

VR Juggler Portable Runtime

Programmer's Guide

VR Juggler Portable Runtime: Programmer's Guide

Version 2.0

Published \$Date: 2007-06-29 17:38:02 -0500 (Fri, 29 Jun 2007) \$

Copyright © 2003–2007 Iowa State University

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with the Invariant Sections being Appendix B, *GNU Free Documentation License*, with no Front-Cover Texts, and with no Back-Cover Texts. A copy of the license is included in Appendix B, *GNU Free Documentation License*.

Table of Contents

| | |
|---|-----|
| Vaporous Programming | vii |
| I. Input/Output | 1 |
| 1. Buffered I/O | 2 |
| Opening and Closing | 2 |
| Setting Attributes for Opening | 2 |
| Blocking Versus Non-Blocking | 2 |
| Reading and Writing | 2 |
| Statistics Collection | 3 |
| 2. Sockets | 4 |
| Internet Addresses | 4 |
| Socket Commonalities | 4 |
| Uses of <code>vpr::Socket</code> | 5 |
| Fixed Blocking State | 5 |
| Datagram-Oriented Sockets | 6 |
| Stream-Oriented Sockets | 6 |
| The Acceptor/Connector Pattern | 6 |
| The Acceptor | 6 |
| The Connector | 6 |
| 3. Serial Ports | 7 |
| Interface Overview | 7 |
| Abstraction Details | 7 |
| 4. Data Marshaling | 8 |
| Endian Conversion | 8 |
| Object Serialization | 8 |
| II. Multi-Threading | 10 |
| 5. Creating Threads | 11 |
| Threads: Using <code>vpr::Thread</code> | 11 |
| Creating Threads | 12 |
| Waiting for a Thread to Complete | 13 |
| Suspending and Resuming a Thread's Execution | 13 |
| Getting and Setting a Thread's Priority | 14 |
| Sending Signals to a Thread | 14 |
| Canceling a Thread's Execution | 14 |
| Requesting the Current Thread's Identifier | 15 |
| The Gory Details | 15 |
| Thread Functors | 16 |
| High-Level Description | 16 |
| Functors from (Non-Static) Member Functions | 18 |
| Functors from Static Member Functions | 21 |
| Functors from Non-Member Functions | 23 |
| Functors from Callable Objects | 23 |
| Updating from VPR 1.0 | 24 |
| 6. Synchronization | 28 |
| Semaphores: Using <code>vpr::Semaphore</code> | 28 |
| High-Level Description | 28 |
| Creating a Semaphore | 28 |
| Locking a Semaphore | 29 |
| Releasing a Locked Semaphore | 29 |
| The Gory Details | 29 |
| Mutual Exclusion: Using <code>vpr::Mutex</code> | 29 |
| High-Level Description | 30 |
| Creating a Mutex | 30 |
| Locking a Mutex | 30 |

| | |
|--|----|
| Attempting to Lock a Mutex | 30 |
| Testing the State of a Mutex | 30 |
| Releasing a Locked Mutex | 31 |
| The Gory Details | 31 |
| Condition Variables: Using <code>vpr::CondVar</code> | 31 |
| High-Level Description | 31 |
| Creating a Condition Variable | 32 |
| Locking a Condition Variable | 32 |
| Attempting to Lock a Condition Variable | 32 |
| Releasing a Locked Condition Variable | 32 |
| Putting Condition Variables to Use | 33 |
| 7. Signal Handling | 35 |
| III. Miscellaneous Utilities | 36 |
| 8. Simulated Sockets | 38 |
| Sim Socket Interface | 38 |
| Sim Socket Implementation | 38 |
| 9. Globally Unique Identifiers | 39 |
| GUID Creation | 39 |
| GUID Operations | 39 |
| 10. Intervals | 40 |
| Interval Creation | 40 |
| Interval Operations | 40 |
| 11. Singletons | 41 |
| 12. Factories | 44 |
| 13. Performance Monitoring | 45 |
| IV. Appendices | 46 |
| A. I/O Implementation Information | 48 |
| B. GNU Free Documentation License | 49 |
| PREAMBLE | 49 |
| APPLICABILITY AND DEFINITIONS | 49 |
| VERBATIM COPYING | 50 |
| COPYING IN QUANTITY | 50 |
| MODIFICATIONS | 51 |
| COMBINING DOCUMENTS | 52 |
| COLLECTIONS OF DOCUMENTS | 53 |
| AGGREGATION WITH INDEPENDENT WORKS | 53 |
| TRANSLATION | 53 |
| TERMINATION | 53 |
| FUTURE REVISIONS OF THIS LICENSE | 54 |
| ADDENDUM: How to use this License for your documents | 54 |
| Bibliography | 55 |
| Glossary of Terms | 56 |
| Index | 57 |

List of Examples

| | |
|--|----|
| 5.1. Using Boost.Bind to Create Thread Functors | 17 |
| 5.2. Member Function for Thread Functor (1) | 19 |
| 5.3. Member Function for Thread Functor (2) | 20 |
| 5.4. Member Function for Thread Functor (3) | 20 |
| 5.5. Member Function for Thread Functor (4) | 21 |
| 5.6. Static Member Function for Thread Functor (1) | 21 |
| 5.7. Static Member Function for Thread Functor (2) | 22 |
| 5.8. Static Member Function for Thread Functor (3) | 22 |
| 5.9. Non-Member Function for Thread Functor | 23 |
| 5.10. Callable Object for Thread Functor (1) | 23 |
| 5.11. Callable Object for Thread Functor (2) | 24 |
| 5.12. Callable Object for Thread Functor (3) | 24 |
| 5.13. VPR 1.0 Use of Thread Non-Member Functor | 25 |
| 5.14. Updated Use of Static Member Function for Thread Functor | 26 |
| 5.15. VPR 1.0 Use of Thread Member Functor | 26 |
| 5.16. Updated Use of Member Function for Thread Functor | 27 |
| 11.1. Use of <code>vprSingletonHeader()</code> | 41 |
| 11.2. Use of <code>vprSingletonImp()</code> | 41 |
| 11.3. Use of <code>vprSingletonHeaderWithInitFunc()</code> | 42 |
| 11.4. Use of <code>vprSingletonImpWithInitFunc()</code> | 42 |
| 11.5. Use of <code>vprSingletonImpLifetime()</code> | 42 |
| 11.6. Use of <code>vpr::Singleton<T></code> | 43 |

Vaporous Programming

For those developers new to the VR Juggler Portable Runtime [<http://www.vrjuggler.org/vapor/>] (VPR), VPR provides an cross-platform, object-oriented abstraction layer to common operating system features. VPR is the key to the portability of Gadgeteer, Tweek, VR Juggler, and other middleware written at the Virtual Reality Applications Center. It has been in development since January 1997, and it has grown to be a highly portable, robust tool. Software written on top of VPR can be compiled on IRIX, Linux, Windows, FreeBSD, and Solaris, usually without modification.

Internally, VPR wraps platform-specific APIs such as BSD sockets, POSIX threads, Win32 threads, and Win32 overlapped I/O. Depending upon how it is compiled, it may also wrap the Netscape Portable Runtime [<http://www.mozilla.org/projects/nspr/index.html>] (*NSPR*), another cross-platform OS abstraction layer written in C. By wrapping NSPR, VPR provides developers with an object-oriented interface and gains even better portability. These details are all hidden behind the classes that make up VPR, and users of VPR do not need to worry about platform-specific details as a result.

VPR is basically a collection of utility classes. As such, the biggest part of using VPR is knowing the interface for a given class. In this book, we provide high-level information about various pieces of VPR in hopes of making VPR easier to use. The book itself is designed so that readers can focus on what they need to know about VPR classes. For example, someone who wants to learn about using the VPR thread abstraction can go straight to that part of the book (i.e., Part II, “Multi-Threading”). Within each part, however, the chapters build up the concepts incrementally, so it is advisable, for example, to understand the basics of VPR I/O before trying to learn about the serial port abstraction.

Part I. Input/Output

To begin, we will cover the components of VPR that will be used for I/O programming. This includes how to use VPR sockets and serial ports. We assume that the reader has at least some familiarity with operating system programming, in particular with serial device I/O and socket I/O.

Chapter 1. Buffered I/O

One of the largest components of VPR is its I/O abstraction. All I/O classes (file handles, serial ports, and sockets) share the base class `vpr::BlockIO`. Reads and writes are performed using contiguous blocks of memory (buffers). This design provides an API that closely resembles that of the underlying operating system (with methods called `read()` and `write()`), but it is in contrast to stream-oriented I/O that is usually seen in C++. Streams could be written on top of the buffered I/O classes, but thus far, the need has not arisen. With this in mind, the design provides an API that is immediately familiar to programmers used to POSIX-based interfaces, but the API may seem clumsy to C++ programmers who are accustomed to using `std::ostream` and friends.

Readers interested in the implementation of the I/O component of VPR are referred to Appendix A, *I/O Implementation Information*. We discuss the use of the VPR socket abstraction, and we provide some insight into how the abstraction is implemented. By providing some implementation details, it is our hope that the online API reference will be easier to understand and navigate.

Opening and Closing

Opening and closing I/O devices is quite simple. There are two methods for performing these actions: `vpr::BlockIO::open()` and `vpr::BlockIO::close()`. However, at the `vpr::BlockIO` level, these methods are pure virtual (i.e., abstract), and thus, the implementation varies depending on the actual I/O device, be it a socket, serial port, or file descriptor. Regardless of the implementation, the preconditions for `vpr::BlockIO::open()` state that the device must not already be open. For `vpr::BlockIO::close()`, the device must be open before an attempt is made to close it.

Setting Attributes for Opening

Prior to opening an I/O device, some attributes can be set. These in turn affect how the device is opened. In the general case of `vpr::BlockIO`, the only attribute that is available determines whether the device will be opened in blocking mode or non-blocking mode. By default, all devices open in blocking mode, and in most cases, this is the desired mode.

Blocking Versus Non-Blocking

The decision to use blocking or non-blocking I/O depends on the needs of the application or library being developed on top of VPR. While the decision can be made before opening the device, it can also be made after the device is open using the methods `vpr::BlockIO::enableBlocking()` and `vpr::BlockIO::enableNonBlocking()`. Typically, the blocking/non-blocking state should be set exactly once (either before or after opening the device). In some cases, it is not possible to change the state after a critical “point of no return.” Refer to the section called “Fixed Blocking State” for more information on this.

Reading and Writing

Reads and writes occur using the `read()` and `write()` methods respectively. These methods are overloaded for common data structures that may be used to store the information being read or written. For example, strings are used frequently in I/O handling, so the type `std::string` can be used for easy management of string data. When reading `n` bytes, the `std::string` object will be resized internally by `read()` to ensure that it has enough room to store the full buffer. The same is true for the `read()` variant that takes a `std::vector<vpr::Unit8>` object reference. This overloaded version of `read()` is helpful when dealing in arrays of bytes. Of course, the lowest level variant of `read()` is the version that takes a `void*` buffer. In this case, the buffer pointed to must have at least `n` bytes of contiguous storage prior to calling `read()`.

There is also a special method called `readn()` that guarantees that `n` bytes will be read. (The `read()` method only guarantees that it read *at most* `n` bytes.) As such, `readn()` is a blocking call, even when a non-blocking data source is being used behind the scenes. It will not return until all `n` bytes have been read or an error occurs while reading.

Writing to an I/O object works as one might expect. The same overloads are available for `write()` as are available for `read()` and `readn()`. The buffer passed in to `write()` must be at least as big as the amount of data to be written (in bytes), or a memory access error can occur.

Tip

Always make sure that the buffer size matches the amount of data to be read or written. Buffer overflows have long been a source of security problems in software, and they can be avoided by managing memory carefully.

Statistics Collection

All the I/O classes in VPR have built-in statistics collection capabilities. By default, the code is not activated so as to prevent unwanted overhead. However, it can be enabled quite simply using the method `vpr::BlockIO::setIOStatStrategy()`. This method takes a single parameter, a statistics collection object, and invokes the correct methods whenever I/O occurs. Within the specific implementation, any form of statistics related to reading and writing of data may be collected.

From the name of the method in `vpr::BlockIO`, we see the first indication that a Strategy pattern [Gam95] is used to implement the pluggable statistics collection code. All statistics strategy classes must derive from `vpr::BaseIOStatsStrategy`, and strategies can be mixed using the templated class `vpr::IOStatsStrategyAdapter<S, T>`. Currently, the only strategy class is `vpr::BandwidthIOStatsStrategy`, used for collecting information about bandwidth usage of a given I/O object.

Chapter 2. Sockets

Socket programming can be a very difficult task, and the API used to write network code is difficult to understand in and of itself. The purpose of the VPR socket abstraction is thus two-fold: it abstracts the platform-specific API, and it aims to simplify the interface so that developers can focus on protocol implementations.

Note

Readers not familiar with socket programming should consult a reference manual ([Ste98] is recommended). We do not attempt to explain the ins and outs of socket programming. Instead, we assume that readers are familiar with socket-level I/O and the ideas involved with various types of network communication.

The socket abstraction follows the concepts set forth by the *BSD sockets* API, which was also the model for the Winsock API used on Windows. In VPR, two types of sockets may be instantiated: stream-oriented (TCP, `vpr::SocketStream`) and datagram (UDP, `vpr::SocketDatagram`). The helper class `vpr::InetAddr` makes use of Internet Protocol (v4) addresses easier. Built on top of `vpr::SocketStream` are two classes that make writing client/server code easier: `vpr::SocketConnector` and `vpr::SocketAcceptor`. Finally, VPR provides cross-platform data conversion functions (see Chapter 4, *Data Marshaling*) to deal with endian issues.

We begin our discussion by diving right into the common features of sockets, as collected in the class `vpr::Socket`. We assume that readers already have an understanding of the buffered I/O concepts (see Chapter 1, *Buffered I/O*) used in VPR I/O programming. The following sections cover datagram-oriented sockets and stream-oriented sockets respectively. We will conclude this chapter with a review of the high-level patterns implemented for simplifying the authoring of client/server architectures.

Internet Addresses

All socket code written using the VPR socket abstraction must use Internet Protocol (IP) addresses. The class `vpr::InetAddr` neatly abstracts the low-level details of using Internet addresses¹. This class encapsulates both the IP address and the port number. It manages all the endian issues and the lookup of host names as necessary.

When constructed, a new `vpr::InetAddr` is initialized to the constant value `vpr::InetAddr::AnyAddr`. This value corresponds with the OS-level constant `INADDR_ANY`. Typically, either a host name, a port number, or both must be set after the object is constructed. Such details will vary depending on the application needs. The IP address can be set using a symbolic host name (which will be resolved through DNS queries) or using the human-readable “dotted-decimal” notation. The port number is set using the native byte ordering; it is converted internally to network byte order. It is also possible to set the host name and port number together in a single string that uses the format “host:port”. This format is convenient when the values for the host name and port come in as string values.

Socket Commonalities

At the lowest level, all sockets have several things in common. For example, all sockets must be opened before they are used, and they must be closed when communication is complete. During communication,

¹The current implementation of `vpr::InetAddr` only supports IPv4, though support for IPv6 will be added when the need arises.

data is read from and written to a socket, and reads and writes may be blocking (synchronous) or non-blocking (asynchronous). All sockets are bound to a local address, and *connected* sockets have a remote address².

Note

It is important to note that a socket does not have to be stream-oriented to be in a connected state. A datagram-oriented socket may be “connected” to a remote address so that it has a default destination. This alleviates the need to specify the destination address at every send.

These commonalities are collected into the class `vpr::Socket`, which serves as the base interface for datagram- and stream-oriented sockets. The API for this class includes methods such as `open()`, `close()`, `send()`, `recv()`, and `connect()`. Note that `recv()` and `send()` are provided as analogues to `read()` and `write()` respectively. These are included because the BSD sockets API defines the system calls `recv(2)` and `send(2)`, in addition to `read(2)` and `write(2)`, for use with socket file descriptors. The extra methods are thus provided to give programmers already familiar with the BSD sockets API an easily recognizable interface.

Uses of `vpr::Socket`

Instances of `vpr::Socket` cannot be created because the constructors are not public. Instances of the concrete types `vpr::SocketDatagram` and `vpr::SocketStream` can be used polymorphically as instances of `vpr::Socket` (and `vpr::BlockIO`, of course). Because the basic operations such as `read()` and `write()` are defined by the base class, using the concrete socket types polymorphically could be a convenient mechanism for mixing socket communication protocols in an application.

Fixed Blocking State

Due to the semantics of sockets on Windows NT, the VPR socket abstraction contains a slight variation of the behavior that is available on UNIX-based systems. In Windows, once a call to `read()`, `write()`, `accept()`, etc., is made, the blocking state of the socket is fixed³. That is, if the socket is a blocking socket, it will forever remain in a blocking socket after one of these calls. The same is true for non-blocking sockets. Furthermore, for a stream-oriented socket that is accepting connections, the sockets created as clients connect inherit the blocking state of the accepting socket. The full list of methods that fix the blocking state is as follows:

- `vpr::Socket::read()`, `vpr::Socket::readn()`, `vpr::Socket::recv()`,
`vpr::Socket::recvn()`, `vpr::SocketDatagram::recvfrom()`
- `vpr::Socket::write()`, `vpr::Socket::send()`,
`vpr::SocketDatagram::sendto()`
- `vpr::SocketStream::accept()`
- `vpr::SocketStream::connect()`

The NSPR documentation [<http://www.mozilla.org/projects/nspr/>] has a more complete description of this issue. We must implement our socket abstraction in this way in order to provide consistent semantics (not just consistent syntax) across platforms.

²Unconnected sockets may send data to a different destination at every write. They may also receive data from any remote address.

³UNIX-based systems allow the blocking state to be changed from blocking to non-blocking or vice versa at any time.

Datagram-Oriented Sockets

The class `vpr::SocketDatagram` provides VPR's abstraction to datagram-oriented sockets, typically known as UDP (user datagram protocol) sockets. Indeed, this class wraps the underlying operating system's implementation of UDP sockets. The interface for `vpr::SocketDatagram` extends `vpr::Socket` to include the methods `sendto()` and `recvfrom()`, overloaded in the same way as `read()` and `write()`. As with the operating system API, these methods are used to send a message to a specific destination address or to receive a message from a specific remote address, respectively.

Stream-Oriented Sockets

The class `vpr::SocketStream` wraps the use of TCP (transmission control protocol) sockets. TCP sockets are also known more abstractly as stream-oriented sockets. All such sockets must be connected to a specific peer, and thus there is no interface comparable to `vpr::SocketDatagram::sendto()` or `vpr::SocketDatagram::recvfrom()`.

In order for connections to be made, a socket must be listening for incoming connection requests. For that purpose, `vpr::SocketStream` introduces the methods `listen()` (to put a socket into a listening state) and `accept()` (for accepting new connections). These work the same way as the system calls after which they are named. However, `accept()` is somewhat unique in that it takes an unopened `vpr::SocketStream` object as a parameter. The object reference is “set up” when a successful connection occurs. Thus, when `vpr::SocketStream::accept()` returns successfully, the caller can be certain that the `vpr::SocketStream` reference passed in is now a valid, connected socket.

Finally, since stream-oriented sockets always have an accepting socket that handles incoming connection requests, `vpr::SocketStream` provides a convenience method called `openServer()`. This can be used in place of the usual open-bind-listen sequence of calls for setting up an accepting (server) socket. Use of this method is not required for putting a socket into a listening state; rather, it exists to shorten user code slightly. The drawback of using it is that, in the case of failure, the returned status will not tell the caller what stage of setting up the listening socket failed.

The Acceptor/Connector Pattern

Building on the foundation of stream-oriented, connected sockets, VPR implements the Acceptor/Connector Pattern [Sch00]. The classes used in the implementation are `vpr::SocketAcceptor` and `vpr::SocketConnector`. This pattern captures the concepts used in writing stream-oriented network software. The software may use a client/server protocol or a peer-to-peer protocol, but in either case, an initial connection must be made to an accepting socket.

The Acceptor

The acceptor is created using a `vpr::InetAddr` object that specifies the address on which the acceptor listens for incoming connection requests. Once opened, the acceptor is ready to accept new connections. The call to `vpr::SocketAcceptor::accept()` uses the same arguments and behavior as `vpr::SocketStream::accept()`, so programmers already familiar with setting up an accepting socket with `vpr::SocketStream` will find `vpr::SocketAcceptor` very easy to use.

The Connector

The connector is designed to make non-blocking connections easy to manage. Depending on the arguments passed to `vpr::SocketConnector::connect()`, a socket may be put into non-blocking mode if it is not already set as such. Thus, a connection can be made “in the background” if necessary. However, due to the semantics described in the section called “Fixed Blocking State”, after a background connection is made, the socket must remain in non-blocking mode for the duration of its lifetime.

Chapter 3. Serial Ports

Most input devices used for virtual reality systems today make use of a computer's serial port for data communication. In our experience, serial port programming is not much different than other I/O programming. Implementing the communication protocol used by a given device tends to be the hard part, and that will likely be the case regardless of the underlying hardware.

The VPR serial port abstraction is based on the concepts implemented by the standard termios serial interface used by most modern UNIX-based operating systems [Ste92]. As such, the API allows enabling and disabling of a subset of the serial device features that can be manipulated using termios directly. To provide cross-platform semantics, however, some termios features are not included because there is no corresponding capability with Win32 overlapped I/O. Furthermore, any termios settings that relate only to modems are not included in the VPR serial port abstraction.

Interface Overview

In termios, serial ports are configured by setting or clearing a wide variety of bits in various data structures. Based on this, the VPR serial port API includes methods for enabling a feature, disabling a feature, and testing the current status of a feature. For example, the following methods deal with the hardware flow control bit:

- `enableHardwareFlowControl()`: Enables hardware flow control (if it was not already enabled)
- `disableHardwareFlowControl()`: Disables hardware flow control (if it was not already disabled)
- `getHardwareFlowControlState()`: Returns the current state of hardware flow control (true for on, false for off)

When changing the enabled state of a serial port feature, the change may not take effect immediately. This is determined by the update action setting, which is manipulated by `vpr::SerialPort::setUpdateAction()`. There are three possible states (corresponding to the enumerated type `vpr::SerialTypes::UpdateActionOption`):

1. Now: Perform the change immediately
2. Drain: Perform the change after all output is transmitted
3. Flush: Perform the change after all output is transmitted and discard all unread input

The right setting to use may depend on the specific hardware or on the desired behavior.

Abstraction Details

The serial port abstraction is handled differently than the other I/O abstraction components. We wrap two serial port interfaces: termios and Win32 overlapped I/O. Because NSPR does not provide a serial port layer, we have to allow the termios to be used with NSPR on UNIX-based platforms. While this makes the implementation a little clumsy and the build system a little more complicated, it has little if any impact on users. The point of the abstraction is to hide the low-level details to provide a consistent interface across platforms.

Chapter 4. Data Marshaling

Network communication involves the transfer of data between computers, and for it to work, the two computers must be able to talk to each other using the same language. This must occur even if the two have different internal representations of the data they hold. Thus, the data must be marshaled into a common format when it is sent out and demarshaled into the local native format when it is received. VPR provides some helper functions and utility classes to simplify the efforts of network programmers.

Endian Conversion

A very common data marshaling activity is the conversion of a multi-byte data unit from host byte order to network byte order. Such conversions are necessary for elements of data that occupy 16 or more bits. In VPR terms, that means the types `vpr::Int16`, `vpr::int32`, `vpr::Int64`, and the unsigned variants thereof. The interface `vpr::System` provides conversion functions from host to network byte order and vice versa for all of these types. All the functions operate in terms of the unsigned version of the aforementioned integer types, but they work with the signed versions as well since they simply manipulate the actual bits. The full list of functions is as follows:

- `vpr::System::Htons()`: Converts a 16-bit integer from host to network byte order.
- `vpr::System::Ntohs()`: Converts a 16-bit integer from network to host byte order.
- `vpr::System::Htonl()`: Converts a 32-bit integer from host to network byte order.
- `vpr::System::Ntohl()`: Converts a 32-bit integer from network to host byte order.
- `vpr::System::Htonll()`: Converts a 64-bit integer from host to network byte order.
- `vpr::System::Ntohll()`: Converts a 64-bit integer from network to host byte order.

Single-precision floating-point values (which occupy 32 bits of memory) can be converted using `vpr::System::Htonl()` and `vpr::System::Ntohl()`. Similarly, double-precision floating-point values (which occupy 64 bits of memory) can be converted using `vpr::System::Htonll()` and `vpr::System::Ntohll()`.

Note

Programmers already familiar with the operating system-level calls such as `ntohs(3)` and `htonl(3)` may wonder why the above functions are named with a capital letter (i.e., `vpr::System::Htonl()` versus `vpr::System::htonl()`). We have used this naming convention because the byte order conversion functions are preprocessor macros on some platforms, and the C preprocessor cannot tell the difference between a method declaration and the use of a macro. In other words, the code would not compile on platforms where the functions are really macros.

Object Serialization

Serializing objects is more complicated than dealing with individual integer variables, but ultimately, a class is composed of other data types. If the internal data types can be serialized, then the object that holds them can be serialized as well. To enable this functionality, VPR defines the interface `vpr::SerializableObject`. It operates in terms of two other interfaces: `vpr::ObjectReader` and `vpr::ObjectWriter`. Together, these allow an object and all the data it aggregates to be serialized into an array of bytes that can be sent over the network. Once received, the array can be de-

serialized into a duplicate of the original object.

The basic idea behind the object serialization interface in VPR is the same as in Java (see the API documentation on `java.io.Serializable`). An class identifies itself as being serializable by adding `vpr::SerializableObject` to its list of parent classes. Two pure virtual methods must then be implemented: `readObject()` and `writeObject()`. When a class instance must be serialized, `writeObject()` is invoked with an argument that provides the class with a `vpr::ObjectWriter` instance. The implementation of `writeObject()` would then add the instance data to the object writer and return. De-serializing an object occurs in `readObject()` using an instance of `vpr::ObjectReader`. A full class hierarchy can be serialized and de-serialized through polymorphism. The derived classes must simply call the parent class' `writeObject()` and `readObject()` methods, thus following the class hierarchy up to the first class that identified itself as serializable.

Because `vpr::ObjectReader` and `vpr::ObjectWriter` are abstract types, the actual implementation of these may vary. This is similar to the way that Java can serialize an object to a variety of data streams. Currently, VPR can serialize a class to an array of bytes (`vpr::BufferObjectReader` and `vpr::BufferObjectWriter`) or to XML (`vpr::XMLObjectReader` and `vpr::XMLObjectWriter`). The array of bytes is suitable for network transmission and makes sharing of classes between hosts easy.

Part II. Multi-Threading

In this part, we present the capabilities VPR provides for writing cross-platform multi-threaded software. It is assumed that readers already know the basics of multi-threaded programming including the definition of *thread of control*. What is described here is how to use the VPR thread interface, `vpr::Thread`, not how to write multi-threaded software. For that reason, it is recommended that readers be familiar with the following publications before continuing:

- *Pthreads Programming* [<http://www.oreilly.com/catalog/pthread/>] by Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell.
 - The `sproc(2)` manual page on IRIX or on SGI's technical publications site [http://techpubs.sgi.com/library/tpl/cgi-bin/getdoc.cgi?coll=0650&db=man&fname=/usr/share/catman/p_man/cat2/standard/nsproc.z&srch=sproc].
 - The `pthread(3)` manual page for your operating system. The `pthread` functions are part of a POSIX standard and will be the same across platforms.
-

Chapter 5. Creating Threads

When considering multi-threaded programming, it is important to know that with great power comes great responsibility. The power is being able to provide multiple threads of control in a single application. The responsibility is making sure those threads get along with each other and do not step on each other's data. VR Juggler is a multi-threaded library which makes it very powerful and very complex.

As a cross-platform framework, VR Juggler uses an internal threading abstraction that provides a uniform interface to platform-specific threading implementations. That cross-platform interface is available to programmers to make applications multi-threaded without tying them to a specific operating system's threading implementation.

Threads: Using `vpr::Thread`

The threading interface in VPR is modeled after the POSIX thread specification of POSIX.1b (formerly POSIX.4). The main difference is that VPR's interface is object-oriented while POSIX threads (pthreads) are procedural. The basic principles are exactly the same, however. A function (or class method) is provided to the `vpr::Thread` class, and that function is executed in a thread of control that is independent of the creating thread.

Threads are spawned (initialized and begin execution) when the `vpr::Thread` constructor is called. That is, when instantiating a `vpr::Thread` object, a new thread of execution is created. The semantics of threads says that a thread can begin execution at any time after being created, and this is true with `vpr::Threads`. Do not make any assumptions about when the thread will begin running. It may happen before or after the constructor returns the `vpr::Thread` object.

To pass arguments to threads, the common mechanism of encapsulating them in a C++ struct must be used. The function executed by the thread takes only a single argument of type `void*`. An argument is not required, of course, but to pass more than one argument to a thread, the best way to do this is to create a structure and pass a pointer to it to the `vpr::Thread` constructor.

Once a `vpr::Thread` object is created, it acts as an interface into controlling the thread it encapsulates. Thread signals can be sent, priority changes can be made, execution can be suspended, etc. This interface is the focus of this section.

We begin our discussion of creating threads with VPR by explaining the use of the class `vpr::Thread`. Use of `vpr::Thread` is intended to be easy. Multi-threaded programming has enough complications without having a difficult API as well. In almost all cases, thread creation can be done in a single step, executed one of two ways:

1. Pass a function pointer to the `vpr::Thread` constructor along with any argument that should be passed to the function when the thread is created
2. Pass a *functor* (a callable object) to the `vpr::Thread` constructor

The second appears easier, but to create the functor, parameters to the function executed by the thread may still have to be passed. The presence of parameters depends on the specific function being run by the thread. In addition to the function pointer or functor, parameters such as the priority and the stack size may be passed to the `vpr::Thread` constructor, but the defaults for the constructor are quite reasonable.

A minor issue with creating a `vpr::Thread` is the concept of functors. The topic of functors will be put off until the next section. For now, just think of them as callable objects.

Before writing code that uses `vpr::Threads`, make sure that the header file `vpr/Thread/Thread.h` is included. Never include the platform-specific headers such as `vpr/md/POSIX/Thread/ThreadPosix.h`. The single file `vpr/Thread/Thread.h` is all that is required.

Creating Threads

The following example illustrates how to create a thread that will execute a free function called `run()` that takes no arguments. The prototype for `run()` is:

```
void run();
```

This will be the same across all platforms. The thread creation code is then:

```
vpr::Thread* thread;  
thread = new vpr::Thread(run);
```

At this point, a newly spawned thread is executing the code in `run()`. It is advisable to hang onto the variable `thread` so that the thread may be controlled as necessary.

That was pretty easy. What if you want to pass one or more arguments to `run()` so that its behavior can be modified based on some variables? One approach is to define the function so that it takes a parameter that is an aggregate type (a struct or a class). The data needed by the function is then collected into this aggregate type and packaged with the function pointer. A common way to do this is as follows:

```
struct ThreadArgs  
{  
    int id;  
    std::string name;  
    // And so on...  
};  
  
void run(const ThreadArgs& args)  
{  
    // Do work ...  
}  
  
void someFunc()  
{  
    // Other code ...  
  
    ThreadArgs args;  
    args.id = 50;  
    args.name = "My Thread";  
    // And so on ...  
  
    vpr::Thread* thread;  
    thread = new vpr::Thread(boost::bind(run, args));  
}
```

When creating a single thread, this works beautifully. If multiple threads are needed, all taking the same type of argument, there would usually have to be a separate argument structure instance for each one. A bunch of objects can be declared, or the same objects can be reused over and over.

The preceding example made use of `Boost.Bind` [<http://www.boost.org/libs/bind/bind.html>] to couple the `run()` function with a struct instance. Instead of declaring a struct to bundle all the arguments together, we could take advantage of the flexibility of `Boost.Bind` to couple multiple arguments with the

function. This is shown below:

```
void run(int id, char* name)
{
    // Do work ...
}

void someFunc()
{
    // Other code ...

    vpr::Thread* thread;
    thread = new vpr::Thread(boost::bind(run, 50, "My Thread"));
}
```

There is a limit to the number of parameters that can be passed in this way, so this approach must be used judiciously. We will explain more about function objects and Boost.Bind in the section called “Thread Functors”.

Waiting for a Thread to Complete

Once we have a thread running, it is often useful to synchronize another thread so that its execution halts until the running thread has completed. This is called “joining threads”. The following example illustrates how this can be done:

```
vpr::Thread* thread;

thread = new vpr::Thread(run);

// Do other things while the thread is going ...

thread->join();

// Now that the thread is done, continue.
```

Here, the creator of thread can be another `vpr::Thread`, or it can be the main thread of execution. In other words, any thread can create more threads and control them. What happens in this example is that thread is created and begins running. Meanwhile, the creator thread continues to do some more work and then must wait for `thread` to finish its work before continuing. It calls the `join()` method, a blocking call, and it will not return until `thread` has completed.

While it is not demonstrated here, the `join()` method can take a single argument of type `void**`. It is a pointer to a pointer where the exit status of the joined thread is stored. The operating system fills the pointed to pointer with the exit status when the thread exits.

Suspending and Resuming a Thread's Execution

Sometimes, it may be necessary to suspend the execution of a running thread and resume it again later. There are two methods in the `vpr::Thread` interface that do just this. Assuming that there is already a running thread pointed to by the object `thread`, it can be suspended as follows:

```
thread->suspend();
```

Resuming execution of the suspended thread is just as easy:

```
thread->resume();
```

If something goes wrong when suspending or resuming, `vpr::IllegalArgumentException` is thrown. Otherwise, these methods return nothing upon successful completion.

Getting and Setting a Thread's Priority

Changing the priority of a thread tells the underlying operating system how important a thread is and gives it hints about how to schedule the threads. If no value for the priority is given to the constructor, all `vpr::Threads` are created with the default priority for all threads. Values higher than 0 for the priority request a higher priority when the thread is created.

Besides being able to set the priority when the thread is created, it is possible to query and to adjust the priority of a running thread. Assuming that there is already a running thread pointed to by the object `thread`, its priority can be requested as follows:

```
int prio;
thread->getPrio(&prio);
```

The thread's priority is stored in `prio` and returned via the pointer passed to the `getPrio()` method. Setting that thread's priority is also easy:

```
int prio;
// Assign some priority value to prio ...
thread->setPrio(prio);
```

If something goes wrong when querying or changing the priority of the thread, `vpr::IllegalArgumentException` is thrown. Otherwise, these methods return nothing upon successful completion.

Sending Signals to a Thread

On UNIX-based systems, a signal is sent to a process using the `kill(2)` system call. With POSIX threads, signals are sent using `pthread_kill(3)`. VPR's thread interface implements these ideas using a `kill()` method. There are two ways to call this method: with an argument naming the signal to be delivered to the thread or without an argument which cancels the thread's execution. The first of these is described in this section, and the second is described in the next section.

A problem does arise here, unfortunately. Signals are not supported on all operating systems (notably, Win32). The interface is consistent, but code written on IRIX will not compile on Win32 if, for example, it sends a `SIGHUP` to a thread. An improved thread interface is being designed to overcome problems such as this one. For now, we describe this part of the interface as though it is supported completely on all platforms.

As usual, assume there is a running thread, a pointer to which is stored in `thread`. To send it a signal (`SIGINT`, for example), use the following:

```
thread->kill(SIGINT);
```

The signal will be delivered to the thread by the operating system, and the thread is expected to handle it properly. This version of the `kill()` method throws `vpr::IllegalArgumentException` if an error occurs. Otherwise, this method returns nothing upon successful completion.

Canceling a Thread's Execution

As described in the previous section, using the `kill()` method with no argument cancels the execution of the thread. When using POSIX threads, this is actually implemented using `pthread_cancel(3)`. On IRIX with SPROC threads, a `SIGKILL` is sent to the thread to end its execution forcibly. The syntax for using this method is basically the same as in the previous section, but it is repeated to make that clear. Again assuming that there is a running thread with a pointer to its `vpr::Thread` object stored in `thread`, use the following:

```
thread->kill();
```

Unlike the syntax used to send a signal to a thread, this version of `kill()` does not have a return value.

Users of POSIX threads may be wondering if the `vpr::Thread` API provides a way to set cancellation points in the code. Unfortunately, it does not at this time. Extending the interface in this way is being considered, but cancellation points do not have meaning with all thread implementations.

Requesting the Current Thread's Identifier

Lastly, it is common to request the currently running thread's identifier. This only makes sense when called from a point on that thread's flow of execution. (In POSIX threads, this is the notion of "self". For IRIX SPROC threads, this means getting the process ID.) The `vpr::Thread` API provides a static method that can be called at any time in the thread that is currently running. It returns a pointer to a `vpr::BaseThread` (the basic type from which `vpr::Thread` inherits its interface). The syntax is as follows:

```
vpr::BaseThread* my_id;
my_id = vpr::Thread::self();
```

The returned pointer can then be used to perform all of the previously described operations on the current thread.

The Gory Details

The current threading implementation in VPR is a little difficult to understand. The code is not complicated at all, but because all platform-specific implementations are referred to as `vpr::Threads`, the details can get lost in the shuffle. To begin, the current list of platform-specific thread implementation wrapper classes are:

- `vpr::ThreadSGI`: A wrapper around IRIX SPROC threads (refer to the `sproc(2)` manual page for more information)
- `vpr::ThreadPosix`: A wrapper around POSIX threads (both Draft 4 and Draft 10 of the standard are supported)
- `vpr::ThreadNSPR`: A wrapper around Netscape Portable Runtime threads
- `vpr::ThreadWin32`: A wrapper around Win32 threads

The interface itself is defined in `vpr::BaseThread`, and all of the above classes inherit from that class.

The threading implementation used is chosen when VPR is compiled. To use a certain type of thread system, be sure that the version of VPR in use was compiled with the type of threads desired. When the VPR build is configured, preprocessor `#define` statements are made in `vpr/vprDefines.h` that

describe the threading system to use. Based on that, the header file `vpr/Thread/Thread.h` makes several typedefs that set up one of the platform-specific thread implementations to act as the `vpr::Thread` interface. For example, if compiling on Win32, the class `vpr::ThreadWin32` is typedef'd to be `vpr::Thread`. Since the interface is consistent among all the wrappers, everything works as though that was the way it was written to behave.

The current implementation is modeled after the POSIX thread API for the most part. When designing it, we approached it with the idea that having a more complete API was more important than having a “lowest-common-denominator” API. That is, just because not all threading implementations support a specific feature does not mean that the API should suffer by not having that feature. Whether this was a good approach or not is an open debate.

Note

VPR has a wrapper around Netscape Portable Runtime [<http://www.mozilla.org/projects/nspr/index.html>] (NSPR) threads. NSPR threads do not support all the features we have, however, because they took the lowest-common-denominator approach. As with all technology, there is a trade-off in relieving some of our work load by using an existing cross-platform thread implementation: our interface becomes limited to what features that implementation provides. It remains to be seen exactly how much of VPR's threading subsystem will be removed, and those programmers who choose to use it should be careful to watch the mailing lists for discussions and announcements about changes.

Thread Functors

In this section, we explain the concept and use of *functors*. A functor is a high-level concept that encapsulates something quite simple. A functor is defined as “something that performs an operation or a function.” In VPR functors are used as the code executed by a thread (refer to the section called “Threads: Using `vpr::Thread`” for more detail on the topic of `vpr::Threads`). This section describes how to use functors for exactly that purpose.

Important

Users of the VPR 1.0 thread API are encouraged to read this section very carefully. At the end, there is an explanation of how to update VPR 1.0 code to use the new thread functor interface. The flexibility offered by the new approach in VPR 1.1 and beyond should offer programmers many new opportunities for how they handle and utilize threads in their software.

High-Level Description

As mentioned, a functor is used in VPR with `vpr::Threads`. VPR threads utilize `Boost.Function` [<http://www.boost.org/doc/html/function.html>] as the functor implementation. A `Boost.Function` object is a callable object, meaning that it has an overload of `operator()` that takes zero or more arguments. Earlier in the section called “Creating Threads”, we saw how to use raw function pointers. `Boost.Function` can wrap four different types of callable types:

1. Free functions
2. Static class functions
3. Non-static member functions
4. Objects overloading the member function `operator()`

Programmers already familiar with the use of `Boost.Function` and `Boost.Bind` can skip this entire section

about thread functors. The remainder of this section is an overview of how to use `Boost.Function` and `Boost.Bind` for programmers who are unfamiliar with these tools.

Getting back to functors, a functor is simply another object type that happens to encapsulate a user-defined function. The details on how this is done are not important here. What is important to know is that a functor can be thought of as a normal function. When using them, programmers usually just implement a function and then pass the function pointer to the `Boost.Function` constructor. `Boost.Function` does the rest.

The functor must behave as a function that returns nothing and takes no parameters. In terms of function pointers, the type must be `void(*)()`. A more readable form of this is the `Boost.Function` type: `boost::function<void()>`. Yet another form is the portable `Boost.Function` type used for older compilers: `boost::function0<void>`. Remember that we are not restricted to using function pointers to create functions. We are describing here callable objects that *behave* as functions.

If the functor needs to be passed arguments, then `Boost.Bind` [<http://www.boost.org/libs/bind/bind.html>] must be used to couple parameters with the functor. Those parameters will be passed into the function later when the functor is invoked. `Boost.Bind` is very powerful, and the full usage of it is beyond the scope of this document. We will demonstrate its use more fully later in this section, but we provide some uses in Example 5.1, “Using `Boost.Bind` to Create Thread Functors”.

Example 5.1. Using `Boost.Bind` to Create Thread Functors

```
struct ThreadArgs
{
    int id;
    std::string name;
};

void run1(const ThreadArgs& args);
void run2(const int id, const std::string& name);

class MyClass
{
public:
    // Bind a free function with an instance of an aggregate type.
    void spawnThread1()
    {
        ThreadArgs args = { 50, "My Thread" };
        mThread = new vpr::Thread(boost::bind(run1, args));
    }

    // Bind a free function with an instance of an aggregate type that
    // is a data member.
    void spawnThread2()
    {
        // Copies mArgs.
        mThread = new vpr::Thread(boost::bind(run1, mArgs));

        // Passes a reference to mArgs.
        mThread = new vpr::Thread(boost::bind(run1, boost::ref(mArgs)));
    }

    // Bind a free function with multiple parameters.
    void spawnThread2()
    {
        mThread =
            new vpr::Thread(boost::bind(run2, 50,
                                       std::string("My Thread")));
    }
}
```

```
// Bind a static member function with an argument.
void spawnThread3()
{
    mThread = new vpr::Thread(boost::bind(MyClass::staticRun, 10));
}

// Bind a non-static member function.
void spawnThread4()
{
    mThread = new vpr::Thread(boost::bind(&MyClass::run, this));
}

// Bind a non-static member function with multiple parameters.
void spawnThread5()
{
    mThread = new vpr::Thread(boost::bind(&MyClass::run, this,
                                           3.14159f, 16));
}

private:
    static void staticRun(int id);

    void run();

    void otherRun(const float someParam, short otherParam);

    vpr::Thread* mThread;
    ThreadArgs  mArgs;
};
```

Once a functor object exists, it is passed to the `vpr::Thread` constructor, and the new thread will execute the functor (which knows what to do with its held callable). The end result is the same as using a normal C/C++ free function or a static class member function, but there is one special benefit: with functors, non-static class member functions can be passed. In many cases, there arises a need to run a member function in a separate thread, but making it static is infeasible or awkward. Thus, it would be best to pass a non-static member function to the created thread. To get access to the non-static data members, however, the C++ `this` pointer must be available to the thread. By using `Boost.Bind` with `Boost.Function`, that is all handled behind the scenes so that passing a non-static member function is straightforward. We have seen how to do this in the methods `MyClass::spawnThread4()` and `MyClass::spawnThread5()`.

Before getting into specifics, there is a header file that must be included to use `Boost.Function` VPR thread functors. In this case, the header is `boost/function.hpp`. If `Boost.Bind` is needed, then `boost/bind.hpp` must be included. Finally, if `boost::ref()` is needed, then `boost/ref.hpp` must be included.

Functors from (Non-Static) Member Functions

We have already seen examples of how to create a `Boost.Function` functor for a member function. In this section, we will review in more detail using `Boost.Function` and `Boost.Bind` to accomplish this. Just as with free functions, the member functions (heretofore referred to as “methods”) must have the following prototype:

```
void methodName();
```

A common example of using `vpr::Thread` with a member function as the functor callable is shown in Example 5.2, “Member Function for Thread Functor (1)”. The key aspect in this example is the im-

plementation of the method `MyObject::start()`. The usage of `Boost.Bind` here is required because the object pointer has to be bound with the member function pointer in order for it to be invoked properly. Fortunately, this use is very simple—much simpler than the use of `vpr::ThreadMemberFunctor<T>` in VPR 1.0.

Note

This is not the only way of using `vpr::Thread`, nor is it strictly a recommended way of using it. It is simply an example. One missing aspect is protection of `mRunning` using a synchronization primitive such as a condition variable.

Example 5.2. Member Function for Thread Functor (1)

```
class MyObject
{
public:
    MyObject()
        : mThread(NULL)
        , mRunning(false)
    {}

    ~MyObject()
    {
        if ( mRunning )
        {
            stop();
        }

        if ( NULL != mThread )
        {
            delete mThread;
            mThread = NULL;
        }
    }

    void start()
    {
        mThread = new vpr::Thread(boost::bind(&MyObject::run, this));
        mRunning = true;
    }

    void stop()
    {
        mRunning = false;
        mThread->join();
    }

private:
    void run()
    {
        while ( mRunning )
        {
            // Do work ...
        }
    }

    vpr::Thread* mThread;
    bool mRunning;
};
```

Now let us say that there is the method `MyObject::run()` needs to take one or more arguments. `Boost.Bind` makes this possible, and it is just as easy as what we have seen in the previous example. We simply have to bind the method arguments along with the `this` pointer, as shown in Example 5.3, “Member Function for Thread Functor (2)”. For the purposes of this example, we have changed `MyObject::start()` so that it takes an argument that specifies how many iterations the thread loop will perform before exiting.

Example 5.3. Member Function for Thread Functor (2)

```
void MyObject::start(const int runCount)
{
    mThread = vpr::Thread(boost::bind(&MyObject::run, this, runCount));
}

void MyObject::run(const int runCount)
{
    for ( int i = 0; i < runCount; ++i )
    {
        // Do work ...
    }
}
```

What if the thread spawning is happening externally to the class that does the work? Once again, `Boost.Bind` will be necessary, and as we will see, the usage is familiar. Instead of using `this`, we use the object instance created by the external code. We will change the declaration of `MyObject` and assume that there is an aggregate type `thread_args_t` declared somewhere. The code shown in Example 5.4, “Member Function for Thread Functor (3)” demonstrates how we make this happen, though it is a contrived example. Note that the memory allocated for `args` would have to be deleted at some point after the thread is done using the data. That could be done at the end of `MyObject::run()` or after the thread is known to have completed its execution.

Example 5.4. Member Function for Thread Functor (3)

```
struct thread_args_t;

class MyObject
{
public:
    void run(thread_args_t* args)
    {
        // Do work ...
    }
};

void spawnThread()
{
    MyObject* my_obj = new MyObject();
    thread_args_t* args = new thread_args_t();
    // Fill in the arguments to be passed to the thread...

    vpr::Thread* thread = new vpr::Thread(boost::bind(&MyObject::run,
                                                    my_obj, args));
}
```

Creating so many heap-allocated objects is rather a hassle. Parameters passed to `boost::bind()` are copied by default. This allows the memory to be coupled with the functor so that it is available when the functor is invoked later. This means that it is safe to use stack-allocated memory when calling `boost::bind()`. It is not always desirable to have all the data copied, and that is where `boost::ref()` comes into the picture. In Example 5.5, “Member Function for Thread Functor (4)”, we see an example of using `boost::ref()` to create a reference to the instance of `MyObject` while copying in the object of type `thread_args_t`. Since `my_obj` is passed by reference, the memory should not actually be allocated on the heap. Rather, it would normally be a data member for the class creating the thread. More generally, `my_obj` cannot be deleted before the created thread exits. If `my_obj` is deleted while the thread is running, the application will crash. This is true of all data bound to the function called by the spawned thread.

Example 5.5. Member Function for Thread Functor (4)

```
MyObject my_obj;
thread_args_t args;

// Fill in the arguments to be passed to the thread...

vpr::Thread* thread = new vpr::Thread(boost::bind(&MyObject::run,
                                                boost::ref(my_obj),
                                                args));
```

Important

Objects bound as parameters to a function call through `boost::bind()` may be copied many, many times. If this copying of objects will be expensive, strongly consider using `boost::ref()` to prevent this from happening. Bear in mind that memory may have to be managed differently to ensure that the referenced bound parameter does not get deleted prematurely.

Functors from Static Member Functions

Example 5.6. Static Member Function for Thread Functor (1)

```
class MyObject
{
public:
    MyObject()
        : mThread(NULL)
    {
    }

    ~MyObject()
    {
        // Need to ensure that the thread is not running.
        if ( NULL != mThread )
        {
            delete mThread;
            mThread = NULL;
        }
    }

    void start()
```

```

    {
        mThread = new vpr::Thread(MyObject::run);
    }

    void stop()
    {
        // Tell the thread to exit ...
        mThread->join();
    }

private:
    static void run()
    {
        // Do work ...
    }

    vpr::Thread* mThread;
};

```

Example 5.7. Static Member Function for Thread Functor (2)

```

void MyObject::start(const int runCount)
{
    mThread = vpr::Thread(boost::bind(MyObject::run, runCount));
}

void MyObject::run(const int runCount)
{
    for ( int i = 0; i < runCount; ++i )
    {
        // Do work ...
    }
}

```

Example 5.8. Static Member Function for Thread Functor (3)

```

struct thread_args_t;

class MyObject
{
public:
    static void run(thread_args_t* args)
    {
        // Do work ...
    }
};

void spawnThread()
{
    thread_args_t* args = new thread_args_t();
    // Fill in the arguments to be passed to the thread...

    vpr::Thread* thread = new vpr::Thread(boost::bind(MyObject::run,
                                                    args));
}

```

Functors from Non-Member Functions

Functors for threads can be created for non-member functions. The process is basically the same as for using static member functions. The only real difference is that the class scoping does not need to be used since the non-member functions will not be in a class. A simple example of this is shown in Example 5.9, “Non-Member Function for Thread Functor”. As usual, proper memory management would be needed for the allocated `vpr::Thread` object. And, of course, parameters to the non-member function can be bound to the function using `Boost.Bind`.

Example 5.9. Non-Member Function for Thread Functor

```
void run()
{
    // Do some work ...
}

void startThread()
{
    vpr::Thread* thread = new vpr::Thread(run);
}
```

Functors from Callable Objects

A new capability not (easily) available with VPR 1.0 is the use of callable objects as functors. This means using an instance of a type that overloads `operator()` as the callable handled by `Boost.Function`. One way of doing this is shown in Example 5.10, “Callable Object for Thread Functor (1)”. Note that a *copy* of `c` will be made for use when the functor is invoked in the spawned thread.

Example 5.10. Callable Object for Thread Functor (1)

```
class Callable
{
public:
    void operator()()
    {
        // Do some work ...
    }
};

void startThread()
{
    Callable c;
    vpr::Thread* thread = new vpr::Thread(c);
}
```

A more interesting use of a callable object would be one that has state. There are two ways of using a stateful callable object. The first is to use data members in the class (or struct) declaration that are then accessed by the overload of `operator()`. This is a very simple thing to do since having data members in a class is so common.

The other is to use our old friend `Boost.Bind`, though it is not as simple as in previous cases. Since `Boost.Bind` is designed for binding parameters to function pointers, we have to use a proper function

pointer—or use `Boost.Function`. In Example 5.11, “Callable Object for Thread Functor (2)”, we see how `Boost.Bind` is used to bind the parameter value 5 to the invocation of `c.operator()(int)`. This is not nearly as simple as what was shown in Example 5.10, “Callable Object for Thread Functor (1)”, even if data members of `Callable` had to be initialized.

Example 5.11. Callable Object for Thread Functor (2)

```
class Callable
{
public:
    void operator()(int arg)
    {
        // Do some work ...
    }
};

void startThread()
{
    Callable c;
    vpr::Thread* thread =
        new vpr::Thread(boost::bind(&Callable::operator(), c, 5));
}
```

Since we are working with a callable object, it stands to reason that we should be able to leverage `Boost.Function` somehow. In Example 5.12, “Callable Object for Thread Functor (3)”, we see how `boost::function<T>` can be used to reduce the apparent complexity of binding the parameter value 5 to the invocation of `c.operator()(int)`. To some, this may look a little more friendly since there is no member function pointer required. Ultimately, it requires slightly more typing than the previous example. Either way, neither of these approaches are very pleasing to the eye, but they do work. One of these approaches may be necessary to use a legacy callable type with `vpr::Thread`.

Example 5.12. Callable Object for Thread Functor (3)

```
class Callable
{
public:
    void operator()(int arg)
    {
        // Do some work ...
    }
};

void startThread()
{
    Callable c;
    vpr::Thread* thread =
        new vpr::Thread(boost::bind(boost::function<void(int)>(c), 5));
}
```

Updating from VPR 1.0

In VPR 1.0, thread functors were handled by subclasses of `vpr::BaseThreadFunctor`. While this approach worked, it was not anywhere near as flexible as what is possible with `Boost.Function` and

Boost.Bind. What has been presented here is for VPR 1.1 and beyond. For those users updating from VPR 1.0, we now present the simple process of changing code using the VPR 1.0 thread API to the VPR 1.1 API.

First, the signature for the function called by the spawned thread in VPR 1.0 took a single `void*` parameter. In VPR 1.1, the function takes no argument. This change was made for two reasons. First, most functions ignored this parameter, so it was wasted memory on the stack. Second, `void*` is a notoriously poor choice for a type since it can point to anything. C++ is a strongly typed language, and we should be taking advantage of that feature. However, since the presence of the parameter acknowledged that it was often necessary to pass data into the thread function, it is still possible to do this using Boost.Bind. What this means is that the callable invoked by the functor can have *any* signature, but the default behavior is for it to return nothing and take no parameters.

Next, the types `vpr::ThreadNonMemberFunctor`, `vpr::ThreadMemberFunctor<T>`, and `vpr::ThreadRunFunctor<T>` have been removed, as has the header file `vpr/Thread/ThreadFunctor.h`. The functionality provided by those types has been offloaded to Boost.Function [<http://www.boost.org/doc/html/function.html>] and Boost.Bind [<http://www.boost.org/libs/bind/bind.html>]. In so doing, the flexibility of how threads are spawned has been increased immensely while actually making it simpler to spawn them.

For uses of `vpr::ThreadNonMemberFunctor`, quite a bit less code has to be written to spawn a thread. We see in Example 5.13, “VPR 1.0 Use of Thread Non-Member Functor” how this type was used with VPR 1.0 to spawn a thread that executed a static member function. Then, in Example 5.14, “Updated Use of Static Member Function for Thread Functor”, we see the equivalent code using the VPR 1.1 (and newer) interface. No longer must a functor object be instantiated on the heap and stored for later deletion¹. Moreover, there is no need to pass `NULL` as the parameter value, though if a parameter value is needed, then Boost.Bind must be used. Refer to the section called “Functors from Static Member Functions” for more details on this topic.

Example 5.13. VPR 1.0 Use of Thread Non-Member Functor

```
#include <vpr/Thread/ThreadFunctor.h>
#include <vpr/Thread/Thread.h>

class MyClass_VPR10
{
public:
    void start()
    {
        mFunctor =
            new vpr::ThreadNonMemberFunctor(MyClass_VPR10::run, NULL);
        mThread = new vpr::Thread(mFunctor);
    }

private:
    static void run(void* args)
    {
        // Do some work ...
    }

    vpr::ThreadNonMemberFunctor* mFunctor;
    vpr::Thread* mThread;
};
```

¹Leaking the instantiated functor was an all-too-common error with VPR 1.0.

Example 5.14. Updated Use of Static Member Function for Thread Functor

```
#include <vpr/Thread.h>

class MyClass
{
public:
    void start()
    {
        mThread = new vpr::Thread(MyClass::run);
    }

private:
    static void run()
    {
        // Do some work ...
    }

    vpr::Thread* mThread;
};
```

To use a non-static member function, either `vpr::ThreadMemberFunctor<T>` or `vpr::ThreadRunFunctor<T>` was used with VPR 1.0, as shown in Example 5.15, “VPR 1.0 Use of Thread Member Functor”. Now, `Boost.Bind` comes to our rescue and vastly simplifies the code needed to accomplish the very same thing. In Example 5.16, “Updated Use of Member Function for Thread Functor”, we see just how much less code is required. The nice thing is that the parameters that were passed to the `vpr::ThreadMemberFunctor<T>` constructor are nearly the same as what must be passed to `boost::bind()`. The order for the `this` pointer and the member function pointer are simply reversed—and the `NULL` value for the function argument is removed. If arguments need to be passed to the function, pass them in as arguments to `boost::bind()` after the `this` pointer. Refer to the section called “Functors from (Non-Static) Member Functions” for more details on this topic.

Example 5.15. VPR 1.0 Use of Thread Member Functor

```
#include <vpr/Thread/ThreadFunctor.h>
#include <vpr/Thread/Thread.h>

class MyClass_VPR10
{
public:
    void start()
    {
        mFunctor =
            new vpr::ThreadMemberFunctor<MyClass_VPR10>(
                this, &MyClass_VPR10::run, NULL
            );
        mThread = new vpr::Thread(mFunctor);
    }

private:
    void run(void* args)
    {
        // Do some work ...
    }

    vpr::ThreadMemberFunctor<MyClass_VPR10>* mFunctor;
```

```
    vpr::Thread* mThread;  
};
```

Example 5.16. Updated Use of Member Function for Thread Functor

```
#include <boost/bind.hpp>  
#include <vpr/Thread/Thread.h>  
  
class MyClass  
{  
public:  
    void start()  
    {  
        mThread = new vpr::Thread(boost::bind(&MyClass::run, this));  
    }  
  
private:  
    void run()  
    {  
        // Do some work ...  
    }  
  
    vpr::Thread* mThread;  
};
```

Chapter 6. Synchronization

When multiple processes or threads have access to the same data, synchronization of reads and writes becomes an important concern. For example, if one thread writes to a shared variable when another thread is reading, the value read will be corrupted. If two threads try to write to the same shared variable at the same time, one of the two writes will be lost. These situations can lead to unexpected, and often undesirable, program execution. For that reason, it is important to understand how to protect access to shared data so that the multi-threaded software will execute correctly.

Semaphores: Using `vpr::Semaphore`

The most important part of multi-threaded programming is proper thread synchronization so that access to shared data is controlled. Doing so results in consistency among all threads. Semaphores are a very common synchronization mechanism and have been used widely in concurrent systems. This section describes the cross-platform semaphore interface provided with and used by VPR. It does not explain what semaphores are or how to use them—it is assumed that readers are already familiar with the topic lest they probably would not be reading this chapter at all.

High-Level Description

As with threads, a cross-platform abstraction layer has been written to provide a consistent way to use semaphores on all supported platforms. The primary goal behind the interface design is to provide the common *P* (acquire) and *V* (release) operations. The interface does include methods for read/write semaphores, but as of this writing, that part of the interface is not complete. Because of that, the use section does not cover that part of the interface. When the implementation is complete, this section will be expanded.

As always, there is a header file that must be included to use `vpr::Semaphore`. This time around, the file is `vpr/Sync/Semaphore.h`. Do not include any of the platform-specific implementation files. That is all handled appropriately within `vpr/Sync/Semaphore.h`.

Creating a Semaphore

When creating a `vpr::Semaphore` object, give the initial value that represents the number of resources being controlled by the semaphore. If no value is given, the default is 1 which of course gives a binary semaphore. Binary semaphores are better known as mutexes (see the section called “Mutual Exclusion: Using `vpr::Mutex`” for more information about mutex use in VPR). An example of creating a simple semaphore to control access to five resources is as follows:

```
vpr::Semaphore sema(5);
```

This creates a semaphore capable of controlling concurrent access to five resources. At some point, if there is a need to change the number of resources, a method called `reset()` is provided. Pass the new number of resources, and the semaphore object is updated appropriately:

```
sema.reset(4);
```

The semaphore `sema` now controls access to only four resources.

Creation of a semaphore can fail, and if it does, the `vpr::Semaphore` constructor throws an exception of type `vpr::ResourceException`. If creation of the semaphore succeeds, then lock and unlock operations on that semaphore are guaranteed to succeed.

Locking a Semaphore

When a thread needs to acquire access to shared data, it locks a semaphore. In the `vpr::Semaphore` interface, this is accomplished using the `acquire()` method:

```
sema.acquire();
```

As expected, `acquire()` is a blocking call, so if the semaphore's value is less than or equal to 0, the thread requesting the lock will block until the semaphore's value is greater than 0. If the acquisition of the semaphore would cause a deadlock (because the thread already holds the semaphore lock), `vpr::DeadlockException` is thrown. Otherwise, the method is guaranteed to return when the lock is acquired.

Releasing a Locked Semaphore

Finally, when access to the critical section is complete, the semaphore is released using the `release()` method:

```
sema.release();
```

This method is guaranteed to return successfully.

The Gory Details

Those who have read the Gory Details section for `vpr::Threads` will find this section very familiar. As with `vpr::Threads`, there are several platform-specific semaphore implementation wrapper classes:

- `vpr::SemaphoreSGI`: A wrapper around IRIX shared-arena semaphores (refer to the `usnews-ema(3P)` and related manual pages for more information)
- `vpr::SemaphorePosix`: A wrapper around POSIX real-time semaphores (POSIX.1b, formerly POSIX.4)
- `vpr::SemaphoreNSPR`: An implementation of semaphores using NSPR primitives
- `vpr::SemaphoreWin32`: A wrapper around Win32 semaphores

Unlike `vpr::Thread`, however, there is no base interface from which these implementations inherit. Performance decreases caused by virtual functions are avoided this way.

The semaphore implementation used is chosen when VPR is compiled and will always match the thread implementation being used. When the VPR build is configured, preprocessor `#define` statements are made in `vpr/vprDefines.h` that describe the threading system and thus the semaphores to use. Based on that, the header file `vpr/Sync/Semaphore.h` makes several typedefs that set up one of the platform-specific implementations to act as the `vpr::Semaphore` interface. For example, if compiling on Linux, the class `vpr::SemaphorePosix` is typedef'd to `vpr::Semaphore`. Since the interface is consistent among all the wrappers, everything works as though that was the way it was written to behave.

Mutual Exclusion: Using `vpr::Mutex`

In addition to cross-platform semaphores, VPR provides an abstraction for cross-platform mutexes. Mu-

mutexes are a special type of semaphore known as a binary semaphore. Exactly one thread can hold the lock at any time. This very short section, however, is not about mutexes but rather about the `vpr::Mutex` interface provided with and used by VPR.

High-Level Description

The cross-platform mutex abstraction in VPR is critical for synchronizing access to shared data. Those who have read the section on `vpr::Semaphore` will find this section very, very familiar. The interface for `vpr::Mutex` is a subset of that for `vpr::Semaphore` since mutexes are binary semaphores. They can be locked and unlocked. That is all there is to know. The `vpr::Mutex` interface does include some methods for read/write mutexes, but this implementation is incomplete and is not documented here for that reason. When the implementation is finished, this documentation will be expanded.

The header file to include for using `vpr::Mutex` is `vpr/Sync/Mutex.h`. As with other classes discussed in this chapter, it is important not to include the platform-specific header files.

Creating a Mutex

When creating a `vpr::Mutex` object, there are no special parameters to pass or considerations to be made. An example of creating a mutex is as follows:

```
vpr::Mutex mutex;
```

Creation of a mutex can fail, and if it does, the `vpr::Mutex` constructor throws an exception of type `vpr::ResourceException`. If creation of the mutex succeeds, then lock and unlock operations on that mutex are guaranteed to succeed.

Locking a Mutex

When a thread needs to acquire access to shared data, it can lock a mutex. In the `vpr::Mutex` interface, this is accomplished using the `acquire()` method:

```
mutex.acquire();
```

As expected, `acquire()` is a blocking call, so if the mutex is already locked by another thread, the thread requesting the lock will block until the mutex is released by the other thread. If the acquisition of the mutex would cause a deadlock (because the thread already holds the mutex), `vpr::DeadlockException` is thrown. Otherwise, the method is guaranteed to return when the lock is acquired.

Attempting to Lock a Mutex

If there is a need to lock a mutex only when the call would *not* block, a method is provided to do this. It is called `tryAcquire()`, and it will not block if the mutex is already locked. It works as follows:

```
const bool locked = mutex.tryAcquire();
```

If the mutex is locked, `true` is returned. Otherwise, `false` is returned. The call does not block.

Testing the State of a Mutex

In addition to conditional locking, the state of a mutex can be tested to see if it is locked or unlocked. This is done using the `test()` method as follows:

```
const bool locked = mutex.test();
```

If the mutex is *not* locked, `false` is returned. Otherwise, `true` is returned.

Releasing a Locked Mutex

When access to the critical section is complete, a locked mutex is released using the `release()` method:

```
mutex.release();
```

This method is guaranteed to return successfully.

The Gory Details

Those who have read the Gory Details sections for `vpr::Threads` or for `vpr::Semaphores` will find this last section very familiar (and probably uninteresting at this point). As with `vpr::Threads` and `vpr::Semaphores`, there are several platform-specific mutex implementation wrapper classes:

- `vpr::MutexSGI`: A wrapper around IRIX shared-arena mutexes (refer to the `usnewlock(3P)` and related manual pages for more information)
- `vpr::MutexPosix`: A wrapper around POSIX real-time mutexes (POSIX.1b, formerly POSIX.4)
- `vpr::MutexNSPR`: A wrapper around NSPR mutexes
- `vpr::MutexWin32`: A wrapper around Win32 mutexes

Similar to `vpr::Semaphore`, there is no base interface from which these implementations inherit. Performance issues caused by virtual functions are avoided by doing this.

The mutex implementation used is chosen when VPR is compiled and will always match the thread implementation being used. When the VPR build is configured, preprocessor `#define` statements are made in `vpr/vprDefines.h` that describe the threading system and thus the mutexes to use. Based on that, the header file `vpr/Sync/Mutex.h` makes several typedefs that set up one of the platform-specific implementations to act as the `vpr::Mutex` interface. For example, if compiling on Solaris, the class `vpr::MutexPosix` is typedef'd to be `vpr::Mutex`. Since the interface is consistent among all the wrappers, everything works as though that was the way it was written to behave.

Condition Variables: Using `vpr::CondVar`

Condition variables are a helpful extension to mutexes. Every condition variable has an associated mutex, and thus they can be used to control mutually exclusive access to some resource. A condition variable adds the ability to test a condition and wait for its state to change in some meaningful way. Waiting threads are awakened when the state of the variable changes or when a time interval expires. When awoken, the relevant condition is tested. If the state has changed to the desired result, the thread continues its execution. If not, it goes back into the waiting state until it is awakened again to repeat the process.

High-Level Description

The interface for `vpr::CondVar` is very similar to that of `vpr::Mutex`. This is because every in-

stance of `vpr::CondVar` contains an instance of `vpr::Mutex`. Thus, acquiring and releasing a condition variable actually acquires and releases the contained mutex.

The header file to include for using `vpr::CondVar` is `vpr/Sync/CondVar.h`. As with other classes discussed in this chapter, it is important not to include the platform-specific header files.

Creating a Condition Variable

When creating a `vpr::CondVar` object, there are no special parameters to pass or considerations to be made. An example of creating a condition variable is as follows:

```
vpr::CondVar cv;
```

Creation of a condition variable can fail, and if it does, the `vpr::CondVar` constructor throws an exception of type `vpr::ResourceException`. If creation of the condition variable succeeds, then lock and unlock operations on that condition variable are guaranteed to succeed.

In addition to the `vpr::CondVar` instance, there is usually some associated variable whose value will be tested and modified by various threads. For our purposes, we will use a boolean variable:

```
bool state_var;
```

Locking a Condition Variable

When a thread needs to acquire access to shared data, it can lock a condition variable. In the `vpr::CondVar` interface, this is accomplished using the `acquire()` method:

```
cv.acquire();
```

As expected, `acquire()` is a blocking call, so if the condition variable's mutex is already locked by another thread, the thread requesting the lock will block until the mutex is released by the other thread. If the acquisition of the condition variable's lock would cause a deadlock (because the thread already holds the lock), `vpr::DeadlockException` is thrown. Otherwise, the method is guaranteed to return when the lock is acquired.

Attempting to Lock a Condition Variable

If there is a need to lock a condition variable's mutex only when the call would *not* block, a method is provided to do this. It is called `tryAcquire()`, and it will not block if the mutex is already locked. It works as follows:

```
const bool locked = cv.tryAcquire();
```

If the condition variable's mutex is locked, `true` is returned. Otherwise, `false` is returned. The call does not block.

Releasing a Locked Condition Variable

When access to the critical section is complete, a locked condition variable is released using the `release()` method:

```
cv.release();
```

This method is guaranteed to return successfully.

Prior to calling `vpr::CondVar::release()`, it is almost always necessary to call `vpr::CondVar::signal()` (or `vpr::CondVar::broadcast()`) to inform any waiting threads that the condition on which they are waiting may have changed. The next section describes this procedure in greater detail.

Putting Condition Variables to Use

So far, we have seen how to lock and unlock a condition variable, but we have not seen how to *use* a condition variable. Let us say that we have two threads running concurrently. One thread is performing an operation, and the other thread must wait until that operation is complete. Condition variables work wonderfully in such a situation. For the following description, we will call the first thread the worker thread and the second thread the waiting thread. For the example, we will use the condition variable `cv` and the boolean flag `state_var`, introduced above. Initially, `state_var` will be set to `false` to indicate that the worker thread has not done its job yet.

The Waiting Thread

While it may seem more logical to focus on the worker thread first, our example will flow better if we start with the waiting thread. That way, the waiting thread can be in its waiting state (at least conceptually) when we talk about the worker thread. The waiting thread will of course be waiting for the condition (the value of `state_var`) to change.

In order for the waiting thread to be waiting, it must have tested the value of `state_var`. Because `state_var` is shared data, the condition variable must be locked before it can read the current value of `state_var`. Depending on the current value, the waiting thread will either wait or continue its execution. The usual process for doing this is shown below:

```
cv.acquire();
{
    while ( state_var == false )
    {
        cv.wait();
    }
}
cv.release();
```

The call to `vpr::CondVar::wait()` is special: the mutex associated with `cv` is unlocked so that other threads can manipulate `state_var`, but the waiting thread will block. When the waiting thread is awakened, it will regain the lock it previously held on the condition variable. The thread will test `state_var` again, and if its value is now `true`, the waiting thread will exit the loop and release the condition variable. If the value of `state_var` is still `false`, the waiting thread will go back to its waiting state.

The Worker Thread

The worker thread has some task to perform that is critical to the proper execution of the two active threads. In order to inform the waiting thread about the current status of the task, we will use the condition variable `cv` and the boolean flag `state_var`, shown above. While the worker thread is performing its task, it must hold the lock on the condition variable, as shown below:

```
cv.acquire();
{
    // Perform our critical task ...

    state_var = true;
```

```
    cv.signal();  
}  
cv.release();
```

Once the “critical task” is complete, the value of `state_var` is changed to `true` to indicate that the job is done. Then, `vpr::CondVar::signal()` is invoked. This will wake up the waiting thread, and the condition variable is released. The result is that the waiting thread will wake up to discover that the job has been completed, and it will continue with its execution.

Note

The method `vpr::CondVar::signal()` will wake up at most one waiting thread. Which thread is awakened is determined by the operating system scheduling algorithms. To wake up all threads, use `vpr::CondVar::broadcast()`. Of course, each waiting thread will have to wait its turn to get exclusive access to test the condition, but this can be useful when it is known that many threads are all waiting on the same result.

Chapter 7. Signal Handling

VPR contains an abstraction for allowing cross-platform signal handling. The interface is based on that used in the ADAPTIVE Communication Environment [<http://www.cs.wustl.edu/~schmidt/ACE.html>] (ACE). The basic idea is that a set of signals is associated with a signal handler. The handler is registered with the operating system, and whenever one of the signals in the signal set is delivered to the process (thread), the handler is invoked.

The signal set is encapsulated within an instance of `vpr::SignalSet`, and a signal handler is simply a callback function. The two are combined using the concept of a signal action which is implemented in the class `vpr::SignalAction`. The signal action registers the handler for the given set of signals with the operating system.

Part III. Miscellaneous Utilities

Table of Contents

| | |
|--------------------------------------|----|
| 8. Simulated Sockets | 38 |
| Sim Socket Interface | 38 |
| Sim Socket Implementation | 38 |
| 9. Globally Unique Identifiers | 39 |
| GUID Creation | 39 |
| GUID Operations | 39 |
| 10. Intervals | 40 |
| Interval Creation | 40 |
| Interval Operations | 40 |
| 11. Singletons | 41 |
| 12. Factories | 44 |
| 13. Performance Monitoring | 45 |

Chapter 8. Simulated Sockets

The simulated sockets (often referred to as “sim sockets”) in VPR are designed to allow testing of network algorithms without requiring the use of a physical computer network. All communication between “nodes” occurs in memory by simulating the actual process of sending and receiving data on a network. This is done through a discrete event simulation where the events are the arrival of a message (packet) at a node in the network. Because we have chosen the message arrival event as the step size, the simulation can proceed at a very high rate. Within the network, timing calculations are performed so that messages are delivered based on the constraints of the network (bandwidth and transmission latency are the primary constraints in the current version).

Sim Socket Interface

The interface for sim sockets is identical to that of the operating system abstraction layer for sockets, namely `vpr::Socket`, `vpr::SocketStream`, and `vpr::SocketDatagram`. Indeed, user code actually uses instances of those classes. The use of sim sockets versus real sockets is made when VPR is compiled, in the same way that the threading abstraction (NSPR versus POSIX versus Win32 versus SPROC) is chosen at compile time. Ideally, user code should not have to change at all to use sim sockets, thus making it possible to test network algorithms with exactly the same code as would be used with real sockets.

Of course, the real world is not ideal. While it is very true that most code does not have to change to use sim sockets, there is one major exception. Because the sim sockets are based on an event-driven simulation, something must be driving the simulation. When using sim sockets, user code must have a separate thread running that makes use of the singleton `vpr::sim::Controller`. The thread contains a loop that invokes `vpr::sim::Controller::processNextEvent()` or `vpr::sim::Controller::processEvents()` at each iteration.

Another aspect of the current sim socket implementation is that it is designed to execute in a single thread using non-blocking socket semantics. We have chosen this design because we felt that the complexity of writing a multi-threaded socket simulation that allowed sockets to block would be far too difficult. Future versions may allow blocking or non-blocking semantics to be chosen by user code, but such a feature is not expected in the near future.

Sim Socket Implementation

The implementation of the sim socket code is based on the Boost Graph Library [<http://www.boost.org/libs/graph/doc/>]. Vertices represent network nodes (also known as hosts), and edges represent network lines. The network topology is constructed using output from the Tiers software [<http://www.geocities.com/ResearchTriangle/3867/sourcecode.html>]. Once constructed, an in-memory network serves as the backend for the higher level socket API. Messages (blocks of memory) are passed between nodes until they reach their destination.

Chapter 9. Globally Unique Identifiers

Globally unique identifiers (GUIDs), also known as universally unique identifiers (UUIDs), provide 128-bit identifiers that are guaranteed to be unique (speaking statistically). These are based on an Internet draft written by Paul L. Leach at Microsoft. The likelihood of two identifiers being the same varies depending on the generation mechanism used. In VPR, we define the type `vpr::GUID` as a cross-platform wrapper around the process of generating GUIDs and the process of serializing GUIDs for network transmission.

GUID Creation

Creating GUIDs is easy. The default `vpr::GUID` constructor will create a “null” GUID. A fresh identifier can be generated by invoking the method `vpr::GUID::generate()` on the newly created null GUID. To create a constant GUID, the string representation of the GUID can be passed to the `vpr::GUID` constructor. Such a string can be created using a command-line utility such as **uuidgen**.

The GUID specification describes creating identifiers using a “namespace” concept. The last `vpr::GUID` constructor takes two arguments: a namespace GUID and a string name. The namespace GUID is the same for all GUIDs created in the namespace. The string name is used to create a hash value that makes the GUID unique for that name within the namespace. The full algorithm and semantics of GUID namespaces are beyond the scope of this document, but the functionality is provided to offer a complete (cross-platform) implementation of the specification.

GUID Operations

The `vpr::GUID` interface provides overloaded methods for comparison operations (equality and less-than) between two `vpr::GUID` instances. A function is also provided for generating a hash value for a given GUID so that they may be used in STL indexed containers.

Serialization of `vpr::GUID` objects uses the same mechanisms described in Chapter 4, *Data Marshaling*.

Chapter 10. Intervals

The `vpr::Interval` class defines a high-resolution interval that is based on an unsigned, always-increasing counter. These intervals are only valid for about twelve (12) hours, which implies that they are only useful for a range of roughly six hours. When the internal counter reaches its maximum value, the interval overflows. The overloaded operator- will deal with overflows so that two intervals can be compared.

Interval Creation

Intervals are constructed with a constant value and a time unit, either microseconds, milliseconds, or seconds. Otherwise, an empty interval (whose internal counter is initialized to zero) can be created using the default value. The internal counter of any interval instance is modified using one of the “setter” methods.

Interval Operations

The set methods take various forms. For example, the method `vpr::Interval::setNow()` sets the internal counter to the current time. The variants of `vpr::Interval::set()` (including `vpr::Interval::setf()` and `vpr::Interval::setd()`) set the counter to an explicit value. As with the constructor, these take a second argument that determines the time units. There are helper methods such as `vpr::Interval::sec(vpr::Uint64)` and `vpr::Interval::usecf(float)` that use the correct time unit internally. Use of these can be less error prone than giving both the time and the unit, but which version to use is up to the user.

The value of an interval is queried to return using a specific time unit. For example, `vpr::Interval::msecf()` returns the interval value as a floating-point number representing milliseconds. The “base” counter value can be returned. Conceptually, this is a unit-less value. Strictly speaking, its units are defined by platform-specific details that determine how accurate the counter can be.

Mathematical operations can be performed on `vpr::Interval` instances. They allow addition, subtraction, and comparison. As mentioned above, the overloaded subtraction operator will deal with overflow to allow proper comparison of two `vpr::Interval` instances. However, users should always bear in mind the fact that the intervals are only good for ranges of about six (6) hours.

Chapter 11. Singletons

A *singleton* is a common design pattern [Gam95]. VPR offers two ways to implement a singleton, both of which come from the header `vpr/Util/Singleton.h`. Libraries such as Loki [<http://sourceforge.net/projects/loki-lib/>] offer other approaches.

The first way of creating a singleton using VPR is to use the C preprocessor approach. In this case, we utilize two macros: `vprSingletonHeader()` and `vprSingletonImp()`. One goes in the class declaration (in the header file), and the other goes with the class definition (in the `.cpp` file). The basic usage of these macros is shown in Example 11.1, “Use of `vprSingletonHeader()`” and Example 11.2, “Use of `vprSingletonImp()`”.

Example 11.1. Use of `vprSingletonHeader()`

```
#include <boost/noncopyable.hpp>
#include <vpr/Util/Singleton.h>

class MySingleton : boost::noncopyable
{
public:
    // Public operations ...
    void doSomething();

private:
    // Prevent instantiation by user code.
    MySingleton()
    {
        // Some constructor actions ...
    }

    // Bring in the singleton declaration pieces.
    vprSingletonHeader(MySingleton);
};
```

Example 11.2. Use of `vprSingletonImp()`

```
#include "MySingleton.h"

// Bring in the singleton definition pieces.
vprSingletonImp(MySingleton);

void MySingleton::doSomething()
{
    // Do something ...
}
```

In some cases, an object may need to perform some initialization steps after being instantiated. In that case, use the macro `vprSingletonHeaderWithInitFunc()` instead of `vprSingletonHeader()` and `vprSingletonImpWithInitFunc()` instead of `vprSingletonImp()`. An example of this is shown in Example 11.3, “Use of `vprSingletonHeaderWithInitFunc()`” and Example 11.4, “Use of `vprSingletonImpWithInitFunc()`”. The initialization method must take no parameters and return nothing.

Example 11.3. Use of `vprSingletonHeaderWithInitFunc()`

```
#include <boost/noncopyable.hpp>
#include <vpr/Util/Singleton.h>

class MySingleton : boost::noncopyable
{
public:
    // Public operations ...
    void doSomething();

private:
    // Prevent instantiation by user code.
    MySingleton()
    {
        // Some constructor actions ...
    }

    void init()
    {
        // Perform initialization operations ...
    }

    // Bring in the singleton declaration pieces.
    vprSingletonHeaderWithInitFunc(MySingleton, init);
};
```

Example 11.4. Use of `vprSingletonImpWithInitFunc()`

```
#include "MySingleton.h"

// Bring in the singleton definition pieces.
vprSingletonImp(MySingleton);

void MySingleton::doSomething()
{
    // Do something ...
}
```

In VPR 1.1 and beyond, the destructor for singleton objects is called when the application exits. This can result in some undesirable behavior if one singleton depends on another, thus meaning that the order in which they are deleted can make a difference. Without any specific ordering being imposed, singleton objects are deleted in the reverse order in which they were instantiated. Since it is not always possible to control the order of instantiation, it can be useful to control the order of destruction. This can be done by setting the “lifetime” or “longevity” of a singleton. The higher the longevity, the later that the singleton will be deleted when the application is exiting. This is accomplished using either `vprSingletonImpLifetime()` or `vprSingletonImpLifetimeWithInitFunc()`. The macro used must be paired with either `vprSingletonHeader()` or `vprSingletonHeaderWithInitFunc()`. Usage of `vprSingletonImpLifetime()` is shown in Example 11.5, “Use of `vprSingletonImpLifetime()`”.

Example 11.5. Use of `vprSingletonImpLifetime()`

```
#include "MySingleton.h"

// Bring in the singleton definition pieces. Delete it early
// in the application exit process.
vprSingletonImpLifetime(MySingleton, 10);

void MySingleton::doSomething()
{
    // Do something ...
}
```

The other approach uses a template class called `vpr::Singleton<T>`. An example of its use is shown in Example 11.6, “Use of `vpr::Singleton<T>`”.

Example 11.6. Use of `vpr::Singleton<T>`

```
1 #include <boost/noncopyable.hpp>
  #include <vpr/Util/Singleton.h>

  class MySingleton
5   : public vpr::Singleton<MySingleton>
    , boost::noncopyable
  {
  public:
    // Public operations ...
10  void doSomething()
    {
        // Do something ...
    }

15 private:
    // Required so that vpr::Singleton can instantiate this class.
    friend class vpr::Singleton<MySingleton>;

    // Prevent instantiation by user code.
20  MySingleton()
    {
        // Some constructor actions ...
    }
  };
```

Warning

Singletons implemented using `vpr::Singleton<T>` do not work across DLLs on Microsoft Windows® or shared libraries (`.dylib` files) on Mac OS X. If the singleton is for internal use only, such as in the case of an application, then using `vpr::Singleton<T>` is fine. Otherwise, the C preprocessor approach must be used.

In all singleton implementations, getting a reference to the singleton object (and calling the `doSomething()` method) is then done using the following syntax:

```
MySingleton::instance()->doSomething();
```

Chapter 12. Factories

Chapter 13. Performance Monitoring

The Performance Monitoring part of the VPR library enables users to obtain and use performance metrics easily. The basic concept for VPR's performance monitoring is that there is a manager that is in charge of keeping track of all the different data coming in. This is called the Profile Manager (from the class `vpr::ProfileManager`).

The Profile Manager keeps a one-to-many tree of the different metrics (instances of `vpr::ProfileSample`) being used. When the Profile Manager starts profiling a section of code, it looks to see if it had been profiled before and if so adds another sample to that profile otherwise it is added to the tree. Upon the completion of the function the destructor asks the Profile Manager to stop profiling.

Along with being able to keep track of the named profiles, there is an ability to keep a specified number of samples so the Profile Manager will keep more than just the last sample obtained. There are also two macros provided for ease of use and to minimize the intrusiveness of the API. These are the `VPR_PROFILE_GUARD(name)` and `VPR_PROFILE_GUARD_HISTORY(name, queue_size)` macros. These can be simply added at the top of functions that are to be profiled. Also, these macros give the ability at compile time to disable all profiling by way of the variable `VPR_DISABLE_PROFILE`. All of the profiling data is obtained by using the `vpr::Interval` class to measure the time spent in a particular piece of code using the highest precision timer available to the VPR.

Calls may be made to `vpr::ProfileManager::startProfile("name")` and to `vpr::ProfileManager::stopProfile()` so that just a certain part of the code may be profiled. A name needs to be specified when a profile is started but when `vpr::ProfileManager::stopProfile()` is called it simply stops the last profile that was started. Starts and stops can be nested within a function or even a program as long as their is a corresponding start with every stop.

An iterator class is provided to allow the user to use the captured data for statistic calculations, overloaded output stream operator for printing out, etc. The iterator class is what the user can use to collect information about the tree structure and interpreting the tree structure for hierarchical calls within the profiles. For profiles that contain more than one sample there are functions provided to get the short term average. Also, to quickly just print out the statistics at any point the user may call `VPR_PROFILE_RESULTS()` and the current status of all profiles will be printed out via `vpr::DEBUG`.

Part IV. Appendices

Table of Contents

| | |
|--|----|
| A. I/O Implementation Information | 48 |
| B. GNU Free Documentation License | 49 |
| PREAMBLE | 49 |
| APPLICABILITY AND DEFINITIONS | 49 |
| VERBATIM COPYING | 50 |
| COPYING IN QUANTITY | 50 |
| MODIFICATIONS | 51 |
| COMBINING DOCUMENTS | 52 |
| COLLECTIONS OF DOCUMENTS | 53 |
| AGGREGATION WITH INDEPENDENT WORKS | 53 |
| TRANSLATION | 53 |
| TERMINATION | 53 |
| FUTURE REVISIONS OF THIS LICENSE | 54 |
| ADDENDUM: How to use this License for your documents | 54 |

Appendix A. I/O Implementation Information

All of the buffered I/O code is based around a Bridge pattern [Gam95], though the actual implementation of the bridge is not as straightforward as described in the literature. However, the implementation can be divided into the platform-specific wrapper classes (e.g., `vpr::SocketStreamImplNSPR`) and “containers” for those classes (e.g., `vpr::SocketStream_t<T>`). Ultimately, the type seen in user code (e.g., `vpr::SocketStream`) is a typedef for a specific container class instantiation.

The typedef is based on the concept of a platform-specific “domain”, as specified in the header `vpr/vprDomain.h`. These domains allow a form of parameterization of VPR components using the C pre-processor. For example, when compiling on IRIX with SPROC threads, the domain defines the use of the SPROC threading subsystem, BSD sockets, and termios serial I/O. When compiling on Windows, where only NSPR is used, the domain specifies the use of native Win32 serial I/O and NSPR for socket I/O. Moreover, the simulated sockets can be mixed with any of the threading subsystems using this paradigm.

Appendix B. GNU Free Documentation License

Version 1.2, November 2002

FSF Copyright note

Copyright (C) 2000,2001,2002 Free Software Foundation, Inc. 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondly, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sec-

tions then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you

as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

GNU FDL Modification Conditions

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
- D. Preserve all the copyright notices of the Document.
- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- H. Include an unaltered copy of this License.
- I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the

title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

- J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- M Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
- N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
- O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the

combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Sample Invariant Sections list

Copyright (c) YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

Sample Invariant Sections list

with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Bibliography

- [Gam95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Elements of Reusable Object-Oriented Software. Addison-Wesley. 1995.
- [Nic96] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads Programming*. A POSIX Standard for Better Multiprocessing. O'Reilly & Associates. 1996.
- [Sch00] Douglas Schmidt. *Pattern-Oriented Software Architecture*. Volume 2. Patterns for Concurrent and Networked Objects. John Wiley & Sons. 2000.
- [Ste92] W. Richard Stevens. *Advanced Programming in the UNIX Environment*. Addison-Wesley. 1992.
- [Ste98] W. Richard Stevens. *UNIX Network Programming*. Volume 1. Network APIs: Sockets and XTI. Second Edition. Prentice-Hall PTR. 1998.

Glossary of Terms

B

BSD sockets

The socket programming interface introduced with the Berkeley Software Distribution version of the UNIX operating system. It is made up of a collection of system calls that allow highly flexible socket programming. Most UNIX variants in use today use the BSD sockets API. Moreover, the Winsock API used on Windows is based on this API.

F

functor

See Something that performs an operation or function. In the scope of object-oriented programming, a functor is an object that behaves similarly to a function pointer in C..

N

Netscape Portable Runtime

More information can be found at <http://www.mozilla.org/projects/nspr/index.html>

Index

B

byte order
 host/network conversion, 8

C

classes
 vpr::BaseIOStatsStrategy, 3
 vpr::BlockIO, 2, 2
 vpr::BufferObjectReader, 9
 vpr::BufferObjectWriter, 9
 vpr::CondVar, 31
 vpr::GUID, 39
 vpr::InetAddr, 4
 vpr::Interval, 40
 vpr::IOStatsStrategyAdapter<S, T>, 3
 vpr::Mutex, 31
 vpr::ObjectReader, 8
 vpr::ObjectWriter, 8
 vpr::Semaphore, 30
 vpr::SerializableObject, 8
 vpr::Singleton<T>, 43
 vpr::Socket, 4, 5, 6
 vpr::SocketAcceptor, 4, 6
 vpr::SocketConnector, 4, 6
 vpr::SocketDatagram, 4, 5, 6
 vpr::SocketStream, 4, 5, 6, 48
 vpr::SocketStream_t<T>, 48
 vpr::SocketStreamImplNSPR, 48
 vpr::System, 8
 vpr::Thread, 16
 vpr::XMLObjectReader, 9
 vpr::XMLObjectWriter, 9
constants
 vpr::InetAddr::AnyAddr, 4

E

exceptions
 vpr::DeadlockException, 29, 30, 32
 vpr::IllegalArgumentException, 14, 14, 14
 vpr::ResourceException, 28, 30, 32

F

functor, 11, 16
 use with vpr::Thread, 16
 (see also vpr::Thread)
functors
 member functions, 18
 non-member functions, 23
 non-static member functions, 18
 static member functions, 21

I

input/output abstraction, 2
 built-in statistics collection, 3

M

macros
 vprSingletonHeader, 41
 vprSingletonHeaderWithInitFunc, 41
 vprSingletonImp, 41
 vprSingletonImpLifetime, 42
 vprSingletonImpLifetimeWithInitFunc, 42
 vprSingletonImpWithInitFunc, 41
multi-threading
 vpr::Thread, 11

N

NSPR, 16

O

object serialization, 7, 8

S

serial port abstraction, 7
socket abstraction, 4
sockets
 acceptor/connector pattern, 6
 blocking semantics, 5
 common features of, 4
 datagram, 6
 Internet addresses, 4
 stream, 6

T

thread of control, 10

V

vpr::BaseThreadFunctor
 description of, 16
vpr::CondVar, 31
 creating, 32
 description of, 31
 locking, 32
 without blocking, 32
 releasing, 32
vpr::Mutex, 29
 creating, 30
 description of, 30
 details, 31
 locking, 30
 without blocking, 30
 releasing, 31
 testing state, 30
vpr::Semaphore, 28
 creating, 28
 description of, 28
 details, 29
 locking, 29

- releasing, 29
- reset() method, 28
- vpr::Socket
 - polymorphism and, 5
- vpr::Thread, 11
 - canceling, 14
 - creating, 12
 - details, 15
 - joining threads, 13
 - priority, 14
 - self, 15
 - sending signals, 14
 - suspend, resume, 13