBadParameter	could not start notification request		
Example			
`cppvmSpawnMultiConnection children("hi-there", 5);` `cppvmNotificationTaskExit notify(777, children);`			
Discussion			
One notification message is send for each requested task id. You have to receive the notification messages and read the results with the `getExitTid` method later on.			

Table 11.479: cppvmNotificationTaskExit: `constructor`

`constructor`			cppvmNotificationTaskExit
constructor			
Synopsis			
`cppvmNotificationTaskExit(int msgTag, cppvmTask& task);`			
Parameter			
Parameter	Type	Default	Description
`msgTag`	in		message tag to be used for the notification messages
`task`	in		get notification for this task
Exceptions			
cppvmExceptionBadParameter	could not start notification request		
Example			

`cppvmSpawnConnection child("slave");`
`cppvmTask task = child.getChildTask();`
`cppvmNotificationTaskExit notify(666, task);`			

Discussion			
You have to receive the notification message and read the result with the `getExitTid` method later on.			

Table 11.480: cppvmNotificationTaskExit: `constructor`

`constructor`			cppvmNotificationTaskExit
constructor			
Synopsis			
`cppvmNotificationTaskExit(int msgTag, int number, int* tids);`			
Parameter			
Parameter	Type	Default	Description
`msgTag`	in		message tag to be used for the notification messages
`number`	in		length of the `tids` array
`tids`	in		array of length `number` of task id's to be notified about
Exceptions			
cppvmExceptionBadParameter	could not start notification request		
Example			

`int tids[1];`
`tids[0]=1234;`
`cppvmNotificationTaskExit notify(111, 1, tids);`			

Discussion			
One notification message is send for each requested task id. You have to receive the notification messages and read the results with the `getExitTid` method later on.			

Table 11.481: cppvmNotificationTaskExit: constructor

constructor			cppvmNotificationTaskExit
constructor			
Synopsis			
`cppvmNotificationTaskExit(int msgTag, cppvmArrayInt& tids);`			
Parameter			
Parameter	Type	Default	Description
`msgTag`	in		message tag to be used for the notification messages
`tids`	in		array of task id's to be notified about
Exceptions			
cppvmExceptionBadParameter	could not start notification request		
Example			

`cppvmArrayInt tids(1);`
`tids[0]=5678;`
`cppvmNotificationTaskExit notify(111, tids);`			

Discussion			
One notification message is send for each requested task id. You have to receive the notification messages and read the results with the `getExitTid` method later on.			

Table 11.482: cppvmNotificationTaskExit: constructor

constructor			cppvmNotificationTaskExit
constructor			
Synopsis			
`cppvmNotificationTaskExit(int msgTag, char *taskName);`			
Parameter			
Parameter	Type	Default	Description
`msgTag`	in		message tag to be used for the notification messages
`taskName`	in		name of the task to notify about
Exceptions			
cppvmExceptionBadParameter	could not start notification request		
cppvmExceptionBadResult	pvmd is not responding. name is a NULL pointer a parameter is a NULL pointer. could not get task informations of the virtual machine.		
Example			
`cppvmNotificationTaskExit notify(444, "counter");`			
Discussion			
You have to receive the notification message and read the result with the `getExitTid` method later on.			

Table 11.483: cppvmNotificationTaskExit: `getExitTid`

`getExitTid`			cppvmNotificationTaskExit
get the task id of the killed or exited task			
Synopsis			
int getExitTid();			
Parameter			
Parameter	Type	Default	Description
`int`	return		task id of the exited or killed task.
Exceptions			
cppvmExceptionForbiddenCall	object not initialized, it is not allowed to use it		
Example			

```
// receive notification message steam (blocking)
cppvmReceiveStream recStrm(CPPvmRBany);
cppvmNotificationTaskExit notify(333, "calcit");
recStrm >> notify;
cout << "Task with tid: "
<< notify.getExitTid()
<< " ended." << endl;
```

Discussion

Each notification message contains only one id. If you have initialized the notification object with more than one task id's, you have to receive several messages.

Table 11.484: cppvmNotificationTaskExit: `operator==`

`operator==`			cppvmNotificationTaskExit
test if objects are equal			
Synopsis			
`bool operator== (const cppvmNotificationTaskExit& obj)`			
Parameter			
Parameter	Type	Default	Description
obj	in		object to compare with
bool	return		compare result
Example			

```
cppvmNotificationTaskExit n1(111, "counter1");
cppvmNotificationTaskExit n2(222, "counter2");
...
if(n1==n2){
...
```

Table 11.485: cppvmNotificationTaskExit: operator!=

operator!=			cppvmNotificationTaskExit
test if objects are not equal			
Synopsis			
bool operator!= (const cppvmNotificationTaskExit& obj)			
Parameter			
Parameter	Type	Default	Description
obj	in		object to compare with
bool	return		compare result
Example			

```
cppvmNotificationTaskExit n1(111, "counter1");
cppvmNotificationTaskExit n2(222, "counter2");
...
if(n1!=n2){
...
```

Table 11.486: cppvmNotificationTaskExit: operator<<

operator<<			cppvmNotificationTaskExit
write object on ostream			
Synopsis			
ostream& operator<<(ostream& strm, cppvmNotificationTaskExit& obj);			
Parameter			
Parameter	Type	Default	Description
obj	in		write this object on ostream
strm	in		ostream
ostream	return		ostream after writing the object
Example			

```
cppvmNotificationTaskExit n(555, "calcit");
cout << n;
```

Table 11.487: cppvmNotificationTaskExit: set

set			cppvmNotificationTaskExit
Initialize the notification event for receive only.			
Synopsis			
void set(int msgTag);			
Parameter			
Parameter	Type	Default	Description
msgTag	in		message tag to be used for the notification messages
Example			

continued on next page...

... continued from previous page	
set	cppvmNotificationTaskExit

```
cppvmNotificationTaskExit notify;
notify.set(888);
```

Discussion
In contrast to the other `set` method, this method does not start notification requests. This can be used to receive notifications which have been already requested by another cppvmNotificationTaskExit object. You have to receive the notification messages and read the results with the `getReceivedTids` method later on.

Table 11.488: cppvmNotificationTaskExit: `set`

set	cppvmNotificationTaskExit
Initialize the notification event.	

Synopsis
void set(int msgTag, cppvmSpawnConnection& child);

Parameter			
Parameter	Type	Default	Description
`msgTag`	in		message tag to be used for the notification messages
`child`	in		get notification for this task

Exceptions	
cppvmExceptionBadParameter	could not start notification request

Example

```
cppvmSpawnConnection child("slave");
cppvmNotificationTaskExit notify;
notify.set(888, child);
```

Discussion
You have to receive the notification message and read the result with the `getExitTid` method later on. By calling the `set` method, the notification object is initialized. You should not use the set method to modify a notification. E.g. if you set a different message tag than in a previous initialization, two notification messages are send to the calling process.

Table 11.489: cppvmNotificationTaskExit: `set`

set	cppvmNotificationTaskExit
Initialize the notification event.	

Synopsis
void set(int msgTag, cppvmSpawnMultiConnection& children);

Parameter			
Parameter	Type	Default	Description

continued on next page...

... continued from previous page			
`set`			cppvmNotificationTaskExit
`msgTag`	in		message tag to be used for the notification messages
`children`	in		get notifications for these tasks
Exceptions			
cppvmExceptionBadParameter	could not start notification request		
Example			

```
cppvmSpawnMultiConnection children("gimme5", 5);
cppvmNotificationTaskExit notify;
notify.set(222, children);
```

Discussion

You have to receive the notification messages and read the results with the `getExitTid` method later on. By calling the `set` method, the notification object is initialized. You should not use the set method to modify a notification. E.g. if you set a different message tag than in a previous initialization, two notification messages are send to the calling process.

Table 11.490: cppvmNotificationTaskExit: `set`

`set`			cppvmNotificationTaskExit
Initialize the notification event.			
Synopsis			
void set(int msgTag, cppvmTask& task);			
Parameter			
Parameter	Type	Default	Description
`msgTag`	in		message tag to be used for the notification messages
`task`	in		get notification for this task
Exceptions			
cppvmExceptionBadParameter	could not start notification request		
Example			

```
cppvmSpawnConnection child("hello");
cppvmTask task = child.getChildTask();
cppvmNotificationTaskExit notify;
notify.set(999, task);
```

Discussion

You have to receive the notification message and read the result with the `getExitTid` method later on. By calling the `set` method, the notification object is initialized. You should not use the set method to modify a notification. E.g. if you set a different message tag than in a previous initialization, two notification messages are send to the calling process.

Table 11.491: cppvmNotificationTaskExit: set

set			cppvmNotificationTaskExit
Initialize the notification event.			
Synopsis			
void set(int msgTag, int number, int* tids);			
Parameter			
Parameter	Type	Default	Description
`msgTag`	in		message tag to be used for the notification messages
`number`	in		length of the `tids` array
`tids`	in		array of length `number` of task id's to be notified about
Exceptions			
cppvmExceptionBadParameter	could not start notification request		
Example			

```
cppvmNotificationTaskExit notify;
int tids[1];
tids[0] = 5678;
notify.set(222, 1, tids);
```

Discussion

You have to receive the notification messages and read the results with the `getExitTid` method later on. By calling the `set` method, the notification object is initialized. You should not use the set method to modify a notification. E.g. if you set a different message tag than in a previous initialization, two notification messages are send to the calling process.

Table 11.492: cppvmNotificationTaskExit: set

set			cppvmNotificationTaskExit
Initialize the notification event.			
Synopsis			
void set(int msgTag, int number, cppvmArrayInt& tids);			
Parameter			
Parameter	Type	Default	Description
`msgTag`	in		message tag to be used for the notification messages
`tids`	in		array of task id's to be notified about
Exceptions			
cppvmExceptionBadParameter	could not start notification request		
Example			

```
cppvmNotificationTaskExit notify;
cppvmArrayInt tids(1);
tids[0]=1234;
notify.set(222, tids);
```

continued on next page...

... continued from previous page

set	cppvmNotificationTaskExit

Discussion

You have to receive the notification messages and read the results with the `getExitTid` method later on. By calling the `set` method, the notification object is initialized. You should not use the set method to modify a notification. E.g. if you set a different message tag than in a previous initialization, two notification messages are send to the calling process.

Table 11.493: cppvmNotificationTaskExit: `set`

set	cppvmNotificationTaskExit
Initialize the notification event.	

Synopsis

void set(int msgTag, char *taskName);

Parameter

Parameter	Type	Default	Description
`msgTag`	in		message tag to be used for the notification messages
`taskName`	in		name of the task to notify about

Exceptions

cppvmExceptionBadParameter	could not start notification request
cppvmExceptionBadResult	pvmd is not responding.
	name is a NULL pointer
	a parameter is a NULL pointer.
	could not get task informations of the virtual machine.

Example

```
cppvmNotificationTaskExit notify;
notify.set(555, "calcit");
```

Discussion

You have to receive the notification message and read the result with the `getExitTid` method later on. By calling the `set` method, the notification object is initialized. You should not use the set method to modify a notification. E.g. if you set a different message tag than in a previous initialization, two notification messages are send to the calling process.

11.19 cppvmException

In CPPVM the complete error handling is done by C++ exceptions. Every CPPVM error throws an exception object which can be caught by the user and handled individually. The exception object describes the error type (exception class name), the module (file), class and method in which the error occurred, and it contains also an error message. The exception objects can be of the classes:

- cppvmException base class for all exceptions
- cppvmExceptionForbiddenCall (currently) illegal call of a method or function
- cppvmExceptionBadParameter a parameter has a invalid value
- cppvmExceptionOutOfRange a parameter is out of range
- cppvmExceptionBadResult a function/method called by CPPVM returned an error

All classes implement the same methods. They only differ in the name of the classes which specify the error type. Therefore only cppvmException is described, use the other classes equivalent. The base class cppvmException can also be used to catch all CPPVM errors. See section 10.1 how to use exception handling within CPPVM programs.

Table 11.494: cppvmException: `constructor`

`constructor`			cppvmException
constructor			
Synopsis			
`cppvmException(char *mo, char *cl, char *me, char *er);`			
Parameter			
Parameter	Type	Default	Description
`mo`	in		module, in which the error occurred
`cl`	in		class, in which the error occurred
`me`	in		method, in which the error occurred
`er`	in		error message
Example			
`cppvmException myExcep("myfile.cpp", "myCppvmClass", "constructor", "no memory");`			

Table 11.495: cppvmException: `copy constructor`

`copy constructor`	cppvmException
copy constructor	
	continued on next page...

... continued from previous page			
`copy constructor`			cppvmException
Synopsis			
`cppvmException(const cppvmException& obj);`			
Parameter			
Parameter	Type	Default	Description
`obj`	in		object to copy
Example			
`cppvmException myExcep = previousDefinedExcep;`			

Table 11.496: cppvmException: `CPPvmTry`

`CPPvmTry`			cppvmException
All exception classes in this module are handled with this macro. Other exceptions are not handled.			
Synopsis			
`CPPvmTry(expression)`			
Parameter			
Parameter	Type	Default	Description
`expression`	in		expression to evaluate
Example			
`CPPvmTry(myCppvmObject.myMethod())`			
Discussion			
If the method `myMethod()` of the object `myCppvmObject` throws an CPPvm exception object, it will be caught. The error message is printed on `cerr`. Other exceptions are not handled and must be caught somewhere else.			

Table 11.497: cppvmException: `CPPvmTryAll`

`CPPvmTryAll`			cppvmException
Global exception handling. All exceptions will be handled.			
Synopsis			
`CPPvmTryAll(expression)`			
Parameter			
Parameter	Type	Default	Description
`expression`	in		expression to evaluate
Example			
`CPPvmTryAll(myCppvmObject.myMethod())`			
Discussion			
If the method `myMethod()` of the object `myCppvmObject` throws any exception, it will be caught. The error message is printed on `cerr`.			

Table 11.498: cppvmException: CPPvmTryAndErr

CPPvmTryAndErr			cppvmException
Global exception handling. All exceptions will be handled. If an exception occurs, the error_expression' is evaluated.			
Synopsis			
CPPvmTryAndErr(check_expression, error_expression)			
Parameter			
Parameter	Type	Default	Description
check_expression	in		expression to evaluate
error_expression	in		expression to evaluate if an exception occurs
Example			
CPPvmTryAndErr(myCppvmObject.myMethod(),print("Exception occurred."))			
Discussion			
If the method myMethod() of the object myCppvmObject throws an exception object, the print function will be called.			

Table 11.499: cppvmException: CPPvmTryAndSpecificErr

CPPvmTryAndSpecificErr			cppvmException
Global exception handling. All exceptions will be handled. If the exception is a CPPvm exception the cppvm_error_expression' is evaluated. If the exception is not part of CPPvm, the error_expression' is called.			
Synopsis			
CPPvmTryAndSpecificErr(check_expression, cppvm_error_expression, error_expression)			
Parameter			
Parameter	Type	Default	Description
check_expression	in		expression to evaluate
cppvm_error_expression	in		expression to evaluate if an CPPvm exception occurs
error_expression	in		expression to evaluate if an other exception occurs
Example			
CPPvmTryAndSpecificErr(myCppvmObject.myMethod(),print("CPPvm exception occurred."), print("Other exception occurred."))			
Discussion			
If the method myMethod() of the object myCppvmObject throws an CPPvm exception object, the print("CPPvm exception occurred") function will be called. If the method throws an exception which is not part of CPPvm, print("Other exception occurred") function is evaluated.			

Table 11.500: cppvmException: `CPPvmWarn`

`CPPvmWarn`			cppvmException
Print a warning message to `cerr`.			
Synopsis			
`CPPvmWarn(module_name, class_name, method_name, msg)`			
Parameter			
Parameter	Type	Default	Description
`module_name`	in		module name, in which the error occurs
`class_name`	in		class name, in which the error occurs
`method_name`	in		method name, in which the error occurs
`msg`	in		error message
Example			
`CPPvmWarn("myFile.cpp", "myClass", "myMethod", "Bad style! All names have the prefix 'my'!")`			

Table 11.501: cppvmException: `getClassName`

`getClassName`			cppvmException
get the class name as string			
Synopsis			
`char *getClassName();`			
Parameter			
Parameter	Type	Default	Description
`char *`	return		class name
Example			
`cout << myExcep.getClassName();`			

Table 11.502: cppvmException: `getErrMsg`

`getErrMsg`			cppvmException
get the error message as string			
Synopsis			
`char *getErrMsg();`			
Parameter			
Parameter	Type	Default	Description
`char *`	return		error message
Example			
`cout << myExcep.getErrMsg();`			

Table 11.503: cppvmException: getMethodName

getMethodName			cppvmException
get the method name as string			
Synopsis			
`char *getMethodName();`			
Parameter			
Parameter	Type	Default	Description
`char *`	return		method name
Example			
`cout << myExcep.getMethodName();`			

Table 11.504: cppvmException: getModuleName

getModuleName			cppvmException
get the module name as string			
Synopsis			
`char *getModuleName();`			
Parameter			
Parameter	Type	Default	Description
`char *`	return		module name
Example			
`cout << myExcep.getModuleName();`			

Table 11.505: cppvmException: operator=

operator=			cppvmException
assign object			
Synopsis			
`cppvmException& operator= (const cppvmException& obj)`			
Parameter			
Parameter	Type	Default	Description
`obj`	in		object to assign
`cppvmException`	return		this object
Example			

```
cppvmException e1("myfile.cpp", "myCppvmClass", "constructor", "no
memory");
cppvmException e2("myfile.cpp", "myCppvmClass", "setValue", "common
error");
...
e1 = e2;
```

Table 11.506: cppvmException: operator==

operator==			cppvmException
test if objects are equal			
Synopsis			
`bool operator== (const cppvmException& obj)`			
Parameter			
Parameter	Type	Default	Description
obj	in		object to compare with
bool	return		compare result
Example			

```
cppvmException e1("myfile.cpp", "myCppvmClass", "constructor", "no
memory");
cppvmException e2("myfile.cpp", "myCppvmClass", "setValue", "common
error");
...
if(e1==e2){ ...
```

Table 11.507: cppvmException: operator!=

operator!=			cppvmException
test if objects are not equal			
Synopsis			
`bool operator!= (const cppvmException& obj)`			
Parameter			
Parameter	Type	Default	Description
obj	in		object to compare with
bool	return		compare result
Example			

```
cppvmException e1("myfile.cpp", "myCppvmClass", "constructor", "no
memory");
cppvmException e2("myfile.cpp", "myCppvmClass", "setValue", "common
error");
...
if(e1!=e2){ ...
```

Table 11.508: cppvmException: operator<<

operator<<	cppvmException
write object on ostream	
Synopsis	
`ostream& operator<<(ostream& strm, cppvmException& obj);`	

continued on next page...

. . . continued from previous page	
`operator<<`	cppvmException

Parameter			
Parameter	Type	Default	Description
`obj`	in		write this object on ostream
`strm`	in		ostream
`ostream`	return		ostream after writing the object

Example

```
cppvmException e("myfile.cpp", "myCppvmClass", "setValue", "common
error");
cout << e;
```

Chapter 12

Examples

There are some examples to give you a better understanding of the CPPvm possibilities:

access	access standard CPPvm message classes
adminis	administrator which spawns two servers and some applications; explains also pvm contexts and the exception handling (see section 4.6)
aramis	group broadcast (see section 4.4)
array	array size and length examples
ba	use the function `cppvmTransferBool`
big	user written CPPvm class derived from user written base class (see chapter 8)
bool	use CPPvm bool classes
catch	redirection of children stdout/stderr output
cham	use different message tags for each object
coll	collections: use, sort, send/receive, distributed
conn	message passing without spawning; two processes connect to pvm and pass messages (start 'conn' and 'conn2' one after the other very fast)
cook	message mailbox example with multiple instances (see chapter 6)
cpptypes	explicit message passing using standard C++ types (see section 4.1)
cppvmvspvm	compare the transfer rates of CPPvm and PVM (see chapter A)
dartagnan2	multicast example (see section 4.5)
dhello	hello world example (with distributed object)
distint	integer object: distributed and send/receive
diststr	string object: distributed and send/receive (see chapter 11)

external	send/receive/distributed transfer of an external (non-CPPVM) classes (see chapter 9)
frontdoor	barrier process synchronization (see section 7.3)
hello	hello world example (see chapter 4)
hiddenexternal	send/receive/distributed transfer of an external (non-CPPVM) class; this examples shows how to handle external classes when their class members can not be accessed directly (see section 9.1)
mbox	message mailbox example
mcast	multi-cast and group broadcast
mini	user written small CPPVM class (see chapter 8)
mutexfunction	example for the mutual exclusion concept using functions (see subsection 7.2.1)
mutexobject	example for the mutual exclusion concept using objects (see subsection 7.2.2)
myclass	user written class/subclass: send/receive, distributed, collection
newtag	change the message tags of CPPVM classes
notify	notification messages example (see section 10.3)
numcrunch	process synchronization example (see section 7.1)
oktoberfest	synchronization example with changing barriers (see section 7.3)
pavec	STL wrapper class example
peter	transfer to far related processes 1 (see section 4.3)
peter2	transfer to far related processes 2 (see section 4.3)
producer	producer/consumer semaphore example (see subsection 7.1.1)
sendrec	several send/receive/forward examples
status	detect failed task and hosts (see section 10.2)
stl	simple STL example; send/receive a vector object (see section 4.2)
stlbitset	send/receive/distributed STL objects: bitset and cppvmStlBitset
stlcomplex	send/receive/distributed STL objects: complex and cppvmStlComplex
stldeque	send/receive/distributed STL objects: deque and cppvmStlDeque (see section 5.1)
stllist	send/receive/distributed STL objects: list and cppvmStlList
stlmap	send/receive/distributed STL objects: map and cppvmStlMap
stlmultimap	send/receive/distributed STL objects: multimap and cppvmStlMultimap
stlpair	send/receive/distributed STL objects: pair and cppvmStlPair

stlpriorityqueue	send/receive/distributed STL objects: priority_queue and cppvmStlPriorityQueue
stlqueue	send/receive/distributed STL objects: queue and cppvmStlQueue
stlset	send/receive/distributed STL objects: set and cppvmStlSet
stlslist	send/receive/distributed STL objects: slist and cppvmStlSlist
stlstack	send/receive/distributed STL objects: stack and cppvmStlStack
stlstring	send/receive/distributed STL objects: string and cppvmStlString
stlvalarray	send/receive/distributed STL objects: valarray and cppvmStlValarray
stlvector	send/receive/distributed STL objects: vector and cppvmStlVector
templ	use C++ templates for message passing (see section 8.1)
terminalA	critical sections semaphore example (see section 7.1)

The examples are part of the CPPVM package. Please refer the files in `examples/h` and `examples/src` for the source code. How to compile and start the examples is also described in the `examples/README` file:

```
>cd $HOME
>cd cppvm/examples
>make all
>make install
```

Execute the examples, e.g. `hello`:

```
>cd $PVM_ROOT/bin/$PVM_ARCH
>hello
```

12.1 access

The programs `access` shows how to access to standard CPPVM message classes. It is very easy to assign message objects to standard C types and vice versa:

access.cpp	program `access`; assign C types and CPPVM classes

The example contains:

- access to CPPVM message classes

12.2 array

The "array" programs `array` and `array2` show the difference between the array length and the array size:

array.cpp	program `array`; spawn program `array2` and call some resize methods
array2.cpp	program `array2`; send array object to `array`

The example contains:

- send and receive an array
- receive array in append mode
- append an array to another array
- resize arrays in different modes
- fill arrays

12.3 bool

The programs `bool` and `bool2` are small examples how to use bool objects:

bool.cpp	program `bool`; spawn program `bool2` and receive cppvmArrayBool and cppvmBool objects
bool2.cpp	program `bool2`; send cppvmArrayBool and cppvmBool objects to `bool`

The example contains:

- send and receive of the standard CPPVM classes cppvmBool and cppvmArrayBool

12.4 ba

The programs `ba` and `ba2` are small examples how to use bool objects:

ba.cpp	program `ba`; spawn program `ba2`; read distributed boolArray object; receive boolArray object
ba2.cpp	program `ba2`; write distributed boolArray object; send boolArray object to `ba`
boolar.{h|cpp}	new user defined class `boolArray`

The example contains:

- implementation of a new user defined pvm class boolArray
- send/receive a boolArray object
- distributed boolArray object

12.5 catch

The "catch" programs `catch` and `catch2` show how to redirect the output of child processes:

catch.cpp	program `catch`; spawn two `catch2` programs and redirect their output
catch2.cpp	program `catch2`; write a message on `cout`

The example contains:

- the redirection of the children output to `cout` and into a file
- the influence of new redirections on existing redirections

12.6 cham

The "cham" programs `cham` and `cham2` show how to use different message tags for each object:

chameleon.{h|cpp}	new pvm class chameleon
cham.cpp	program `cham`; spawn program `cham2`; receive chameleon objects
cham2.cpp	program `cham2`; send objects to `cham`

The example contains:

- implementation of a new user defined pvm class chameleon
- show how to give each chameleon a different message tag by calling the constructor of cppvmObject with a message tag as parameter
- change the object message tag of a chameleon object
- use the class message tag again instead of the object message tag
- send/receive chameleon objects

12.7 coll

The "coll" programs `coll` and `coll2` explain how to use the standard CPPVM class cppvmObjectCollection:

coll.cpp	program `coll`; spawn program `coll2` and use several collection features
coll2.cpp	program `coll2`; receive collection, write in distributed collection

The example contains:

- use collections
- sort collections
- nested collections
- send and receive collections:
  - receive collection and replace old collection
  - append received collection to old collection
- distributed collections

12.8 conn

The "conn" programs `conn` and `conn2` are a examples how to connect to PVM without spawning a process. The programs "conn2" must be started after "conn2" very fast:

conn.cpp	program `conn`; send string object to all other tasks; read distributed string object
conn2.cpp	program `conn2`; receive string object; write distributed string object

The example contains:

- connect to PVM without spawning
- send and receive of the standard CPPVM class cppvmInt
- distributed object of standard CPPVM class cppvmInt

12.9 dhello

The "hello world" programs `dhello` and `dhello2` are small examples how work with distributed pvm objects:

dhello.cpp	program `dhello`; spawn program `dhello2` and use a distributed string object
dhello2.cpp	program `dhello2`; read distributed string object

The example contains:

- distributed standard CPPVM object of class cppvmString
- redirect output of a spawned process

12.10 distint

The "distint" programs `distint` and `distint2` are small examples for a distributed integer object:

distint.cpp	program `distint`; spawn program `distint2` and read a distributed integer object
distint2.cpp	program `distint2`; write distributed integer object

The example contains:

- distributed standard CPPVM integer object of class cppvmInt
- update of the local distributed object by the message mailbox with the standard cppvmInt methods `read` and `write`
- distributed objects can still be used for normal send/receive

12.11 mbox

The "mbox" programs `mbox` and `mbox2` are a small message mailbox examples:

mbox.cpp	program `mbox`; spawn program `mbox2` and read mailbox string object
mbox2.cpp	program `mbox2`; write mailbox string object

The example contains:

- standard CPPvm class cppvmString as mailbox object
- read/write the global debug level variable (for all processes in the current parallel virtual machine)

12.12 mcast

The "mcast" programs `mcast` and `mcast2` are multicast and group broadcast examples. Broadcast and multicast are different techniques. Multicasted objects are sent to a specified collection of processes whereas broadcasted objects are sent to all members of a specified group. Processes can join or leave a group dynamically.

mcast.cpp	program `mcast`; spawn program `mcast2` multicast/broadcast objects
mcast2.cpp	program `mcast2`; receive broadcasted/multicasted objects

The example contains:

- spawn multiple instances of a program (children)
- multicast to all children
- multicast to a subset of 'tagged' children
- receive array:
  - replace old array by received array
  - append received array to old array
- group broadcast to all group members

12.13 myclass

The "myclass" programs `myclass` and `myclass2` show the possibilities of user defined pvm classes:

group.{h|cpp}	new user defined pvm class groupClass; used as part of userClass
user.{h|cpp}	new user defined pvm class userClass
userkey.{h|cpp}	new user defined pvm class userKeyClass; subclass of userClass
myclass.cpp	program `myclass`; spawn program `myclass2`; use new user defined classes
myclass2.cpp	program `myclass2`; use new user defined classes

The example contains:

- implementation of new user defined pvm classes
- use standard pvm classes as parts of the user defined classes
- use user defined classes as parts of other user defined classes
- implementation of subclasses from user defined pvm classes
- send/receive of user defined classes
- distributed user defined classes
- collection of user defined classes
  - sort collection
  - nested collection
  - send/receive collection
- use in/out methods for user defined classes

12.14 newtag

The "newtag" programs `newtag` and `newtag2` are examples how to modify the message tag values for CPPVM classes and the object message tag values during runtime:

newtag.cpp	program `newtag`; spawn program `newtag2` and try to receive string objects
newtag2.cpp	program `newtag2`; send string objects to `newtag`

The example contains:

- send and receive of the standard CPPVM class cppvmString
- modify the message tag of a CPPVM class during runtime
- modify the message tag of an CPPVM object during runtime

12.15 pavec

The programs `pavec` and `pavec2` are examples for a STL wrapper class:

pavec.cpp	program `pavec`; spawn program `pavec2`; read distributed paramVector object; receive paramVector object
pavec2.cpp	program `pavec2`; write distributed paramVector object; send paramVector to `pavec`

The example contains:

- STL wrapper class paramVector; transfer STL objects within a user written class
- distributed paramVector object
- send/receive a paramVector object

12.16 sendrec

The programs `sendrec` and `sendrec2` show the different send, receive and forward possibilities of CPPVM:

sendrec.cpp	program `sendrec`; spawn program `sendrec2` and send/receive/forward pvm objects
sendrec2.cpp	program `sendrec2`; send/receive/forward pvm objects

The example contains:

- blocking/nonblocking send of the standard CPPVM classes
- blocking/nonblocking/timeout receive of the standard CPPVM classes
- forwarding of messages
- receive objects in a different order than they are sent

12.17 stlbitset

The programs `stlbitset` and `stlbitset2` are examples how to use standard template library objects:

stlbitset.cpp	program `stlbitset`; spawn program `stlbitset2`; read distributed cppvmStlBitset object; receive cppvmStlBitset and bitset object
stlbitset2.cpp	program `stlbitset2`; write distributed cppvmStlBitset object; send cppvmStlBitset and bitset object to `stlbitset`

The example contains:

- send/receive a bitset object
- send/receive a cppvmStlBitset object
- distributed cppvmStlBitset object

12.18 stlcomplex

The programs `stlcomplex` and `stlcomplex2` are examples how to use standard template library objects:

stlcomplex.cpp	program `stlcomplex`; spawn program `stlcomplex2`; read distributed cppvmStlComplex object; receive cppvmStlComplex and complex object
stlcomplex2.cpp	program `stlcomplex2`; write distributed cppvmStlComplex object; send cppvmStlComplex and complex object to `stlcomplex`

The example contains:

- send/receive a complex object
- send/receive a cppvmStlComplex object
- distributed cppvmStlComplex object

12.19 stllist

The programs `stllist` and `stllist2` are examples how to use standard template library objects:

stllist.cpp	program `stllist`; spawn program `stllist2`; read distributed cppvmStlList object; receive cppvmStlList and list object
stllist2.cpp	program `stllist2`; write distributed cppvmStlList object; send cppvmStlList and list object to `stllist`

The example contains:

- send/receive a list object
- send/receive a cppvmStlList object
- distributed cppvmStlList object

12.20 stlmap

The programs `stlmap` and `stlmap2` are examples how to use standard template library objects:

stlmap.cpp	program `stlmap`; spawn program `stlmap2`; read distributed cppvmStlMap object; receive cppvmStlMap and map object
stlmap2.cpp	program `stlmap2`; write distributed cppvmStlMap object; send cppvmStlMap and map object to `stlmap`

The example contains:

- send/receive a map object
- send/receive a cppvmStlMap object
- distributed cppvmStlMap object

12.21 stlmultimap

The programs `stlmultimap` and `stlmultimap2` are examples how to use standard template library objects:

stlmultimap.cpp	program `stlmultimap`; spawn program `stlmultimap2`; read distributed cppvmStlMultimap object; receive cppvmStlMultimap and multimap object
stlmultimap2.cpp	program `stlmultimap2`; write distributed cppvmStlMultimap object; send cppvmStlMultimap and multimap object to `stlmultimap`

The example contains:

- send/receive a multimap object
- send/receive a cppvmStlMultimap object
- distributed cppvmStlMultimap object

12.22 stlmultiset

The programs `stlmultiset` and `stlmultiset2` are examples how to use standard template library objects:

stlmultiset.cpp	program `stlmultiset`; spawn program `stlmultiset2`; read distributed cppvmStlMultiset object; receive cppvmStlMultiset and multiset object
stlmultiset2.cpp	program `stlmultiset2`; write distributed cppvmStlMultiset object; send cppvmStlMultiset and multiset object to `stlmultiset`

The example contains:

- send/receive a multiset object
- send/receive a cppvmStlMultiset object
- distributed cppvmStlMultiset object

12.23 stlpair

The programs `stlpair` and `stlpair2` are examples how to use standard template library objects:

stlpair.cpp	program `stlpair`; spawn program `stlpair2`; read distributed cppvmStlPair object; receive cppvmStlPair and pair object
stlpair2.cpp	program `stlpair2`; write distributed cppvmStlPair object; send cppvmStlPair and pair object to `stlpair`

The example contains:

- send/receive a pair object
- send/receive a cppvmStlPair object
- distributed cppvmStlPair object

12.24 stlpriorityqueue

The programs `stlpriorityqueue` and `stlpriorityqueue2` are examples how to use standard template library objects:

stlpriorityqueue.cpp	program `stlpriorityqueue`; spawn program `stlpriorityqueue2`; read distributed cppvmStlPriorityqueue object; receive cppvmStlPriorityqueue and priority_queue object
stlpriorityqueue2.cpp	program `stlpriorityqueue2`; write distributed cppvmStlPriorityqueue object; send cppvmStlPriorityqueue and priority_queue object to `stlpriorityqueue`

The example contains:

- send/receive a priority_queue object
- send/receive a cppvmStlPriorityqueue object
- distributed cppvmStlPriorityqueue object

12.25 stlqueue

The programs `stlqueue` and `stlqueue2` are examples how to use standard template library objects:

stlqueue.cpp	program `stlqueue`; spawn program `stlqueue2`; read distributed cppvmStlQueue object; receive cppvmStlQueue and queue object
stlqueue2.cpp	program `stlqueue2`; write distributed cppvmStlQueue object; send cppvmStlQueue and queue object to `stlqueue`

The example contains:

- send/receive a queue object
- send/receive a cppvmStlQueue object
- distributed cppvmStlQueue object

12.26 stlset

The programs `stlset` and `stlset2` are examples how to use standard template library objects:

stlset.cpp	program `stlset`; spawn program `stlset2`; read distributed cppvm-StlSet object; receive cppvmStlSet and set object
stlset2.cpp	program `stlset2`; write distributed cppvmStlSet object; send cp-pvmStlSet and set object to `stlset`

The example contains:

- send/receive a set object
- send/receive a cppvmStlSet object
- distributed cppvmStlSet object

12.27 stlslist

The programs `stlslist` and `stlslist2` are examples how to use standard template library objects:

stlslist.cpp	program `stlslist`; spawn program `stlslist2`; read distributed cppvmStlSlist object; receive cppvmStlSlist and slist object
stlslist2.cpp	program `stlslist2`; write distributed cppvmStlSlist object; send cppvmStlSlist and slist object to `stlslist`

The example contains:

- send/receive a slist object
- send/receive a cppvmStlSlist object
- distributed cppvmStlSlist object

12.28 stlstack

The programs `stlstack` and `stlstack2` are examples how to use standard template library objects:

stlstack.cpp	program `stlstack`; spawn program `stlstack2`; read distributed cppvmStlStack object; receive cppvmStlStack and stack object
stlstack2.cpp	program `stlstack2`; write distributed cppvmStlStack object; send cppvmStlStack and stack object to `stlstack`

The example contains:

- send/receive a stack object
- send/receive a cppvmStlStack object
- distributed cppvmStlStack object

12.29 stlstring

The programs `stlstring` and `stlstring2` are examples how to use standard template library objects:

stlstring.cpp	program `stlstring`; spawn program `stlstring2`; read distributed cppvmStlString object; receive cppvmStlString and string object
stlstring2.cpp	program `stlstring2`; write distributed cppvmStlString object; send cppvmStlString and string object to `stlstring`

The example contains:

- send/receive a string object
- send/receive a cppvmStlString object
- distributed cppvmStlString object

12.30 stlvalarray

The programs `stlvalarray` and `stlvalarray2` are examples how to use standard template library objects:

stlvalarray.cpp	program `stlvalarray`; spawn program `stlvalarray2`; read distributed cppvmStlValarray object; receive cppvmStlValarray and valarray object
stlvalarray2.cpp	program `stlvalarray2`; write distributed cppvmStlValarray object; send cppvmStlValarray and valarray object to `stlvalarray`

The example contains:

- send/receive a valarray object
- send/receive a cppvmStlValarray object
- distributed cppvmStlValarray object

12.31 stlvector

The programs `stlvector` and `stlvector2` are examples how to use standard template library objects:

stlvector.cpp	program `stlvector`; spawn program `stlvector2`; read distributed cppvmStlVector object; receive cppvmStlVector and vector object
stlvector2.cpp	program `stlvector2`; write distributed cppvmStlVector object; send cppvmStlVector and vector object to `stlvector`

The example contains:

- send/receive a vector object
- send/receive a cppvmStlVector object
- distributed cppvmStlVector object

Appendix A

Data Transfer Rate

A good indicator to measure the performance of a message passing library is the data transfer rate. The following measurement methods will give the answer to two important questions concerning the transfer rate:

- Is CPPVM fast enough for my application?
- How fast is CPPVM compared to PVM?

The first question is addressed by the programs `cppvmspeed` and `cppvmspeed2`. These programs use CPPVM for message passing and are implemented in C++. The second question is answered by the programs `pvmspeed` and `pvmspeed2`: They implement the same measurement methods as the CPPVM programs but are implemented in C using PVM functions. The data transfer rate is measured in megabytes per seconds (MByte/s). The measured time is the processor time used by the programs. All programs are part of the CPPVM examples.

Since `cppvmspeed` and `pvmspeed` implement the same measurement methods, in the following the name `speed` is used synonymous for both of them.

`speed` is called with several parameters:

```
> speed <MinMsgSize> <MaxMsgSize> <MsgSizeStep> <NumMsg>
        <MsgMode> <MsgMode> <Route> <HostSpeed2> <OutFile>
```

The command line arguments are:

`<MinMsgSize>`	The measurement begins with messages of this size (bytes).
`<MaxMsgSize>`	The measurement ends with messages of this size (bytes).
`<MsgSizeStep>`	This size is added to the current message size until `MaxMsgSize` is reached (bytes).
`<NumMsg>`	Number of messages for each measurement.
`<MsgMode>`	message passing mode: `0` explicit message passing using non-blocking send `1` explicit message passing using blocking send `2` message mailbox using messages for synchronization.
`<MsgEncoding>`	message encoding: `0` XDR encoding (PvmDataDefault) `1` data left in place (PvmDataInPlace) `2` no encoding (PvmDataRaw)
`<Route>`	Message routing policy: `0` route messages via the PVM daemon `1` route messages directly between speed and speed2
`<HostSpeed2>`	The child process is spawned on this host.
`<OutFile>`	Name of the result protocol file. Each measurement is printed in one line with two values. The first one is the message size, the second the transfer rate (Megabyte/s).

The process `speed` is started by the user and spawns the process `speed2` on the host `<HostSpeed2>`. As message data `cppvmSpeed` uses an object of class cppvmArrayChar and for handshakes objects of class cppvmInt. `pvmspeed` uses the C types `char*` and `int` instead.

The `<MsgMode>` determines the measured message passing method:

explicit non-blocking (value `0`) send data non-blocking and receive data blocking

explicit blocking (value `1`) send and receive data blocking

message mailbox (value `2`) write and read data to/from the PVM message mailbox

The explicit non-blocking method is show in figure A.1. The `speed` process sends a non-blocking message to `speed2`. `speed2` receives the message blocking and sends the message back non-blocking. Finally `speed` receives the message blocking. This cycle is done `<NumMsg>` for each message size. After that, the measurement timer stops and the data transfer rate is calculated:

$$\frac{\frac{MsgSize}{1024^2} 2NumMsg}{t_i - t_{i-1}}$$

The message size begins with `<MinMsgSize>` and counts up in `<MsgSizeStep>` steps up to `<MaxMsgSize>`. Each measured data transfer rate is stored in the file `<OutFile>`.

Ensure `<NumMsg>` to be large enough (e.g. 10000 or higher) to get precise results: on some architectures the resolution for time measurement is not very fine-grained. This can lead to problems when calculating the transfer rate for a small number of small messages (for example with a resolution of 0.01 seconds there is no difference between 0.014378 seconds and 0.010001 seconds).

The value `1` for `<MsgMode>` will also activate explicit message passing. But this time, both processes are using a blocking send operation. Since PVM has no blocking send operation, `pvmspeed` emulates the blocking send by using an integer handshake. After every send data call, the process receives the handshake blocking[1]. This will slow down the transfer rate due to the additional traffic.

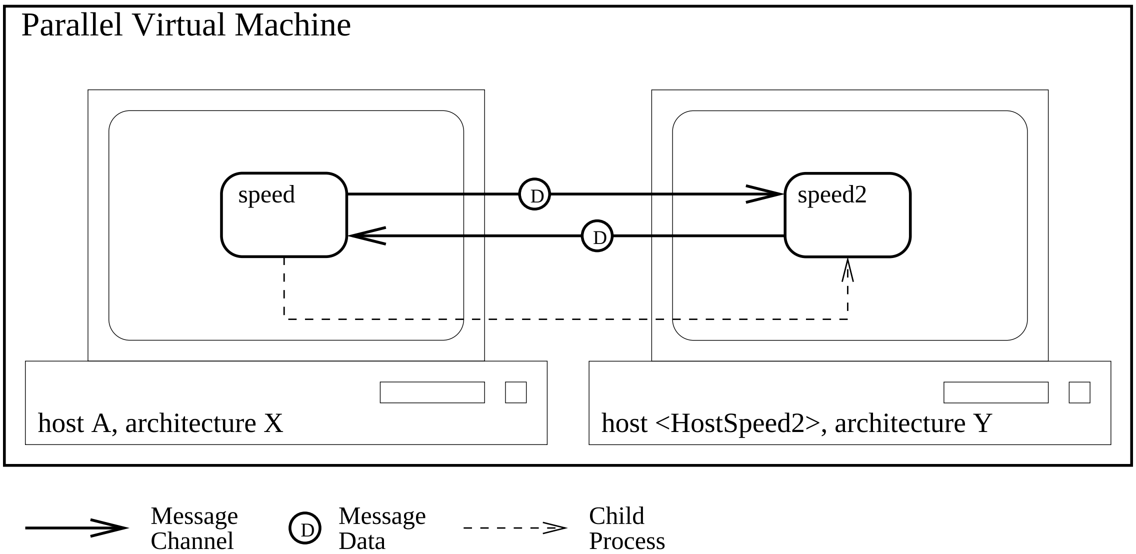


Figure A.1: Data transfer rate measurement for explicit message passing.

The value `2` for `<MsgMode>` will measure the transfer rate for message mailbox data (see figure A.2). The process `speed2` writes data in the message mailbox. After that, `speed` reads the data from the mailbox.

The handshake data ensures that `speed` will wait with reading until `speed2` has written the message. Another handshake will block `speed2` until `speed` has read the message before writing a new one. Both handshakes use a direct message channel between the processes, i.e. explicit message passing. This cycle is done `<NumMsg>` for each message size. After that, the measurement timer stops and the data transfer rate is calculated:

[1] The blocking CPPvm send method works in a similar way.

$$\frac{\frac{MsgSize}{1024^2} NumMsg}{t_i - t_{i-1}}$$

Like for explicit message passing, the message size begins with `<MinMsgSize>` and counts up in `<MsgSizeStep>` steps up to `<MaxMsgSize>`. Each measured data transfer rate is stored in the file `<OutFile>`.

Please notice that this measurement method for distributed objects uses two data transfers: one from `speed2` to the mailbox and one from the mailbox to `speed`. Also additional messages are needed for the handshakes. This will of course reduce the transfer rate.

`cppvmSpeed` uses a distributed object to transfer data from/to the message mailbox. `pvmspeed` use the functions `pvm_putinfo` and `pvm_recvinfo` to exchange data with the mailbox. Since CPPVM internally call these functions for its distributed objects, this is a good method to measure the overhead of CPPVM according to the message mailbox access.

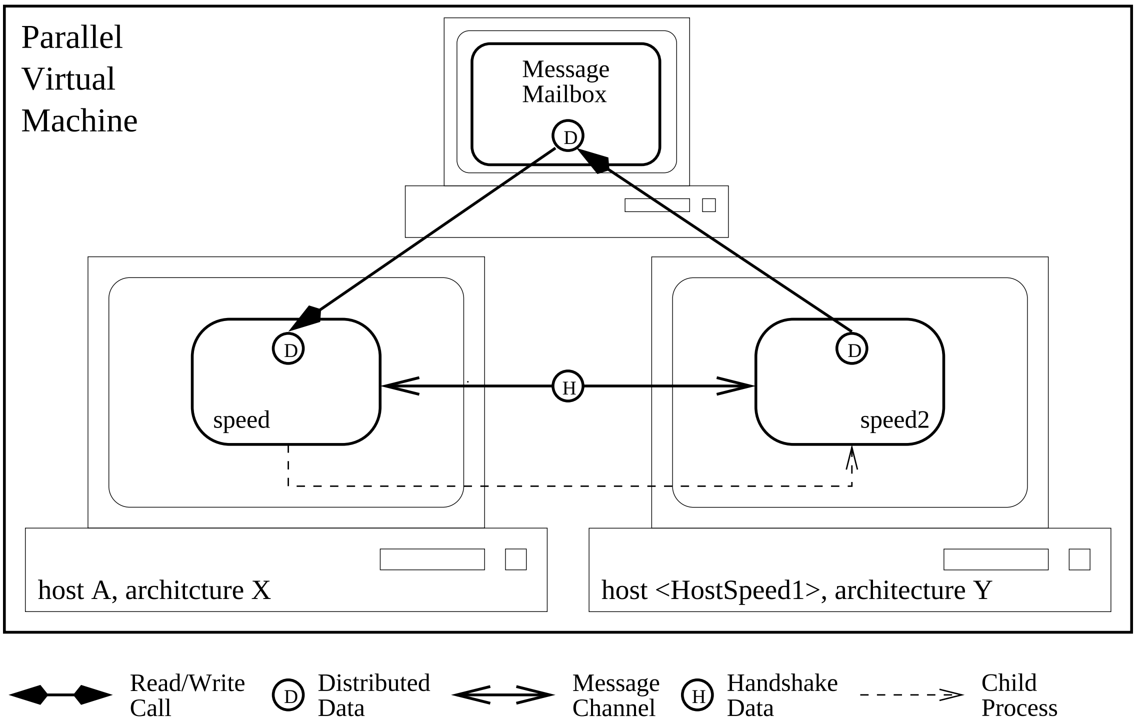


Figure A.2: Data transfer rate measurement for message mailbox data.

There are script files to make the measurement more convenient. On UNIX systems copy the scripts `cppvmvspvm`, `cppvmvspvm2`, and `cppvmvspvmplot.gp` in your executable directory:

```
> cp ~/cppvm/examples/rate/unix/* $PVM_ROOT/bin/$PVM_ARCH
```

The calls of `pvmspeed` and `cppvmspeed` are defined in the file `cppvmvspvm`. Change the host name `sun1` to a host name of your choice and modify message sizes etc. if you like. There is no need to to modify the scripts `cppvmvspvm2` and `cppvmvspvmplot.gp`: they are called internally by `cppvmvspvm`.

For the measurement the semaphore server `cppvms` and the example programs `pvmspeed`, `cppvmspeed`, and `cppvmvspvmmerge` are needed:

```
> cd ~/cppvm
> make
> make install
> cd ~/cppvm/examples/
> make cppvmvspvm
> make install
```

On Windows systems copy the script files `cppvmvspvm.bat`, `cppvmvspvm2.bat`, and `cppvmvspvmplot.gp` from directory `cppvm/examples/rate/WIN32/` in your executable directory `$PVM_ROOT/bin/WIN32`. The calls of `pvmspeed` and `cppvmspeed` are defined in the file `cppvmvspvm.bat`. Like in the UNIX version change the host name `sun1` to a host name of your choice and modify message sizes etc. if you like.

The measurement programs are part of the window package of CPPvm. Copy the semaphore server `cppvms.exe` from `cppvm/bin/WIN32` to `$PVM_ROOT/bin/WIN32`. Copy the executables files `cppvm/examples/bin/WIN32/*` to `$PVM_ROOT/bin/WIN32`.

Now execute the script (UNIX and Windows):

```
> cppvmvspvm
```

This will call the programs `pvmspeed` and `cppvmspeed` several times with different parameters, finally the files `explicit-non-blocking`, `explicit-blocking`, and `message-mailbox` are created. These files contain the results of the measurement: the message size, the transfer rate of PVM, the transfer rate of CPPvm, and the ratio between CPPvm and PVM. A ratio value of 1.0 means CPPvm has the same transfer rate than PVM, a value of 0.8 shows that CPPvm reaches 80 percent of the PVM speed and so on.

If you have `gnuplot` installed, additionally graphical diagrams are created (postscript files `rate.ps` and `ratio.ps`). The file `rate.ps` shows the data transfer rate (MByte/s). The ratio between PVM and CPPvm is shown in `ratio.ps`. To change the output format or the file names modify the `cppvmvspvmplot.gp` file. Examples for measured transfer rates can be found in section A.2.

A.1 Performance

The transfer rate can be influenced by modifying the software and the hardware. On the software side you can modify the message encoding and the message routing. The message encoding scheme can be set by using objects of the send stream class cppvmSendStream. Possible values are:

`PvmDataDefault` XDR encoding (default). This encoding scheme is independent from the host architecture and works every time.

`PvmDataRaw` 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.

`PvmDataInPlace` Leave data in place during packing. The message buffer only contains the sizes and pointers to the items to be sent. When the message is sent the items are copied directly out of the user's memory. This option decreases the number of times a 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. Since the packing and sending is encapsulated in the CPPVM classes, this should work every time.

For example to create a send stream with `PvmDataRaw`, append the value in the parameter list:

```
int
main(){
...
  cppvmConnection pvmConn;

  cppvmSendStream sendStrm(pvmConn, CPPvmSNBparent, PvmDataRaw);
...
}
```

It is also possible to set the message encoding for all send streams of a process. In this case, the local message encodings of all send streams are ignored. To set the global message encoding use the `setOpt` method with the parameters `CPPvmAllPvmDataDefault`, `CPPvmAllPvmDataRaw` and `CPPvmAllPvmDataInPlace`. The settings are only valid for one process, not for all processes in the virtual machine. `CPPvmAllPvmDataDefault` will again use the local settings of the send streams:

```
int
main(){
...
```

```
  cppvmConnection pvmConn;

  // perform all send operations with "PvmDataInPlace"
  pvmConn.setOpt(CPPvmMsgEncoding, CPPvmAllPvmDataInPlace);

...
}
```

The default message routing is from one process to its PVM-daemon, then to the PVM-daemon of the target process and finally from this daemon to the target process itself. You can advise PVM to set up a direct process-to-process link (`PvmRouteDirect`). Once a link is established it persists until the application finishes. If a direct link can not be established because one of the two tasks has requested `PvmDontRoute` or because adequate resources aren't available, then the default route through the PVM daemons is used. You can modify the message routing with the `setOpt` method of the class pvmConnection. This is typically done only once near the beginning of each process.

```
int
main(){
...
  cppvmConnection pvmConn;

  // set direct routing
  pvmConn.setOpt(PvmRoute, PvmRouteDirect);

...
}
```

Each direct route consumes a file descriptor. The number of file descriptors is usually limited for each process. Thus if a process opens to many direct routes to other processes, it can run out of file descriptors. In this case, use the default routing via the PVM-daemons.

Currently the modification of the routing or the encoding will not speed-up the distributed or the message mailbox objects. The transfers of the messages from/to the message mailbox does not allow direct routing. Also, the encoding scheme `PvmDataInPlace` does not work together with the mailbox.

Each object piped into a CPPvm send stream is sent as a separate message. As you can see in section A.2, the transfer rate for small messages is lower than the transfer rate for long messages. This is due to the system overhead which is larger for small messages in comparison to long messages. If you want to send a lot of small messages you can increase the transfer rate by using a cppvmObjectCollection object. Add all of your messages to the list object to generate a larger message and send the list object.

Another way to generate larger messages are new message passing classes with aggregated transfer data. Instead of sending one data after the other:

```
...
  cppvmInt myInt;
  cppvmArrayBool myBoolArray;
  cppvmDouble myDouble;
...
  sendStrm << myInt << myDouble;
  sendStrm << myBoolArray;
...
```

use a new class and send only one object:

```
class myTransferData : public cppvmObject{
...
  cppvmInt myInt;
  cppvmArrayBool myBoolArray;
  cppvmDouble myDouble;
...
  void
  myTransferData::cppvmTransfer()
  {
    myInt.cppvmTransfer();
    myBoolArray.cppvmTransfer();
    myDouble.cppvmTransfer();
  }
...
}
...
  myTransferData myData;
  sendStrm << myData;
...
```

On the other hand, the transfer rate drops down for very long messages. This is due to the huge allocation of system resources. Try to split up your messages, if they are too big. You can figure out the best message size for your system by evaluating the results of the `speed` example.

A general hint to speed-up your programs and increase the data transfer rate is to avoid active waiting for messages. Try to get rid of nonblocking receives if possible. This wastes processor time and network bandwidth. Use a blocking or timeout receive instead.

Another possibility to increase the transfer rate is to use hardware features. PVM supports SMP and shared memory on some systems (see the file `$PVM_ROOT/Readme.mp` for more information). An easy way to increase the transfer rate is to speed-up the network, e.g. from 10 MBit Ethernet to 100 MBit Fast-Ethernet. Currently there are also some approaches to implement PVM for SCI, Myrinet and Atoll high-speed network cards.

Additional performance hints can be found at the web page:

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

A.2 Practical Results

This section gives some practical measurement results. Of course it is recommended to use the `cppvmvspvm` examples described at the beginning of appendix A to get exact results for your configuration. Please read also the description of the measurement methods given there to get a better understanding of how to interpret the values.

The measurements were made on a system with two hosts, each host had one Pentium II processor (Deschutes, 400 MHz, 512 KB cache). The operating system was a desktop LINUX (SuSE 6.4, 2.2.14 kernel). PVM version 3.4.4 and CPPvm version 1.4.0 were used. The network was a standard 10 MBit Ethernet using usual network cards and a 10 MBit Hub. As compiler `gcc` version 2.95.2 with optimize level `-O3` was used. The `pvmspeed` and `cppvmspeed` programs were executed in a shell with no graphical user interface (X-Windows).

The transfer rate depends on the message size. The system overhead, e.g. the latency to establish a transfer, is larger for small messages in comparison to big ones. On the other hand, the transfer rate drops down for very long messages. This is due to the huge allocation of system resources. This causes the typical plot of the graphs: first the transfer rate raises with the message size and later on goes down for large sizes.

The average ratio between CPPvm and PVM is used for performance comparison. As described at the beginning of this appendix, a ratio value of 1.0 means CPPvm has the same transfer rate than PVM, a value 0.8 shows that CPPvm reaches 80 percent of the PVM speed and so on. For some measurements sizes values greater than 1.0 occur. This means that CPPvm is faster than PVM. Since CPPvm is build on top of PVM this can only happen in special cases, e.g. for network collisions or caching effects during the measurement. Of course this also indicates that the overhead of CPPvm compared to PVM is very small.

The results show that the performance of the message mailbox access on one host does not scale very well for large messages compared to explicit message passing. This is due to the message mailbox was at first designed for just storing a few variables and not for extensive data transfer. As you can see in the following examples the network is usually the bottleneck for data transfer between hosts. In this case the performance loss of the mailbox is much less than the performance loss caused by the network. Therefore the gap narrows between message mailbox access and explicit message passing when using networks.

The first measurement was made with encoding scheme `PvmDataDefault` on one host (see figure A.3). The average ratio between CPPvm and PVM is 0.978 for explicit non-blocking message passing, 0.981 for blocking send, and 0.994 for mailbox access.

The next example also measures the transfer rate between processes on one host (see figure A.4). This time two optimizations described in section A.1 are used to speedup the transfer rate: `PvmDataInPlace` leaves data in place during packing (do not copy the data any more) and `PvmRouteDirect` sets up a direct process-to-process link (no communication to the PVM-daemon any more). The results show that these optimization can increase the transfer on one host a lot. This is very useful especially on hosts with more than one processor. The optimizations also increase the gap between PVM and CPPvm. This is especially true for small message sizes and therefore could be a result of caching effects or compiler optimizations[2]. But as the results show (average ratio non-blocking 0.830, blocking 0.875, mailbox 0.991) the total overhead is still acceptable. The graph also shows the described performance problems of the message mailbox access.

Transfer data using an external network is usually slower than between processes on just one host[3]. Figure A.5 shows the results for two processes running on two hosts, the hosts are connected with a 10 MBit Ethernet. Due to the slow network the gap between explicit message passing and the message mailbox decreases. The performance of PVM and CPPvm is very close together (average ratio non-blocking 0.998, blocking 0.998, mailbox 0.998).

The last measurement uses the same configuration as the previous: two hosts and a 10 MBit Ethernet (see figure A.6). Additionally the optimizations `PvmDataInPlace` and `PvmRouteDirect` are applied. Since the network is the bottleneck, the optimizations have no positive effect. In contrast: the transfer rate is lower than in the example above. The average ratio for non-blocking message passing is 0.985, 0.989 for blocking message passing, and 1.027 for message mailbox access.

All results show that the performance of CPPvm is quite close to the performance of PVM. Thus CPPvm allows to take advantage of object-oriented programming without causing to much overhead compared to procedural programming.

---

[2]The size of the executables `pvmspeed` and `pvmspeed2` is smaller than the size of the corresponding executables `cppvmspeed` and `cppvmspeed2`.

[3]Within one host the fast internal memory can be used for transferring data among processes.

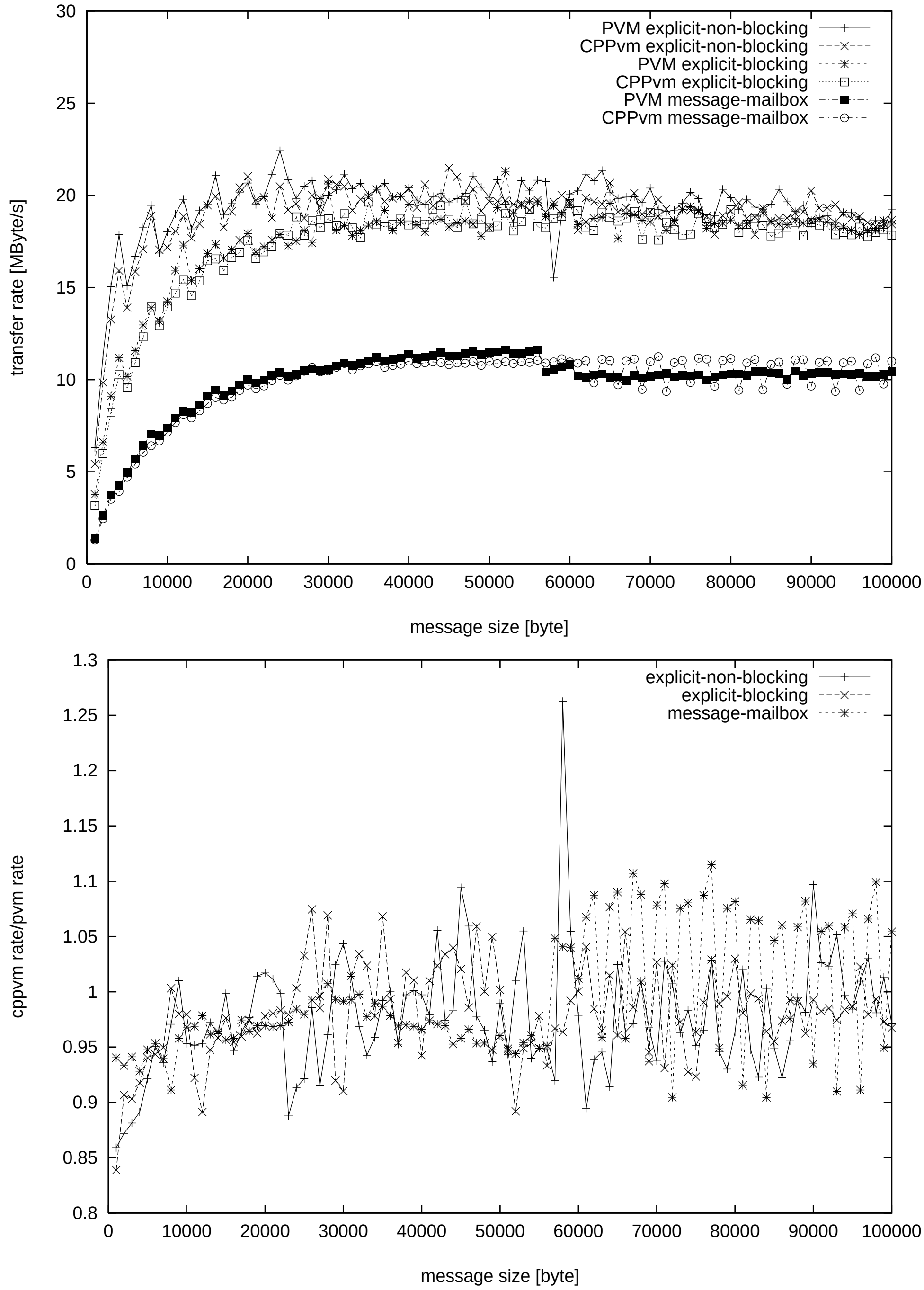
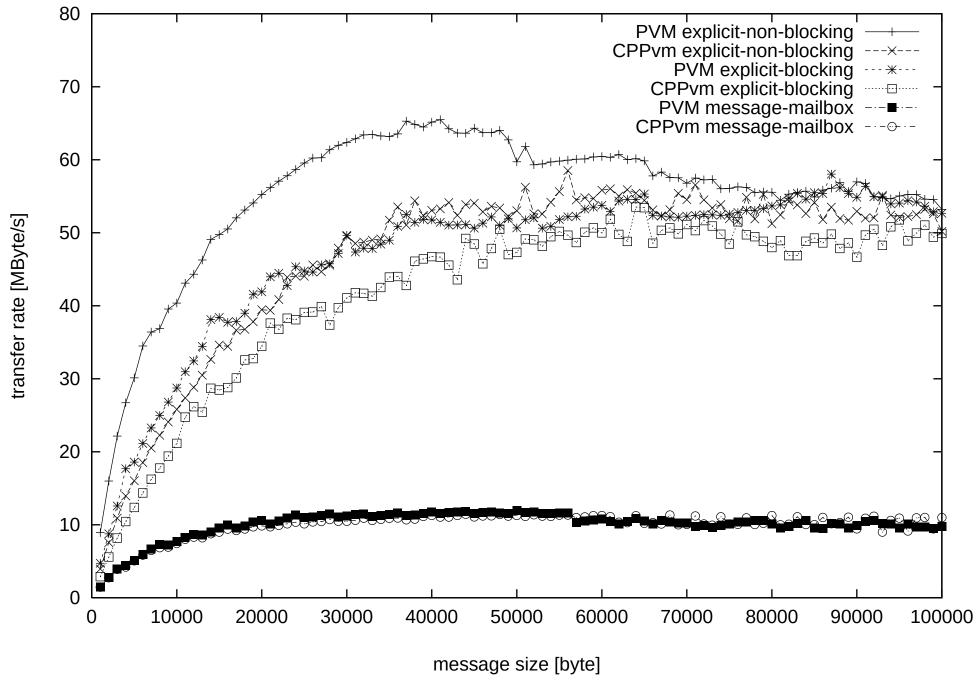


Figure A.3: Data transfer rate (see upper graph) using one processor with encoding scheme `PvmDataDefault`. The average ratio (see lower graph) for explicit message passing with non-blocking send is 0.978, for blocking send 0.981, and for the message mailbox access 0.994.



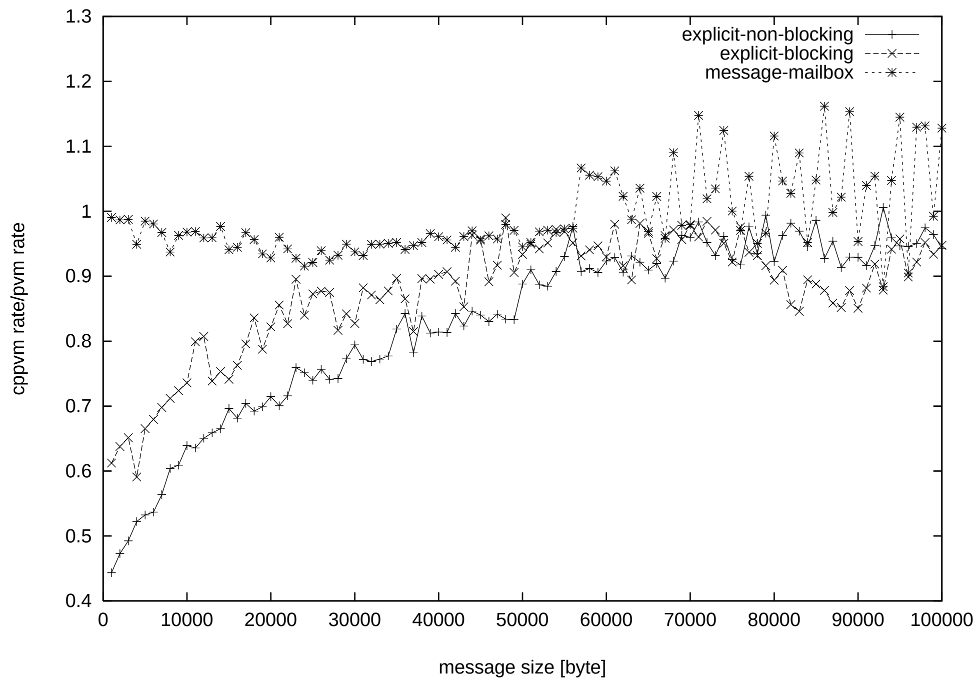
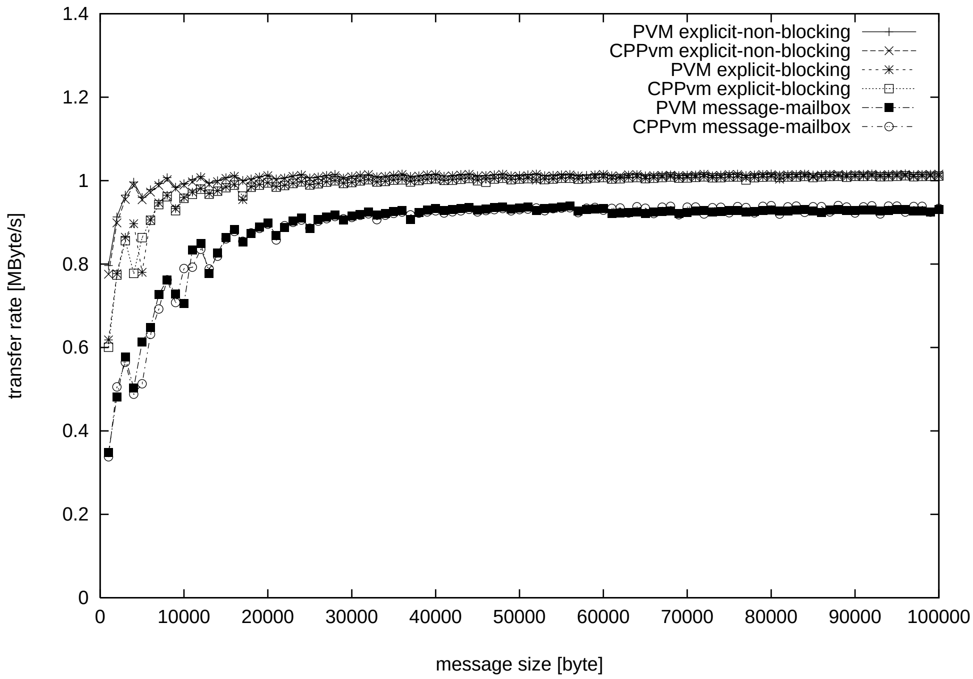


Figure A.4: Data transfer rate (see upper graph) using one processor with encoding scheme `PvmDataInPlace` and direct routing (`PvmRouteDirect`). The average ratio (see lower graph) for explicit message passing with non-blocking send is 0.830, for blocking send 0.875, and for the message mailbox access 0.991.



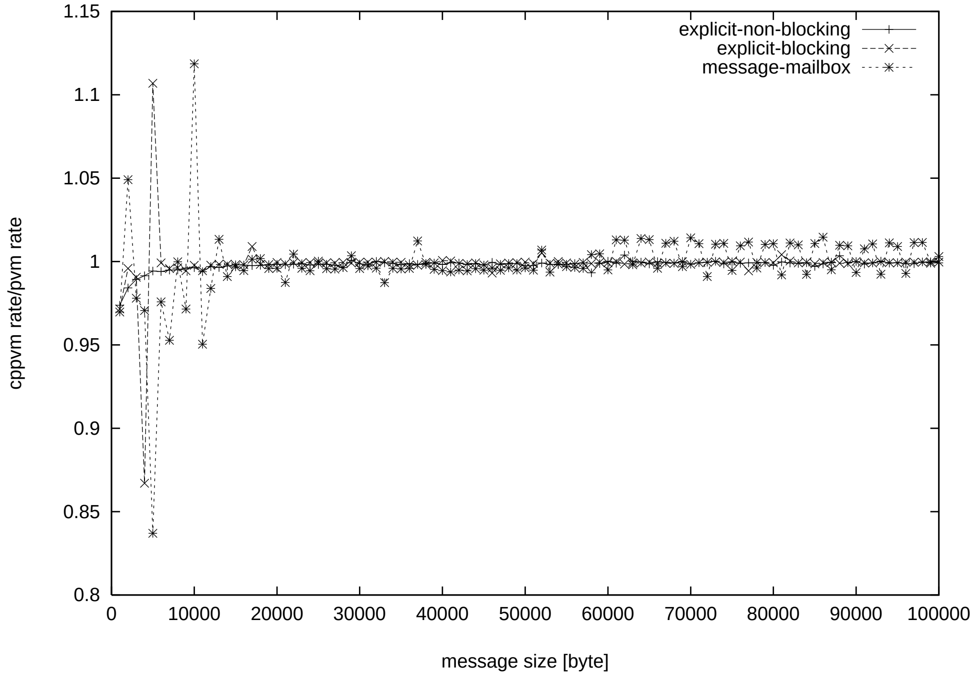


Figure A.5: Data transfer rate (see upper graph) using two hosts connected via a 10 MBit Ethernet with encoding scheme `PvmDataDefault`. The average ratio (see lower graph) for explicit message passing with non-blocking send is 0.998, for blocking send 0.998, and for the message mailbox access 0.998.

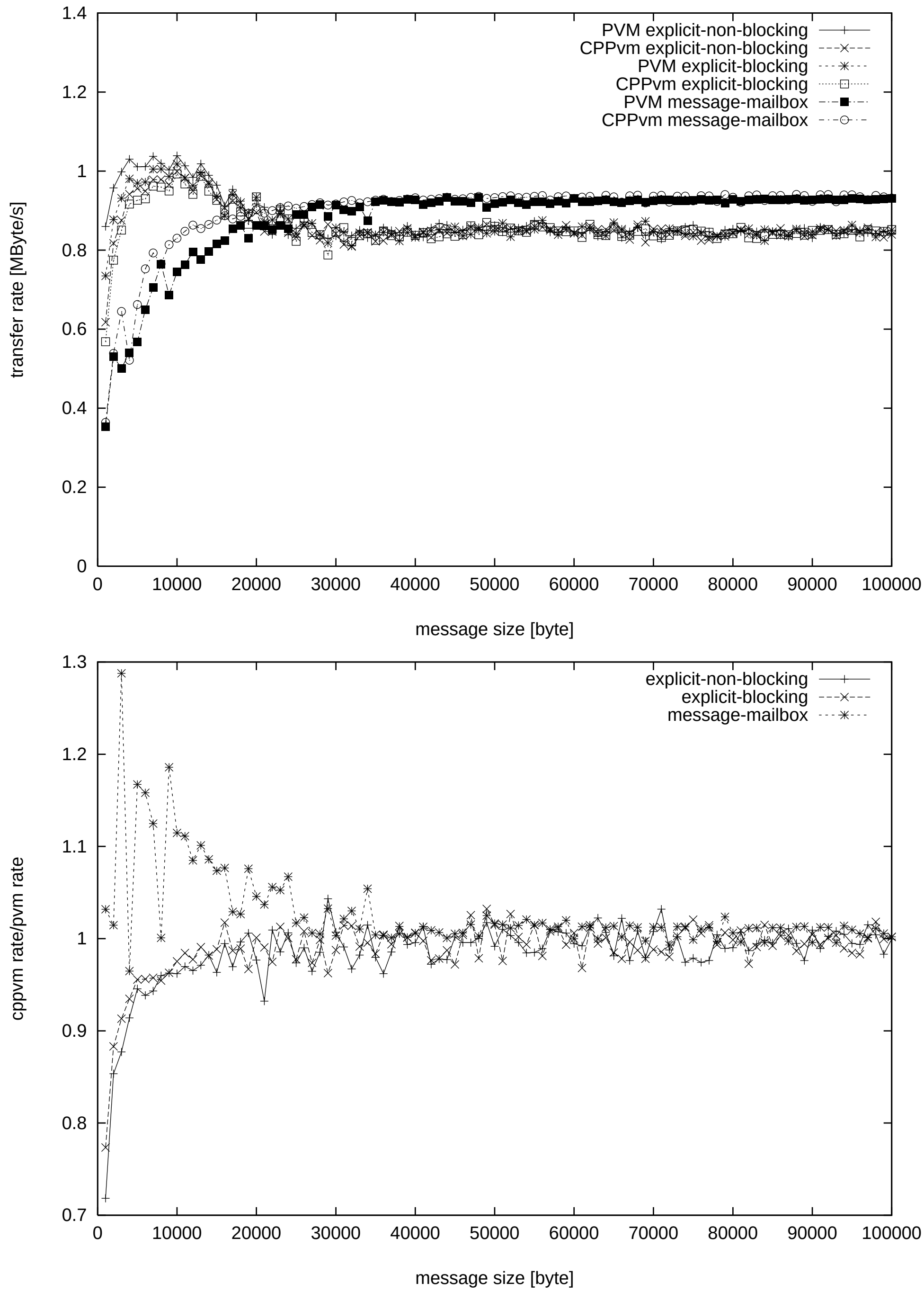


Figure A.6: Data transfer rate (see upper graph) using two hosts connected via a 10 MBit Ethernet with encoding scheme `PvmDataInPlace` and direct routing (`PvmRouteDirect`). The average ratio (see lower graph) for explicit message passing with non-blocking send is 0.985, for blocking send 0.989, and for the message mailbox access 1.027.

Appendix B

FAQ

The Frequently Asked Questions (FAQ) should help you with common PVM and CPPVM problems. This compilation is taken from the previous chapters, the PVM documentation, and the PVM newsgroup. If you have problems which are not described in this list or in the manuals you can also visit the PVM newsgroup `comp.parallel.pvm` or the WWW-pages:

```
http://www.epm.ornl.gov/pvm/pvm_home.html
http://www.informatik.uni-stuttgart.de/ipvr/bv/cppvm
http://groups.yahoo.com/group/CPPvm
```

There is a pre-installed PVM **version on my host but it does not work**

Install your own PVM version in your home directory. See subsection 2.1.1 and the file `$PVM_ROOT/Readme`.

I can not compile PVM

See subsection 2.1.1 and the file `$PVM_ROOT/Readme`. If the linker says e.g. "undefined reference to 'tgoto'", you have to add the library which contains ´tgoto´ (see question "How can I find a library for a specific symbol"). If e.g. the library `/usr/lib/libncurses.a` contains the symbol, you have to edit the `$PVM_ROOT/conf/$PVM_ARCH.def` file. Add the library to the `ARCHLIB` line:

```
ARCHLIB          =        -L/usr/lib -lncurses
```

"`-L`" tells the compiler the path of the library, "`-l`" specifies the library name (without "lib" at the beginning and ".a" at the end).

I can not compile CPPvm

See subsection 2.2.1, the file `cppvm/README` and the trouble shooting section in file `cppvm/makefile`. You can also modify and use the script `make-fallback` instead of the makefile to compile CPPvm.

I can not compile the CPPvm examples

See the file `cppvm/examples/README` and the trouble shooting section in file `cppvm/examples/makefile`. You can also modify and use the script `examples/make-fallback` instead of the makefile to compile CPPvm.

How can I find a specific library?

If you want to find a library (e.g. `libbaz.a`), use the `find` command:

```
find / -name "libbaz*" -print
```

How can I find a library for a specific symbol?

If you are searching for a library which contains a specific symbol (e.g. `foo`), you can:

- Try `man foo`. Maybe there is a man page for the symbol which also tells you the library name.
- Search manually for the symbol in some libraries (e.g. all libraries in /usr/lib):

  ```
  nm --print-file-name /usr/lib/lib* | grep foo
  ```

Can I use different PVM/CPPvm versions at the same time?

No. Ensure that you have the same PVM and CPPvm versions installed on every host which is part of your virtual machine.

I use a new PVM version and CPPvm does not work any more

Since CPPvm uses PVM you have to recompile CPPvm, e.g.:

```
cd cppvm
make clean all install
```

I can not start the PVM daemon on the local host

PVM startup problems are logged in the file /tmp/pvml.<uid>. Check this file for error messages.

Confirm that you have installed PVM and that you have set the environment variables PVM_ROOT and PVM_ARCH (see subsection 2.1.1). Try to start the PVM console ($PVM_ROOT/console/$PVM_ARCH/pvm). If you cannot start the console, kill the processes pvm and the pvmd (if running). After that remove the socket address file /tmp/pvmd.<uid> and the log file /tmp/pvml.<uid> created by former runs of the local PVM–daemon and try to start the console again.

I can not start the PVM daemon on my local Windows host

If you get error messages like "You are required to run on NTFS", this has probably nothing to do with your file-system, but with the PVM socket files. So check the PVM_TMP variable and delete the files pvmd.<uid> and pvml.<uid> in the directory pointed by this variable.

I can not start a PVM daemon on another hosts

PVM startup problems are logged in the file /tmp/pvml.<uid>. Check this file for error messages. Ensure that you have set the environment variables PVM_ROOT and PVM_ARCH on the other host (see subsection 2.1.1). If you have are setting the environment variables in your shell setup file (.cshrc, .bashrc, ...) but /tmp/pvml.<uid> says that PVM_ROOT is not set, ensure that your setup file is evaluated until the variables are set. Put the PVM environment variables at the beginning of your shell setup file to avoid such problems.

The different versions of PVM/CPPVM are not fully compatible to each other. Ensure that you have installed the same version of PVM/CPPVM on each host. Start the PVM console $PVM_ROOT/console/$PVM_ARCH/pvm and use the command version to determine the PVM version on a host. Look in the file cppvm/VERSION for the current CPPVM version. Ensure that you have compiled your programs with these PVM/CPPVM libraries. Recompile the PVM/CPPVM libraries and your programs if you are not sure about this.

Do the steps described in "I can not start the PVM daemon on the local host" on every host which is part of your virtual machine. Please pay special attention to the point about already running PVM daemons on the other hosts.

Check if you can execute a remote commands on the host (rsh <host> ls). If that does not work, check that your .rhost file on the remote host contains the name of the host from which called the command.

Check the path to the daemon on each host ($PVM_ROOT/lib/pvmd). On some architectures the path must be described in the host-file (win1 dx="C:/pvm3/lib/pvmd"). See subsection 2.1.5 and 2.2.5.1 how to use host-files. You can check this by typing `rsh <host> $PVM_ROOT/lib/pvmd`.

If you are using a different login name on the other host, use the host-file options `lo=` and `so=` (see subsection 2.1.5 and 2.2.5.1).

If you still get strange messages, check your `.cshrc` file. It is important that you not have any I/O in the `.cshrc` file because this will interfere with the startup of PVM.

If the PVM daemon can still not be started, you might have a network problem. Check your network (hostname-to-address mappings (`/etc/hosts`), ip-address, net mask, gateway, hub ...).

I can not start PVM daemon on Windows hosts

Ensure that you have installed a working remote shell daemon on the Windows hosts. By default Windows does not provide mechanisms for remote program execution. But you can download a remote shell daemon from the PVM www-pages. Sometimes a host-file has to be used to tell PVM where to find the daemon (`win1 dx="C:/programs/pvm3.4/lib/pvmd"` see subsection 2.1.5 and 2.2.5.1). See question "I can not start PVM daemon on other hosts" for more information.

The remote shell `rsh` is used to start the PVM daemons on other hosts. How can I replace `rsh` by another shell, e.g. `ssh`?

There are two possibilities to another shell. The first possibility is to edit the define `RSHCOMMAND` in the file `$PVM_ROOT/conf/$PVM_ARCH.def`. Replace `rsh` by your shell. For example to use `ssh`, replace
`-DRSHCOMMAND= \"/usr/bin/rsh\"`
by
`-DRSHCOMMAND= \"/usr/bin/ssh\"`
You have to recompile PVM after the modification. If there is no file `$PVM_ROOT/conf/$PVM_ARCH.def`, download the PVM package and compile your own version in your home directory.

The second possibility is to set the `PVM_RSH` environment variable to point to the absolute path to your shell, e.g. to `ssh`. Set the variable in the same shell startup file as the variables `PVM_ROOT` and `PVM_ARCH`.

Do not forget to configure the new remote shell. Check if you can execute remote commands, e.g. by calling `ssh <host> ls`. When manually entering your password, you will not receive a prompt from PVM or `ssh` - the text prompts normally returned by `ssh` are automatically captured and written to the local `/tmp/pvml.<uid>` log file. However, simply typing your password from inside the "pvm" console or when running the "pvmd" script will work.

You can bypass this manual password entry for a given session by executing the following commands (example is for bash, could use for any shell):

```
$ ssh-agent bash
$ ssh-add
<now enter your passphrase ONCE>
$ pvm
$ pvm> add host2 . . .
```

For this to work, you must have run `ssh-keygen` on each host to create a private/public key pair for your login id. These keys are typically stored in the `$HOME/.ssh` directory on each system. The `$HOME/.ssh/identity.pub` public key must be copied to each remote host and added as a line in the `$HOME/.ssh/authorized_keys` file on the remote system. When this has been set up, any PVM hosts can be added without password or passphrase entry using the above ssh-agent session.

Can I start a PVM daemon on every host and afterwards start my PVM/CPPvm program?

No. A program can only connect to an already running local PVM daemon. A program fails when it tries to connect to another host with a running PVM daemon. Stop the daemons on the other hosts (see "I can not start the PVM daemon on the local host").

To avoid running daemons on other hosts of your virtual machine always use the `halt` method at the end of your parallel program (stops all processes and daemons of your virtual machine).

Is it possible to run more than one virtual machine at the same time?

The virtual machines are bound to a user. That is why the user id is part of the files `/tmp/pvml.<uid>` and `/tmp/pvmd.<uid>`. Every user has its own virtual machine. The virtual machines of different users do not affect each other. If two users have started a process `hello` on the same host, the `halt` call of one user does not stop the `hello` process of the other user.

A user can only start one virtual machine at the same time. It is also possible to allow more than one virtual machine. For that you have to compile a special PVM version. Please refer the PVM manual how to accomplish that.

I can not start a CPPvm program

Check if you can start a PVM daemon of every host of your virtual machine (see question "I can not start PVM daemon on other hosts").

Check if the called executables are on the specific hosts and that the executable path points to them (usually `pvm3/bin/$PVM_ARCH/` or `$PVM_ROOT/bin/$PVM_ARCH/`). Remember that the executable path can also be set with the `ep=` option in the PVM host-file (see subsection 2.1.5 and 2.2.5.1).

I can not spawn a process on a host

See question "I can not start a CPPVM program".

How can I test if a process is still alive?

See section 10.2.

How can I test if a host is still alive?

See section 10.2.

I can not add a host

Try to start a PVM daemon on the host (see question "I can not start a PVM daemon on another hosts").

I still can not add a host

There may occur additional problems when adding hosts caused by IP problems. First of all, ensure that the mapping of your hostname to an IP address is correct. For example, you might have to replace the IP configuration via DHCP by a static IP address.

If you see messages like `netoutput() sendto: Invalid argument pvmbailout(0)`, `Master Host IP Address is Loopback!`, or get the `PvmIPLoopback` error when adding hosts, this means that the networking on your Master PVM host (the one you initially started PVM on) is not set up to support multiple, remote hosts in your virtual machine. By default, especially on many Linux systems, the host name alias is appended to the loopback/localhost IP address in the `/etc/hosts` file, for example:

```
127.0.0.1       localhost sun11
```

This is very useful for running in stand-alone mode without networking, however this alias must be removed for interaction with remote hosts in PVM. Use a new record with the IP address of your hostname instead, e.g.:

```
127.0.0.1        localhost
44.55.1.1        sun11
```

If this does not work on your host, you should also refer the Linux Networking HOWTO for information on automatically handling this scenario (via `ifup-local` and `ifdown-local scripts`).

The group server can not be started

The group server `pvmgs` may be spawned on any host. The executable normally resides in `$PVM_ROOT/bin/$PVM_ARCH`. If you change the path using the host-file `ep=` option, make sure this directory is still included.

I can not use semaphores in my program

The semaphore server `cppvms` may be spawned on any host which is part of the virtual machine. The executable normally resides in `pvm3/bin/$PVM_ARCH` or `$PVM_ROOT/bin/$PVM_ARCH`. If it is not there compile and copy it:

```
>cd cppvm
>make
>cp bin/$PVM_ARCH/cppvms $PVM_ROOT/bin/$PVM_ARCH
```

If you change the path using the host-file `ep=` option, make sure the path to `cppvms` is still included.

How can I debug or monitor my CPPvm program?

You can use a standard debugger like `xxgdb` to debug your parallel program (see subsection 2.2.4.1). To monitor your program use tools like `tracer` and XPVM (see subsections 2.2.4.2 and 2.2.4.3).

I think my program is o.k. but PVM/CPPvm crashes or shows strange behavior

The different versions of PVM/CPPvm are not fully compatible to each other. Ensure that you have installed the same version of PVM/CPPvm on each host. Start the PVM console `$PVM_ROOT/console/$PVM_ARCH/pvm` and use the command `version` to determine the PVM version on a host. Look in the file `cppvm/VERSION` for the current CPPvm version. Ensure that you have compiled your programs with these PVM/CPPvm libraries. Recompile the PVM/CPPvm libraries and your programs if you are not sure about this.

The program crashes when I call CPPVM methods

On some compilers, an exception can cause a segmentation fault or an I/O trap. You can switch off the exception handling by adding `-DCPPvmDoNotUseExceptions` to `CFLAGS_ALL` in the file `cppvm/makefile`. After that, you have to recompile the CPPVM library and your program. Subsection 2.2.4 explains how to debug and monitor PVM/CPPVM programs.

I get the error message: `pvm_upk*:  End of buffer`

Ensure that you have balanced transfer methods. Unbalanced transfer methods can occur if you differ between send/receive (`cppvmIsTransferSend`,`cppvmIsTransferReceive`) in the transfer method of your class. The error show up for example when sending an integer value with the method `cppvmTransferInt` and the object on the other side wants to receive a double value using `cppvmTransferDouble`.

Unbalanced transfer can also happen when using standard template library classes. Do not try e.g. to receive a `vector<short>` if you send a `vector<float>` object.

Sometimes the send encoding scheme `PvmDataInPlace` seems to cause problems as well. In this case use the encoding scheme `PvmDataDefault` instead.

Can I use PVM message passing functions for CPPVM message classes?

No. All CPPVM message classes (cppvmInt, cppvmFloat, cppvmArrayInt, cppvmArrayFloat, ...) are derived from cppvmObject. They also contain additional information, e.g. to allow a blocking send within CPPVM. For this reason, it is not possible to use them together with standard PVM send/receive functions. The next example will fail:

```
Process A:
...
cppvmInt value;
sendStrm << value;
...

Process B:
...
int value;
pvm_recv(-1, -1);
pvm_upkint(&value, 1, 1);
...
```

Of course it is possible to use PVM functions together with CPPVM. But do not use them for the CPPVM message classes.

The transfer of standard C++ types fails

The standard C++ types (`bool`, `int`, ...) are internally mapped to the corresponding CPPVM classes (cppvmBool, cppvmInt, ...). Therefore the modification of a class message tag e.g. for cppvmFloat will affect the message passing of the corresponding standard C++ type `float` (method `setCPPvmClassMsgTag`, see section 4.7). Object message tags do not affect standard C++ types.

I can not use the STL classes bitset, slist, and valarray

The STL classes bitset, slist and valarray are not supported by default. This is due to they are not implemented in Visual C++ or in older versions of gcc. To enable these classes within CPPVM use the compiler switches `-DCPPvmUseStlBitset`, `-DCPPvmUseStlSlist`, and `-DCPPvmUseStlValarray`.

My compiler does not support templates or the STL

The switch `-DCPPvmDoNotUseTemplates` disables the template support. It also switches off the standard template library (STL) support. If you want to keep template support but not STL support, use the flag `-DCPPvmDoNotUseStl` instead.

How can I redirect the input of a spawned task?

It is not possible to redirect the command line input (prompt) of spawned processes. Each spawned task has `/dev/null` opened as `stdin`. Only processes started by the user can use a command line input. Of course spawned processes can read files or use a graphical user interface (GUI) for its input.

When opening files remember that the working directory of spawned processes is `$HOME` by default. You can change the working directory by using the host-file option `wd=` (see subsection 2.1.5 and 2.2.5.1). When a processes uses a GUI and is spawned on another host, do not forget to enable the X-Window access on your local host (`xhost <hostname>`) and to export the `DISPLAY` variable. There are to ways to export variables:

- set the `PVM_EXPORT` environment variable before starting the program, e.g.:

  ```
  setenv PVM_EXPORT DISPLAY
  ```

- use the `exportEnvVar` method of the class cppvmConnection within your program:

  ```
  cppvmConnection pvmConn;
  pvmConn.exportEnvVar("DISPLAY");
  ```

```
// redirect output
cppvmCatchout catchout(CPPvmStdout);
// spawn child 'gui-app' on 'sun1'
cppvmSpawnConnection child("gui-app", "", PvmTaskHost, "sun1");
```

Ensure that the environment variable `DISPLAY` is set correctly.

How can I redirect the output of a spawned task?

Use the CPPvm class cppvmCatchout to redirect the output of spawned tasks (see section 11.4). The output can be redirected to `stdout`, `stderr` or into a file. Output from grandchildren (spawned by children) tasks is also collected, provided the children don't reset `PvmOutputTid`.

With PVM and CPPvm it is not possible to have more than one catch out stream. For example, it is not possible to catch the output of process `myprocess1` into the file `myprocess1.out` and the output of process `myprocess2` into the file `myprocess2.out`. Every caught output is redirected to the stream specified by the last cppvmCatchout object.

You have to create a cppvmCatchout object before creating a cppvmSpawnConnection or cppvmSpawnMultiConnection object to redirect their output. It is not possible to redirect the output of processes once they are spawned without catchout.

I can not see the redirected output of a spawned task

You have to create a cppvmCatchout object before creating a cppvmSpawnConnection or cppvmSpawnMultiConnection object to redirect their output (see section 11.4). It is not possible to redirect the output of processes once they are spawned without catchout.

The redirected output of the spawned processes is buffered to reduce the frequency of messages to the catchout process. To flush the output you can call `fflush(stdout)` in a spawned process. The buffering mode of `stdout` can be changed to line-oriented for the entire process by calling `setlinebuf(stdout)`. When a spawned process disconnects from PVM (process exits, calls the CPPvm `exit` method, deletes all cppvmConnection, cppvmSpawnConnection or cppvmSpawnMultiConnection) objects) `stdout` is flushed as well.

Is is also possible that a process calls the `halt` method before all messages are collected. A solution is to synchronize the processes before calling `halt` by using messages or semaphores.

I can not compile my CPPvm program

Make sure that you have installed PVM and CPPvm (see subsection 2.1.1 and 2.2.1). Check if you include `cppvm.h` in your program and that you have set the include paths of

PVM and CPPVM in your makefile (`-I$(PVM_ROOT)/include -I/home/smith/cppvm/h`).

I can not link my CPPVM program

Make sure that you link all PVM and CPPVM libraries to your program:

```
-L/home/smith/cppvm/lib/$(PVM_ARCH) -lcppvm
-L$(PVM_ROOT)/lib/$(PVM_ARCH) -lpvm3 -lgpvm3 -lpvmtrc
```

On some architectures special libraries are needed, e.g. `-lsocket -lnsl` for Sun/Solaris or `wsock32.lib` for PC/Windows.

Where do I have to put the executables?

By default PVM searches first in `<WorkingDirectory>/pvm3/bin/$PVM_ARCH/` (the default working directory is `$HOME`) and then in `$PVM_ROOT/bin/$PVM_ARCH/`. This path list can be changed with the host-file option `ep=` (see section 2.1.5 and 2.2.5.1). If your programs aren´t on a filesystem shared between the hosts, you must compile or copy them to each host.

What is the working directory for spawned processes?

The default working directory is `$HOME`. This path can be changed with the host-file option `wd=` (see 2.1.5 and 2.2.5.1).

I want to start more than one parallel program

It is possible to execute more than one parallel program at a time. Just call one program in one shell and another one in a second shell. Of course you have to care about for possible inter-dependences. This can be for example "receive from any process" messages and distributed or mailbox objects with similar key names. A program should also not call `halt` at the end. This would halt the virtual machine and stop the second program as well. The solution is not to call `halt` but to kill all child processes and exit. This will not halt the PVM daemon.

How can I debug CPPVM programs?

See the debugging subsubsection 2.2.4.

The distributed or mailbox objects have other values than I gave them

If an object is registered as distributed (or mailbox) object with `cppvmDistObj("myKey")`, CPPVM tries to find an object with the given key `myKey` in the PVM message mailbox. If there is an object with `myKey`, the object is send from the message mailbox to the local object. If there is no such object, a object with `myKey` is created in the message mailbox and the local object is send to the mailbox.

This is the reason why you should set values of distributed or mailbox objects *after* registering them. If the values are set before, it is possible that the another process has registered `myKey` first and the local object is overwritten with the data base value. Instead of setting the value first:

```
...
  // integer object
  cppvmInt distInt;
  // set value
  distInt = 4;
  // register the object as a distributed object
  distInt.cppvmDistObj("intKey");
  // 'distInt' may now have
  // a value !=4
...
```

register the object before:

```
...
  // integer object
  cppvmInt distInt;
  // register the object as a distributed object
  distInt.cppvmDistObj("intKey");
  // set value
  distInt = 4;
...
```

It is also possible that you have problems with persistent objects (see the next point B).

When I restart my program, the distributed or mailbox objects have strange values

All objects in the data base can be persistent. Distributed objects are persistent by default. Persistent objects are only deleted when the parallel virtual machine is halted. So when new processes are started, the object is still in the database with its old values and sends this values to all processes which register this distributed object again. To halt the virtual machine use the `halt` method of the class cppvmConnection or the halt command on the PVM console (see sections 11.1 and 2.1.4).

If you don´t want to halt the virtual machine, e.g. if you don´t want to kill all PVM daemons at the end of your program, you can also delete the data base objects (see section 5.2 and section 6.1).

Can I start the semaphore server on a specific host?

By default CPPvm chooses the host for the semaphore server "cppvms". But it is also possible to define the host. There are three ways to do this:

```
// version 1:
cppvmSemaphore barrier("stop","sun1");
// version 2:
cppvmSemaphore dummy1(NULL, "win2");
// version 3:
cppvmSemaphore dummy2;
dummy2.startServer("lin3");
```

Version one creates creates a semaphore variable "stop" and starts the semaphore server on host "sun1". Version two and three start the server on the specified host but do not create a semaphore variable

There is another important thing to know: the first semaphore variable in the parallel virtual machine defines the location of the semaphore server. Imagine there are two processes 'A' and 'B', both defining semaphore variables. 'A' wants to start the semaphore server on "sun1", 'B' on "lin2". If the semaphore variable in 'A' is created faster than in 'B', the semaphore server is started on "sun1". In this case, the call for "lin2" in 'B' has no effect. If you want to assign the semaphore server to a specific host, ensure that this call is the first in the virtual machine.

I am using semaphores and get a strange behaviour when I restart my program

The `cppvms` server process is not killed automatically when all application processes exit from PVM. When application processes are started again, they will connect to the already running semaphore server. This will cause problems when the `setNumber` method is called (forbidden) or if the application processes expect a different counter number for a semaphore. So it is recommended to call `halt` when leaving a semaphore program. All processes in the virtual machine are killed, i.e. `cppvms` as well.

If you don´t want to call the `halt` method, e.g. if you don´t want to kill all PVM daemons at the end of your program, you can also kill the server manually:

```
// semaphore
cppvmSemaphore ready("done");
...
ready.killServer();
```

Ensure not to access to semaphores after this call (this includes all processes currently running in the virtual machine)! Processes blocked by semaphores at this time will be blocked forever.

How can I speed-up the transfer rate between the processes?

You can influence the message encoding and message routing as well as take advantage of hardware features (see section A.1).

How can I use shared memory together with PVM?

PVM supports shared memory on some systems. See the file `$PVM_ROOT/Readme.mp` for more information.

Does CPPvm/PVM support multiple processor hosts (SMP)?

On hosts with more than one processor (SMP) the operating system distributes the processes on the available processors. To use the available processors, just start several CPPvm/PVM processes. For example if you have four processors and four CPPvm/PVM processes, the operating system will probably assign one process to each processor.

Resource limitations in the PVM-daemons

How many processes each PVM-daemon can manage is limited by two factors: the number of processes allowed for a user by the operating system, and the number of file desciptors available to the PVM-daemon. The limit on processes is generally not an issue, since it doesn't make sense to have a huge number of processes running on a uniprocessor machine.

Each process consumes one file descriptor in the PVM-daemon, for the daemon-process TCP stream. Each spawned process (not ones connected anonymously) consumes an extra descriptor, since its output is read through a pipe by the PVM-daemon (closing stdout and stderr in the process would reclaim this slot). A few more file descriptors are always in use by the PVM-daemon for the local and network sockets and error log file. For example, with a limit of 64 open files, a user should be able to have up to 30 processes running per host. The PVM-daemon may become a bottleneck if all these processes try to talk to one another through it.

The PVM-daemon uses dynamically allocated memory to store message packets en route between processes. Until the receiving process accepts the packets, they accumulate in the PVM-daemon in a FIFO procedure. No flow control is imposed by the PVM-daemon: it will happily store all the packets given to it, until it can't get any more memory. If an application is designed so that processes can keep sending even when the receiving end is off doing something else and not receiving, the system will eventually run out of memory.

Resource limitations in the processes

As with the PVM-daemon, a process may have a limit on the number of others it can connect to directly. Each direct route to a process has a separate TCP connection (which is bidirectional), and so consumes a file desciptor. Thus, with a limit of 64 open files, a process can establish direct routes to about 60 other processes. Note that this limit is in effect only when using process-process direct routing. Messages routed via the PVM-daemons use only the default daemon-process connection.

The maximum size of a PVM message is limited by the amount of memory available to the process. Because messages are generally packed using data existing elsewhere in memory, and they must be reside in memory between being packed and sent, the largest possible message a process can send should be somewhat less than half the available memory. Note that as a message is sent, memory for packet buffers is allocated by the PVM-daemon, aggravating the situation. Im-place message encoding alleviates this problem somewhat, because the data is not copied into message buffers in the sender. However, on the receiving end, the entire message is downloaded into the process before the receive call accepts it, possibly leaving no room to unpack it.

In a similar vein, if many processes send to a single destination all at once, the destination process or PVM-daemon may be overloaded as it tries to store the messages. Keeping messages from being freed when new ones are received by using `pvm_setrbuf()` also uses up memory.

These problems can sometimes be avoided by rearranging the application code, for example, to use smaller messages, eliminate bottlenecks, and process messages in the order in which they are generated.

I can not send very large messages

See question "Resource limitations in the PVM-daemons" and "Resource limitations in the processes".

I can not spawn a large number of processes

See question "Resource limitations in the PVM-daemons" and "Resource limitations in the processes".

Bibliography

[Dijkstra 68] E. W. Dijkstra. *The Structure of the THE Multiprogramming System.* In *Commun. of the ACM 11*, pages 341–346, May 1968.

[Geist et al. 94] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, Vaidyalingam S. Sunderam. *PVM: Parallel Virtual Machine A Users' Guide and Tutorial for Network Parallel Computing.* MIT Press, 1994.

List of Tables

Index