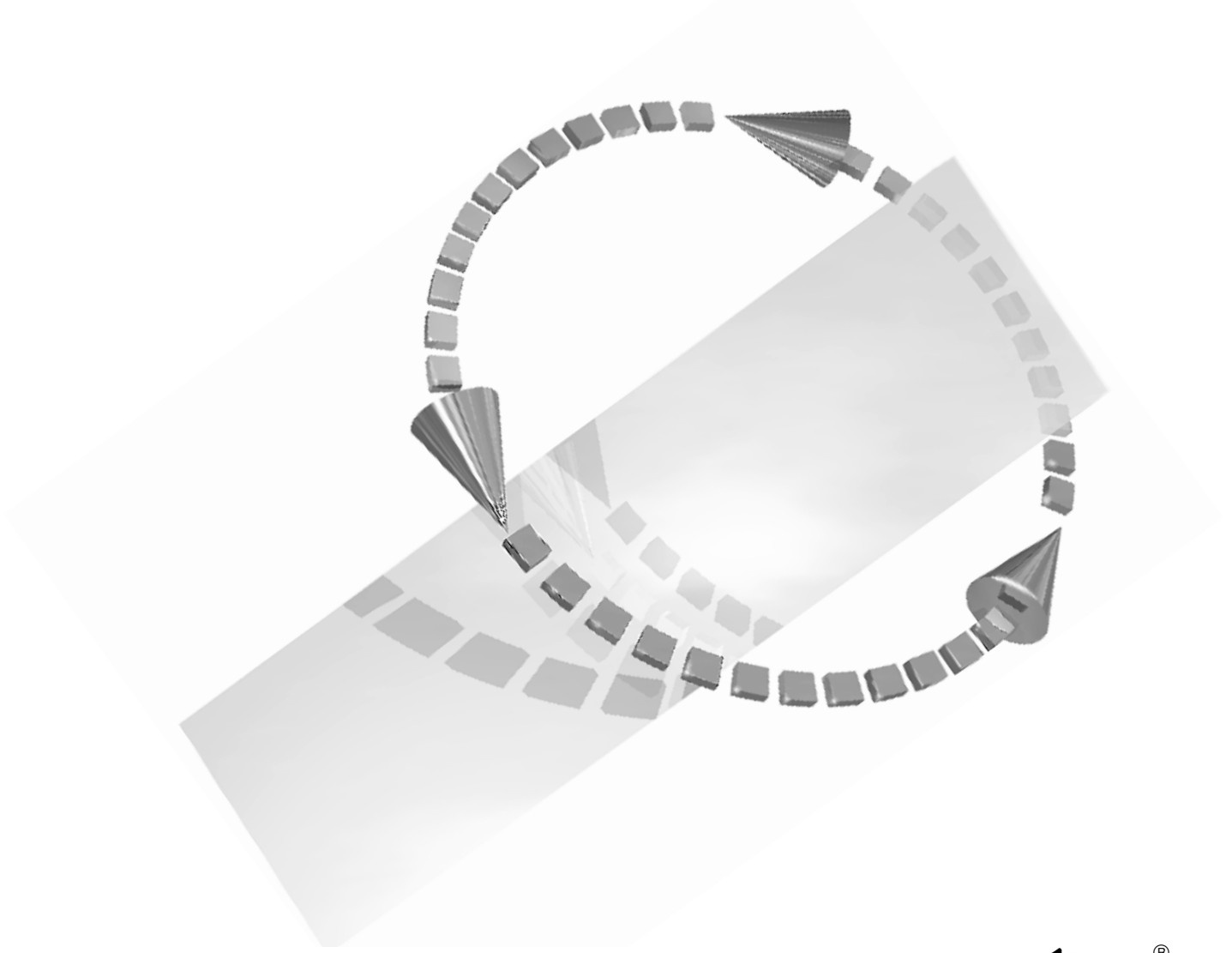
MÄK TECHNOLOGIES

VR-Link

The Networking Toolkit for the Virtual World



MÄK VR-Link®

The Networking Toolkit
for the Virtual World

MÄK TECHNOLOGIES

MÄK Technologies
185 Alewife Brook Parkway
Cambridge, MA 02138 USA

Second Printing: May, 1999

MÄK Technologies
185 Alewife Brook Parkway
Cambridge, MA 02138 USA

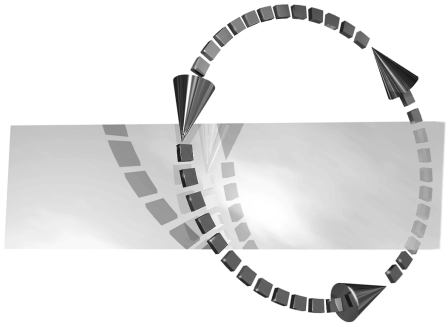
Voice: 617-876-8085

Fax: 617-876-9208

info@mak.com

www.mak.com

Revision VRL-3.4-1-990513



Contents

Preface

Chapter 1 Introduction

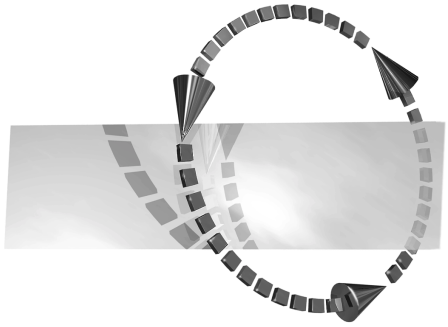
Chapter 2 Installing and Configuring VR-Link

Chapter 5 The Protocol-Independent Interface

Chapter 6 The HLA-Specific Interface

Chapter 7 FOM Agility Examples

Chapter 10 Example and Utility Applications



Preface

This guide, like VR-Link® itself, is intended for use by developers creating simulators and other HLA and DIS applications. It assumes that you are thoroughly familiar with the C++ language and have a working knowledge of HLA and DIS concepts.

We are constantly improving VR-Link to meet user needs. Consequently, some new features may have been implemented after this manual went to press. For the most recent information, please read the *RELNOTES* file or any addendums or release notes distributed with your version of VR-Link.

How this Manual is Organized

The chapters are organized as follows:

Chapter 1, *Introduction*, describes the main features of VR-Link and contains information about simulation protocols.

Chapter 2, *Installing and Configuring VR-Link*, explains how to install the software, and get technical support.

Chapter 3, *VR-Link Concepts*, provides a general description of what VR-Link does and shows how its classes work together. This chapter also discusses code examples for simple VR-Link applications.

Chapter 4, *Compiling VR-Link Applications* describes the procedures for compiling and linking applications built with VR-Link.

Chapter 5, *The Protocol-Independent Interface*, describes the most commonly-used VR-Link classes and functions, and shows examples of their use.

Chapter 6, *The HLA-Specific Interface*, describes classes and functions that support the High Level Architecture for simulations. Most HLA simulator projects do not need to make use of these “lower-level” features.

Chapter 7, *FOM Agility Examples*, describes the examples that demonstrate how to implement FOM mapping and extending FOMs.

Chapter 8, *The DIS-Specific Interface*, describes toolkit functionality that pertains only to the DIS protocol. Most DIS simulator projects do not need to make use of these "lower-level" features.

Chapter 9, *Utilities*, explains how to use VR-Link's utility functions to perform time management, coordinate conversions and other tasks.

Chapter 10, *Example and Utility Applications*, explains how to use the example and utility applications that ship with VR-Link, including *buffgrep*, *netdump*, *nethex*, *hlaNetdump*, *vdc*, *vrlinkd3i*, *buffgrep talk*, *listen*, and *f18*.

Appendix A, *Guide to VR-Link Header Files* lists VR-Link header files and cross references them to VR-Link classes.

The *Glossary* defines important terms that you need to understand.

VR-Link Documentation

The VR-Link documentation set is as follows:

- *MÄK VR-Link Developer's Guide*. This manual is available in printed format and as an online manual in Adobe Portable Document Format (PDF).
- *MÄK VR-Link Reference Manual*. The *Reference Manual* is available only as an online manual in PDF format. It contains the code for all VR-Link header files. Appendix A of the *Developer's Guide* is also included in the *Reference Manual*. The class listings are presented as hypertext links to the header files that define them.
- Class documentation. Classes are documented in a set of protocol-specific PDF files. HLA classes are documented in *vrl3-xHLAclasses.pdf* and DIS classes are documented in *vrl3-xDISclasses.pdf*. Class hierarchy diagrams are in the files *vrl3-xHLAclassdiagram.pdf* and *vrl3-xDISclassdiagram.pdf*.
- Release documentation. Release documentation consists of release notes provided as a text file named *RELNOTES* and in printed format.

The PDF files are in the *./doc* directory.

Online Manuals

The online manuals use hypertext links for the entries in the table of contents, for all index entries, and for cross references. These files also support full text searching.

To view and print PDF documents, you need a copy of the free Acrobat Reader software (version 3.0 or later). It is available for SunOS and Solaris, IRIX, HP-UX, Digital UNIX, Windows, and other platforms. To obtain a copy, visit www.adobe.com.

Other MÄK Products

VR-Link is a member of the MÄK Technologies line of HLA/DIS software products designed to streamline the process of developing and using networked simulated environments.

In addition to VR-Link, the MÄK Technologies product line includes:

- The MÄK RTI. An RTI (Run-Time Infrastructure) is required to run applications using the High Level Architecture (HLA). The MÄK RTI is the first commercial run-time infrastructure for HLA simulations. It is optimized for real-time simulations.
- The MÄK Stealth. The Stealth displays a realistic, three-dimensional view of your virtual world. You can view this world from the inside of a simulated moving vehicle, or place the eyepoint at another moving or stationary location. The Stealth lets you switch rapidly among several predefined viewpoints while the simulation is underway. The Stealth runs on PCs and the SGI computers and supports multi-channel views.
- The MÄK Logger. The Logger, also called the Data Logger, can record HLA and DIS exercises and play them back for after-action review. You can play a recorded file at speeds above or below normal, or in reverse, and locate areas of interest quickly. The Logger lets you choose between a graphical interface, or a shell-style, command-line interface. You can also control the Logger programmatically with logger control PDUs or state update messages.

  The Logger comes with a set of utility programs that let you inspect, combine and edit Logger recordings.
- The MÄK Plan View Display. The Plan View Display (PVD) provides a two-dimensional, “big picture” of a virtual environment. It shows simulated entities, such as vehicles, soldiers, and tanks moving over a 2D-map display.

Please contact MÄK Technologies for more information about these products and their supported platforms.

How to Contact Us

For VR-Link technical support, information about upgrades, and information about other MÄK products, you can reach us in the following ways:

For sales and upgrade information by e-mail:	info@mak.com
For technical support by e-mail:	support@mak.com
For license key files and information:	www.mak.com/support/keyintro.htm
The MÄK web site:	www.mak.com

When requesting support, please tell us the product you are using, the version, and the platform on which you are running.

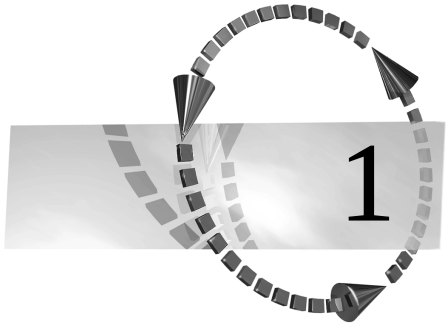
Send postal correspondence to:		MÄK Technologies 185 Alewife Brook Parkway Cambridge, MA, USA 02138
Call or fax us at:	Voice:	617-876-8085 (extension 3 for support)
	Fax:	617-876-9208

Document Conventions

This document uses the following typographic conventions:

`Monospaced Type`	Indicates commands or values you enter.
`Monospaced Italic`	Indicates variables that you replace with appropriate values.
`[ ]`	Indicates optional arguments.
`|`	Separates options in a command where only one option may be chosen at a time.
/	Indicates a directory. Since MÄK products run on both UNIX and PC platforms, we use the / (slash) for generic discussions of path-names. If you are running on a PC, substitute a \ (backslash) when you type pathnames.
Bold type	Indicates a new term.
Italic	Indicates a file name or path, or a class name.
sansSerif	Indicates a parameter or argument.
>	Indicates a one-step procedure.
Menu ③Option	Indicates a menu choice. For example, an instruction to select the Save option from the File menu appears as: Choose File Save.
	Indicates supplemental or clarifying information.
	Indicates additional information that you must observe to ensure the success of a procedure or other task.
	Indicates a warning about potential damage to the program or computer.

Directory names are preceded with dot and slash characters that show their position with respect to the VR-Link home directory. For example, the directory *vrlink/binRPR* appears in the text as *./binRPR*.



Introduction

The VR-Link toolkit is an object-oriented library of C++ classes, functions, and definitions that minimize the effort required to create networked simulators and virtual reality applications. VR-Link's powerful, easy-to-use programmer's interface greatly reduces development cost, time and risk.

This chapter covers the following topics:

1.1. The Protocol-Independent API

With VR-Link's protocol-independent API, you can simulate local entities, set their state, and automatically send entity information to other applications over a network using Distributed Interactive Simulation (DIS) or the High-Level Architecture (HLA) Run-Time Infrastructure (RTI). VR-Link also simplifies receiving and processing information from other applications, providing easy access to the state of remote entities. VR-Link handles dead reckoning, thresholding, responding to attribute requests, filtering, and many other tasks.

Because VR-Link supports both DIS and HLA through very similar APIs, you can often switch your applications between the two by changing just a few lines of initialization code and recompiling. This means that your VR-Link-based applications can maintain the DIS compliance vital to ongoing projects, while migrating to HLA.

DIS only

VR-Link implements all DIS PDUs and you can add support for user-defined PDUs.

1.1.1. FOM Support

VR-Link comes with built-in support for the Realtime Platform Reference FOM (RPR FOM). It can be configured or extended to add new attributes, objects and interactions to the RPR FOM, or to work with other FOMs entirely.

1.2. VR-Link Features

VR-Link provides a range of features to help you create and maintain HLA and DIS applications:

- Exercise Connection: An exercise connection is a VR-Link application's interface to an HLA or DIS exercise. It provides a protocol-independent[1] interface through which to exchange simulation information with other applications, either through the DIS network, or HLA's RTI.
- Object tracking: A VR-Link application uses a Reflected Entity List to keep track of remote participants in your virtual world by processing incoming attribute updates (through either DIS or HLA), and provides a protocol-independent[1] interface to their state. It submits update requests as needed, provides notice when an entity enters or leaves an exercise, and performs dead-reckoning, trajectory smoothing and filtering, as desired. Reflected object lists are available for other types of objects, such as emitters, transmitters, and so on.
- Object publishing: A VR-Link application uses an Entity Publisher to keep remote applications informed about the state of entities that you simulate locally. You periodically set the current state of your objects through its protocol-independent[1] interface, and the Entity Publisher automatically sends state updates when data changes or exceeds configurable thresholds. Object publishers are available for other types of objects, such as emitters, transmitters, and so on.
- Interaction classes: VR-Link provides a protocol-independent[1] interface to the sending and receiving of interaction messages that describe simulation events such as weapon fires, detonations, and radio signal transmissions. C++ classes representing DIS PDUs and HLA interactions usually provide mutator and inspector functions to access each field. Variable length PDUs and parameters are resized transparently. Functions are provided to print human-readable representations of interaction data.
- FOM-Agility: While VR-Link comes with built-in support for the RPR FOM, the FOM Mapper class allows you to map VR-Link's existing protocol-independent[1] API to another FOM's parameters, attributes, and object or interaction classes. In this way, code that uses the API does not need to change when the FOM changes.
- User extensibility: VR-Link's C++ API and implementation allow you to override most of its default functionality through subclassing. You can extend the toolkit to work with new types of HLA object or interaction classes and with new user-defined DIS PDUs.
- Access to low-level details: For developers who want to work below the level of abstraction provided by our top-level API, VR-Link provides protocol-specific classes and functions. Low-level access includes direct access to the HLA RTI and to the details of network configuration in DIS.

1. The interface is mostly protocol-independent. You must compile for the simulation standard on which you will run the application. Some classes have protocol-specific constructors.

- Utility functions: VR-Link includes a rich set of utility functions, including vector and matrix manipulation functions, a platform-independent interface to the system clock, and support for discreet simulation time. Coordinate conversion utilities for geocentric coordinates, geodetic coordinates, topographic coordinates, and UTM coordinates are included as well.
- Example applications: VR-Link comes with a set of HLA and DIS utility programs. For example, the netdump utility prints data received from remote simulations in an easy-to-read format. The f18 utility is a simple networked simulator that serves as a flexible debugging tool. Source code for the examples is provided to demonstrate the use of much of VR-Link's functionality.
- You can use VR-Link's C++ interface from C applications.

1.2.1. Files and Executables in VR-Link

VR-Link includes the header files and libraries necessary to build applications based on VR-Link, plus executables and source code for the following utility applications:

- *f18*
- *netdump*
- *nethex*
- *hlaNetdump*
- *talk*
- *listen*
- *vdc.*

VR-Link includes the *vrlinkd3i* packet server and *buffgrep* executables without source code. The unlimited version of the packet server, *vrlinkdu*, is available as an option. You can find more information about these utility applications in Chapter 10, *Example and Utility Applications*.

We provide source code, but no executables for the following sample applications:

- *simpleC*
- *launcher*
- *testPdu*
- *testInter*
- *testEnc*
- *testDec.*

The "test" applications show how to extend the supplied features of VR-Link by creating your own PDUs, interactions, encoders, and decoders. They are in *./examples/extend*.

1.3. Simulation Standards Supported

VR-Link supports the High Level Architecture and the Distributed Interactive Simulation (DIS) protocol. Please refer to the *RELNOTES* file and release documentation for a list of the versions of these simulation standards supported by your release of VR-Link.

You can also check the MÄK web site for platform support updates at: http://www.mak.com/military/compatibility.html

For a brief discussion of DIS and HLA, see "HLA, DIS, and Protocol Independence," on page 3-4.

1.3.1. HLA FOM Support

VR-Link provides extensive built-in support for the Real-Time Platform Reference FOM (RPR FOM). (See release documentation for version information.) The RPR FOM is a reference FOM developed by the SISO-sanctioned RPR FOM Standards Development Group, which consists of representatives from many companies that used the DIS protocol in the pre-HLA era.

The goal of the RPR FOM is to facilitate a priori interoperability (to about the extent we had in DIS) among HLA simulations that choose to use it. In other words, if you use the RPR FOM (either as is, or with your own extensions), then you know that you will be able to interoperate with anyone else who chooses to use this FOM. Another advantage of using the RPR FOM is that many tools, including many of those offered by MÄK Technologies, are pre-configured for that FOM, meaning a faster road to getting up and running in HLA.

FOM Agility

In addition to its built-in support for the RPR FOM, VR-Link can be configured to work with other FOMs, by using its FOM Mapper to define mappings between VR-Link's protocol-independent interface and the objects, interactions, parameters, and attributes defined in your FOM. For more information about VR-Link's FOM Agility, see "FOM Agility," on page 6-20.

Installing and Configuring VR-Link

This chapter covers the following topics:

2.1. System Requirements

Please read the release documentation for information about system requirements and platform support, or view our web site at: http://www.mak.com/military/compatibility.html

2.2. Installing VR-Link

This section explains how to install VR-Link on different operating systems and describes the file structure. Please check release documentation to see if there have been any changes in installation procedures.

2.3. Installing VR-Link on UNIX

To install VR-Link:

1. Mount the CD-ROM.
2. Run:

   ```
   ./install
   ```

3. Follow the on-screen instructions.

You can install VR-Link into any directory for which you have write permissions.

2.4. Installing VR-Link On A PC

To install VR-Link on a PC:

1. Insert the VR-Link CD into your CD-ROM drive. If the autorun feature is enabled, the MÄK Multiple Installation Program screen opens (Figure 2-1).

   If the MÄK Multiple Installation Program does not open, open the Windows Explorer. In the root directory for your CD drive, double-click *setup.exe*. The MÄK Multiple Installation Program screen opens.

2. In the MÄK Multiple Installation Program screen, click VR-Link.

3. Follow the instructions of the installation program.

Figure 2-1. MÄK Multiple Installation Program

2.5. VR-Link Files Installed

The installation process creates a directory called *vrlinkx.x* and a set of subdirectories beneath it. Table 2-1 describes the contents of the subdirectories.

Table 2-1: Contents of VR-Link subdirectories

Directory	Directory contents
./bin3	Executables for use with DIS version 2.0.3, including utilities and example applications.
./bin5	Executables for use with DIS versions 2.0.4, 2.1.4, and IEEE 1278.x, including utilities and example applications.
./binRPR	Executables for use with the HLA, including utilities and example applications.
./doc	VR-Link documentation in Adobe portable document format (PDF).
./examples	Source directories for the example applications.
./examples/extend	A directory with examples that show how to extend VR-Link by creating new classes and interactions.
./flexlm	Files for the FLEXlm license manager.
./include	C++ header files.
./lib	VR-Link C++ libraries. On PCs, contains multithreaded (MT) and multithread DLL (MD and RT) libraries.

On PCs, the default installation directory is *C:\Program Files\MAK\VRLinkx.x*.

2.6. Setting Up and Using the FLEXlm License Manager

VR-Link, like other MÄK Technologies products, uses the FLEXlm license manager. Before you can use VR-Link, you must obtain a valid license file and configure the license server and client machines. (For information about FLEXlm, view the FAQ page at www.globetrotter.com/flmfaq.htm.) Contact the MÄK Technologies sales department for information about special licensing agreements.

2.6.1. The FLEXlm Client-Server Architecture

This section explains the FLEXlm architecture to provide a context for the setup instructions that follow. FLEXlm uses a client-server architecture. The FLEXlm executable, *maklmgrd* runs on one computer, known as the license server. This machine services every machine (including possibly itself) on which you want to run a MÄK Technologies product. The machines on which the MÄK Technologies products are running are called clients.

The license server has a license file (supplied by MÄK Technologies). This file identifies the MÄK Technologies products for which you have licenses. On each client machine, the MAKLMGRD_LICENSE_FILE environment variable (which you must set) points to the server.

Licenses “float”, so you can install product software on more machines than you have licenses for, but at any given time, you can run only as many instances of a product as you have licensed.

! The license key in the license file is tied to the host id of the server machine. Therefore, if you decide to change the server, you need to get a new license file. Then, you must update the MAKLMGRD_LICENSE_FILE environment variable on every client. This is not a trivial process, and is not recommended.

Figure 2-2 illustrates the client-server architecture for FLEXlm and MÄK products.

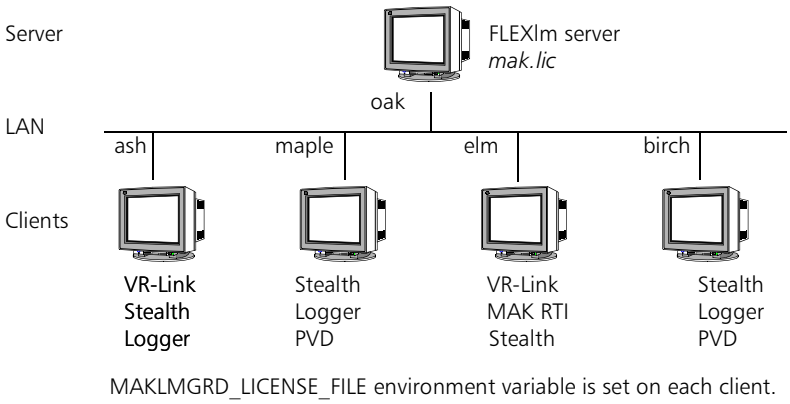


Figure 2-2. FLEXlm example architecture

2.6.2. Installing and Configuring the License Manager

The general procedure for installing and configuring license management is as follows (each step is explained in the rest of this section):

1. Install license manager software.
2. Request a license file.
3. Put the license file in the *flexlm* directory on the license server.
4. Set the MAKLMGRD_LICENSE_FILE environment variable on all client machines.

If you have MÄK Technologies products installed and you purchase additional licenses or products, you must get a license file for the new products or licenses. For this procedure, see "Adding Additional License Files," on page 2-9.

Installing the License Manager Software

As part of the installation of all MÄK products, the FLEXlm software is installed in the *flexlm* directory under the root product directory.

1. Install your MÄK products on the machine(s) on which you want to run them.
2. If this is a networked system, and you did not install MÄK software on the license server, copy the *flexlm* directory from a client machine to the license server machine.

Requesting A License File

License files are keyed to the host id of the license server. You need to supply MÄK with some information so that we can generate a license file for you.

To get a license file:

1. On the license server, in a command window, change to the *flexlm* directory.
2. Execute *lmhostid.* This program generates a unique number that MÄK uses to generate your license activation codes. Write down the number.
3. Go to http://www.mak.com/support/keyintro.htm and complete the license key request form. MÄK will e-mail you a file named *mak.lic*.

To get a license file, you need to know the name of the license server machine. To get the name of your machine follow the procedure for your operating system:

UNIX

At a command prompt, type `hostname`.

Windows 95/98/NT

1. On the Start menu, choose Settings → Control Panel.
2. Open the Network applet.
3. Select the Identification tab. The machine name is listed in the Computer Name field.

Putting the License File in the *flexlm* Directory

After you receive the license file, put it in the *flexlm* directory on the license server.

If you already have a *mak.lic* file in the *flexlm* directory, do not overwrite the file. See the instructions in "Adding Additional License Files," on page 2-9.

Setting the MAKLMGRD_LICENSE_FILE Environment Variable

The MAKLMGRD_LICENSE_FILE environment variable identifies the server machine so that the FLEXlm software on a client can check out a license when you run a MÄK Technologies product. You must set MAKLMGRD_LICENSE_FILE on every client machine.

If you are running MÄK Technologies products on the license server machine, it is also a client, so you must set MAKLMGRD_LICENSE_FILE.

The syntax for the environment variable is: *@Server_name*. For example, if the server machine is oak, set the environment variable to `@oak`.

The following sections explain how to set environment variables on the different platforms that MÄK Technologies products run on.

Windows 95/98

On Windows 95/98, you set environment variables by adding or editing lines in the *Autoexec.bat* file. To set the MAKLMGRD_LICENSE_FILE variable, add a line similar to the following:

```
set MAKLMGRD_LICENSE_FILE=@oak
```

Do not put spaces around the equal (=) sign.

Windows NT

On Windows NT, you set environment variables in the Environment tab of the Properties dialog box for the machine:

1. Right-click the My Computer icon on the Desktop.
2. On the pop-up menu, select Properties. The System Properties dialog box opens (Figure 2-3).
3. Select the Environment tab.
4. In the Variable text field, type MAKLMGRD_LICENSE_FILE.
5. In the Value field, type the value, for example:

   `@oak`

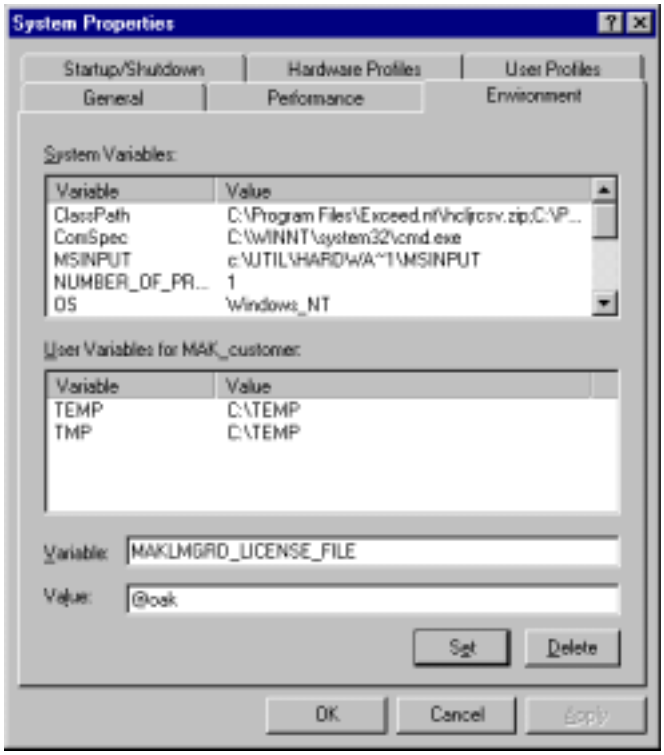


Figure 2-3. Setting an environment variable in Windows NT

UNIX

On UNIX, you set environment variables in your *.cshrc* (or equivalent startup file). Set the variable similarly to the following example:

```
setenv MAKLMGRD_LICENSE_FILE @oak
```

You are ready to run the license server and use your new licenses or MÄK products. See "Running the License Server," on page 2-10.

2.6.3. Adding Additional License Files

If you buy additional MÄK products, or additional licenses for products you already have, you must get an additional license file.

> To request an additional license file, follow the same procedure as for your first license file. See "Requesting A License File," on page 2-7.

To add a new license file to your license server:

1. When you get the new license file, rename it from *mak.lic* to something like *mak2.lic*, or some other name ending in *.lic*, so that it has a different name from the license file(s) currently in the *flexlm* directory on the license server.
2. Put the renamed file in the *flexlm* directory on the license server, with the other license files.

You can now use the new products or additional product licenses.

2.6.4. Running the License Server

The FLEXlm license server must be running on the server machine before you can run your MÄK applications.

> To start the license server, on the server machine only, execute *runLm* from the *./flexlm* directory.

> To obtain the current status of the server, run *lmstat*.

! Do not execute more than one instance of *runLm* at a time. If you aren't sure whether or not the license server is running, execute *lmstat*.

Shutting Down the License Server

> To force all applications to check in their licenses and shut down the license server, run *lmdown*.

2.6.5. Troubleshooting the FLEXlm License Manager

For general troubleshooting information, refer to the FLEXlm FAQ at www.globetrotter.com/flmfaq.htm. It describes common problems, including the majority that have been reported to MÄK support engineers. Please consult the FAQ first when you have a problem.

Unable to Get A License

If you are unable to get a license for a MÄK application and the FLEXlm log file (*LOG*) indicates that the port is in use, it is possible that you have more than one *lmgr*d or *maklmgrd* processes running. It is also possible that an application that was using a license is thought to have terminated, but is still alive. To verify and resolve these possibilities:

1. See if more than one *lmgrd* or *maklmgrd* process is running.

   On Windows 95/98/NT, check the Task Manager. To open the Task Manager, press Ctrl-Alt-Delete.

   On UNIX, enter the following commands:

   ```
   ps -aux | grep lmgrd
   ps -aux | grep maklmgrd
   ```

   If more than one instance of either process is running, write down the process ids (PID).

Some UNIX systems use `ps -aux`, others `ps -ef` to check processes. Use the command appropriate for your operating system.

2. If there are old processes present that may be using a license, kill them.

   On UNIX, kill a process with the kill command, for example:

   ```
   kill -9 1385 878
   ```

   where 1385 and 878 are process IDs.

   On Windows NT, select a process and click End Process.

   It is important that you kill all the processes to ensure a clean restart for the license server.

3. If in step 2 you killed the license server, start it with the `runLm` command.

Preventing Multiple License Manager Processes

We recommend that you keep the license server running and that you not start and stop it as you run applications. If you prefer to stop the license server when you are not using it, always use `lmdown` to stop it.

Whenever you start the license manager, first run *lmstat* to be sure that there is not already a license server process running. Following this practice will ensure that you do not start multiple license servers.

License Manager Cannot Find MÄK License Management Executable

If your license manager gives you an error message indicating that it cannot find the license management executable, try putting the pathname in the license file as follows:

1. Open the license file (*mak.lic*) in a text editor.
2. Edit the license file to specify the location of the *maklmgrd* executable on the server machine. The file supplied by MÄK represents this location with a dot as shown below:

   ```
   DAEMON maklmgrd .
   ```

   Replace the dot with your absolute path to *maklmgrd*, for example:

   ```
   DAEMON maklmgrd "C:\Program Files\MAK\flexlm\maklmgrd"
   ```

! On PCs, you must enclose the path in quotes.
The drive letter must be uppercase.

2.7. Obtaining and Installing a Runtime Infrastructure (RTI)

An RTI (Run-Time Infrastructure) is a software library (and perhaps supporting executables) that implements the HLA Interface Specification. In HLA, applications exchange FOM data through RTI calls, which means that all HLA applications need to use an RTI.

Because of differences in the low-level network mechanisms used by different RTI implementations (which include, but are not limited to packet layout), applications that want to interoperate in the same federation execution must use the same RTI implementation.

Because RTIs are usually provided as dynamic libraries that implement a fixed API, a federation can often switch from one RTI implementation to another between runs (without even recompiling the applications), but during each run, all participants must agree on which RTI to use, much as they must also agree on which FOM to use.

The two RTI implementations against which all MÄK products have been tested are the MÄK RTI (which comes with all MÄK products), and the DMSO RTI (which is available from the DMSO web site). For the most recent information about the RTI versions supported by VR-Link, see the Product Versions page on the MÄK Technologies web site at: http://www.mak.com/military/compatibility.html.

2.7.1. Installing the MÄK RTI

The MÄK RTI is included with all MÄK products and you have the option of installing it as part of product installation.

UNIX

The installation process creates a link called *RTI* in each product's root directory, which points to the *makRti* directory, in which the MÄK RTI software is located.

PCs

The project files point to the default installation directory for the RTI.

Configuring Your System to Use the MÄK RTI

The RTI dynamic library needs to be located somewhere on your dynamic library search path. This path is indicated by an environment variable as follows:

- For IRIX6n32 use LD_LIBRARYN32_PATH
- For DEC Alpha, use PATH
- On other UNIX platforms use LD_LIBRARY_PATH
- On PCs, use PATH

The MÄK RTI needs to know where to find the following configuration files:

- The FED file (*.fed*) for your federation execution (required)
- The RID file (*rid.mtl*) (optional)

Put the configuration files in the directory from which you are running, or set the environment variable RTI_CONFIG to the directory that contains them.

Running Applications with the MÄK RTI

You do not need an RTI server, such as the *rtiexec*, or central application to use the MÄK RTI, however you can run the *rtiexec* if you want to. Be sure the Flexlm license server is running, then start the applications, and they should be able to communicate. (For information about the license server, see "Setting Up and Using the FLEXlm License Manager".) See the *MÄK RTI Reference Manual* for information about the *rtiexec* and changing the networking parameters such as multicast addresses and UDP ports.

Documentation for the MÄK RTI

The documentation for the MÄK RTI consists of:

- *MÄK RTI Reference Manual*
- The documentation set for the DMSO RTI. Aside from the installation and configuration instructions, the documentation for the DMSO RTI also applies to the MÄK RTI. It is included on the MÄK product CD.

2.7.2. Installing the DMSO RTI

You can get the DMSO RTI from DMSO's web site (http://hla.dmso.mil). Click on Software Distribution Center, where you can register for an account, then later download the RTI software. Documentation for the DMSO RTI is provided on the MÄK product CD because the documentation applies to the MÄK RTI as well.

Consult the DMSO RTI Release Notes to determine version information and whether or not any patches are required.

1. Install the RTI according to the instructions provided.
2. Copy the *VR-Link.fed* file from the MÄK products root directory to the DMSO RTI's config directory.

Configuring Your System to Use the DMSO RTI

To configure the environment:

> On UNIX systems, source the *RTI.env* file.

> On Windows systems, set environment variables and add the RTI location to your path. The RTI's README file contains detailed instructions.

Running Applications Using the DMSO RTI

To use the DMSO RTI, you need to run the RTI executive (*rtiexec*). This executable resides in the RTI's *bin/platform_name* directory (UNIX) or in the RTI's *bin* directory (PCs). Run *rtiexec* only on one machine, regardless of how many HLA applications you are running.

Once the *rtiexec* is running, you can run HLA applications.

2.8. Configuring the Network for DIS Applications

In a typical DIS exercise, VR-Link applications communicate with other DIS applications using broadcast UDP/IP over a local area network. In such a configuration:

- All applications in the exercise use the same IP broadcast address, and each application's host machine has a network interface device configured for that address.
- All machines must share a common netmask.
- All participating applications must use the same UDP port.

Typically, machines on a LAN will already have the correct broadcast addresses, so VR-Link applications will be able to communicate with each other immediately. However, if applications are unable to achieve two-way communication, the network devices may be improperly configured.

2.8.1. Broadcast Address and Netmask (UNIX)

Network devices should be configured so that all machines agree on the netmask and on the part of the network address (inet) covered by the netmask. On UNIX-based systems, you can use the `ifconfig` command to examine and set the configuration of network devices. For example, to set the netmask use:

`ifconfig` *`device_name`* `netmask 0xFFFF0000`

Because some machines fill unspecified bits in a broadcast address with 1's (as VR-Link does), while others fill it with 0's, VR-Link computes the broadcast address itself using the device's netmask and host address. Thus, the broadcast address used by VR-Link is the same as that reported by `ifconfig` on an SGI, but not the same under SunOS 4.1.x.

Some machines have more than one network device capable of broadcasting. Unless your application is designed to specify a specific device or address (which is possible using VR-Link routines), a VR-Link application will use the broadcast address of the first device listed in the interface table by default.

See your system administrator if you need help with network issues.

2.8.2. Network Mask (Windows)

Under Windows NT, VR-Link assumes you are running on a class C network using a netmask of 255.255.255.0. (In a class C network, the first three components of the IP address are the same for all machines. In a class B network, the first two components of the IP address are the same; the second two identify the machine.)

If your machine is configured to use a netmask other than 255.255.255.0, create a *DtNetmask* environment variable to specify your netmask.

Under Windows 95/98, VR-Link automatically figures out your machine's netmask. However, if you specify a *DtNetmask* environment variable, VR-Link uses the value you specify.

The DtNetmask variable should specify your netmask in a C-language style hexadecimal format. For example:

if your netmask is:	`255.255.0.0`
the hexadecimal representation is:	`0xFFFF0000`

The default value of DtNetmask is 0xFFFFFF00, which corresponds to the netmask 255.255.255.0.

Setting the Network Mask for Windows NT

To set the DtNetmask variable for Windows NT:

1. Start the Windows NT Control Panel. Open the *System* applet.
2. Enter the DtNetmask variable name and appropriate hexadecimal value in the fields at the bottom of the System dialog box.
3. Click Set.

Setting the Network Mask for Windows 95/98

To set the DtNetmask variable for Windows 95/98:

1. Add the following line to your *AUTOEXEC.BAT* file (replace the hexadecimal number with your netmask):

   ```
   SET DtNetmask=0xFFFF0000
   ```

2. Reboot the PC.

2.8.3. UDP Port

In addition to the broadcast address, the applications in an exercise must also agree on the UDP port number.

The pre-built applications that ship with VR-Link use port 3000 by default (with the exception of *vrlinkd*, which requires that you specify a port number as a command-line argument). Most applications let you set the port number with the `-P` *`port`* option.

VR-Link Concepts

This chapter provides a general explanation of how VR-Link works. It covers the following topics:

3.1. VR-Link is A Multi-Layered Toolkit

VR-Link is a toolkit. It supplies you with the classes, functions, and utilities you need to write a simulation application, but unlike a framework, it doesn't provide a structure that you are required to use. For example, you will probably want to check the state of a locally-simulated entity at periodic intervals, but VR-Link doesn't care whether you do that using a loop (as Example 1 does), or by using a timer or some other mechanism.

3.1.1. Multiple Layers of Access

As a VR-Link developer, you have several options for approaching application development:

- Work through the protocol-independent interface, which with minimal modifications, allows you to create applications that work with both HLA and DIS and shields you from many of the intricacies of the underlying protocol-specific classes and functions. Most VR-Link users take this approach.
- If you want to tailor your application to HLA or DIS, you can go below the protocol-independent interface and take advantage of the protocol-specific classes.
- To fully realize the potential of VR-Link and meet your most demanding simulation needs, you can:
  - Extend VR-Link by creating new classes for HLA objects and interactions or DIS PDUs.
  - Use the FOM mapper to support additional FOMs.

All users will want to read this chapter (*VR-Link Concepts*), Chapter 5, *The Protocol-Independent Interface*, and Chapter 10, *Example and Utility Applications*. Users focused on a specific simulation standard should add the protocol-specific chapter of their choice to this list. Users who want to extend VR-Link will need to understand the basic functioning of VR-Link as well as sections devoted to subclassing and creating your own objects and interactions.

3.2. HLA, DIS, and Protocol Independence

It is beyond the scope of this manual to fully describe DIS and HLA. However, in this section we briefly describe them as a foundation for the concepts and terms discussed in the rest of this chapter.

3.2.1. High Level Architecture (HLA)

HLA is required by the Department of Defense for use in all DoD simulations. In HLA, the types and formats of data to be exchanged among simulators are not standardized, as they are in DIS. Groups of simulators that want to play together must use the same Federation Object Model (FOM) and Run-Time Infrastructure (RTI). FOMs and RTIs are described in the next two sections.

The Federation Object Model (FOM)

Every federation execution requires a Federation Object Model (FOM), which defines the data model for the federation execution. Members of a federation execution can develop a FOM themselves, use an existing reference FOM, or build a FOM by starting with a reference FOM and adding to it or modifying it.

A FOM is implemented through a Federation Execution Data (FED) file, which contains a subset of the FOM that is required by the RTI. VR-Link 3.x has extensive built-in support for the Realtime-Platform Reference FOM (RPR FOM). More specifically, VR-Link supports the FOM defined within the FED file called *VR-Link.fed*, which is stored in the VR-Link home directory. *VR-Link.fed* represents the RPR FOM, plus a few extra classes that we have added.

The RPR FOM is a reference FOM developed by the SISO-sanctioned RPR FOM Standards Development Group, which consists of representatives from many companies that used the DIS protocol in the pre-HLA era.

The goal of the RPR FOM is to facilitate a priori interoperability (to about the extent we had in DIS) among HLA simulations that choose to use it. In other words, if you use the RPR FOM (either as is, or with your own extensions), then you know that you will be able to interoperate with anyone else who chooses to use this FOM. An advantage of using the RPR FOM is that many tools, including many MÄK Technologies products, are pre-configured for that FOM, providing you a faster way to get up and running in HLA.

However, as HLA matures and gains wider use, we can expect that more FOMs will be developed and simulation developers need to consider whether or not their applications will have the flexibility to work with different FOMs. VR-Link has features that allow you to work with new or modified FOMs and you can extend FOMs to meet your needs. The ability to work with a variety of FOMs is called FOM-agility.

As a FOM-agile toolkit, VR-Link can be configured to work with other FOMs, by using its FOM Mapper to define mappings between VR-Link's protocol-independent interface and the objects, interactions, parameters, and attributes defined in your FOM.

For information about how VR-Link implements FOM-agility, see "FOM Agility," on page 6-20. Please see the Release Notes for a list of the RPR FOM versions supported by your version of VR-Link. You can also view product version information on the MÄK web site at http://www.mak.com/military/compatibility.html.

The Run-Time Infrastructure (RTI)

An RTI is an implementation of the HLA interface specification, that is, the services that make up the standard RTI application programmer's interface (API). An application can call the functions in the RTI's API either directly or through VR-Link. The HLA rules specify that federates must use the RTI to exchange all simulation data.

3.2.2. The DIS Protocol

The DIS protocol is a set of standards that govern how participating applications share information about a virtual world. The protocol specifies a set of packets, called Protocol Data Units (PDUs), that communicate this information. Each PDU identifies the sender and contains other information depending on the PDU type. The DIS protocol also specifies when and how frequently PDUs are sent.

DIS 2.0.4 and IEEE 1278.1-1995 are equivalent, as are DIS 2.1.4 and IEEE 1278.1a-1998.

3.2.3. Protocol Independence

In this manual, the Protocol-Independent API refers to a set of classes that encompass most of the features of VR-Link and allow you to create applications that will work in both DIS and HLA without significant modification.[1]

When you create a protocol-independent application, you do not create one executable that works with both protocols; you write one application and compile it for each protocol by specifying the protocol at compile time. In cases where a class uses different constructors or functions depending on the protocol, such as creating an exercise connection, you need to use #ifdef(s) so that your application compiles using the correct constructors.

1. HLA is not actually a protocol, it is an architecture (as its name indicates). However, within HLA, an RTI and FOM combination could be considered a protocol, in that they specify the format and content of data exchanged among simulation participants. And since your application is actually interacting with other simulations via the RTI, it is fair to speak in terms of protocol-independence.

3.2.4. Obtaining Further Information about HLA and DIS

For information about HLA rules and specifications, visit the HLA Web site at:

http://hla.dmso.mil/

You can find general DIS information in the document library at the Simulation Interoperability Standards Organization (SISO) home page:

http://www.sisostds.org/doclib/

The IEEE documentation of IEEE 1278.1-1995 and 1278.1a-1998 has superseded documentation of DIS 2.0.4 and DIS 2.1.4. For documentation of the IEEE 1278.1-1995 or 1278.1a-1998 versions of DIS, you can obtain the publications, *IEEE Standard for Distributed Interactive Simulation, Application Protocols* and *Enumeration and Bit Encoded Values for Use with Protocols for DIS Applications* at the IEEE World Wide Web site:

http://standards.ieee.org/catalog/index.html

Enumeration and Bit Encoded Values for Use with Protocols for DIS Applications (DIS enumerations document) is also available at the SISO site. Portions of the DIS enumerations documents are available for view at: http://siso.sc.ist.ucf.edu/dis/dis-dd/

3.3. A Conceptual Overview of VR-Link

A simulator or other HLA/DIS application performs several tasks in order to interact with other players in a virtual world. These tasks typically include:

- Connecting to an exercise[1]. The application must create an object through which it can send and receive simulation data. That object should provide an efficient way to direct incoming data to other parts of the application.
- Managing state information. The application must inform other exercise participants about the states of its locally generated objects, such as entities, aggregates, transmitters, and so on. The application typically must receive, process, and manage state information about remote objects.
- Managing information about events. The application must inform other participants about interactions such as collisions or fire/detonations involving a locally simulated entity, and receive and process events and interactions sent by other applications.

VR-Link provides classes that facilitate these tasks.

3.3.1. VR-Link's Protocol-Independent Classes

VR-Link provides the following categories of protocol-independent classes:

- Exercise Connection – A class that serves as the application's interface to the RTI or to the DIS network.
- Object management classes – Classes that maintain state information of local and remote objects, and handle the sending and receiving of state updates.
- Interaction classes – Classes that provide a protocol-independent API to either HLA interactions or DIS PDUs that represent events.

VR-Link also provides protocol-specific classes and features. However, in most cases, your code does not need to interact directly with them.

To make the job of writing an application easier, VR-Link includes conversion routines, mathematical functions, and other useful functions. These are described in Chapter 10, *Example and Utility Applications* and Chapter 9, *Utilities*.

1. The HLA counterpart to a DIS exercise is a federation execution. For simplicity, we use the term exercise to represent both.

Figure 3-1 shows how classes in a VR-Link application interact. Your application code can interact with classes at any level, and with the RTI or DIS network directly. However, for most applications, you can rely on the protocol-independent classes, which are generally easier to use. VR-Link utilities, such as the coordinate conversion utilities are also available to application code and VR-Link classes.

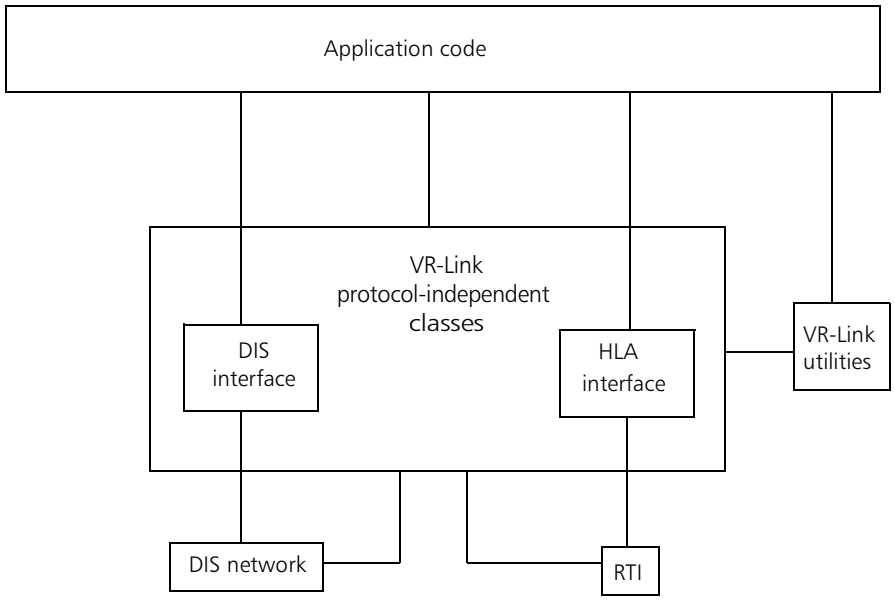


Figure 3-1. Structure of a typical VR-Link application

Protocol-Independent Object Management Classes

VR-Link provides a protocol-independent interface to various logical types of objects that are typically simulated in real-time platform-level simulations. For each logical type of object (entities, aggregates, transmitters, and so on), the following classes provide this interface:

- An object publisher class, for example, *DtEntityPublisher*, which manages the sending of updates for a locally simulated object to the exercise through its *DtExerciseConn*. Object publishers are derived from *DtObjectPublisher*.
- A reflected object class, for example, *DtReflectedEntity,* which represents a remote object. A *DtReflectedEntity* maintains the current state of the object based on updates received from the exercise through its *DtExerciseConn*. Reflected objects are derived from *DtReflectedObject*.

- A reflected object list class, for example, *DtReflectedEntityList*, which keeps track of remote objects. It creates and destroys reflected objects based on information received from the exercise through its *DtExerciseConn*. Reflected object lists are derived from *DtReflectedObjectList*.
- A state repository class, for example, *DtEntityStateRepository*, which is used by both publishers and reflected objects to store the state of the object they represent. State repositories are derived from *DtStateRepository*.

DtEntityStateRepository and *DtAggregateStateRepository* are derived from a common class *DtBaseEntityStateRepository*. *DtBaseEntityStateRepository* is derived from the general base class *DtStateRepository*.

Below the protocol-independent interface are many lower-level objects that help manage objects in HLA. The lower-level objects are for the most part hidden by the higher-level interface and don't typically need to be used by application code directly. These lower-level objects are described in Chapter 6, *The HLA-Specific Interface*.

3.4. Connecting to An Exercise

VR-Link applications connect to an exercise through an exercise connection. Exercise connections are implemented through the class *DtExerciseConn*. Higher level VR-Link classes use the *DtExerciseConn* to set and receive state information and other data. The member functions of *DtExerciseConn* allow an application to do the following:

- Send interactions to the exercise
- Read input from the network
- Generate event IDs
- Register callback functions that are executed each time VR-Link receives interactions and other input from the network.

For more information about exercise connections, see "Connecting to Exercises," on page 5-4.

3.5. Managing State Information

A VR-Link application usually must maintain information about objects being simulated locally and communicate this information to other participants in an exercise. The application must also obtain information about remote objects and represent them locally.

Figure 3-2 illustrates how a VR-Link application manages state information. The elements of the figure are described in the remainder of this section. In a DIS exercise, locally simulated entities are included in the reflected entity list.

In the remainder of this chapter, we talk about concepts primarily in the context of entities. However, most aspects of entity management apply to other objects, such as aggregates and transmitters, as well.

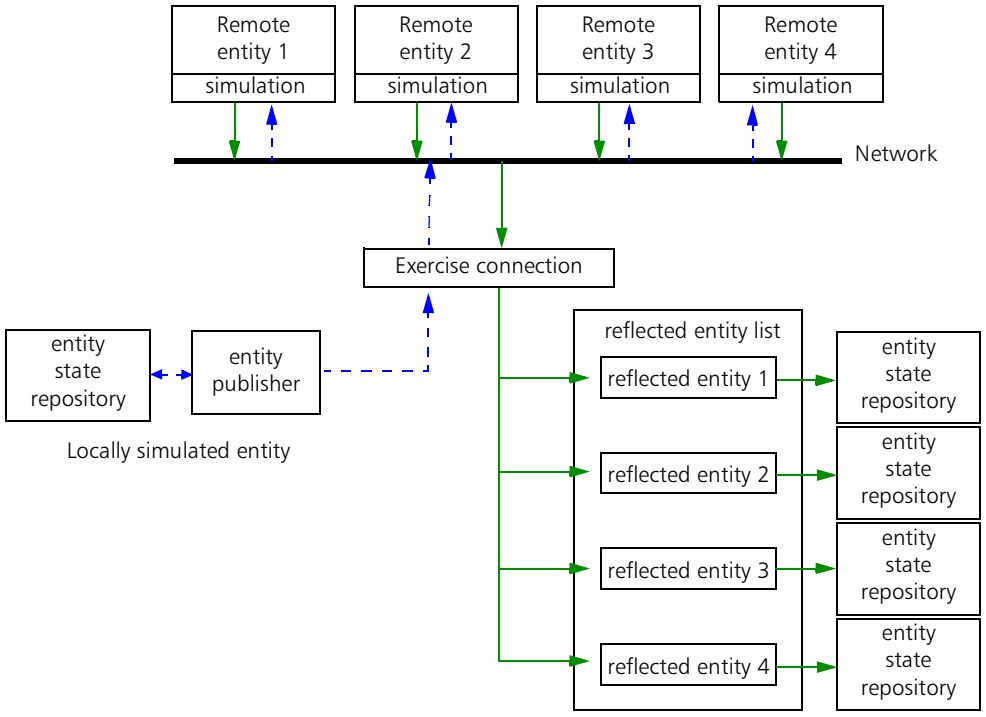


Figure 3-2. Managing state information

3.5.1. Managing Locally Simulated Entities

Each locally simulated entity is represented by a *DtEntityPublisher*, which maintains state information in a *DtEntityStateRepository*. The *DtEntityPublisher* sends state update messages to the other exercise participants through the exercise connection. Figure 3-2 illustrates a *DtEntityPublisher* sending information through an exercise connection. The *DtEntityStateRepository* class has functions that let you inspect the components of an entity's state and change its state.

The frequency with which your application sends updates to the exercise depends on your protocol.

DIS only

In a DIS exercise, state updates are sent when the state of an entity changes and at regular intervals (heartbeats), regardless of whether there are any changes to the entity.

HLA only

The HLA seeks to minimize unnecessary network traffic. Therefore in an HLA federation execution, state updates are sent only under the following conditions:

- When update conditions (as specified by the FOM) have been met and a federate has subscribed to the entity attribute that changed. (Subscription is the process by which a federate tells the RTI that it wants to be notified about changes to an object.) The update condition for most attributes is any change since the attribute was last sent.
- When a specific request for an update is received from another federate or the RTI.

For more information about managing local entities, see "Working with Entities," on page 5-14.

3.5.2. Managing Remote Entities

A VR-Link application receives information about remote entities through the exercise connection. The function DtExerciseConn::drainInput() causes VR-Link to read and process input. VR-Link maintains a list of remote entities in a *DtReflectedEntityList*. Each entity known to the *DtReflectedEntityList* has a *DtReflectedEntity* to represent it. The *DtReflectedEntity* maintains the state of the entity in a *DtEntityStateRepository*. As with locally simulated entities, you can inspect the components of an entity's state. VR-Link also provides functions to track when entities join or leave an exercise.

Figure 3-2 illustrates a *DtReflectedEntityList* and *DtReflectedEntitys*.

For more information about remote entities, see "Working with Remote Entities," on page 5-21.

You may have noticed that the class names we use for managing entities reflect the terminology used by HLA, that is, publishing and reflecting. Although we use this terminology for our protocol-independent classes, for DIS, VR-Link sends the proper PDUs for state updates or interactions.

3.6. Managing Information About Events

We use the term interaction to refer to events such as the firing of a munition, detonation of a munition, or collision of entities.

VR-Link manages interactions through classes derived from *DtInteraction*. Usually, the name of an interaction class includes the interaction type, for example, the class for a fire interaction is *DtFireInteraction*.

3.6.1. Sending Local Interactions

For locally-defined interactions, such as firing a munition from a locally simulated entity, you create an instance of the appropriate interaction class, set its values, and send it through the exercise connection. For more information, see "Sending Interactions," on page 5-11.

3.6.2. Receiving Remote Interactions

A VR-Link application receives notification of a remote interaction through its exercise connection. The function DtExerciseConn::drainInput() causes VR-Link to read and process input. The local application reacts through the use of callback functions.

There are no built-in functions for reacting to remote interactions. You must write a callback function for each type of interaction that you want to respond to and register the callback function with the interaction class. When your application is notified that an interaction event has occurred, it passes the interaction type and affected object to your callback function, which executes whatever code you've written. For information about callbacks, see"Using Callbacks," on page 3-14.

Interactions are transient, that is, they are deleted by VR-Link immediately after the last callback function registered for the interaction is called.

3.7. Other Simulation Concepts

This section describes how VR-Link implements the following simulation and programming concepts:

- Time
- Object identification
- Callbacks
- Coordinate systems
- Dead-reckoning and smoothing
- Time stamps.

3.7.1. Managing Time

The VR-Link toolkit maintains the concept of VR-Link simulation time for use in dead-reckoning of remote entities and thresholding of local entities. Typically, VR-Link simulation time is set once during each iteration of the application's main simulation loop so that all entities are dead-reckoned based on the same value of current time. VR-Link simulation time is set by calling DtSetSimTime().

Usually, you will want to advance VR-Link simulation time each frame, in proportion to the amount of real time that has elapsed since the previous frame, so that it becomes a discrete approximation to (possibly scaled or offset) real time.

In a fixed frame rate application, like VR-Link's *f18* example, VR-Link simulation time can be managed as follows:

```
DtTimeInit();
DtTime dt = .05;// Each time step is .05 seconds
DtTime simTime = 0.0;// Represents current time
...
// Main simulation loop
while(...)
{
    DtSetSimTime(simTime);
    ...
    // Do stuff
    ...
    // Advance simTime by dt
    simTime = simTime + dt;
    // Sleep till the next multiple of .05 seconds
    DtTime timeTillNextFrame = simTime - DtGetElapsedRealTime();
    DtSleep(timeTillNextFrame);
}
```

In a floating frame rate application – where you go on to the next frame as soon as you're done with the previous one – you can pass the current time to DtSetSimTime() each frame:

```
DtTimeInit();
...
// Main simulation loop
while(...)
```

```
{
    DtSetSimTime(DtAbsRealTime());
    ...
    // Do stuff
    ...
}
```

For more information, see "Time-Related Utilities," on page 9-16 and "Timestamps," on page 3-18.

3.7.2. Identifying Objects

One concept that HLA and DIS handle fairly differently object identification.

DIS only

In DIS, entities are identified by a triplet (site, application, entity) known as an Entity Identifier. Other types of objects are usually identified using the IDs of their host entities plus a single additional ID number.

HLA only

In HLA, objects have several different identifiers, including:

- An Object Handle, which is used within a federate to identify a particular object in an RTI service invocation
- An Object Name, which is used within interactions and attribute updates exchanged among federates.

See "Identifying Objects," on page 5-32 for more information about the various types of identifiers, and to see how VR-Link accounts for these differences.

3.7.3. Using Callbacks

Callback functions are functions that a toolkit user writes and registers with the toolkit by passing a pointer to the function as an argument to a callback registration function. The toolkit, in this case VR-Link, will then call your function in response to some specified event.

There are several places in VR-Link where callback functions are necessary to take advantage of certain functionality. For example, in order to process incoming interactions or DIS PDUs of a certain kind, a user registers an interaction callback or PDU callback with the appropriate VR-Link interaction class or PDU class. This is accomplished using the class's addCallback() static member function. Subsequently, whenever VR-Link reads that type of interaction or PDU from the RTI or DIS network, VR-Link calls your callback function, passing it a copy of the relevant interaction or PDU. In most cases, including this one, the callback gets called from within DtExerciseConn::drainInput() - the VR-Link function that causes input to be read from the exercise.

Other places where callback functions are used in VR-Link include the following:

- You can register an entity addition or entity removal callback with a *DtReflectedEntityList*. These functions will be called, also from within drainInput(), whenever we receive word from the exercise that a remote entity has joined or left the exercise.
- A post-drain callback is a general callback function that can be registered with a *DtExerciseConn*, and is called from within drainInput() regardless of what data has been read from the RTI or DIS network. These callbacks are executed after we have finished reading data and have executed other, more specific callbacks.

Because callback functions are passed as regular function pointers to VR-Link's callback registration functions, the callback functions themselves cannot be non-static member functions of a class. They can be global (C-style) functions, or static class members. In order to have a non-static member function of a class be executed as a result of some VR-Link event, you can call the member function from within your callback.

Some callback functions have specific required function signatures, but most types of VR-Link callback functions have a trailing `void*` argument that is usually called **usr**. Corresponding callback registration functions also have a trailing **usr** argument. This allows you to register an arbitrary pointer with VR-Link along with your callback function. When VR-Link executes your callback function, the pointer that you registered is passed to your function as **usr**.

Often, the **usr** argument passes a pointer to an object on which you would like to call a member function. Simply cast **usr** back to the object's type within the callback. For example:

```
class MyObj
{
public:
   // The function we would like to call.
   void someFunc();

   // A static member function to be registered as an interaction
   // callback with the DtFireInteraction class.  When this function
   // is called by VR-Link, usr will contain a pointer to the object
   // passed to addCallback.
   static void theCallback(DtFireInteraction* inter, void* usr)
   {
      MyObj* obj = (MyObj*) usr;
      obj->someFunc();
   }
};

main()
{
   DtExerciseConn exConn(...);
   ...
   MyObj obj;
   // Register our callback function, passing a pointer to obj as usr.
   DtFireInteraction::addCallback(&exConn, MyObj::theCallback, &obj);
   ...
}
```

In cases where you would like to pass more than one object to your callback, you must create a simple structure consisting of pointers to your objects, then pass a pointer to the structure as **usr**.

Any pointer you pass as **usr** must be the address of a variable whose lifetime is at least as long as the period for which the callback function is registered.

3.7.4. Coordinate Systems

By default, *DtEntityStateRepository* works with geocentric coordinates. VR-Link has coordinate conversion routines that let you convert incoming and outgoing entity information to other coordinates systems. VR-Link also has coordinate views that allow you to use various coordinate systems without explicitly calling coordinate conversion routines. The view classes support:

- UTM coordinates
- Cartesian coordinates
- Topographic coordinates.

For more information about views and coordinate conversion, see "Coordinate Views," on page 5-37 and "Coordinate Conversions," on page 9-7.

3.7.5. Dead-Reckoning and Smoothing

To represent the behavior of entities in between state updates, VR-Link uses dead-reckoning, a process of estimating the location of an entity based on its acceleration and velocity. When updated location information arrives, VR-Link must move the simulated entity to its actual position. To ensure that transitions from an entity's dead-reckoned position to its actual position aren't so abrupt as to be visually disconcerting, VR-Link implements smoothing. Figure 3-3 illustrates the difference between a smoothed transition and a non-smoothed transition.

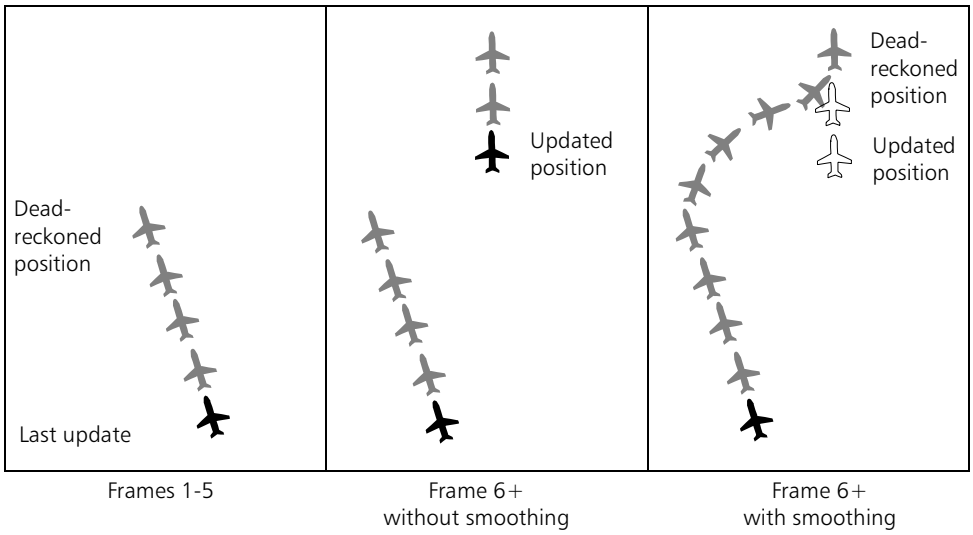


Figure 3-3. Dead-reckoning and smoothing

For more information, see "Dead-Reckoning," on page 5-25.

3.7.6. Timestamps

When interactions or object state updates are sent through a *DtExerciseConn* using sendStamped(), a timestamp is sent with (or within) the message.

A DIS or RPR FOM timestamp indicates only a time (in seconds) past a particular hour at which the data in the message is supposed to be valid. However, it contains no information about what particular hour the sender had in mind. (*DtPdu*, *DtInteraction*, and *DtStateMessage* all have a member function called guessTimeValid() that makes a guess at the full time the sender had in mind by returning the time closest to a reference time such that the number of seconds past the hour equals the value specified in the timestamp).

In addition to the timestamp, DIS and HLA RPR FOM messages contain a timestamp type indicator, which indicates a type of either "relative" or "absolute". Despite the confusion that these terms have generated in the DIS/HLA world, these terms only indicate whether or not the clock the sender used to obtain its timestamp is synchronized to some global exercise clock. A timestamp type of "absolute" means that the sender's clock is synchronized with the clocks of any other applications that claim to be using absolute timestamping. A timestamp type of "relative" means that the sender's clock is not necessarily synchronized with anyone else's clock.

Participants in a DIS or HLA exercise do not have to synchronize their clocks. In fact, in most exercises, clocks are not synchronized. If participants choose to synchronize their clocks, they do so using a method outside the scope of both DIS and HLA. (The HLA RTI has a set of time management services, but synchronizing clocks is not what they do.)

When a sender and a receiver are both using absolute timestamping, that is, their clocks are synchronized, the receiver can immediately make sense of the sender's timestamp. In this case, VR-Link uses the timestamp of incoming entity state updates in dead-reckoning calculations. If one or both are using relative timestamping, then the sender's timestamp is meaningless to the receiver unless he can determine the time difference between the two clocks (which is again, outside the context of DIS and HLA). In this case, like most DIS/HLA applications, we ignore the timestamp, and use the time of receipt in dead-reckoning calculations.

3.8. Basic VR-Link Examples

This section lists and describes two simple VR-Link applications. The first is a listen-only application that observes an exercise without simulating any entity on the network. The second is a write-only program that does not process information about remote entities.

The source code for both examples is included with VR-Link.

3.8.1. A Listen-Only Example

Example 1 illustrates a simple, listen-only VR-Link application. This application can be compiled for either DIS or HLA. The only protocol-specific code is contained in the #if statement that starts at line 22.

With each iteration of the loop, the program prints an entity's updated, dead-reckoned position in topographic coordinates. In addition, if a fire PDU or interaction is detected on the network, the program prints a message showing the entity ID of the attacker.

The online version of this manual has live links between the code example and explanatory text.

Example 1

```
1   #include "stdio.h"
2   #include "exerciseConn.h"
3   #include "reflEntList.h"
4   #include "ProcControl.h"
5   #include "LibMatrix.h"
6   #include "entitySR.h"
7   #include "reflectedEnt.h"
8   #include "fireInter.h"
9   #include "topoView.h"
10
11  int keybrdTick(void);
12
13  // Define a callback to process fire interactions
14  void fireCb(DtFireInteraction *fire, void */*usr*/)
15  {
16     printf("Fire Interaction from %s\n", fire->attackerId().string());
17  }
18
19  main()
20  {
21  // Create a connection to the exercise or federation execution
22  #if DtHLA
23     char *execName          = "VR-Link";
24     char *fedName           = "VR-Link listen";
25     DtExerciseConn exConn(execName, fedName);
```

```
26  #else
27     int port               = 3000;
28     int exerciseId         = 1;
29     int siteId             = 1;
30     int applicationNum     = 15;
31     DtExerciseConn exConn(port, exerciseId, siteId, applicationNum);
32  #endif
33
34     // Register a callback to handle fire interactions
35     DtFireInteraction::addCallback(&exConn, fireCb, NULL);
36
37     // Create an object to manage entities that we hear about
38     // on the network.
39     DtReflectedEntityList rel(&exConn);
40
41     // Initialize VR-Link time.
42     DtTimeInit();
43
44     int forever = 1;
45     while (forever)
46     {
47        // Check if user hit 'q' to quit
48        if (keybrdTick() == -1)
49        break;
50
51        // Tell VR-Link the current value of simulation time
52        DtSetSimTime(DtGetElapsedRealTime());
53        // Process any incoming messages
54        exConn.drainInput();
55
56        // Find the first entity in the reflected entity list
57        DtReflectedEntity *first = rel.first();
58
59        if (first)
60        {
61           // Grab its state repository, where we can inspect its data
62           DtEntityStateRepository *esr = first->entityStateRep();
63
64           // Create a topographic view on the state repository, so we
65           // can look at position information in topographic coordinates
66           double refLatitude  = DtDeg2Rad(  35.699760);
67           double refLongitude = DtDeg2Rad(-121.326577);
68           DtTopoView topoView(esr, refLatitude, refLongitude);
69
70           // Print the position
71           printf("Position of first entity: %s\n",
             topoView.location().string());
72        }
73
74        // Sleep till next iteration
75        DtSleep(0.1);
76     }
77     return 0;
78  }
79
80  int keybrdTick()
81  {
82     char *keyPtr = DtPollInputLine();
83     if (keyPtr && (*keyPtr == 'q' || *keyPtr == 'Q'))
84        return -1;
```

```
85     else
86        return 0;
87  }
```

Connecting to An Exercise

In lines 21-32, the program creates a *DtExerciseConnection*. This connection serves as the program's interface to an exercise. A different constructor is used depending on whether the program is compiled for HLA or DIS. If you compile for HLA, the constructor takes a federation execution and federate name. If you compile for DIS, the constructor takes a port number and the three-part exercise:site:application ID. This is the only protocol-specific part of the code.

Managing State and Interaction Information

Applications based on VR-Link typically use callbacks to handle incoming interactions such as fire, detonations, and collisions. For example, a callback named *fireCb* is registered with the *FireInteraction* class at line 34. This callback (defined at line 13) prints a message containing the attacker ID. It executes whenever the exercise connection receives a Fire PDU or interaction during a call to drainInput().

Tracking Entities

We create a reflected entity list in line 39 to keep track of entities found on the network. The entity list tracks the arrival and departure of entities, performs dead reckoning, manages time outs, and performs other entity-tracking tasks.

Managing Time

DtTimeInit (line 42) initializes VR-Link time. This is necessary for the proper operation of other VR-Link time functions. Make this call once, early in your application, if you are calling other VR-Link time functions, either directly or indirectly.

Listening to the Network

At the start of each iteration, the program sets VR-Link simulation time (line 52) to provide a common time value for use by time-related operations that occur within an iteration of the loop (such as the dead-reckoning of multiple entities).

The drainInput() call (line 54) reads and processes any messages arriving through the exercise connection. This call triggers the execution, if needed, of any callbacks you have registered for that exercise connection.

In line 57, the program finds the first entity in the entity list, then in line 62, retrieves the pointer to the entity's entity state repository. Line 68 creates a topographic view of that entity state repository, allowing us to retrieve its position data in topographic coordinates rather than geocentric. (Lines 66 and 67 hard code the coordinates for this example.)

Line 71 obtains and prints the dead-reckoned entity location.

3.8.2. A Send-Only Example

Example 2 illustrates a simple send-only application that simulates the flight of an F18 aircraft. The program begins by sending a fire PDU or HLA fire interaction. Thereafter, the F18 flies north for 10 seconds, updating its position by sending DIS entity state PDUs or HLA attribute updates.

Example 2

```
1   #include "stdio.h"
2   #include "exerciseConn.h"
3   #include "topoView.h"
4   #include "ProcControl.h"
5   #include "EntityTypes.h"
6   #include "entityPub.h"
7   #include "entitySR.h"
8   #include "fireInter.h"
9
10  main()
11  {
12     // Create a connection to the exercise or federation execution
13  #if DtHLA
14     char *execName        = "VR-Link";
15     char *fedName         = "VR-Link talk";
16     DtExerciseConn exConn(execName, fedName);
17  #else
18     int port           = 3000;
19     int exerciseId     = 1;
20     int siteId         = 1;
21     int applicationNum = 16;
22     DtExerciseConn exConn(port, exerciseId, siteId, applicationNum);
23  #endif
24
25     DtEntityType f18Type(DtPlatform, DtPlatformDomainAir,
          DtUnitedStates, DtFighter, DtF18, 0, 0);
26
27     // Create an entity publisher for the entity we are simulating
28     DtEntityPublisher f18EntityPub(f18Type, &exConn,
          DtDrDrmRvw, DtForceFriendly,
          DtEntityPublisher::guiseSameAsType());
29
30     // Hold on to its state repository, where we can set data
31     DtEntityStateRepository *esr = f18EntityPub.entityStateRep();
32
33     // Create a topographic view on the state repository, so we
34     // can set position information in topographic coordinates
35     double refLatitude  = DtDeg2Rad(  35.699760);
36     double refLongitude = DtDeg2Rad(-121.326577);
37     DtTopoView f18TopoView(esr, refLatitude, refLongitude);
38
39     // We can use the ESR to set state
40     esr->setMarkingText("VR-Link");
41
42     // Initialize VR-Link time.
43     DtTimeInit();
44
45     DtVector position(0, 0, -100);
46     DtVector velocity(20, 0, 0);
```

```
47
48      // Send a Fire Interaction
49      DtFireInteraction fire;
50      fire.setAttackerId(f18EntityPub.id());
51      exConn.sendStamped(fire);
52      // Main loop
53      DtTime timestep = 0.05;
54      DtTime simTime = 0;
55      while (simTime <= 10.0)
56      {
57         // Tell VR-Link the current value of simulation time
58         DtSetSimTime(simTime);
59
60         // Process any incoming messages
61         exConn.drainInput();
62
63         // Set the current position information
64         f18TopoView.setLocation(position);
65         f18TopoView.setVelocity(velocity);
66
67         // Call tick, which insures that any data that needs to be
68         // updated is sent.
69         f18EntityPub.tick();
70
71         // Set up for next iteration
72         position[0] += velocity[0] * timestep;
73         simTime     += timestep;
74
75         // Wait till real time equals simulation time of next step
76         DtSleep(simTime - DtGetElapsedRealTime());
77      }
78
79      return 0;
80  }
```

Connecting to An Exercise

Like the listen-only example, this program creates a *DtExerciseConnection* to provide an interface to the RTI or DIS network (lines 12 through 23).

Managing Entities

Line 25 defines the entity type the F18 will use.

To be visible to other applications in the exercise, each locally-simulated entity requires a *DtEntityPublisher*, created in line 28. The entity publisher manages the generation of messages for this particular entity. It provides an entity state repository where you can set state values, and a tick() function, which causes state information to be sent to the network if necessary.

Line 31 sets up a pointer to the entity state repository, then line 37 creates a topographic view on that repository. This lets us set the entity's positional data using topographic coordinates, rather than the default geocentric coordinates. (Lines 35 and 36 hard code the coordinates for this example.) Line 40 shows an example of storing a non-positional state value in the repository.

Sending Interactions

An example of sending an interaction appears in lines 49-51. You can send the interaction using the exercise connection's sendStamped() function.

You can use sendStamped() to send state updates, but it is preferable to let the entity publisher send state updates for you, using data in the entity state repository, as shown in lines 64 through 69.

Sending State Messages

The main loop executes twenty times per second for ten seconds. As in the listen-only example, this program sets simulation time at the start of each iteration (line 58).

While this program's main purpose is to demonstrate the sending of data to the network, it is not a true send-only application. Incoming data is also processed with the drainInput() call on line 61. This call is required for HLA, because this is where we tick the RTI.

The program updates the F18's positional data in its entity state repository in lines 64 and 65, and ticks the entity publisher in line 69 to send the updated data onto the network. Lines 64 and 65 set topographic coordinates through a view, which VR-Link converts to geocentric coordinates.

Thereafter, the only remaining tasks are to increment the F18's position, increment the simulation time, and sleep until it's time to begin the next iteration.

Compiling VR-Link Applications

This chapter explains how to compile and link VR-Link applications:

4.1. Compiling and Linking a VR-Link Application

To build a VR-Link application, you must tell the compiler where to find the VR-Link header files (and for HLA, the RTI header files). In addition, you must link with VR-Link's libraries (and for HLA, the RTI's libraries). If you are not using a C++ linker to link, you must explicitly link with C++ system libraries such as *libC.a*.

4.1.1. Libraries and Header Files

VR-Link consists of four libraries: *mtl*, *vl*, *matrix*, and *vlutil*. The following versions of the *vl* library are provided:

Table 4-1: Versions of the *vl* library

Simulation standard	UNIX	Windows
HLA	*libvlRPR.a*	*vlRPR*.lib*
DIS 2.0.4, IEEE1278.1, or 2.1.4	*libvl5.a*	*vl5*.lib*
DIS 2.0.3	*libvl3.a*	*vl3*.lib*

*See "Libraries for Windows Platforms," on page 4-3

For each library, we have provided a single header file that includes all other header files in that library. The files are:

- *vlInc.h*
- *mtlInc.h*
- *matrixInc.h*
- *vlutilInc.h*.

For convenience, you can include just these four general header files. However, you might speed up compilation of your source files by including only the header files you need (as our examples do).

Libraries for Windows Platforms

On the PC platform, VR-Link uses multithreaded (MT) libraries by default, but also provides two types of multithreaded DLL (MD) libraries. They are in the *.\lib* directory. We distinguish between the libraries by attaching an MT suffix for the libraries compiled with the /MT compiler switch, and attaching an MD or RT suffix for the libraries compiled with the /MD compiler switch. The file names for the libraries are:

Table 4-2: VR-Link libraries for PCs

/MT compiler switch	/MD compiler switch	
vlutilMT.lib	vlutilMD.lib	vlutilRT.lib
mtlMT.lib	mtlMD.lib	mtlRT.lib
matrixMT.lib	matrixMD.lib	matrixRT.lib
vl3MT.lib	vl3MD.lib	vl3RT.lib
vl5MT.lib	vl5MD.lib	vl5RT.lib
vlRPRMT.lib	vlRPRMD.lib	vlRPRRT.lib

The RT libraries have an associated set of DLL files, as follows:

- *vlutilRT.dll*
- *mtlRT.dll*
- *matrixRT.dll*
- *vl3RT.dll*
- *vl5RT.dll*
- *vlRPRRT.dll.*

Using the RT libraries and DLLs instead of the MD libraries can save memory because only once instance of a DLL is loaded into memory no matter how many VR-Link applications you run. Use of the DLLs is conceptually similar to using shared libraries in UNIX.

Notes for Using the RT libraries

- If you build with the RT libraries, use the /MD compiler switch.
- The DLLs must be in your path.
- You must add the #define statement: `#define use_dll`

4.1.2. Compiling for a Particular Simulation Standard Under UNIX

You can select DIS or HLA at compile time by using the flags described in the following sections. *vlhome* represents your VR-Link home directory. RTI_HOME and RTI_ARCH come from the environment, assuming you've configured the environment according to the instructions in "Obtaining and Installing a Runtime Infrastructure (RTI)," on page 2-13.

Notes

- The order of the libraries in the link line is significant. See the Makefiles in *./examples* for further guidance.
- Other flags may be required on some platforms. See the example makefiles and release notes.

HLA

The compile and link flags for HLA are listed in this section. They are organized so that if you are using the DMSO RTI, compilation is done with DMSO RTI libraries. If DMSO RTI libraries are not present, compilation is done using the MÄK RTI libraries.

Compile Flags

```
-Ivlhome/include                    (VR-Link headers)
-I${RTI_HOME}/lang/C++/include      (path to the RTI include directory)
-Ivlhome/RTI/include                (path to MÄK RTI include directory)
-DDtDISVERS=5 -DDtHLA=1 -DDtRPR=1   (for conditional compilation)
```

Link Flags

```
-Lvlhome/lib -lmtl -lvlRPR -lmatrix -lvlutil
-L${RTI_HOME}/lang/C++/lib/$(RTI_ARCH) (path to the DMSO RTI lib directory)
-Lvlhome/RTI/lib                    (path to MÄK RTI lib directory)
-lRTI                               (path to RTI library)
```

DIS

The compile and link flags for DIS 2.0.4 (IEEE 1278.1) and DIS 2.1.4 (IEEE 1278.1a) are listed in this section.

Compile Flags

```
-Ivlhome/include                    (VR-Link headers)
-DDtDISVERS=5                       (for conditional compilation)
```

Link Flags

```
-Lvlhome/lib -lmtl -lvl5 -lmatrix -lvlutil
```

The compile and link flags for DIS 2.0.3 are listed in this section.

Compile Flags

```
-Ivlhome/include                    (VR-Link headers)
-DDtDISVERS=3                       (for conditional compilation)
```

Link Flags

```
-Lvlhome/lib -lmtl -lvl3 -lmatrix -lvlutil
```

4.1.3. Selecting an Example Application's Protocol under UNIX

When you build the example programs that come with VR-Link, you can choose the protocol without editing the Makefile. Use the VERS argument, as shown in these examples for *netdump*:

```
make VERS=3 f18for DIS 2.0.3
make VERS=5 f18for DIS 2.0.4, IEEE 1278.1 or 2.1.4
make VERS=RPR f18for HLA
```

Depending on the value you specify for VERS, a particular file is included in the Makefile, as determined by the following line in the Makefile:

```
include ./makeMods.mk$(VERS)
```

The included file is *makeMods.mk3*, *makeMods.mk5*, or *makeMods.mkRPR* (which are in the example program's source directory). By including a *makeMods* file in this manner, we effectively insert version-specific commands that control what is built.

If desired, you can still choose the protocol by editing the Makefile manually. To do so, comment and uncomment the appropriate lines according to the instructions in the makefile. You'll also need to remove the include line shown above, or replace the *makeMods* files with empty versions.

4.1.4. Compiling for a Particular Protocol Version under Windows

Under Windows, you can set your program's protocol following the procedures in this section. You need to provide build settings in Microsoft Developer Studio to define constants and specify paths for header files and RTI libraries.

---

i In addition to linking with *netapi.lib* and ws*ock32.lib*, you must link with *comctl32.lib*. See the makefiles.

When Compiling for HLA

1. Choose Project → Settings, then select the C/C++ tab (Figure 4-1).
2. Enter the following #define symbols in the space provided for preprocessor definitions:

   ```
   DtDISVERS=5, DtHLA=1, DtRPR=1
   ```

3. In the space provided for additional include libraries, enter your VR-Link include path and the path to your RTI include directory, for example:

```
..\..\include, "c:\Program Files\MAK Technologies\MakRti1.3\include"
```

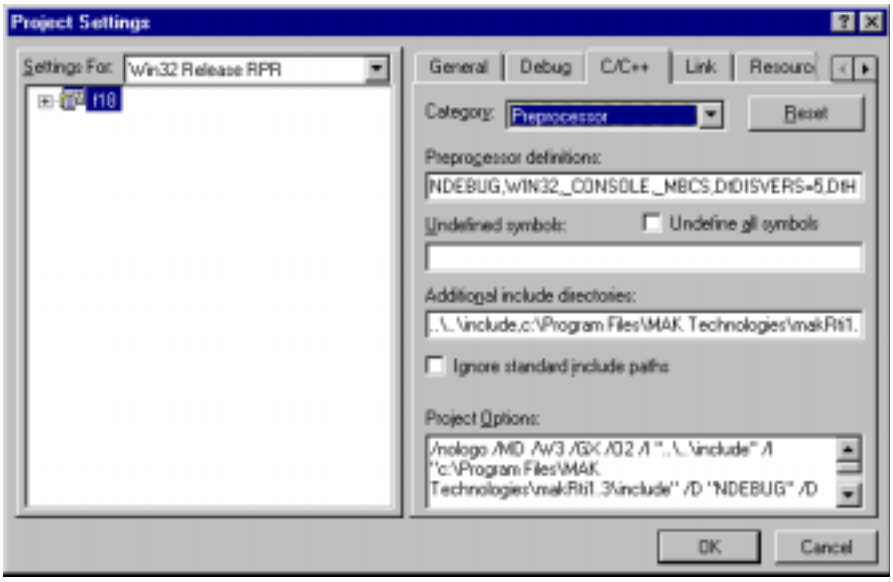


Figure 4-1. Developer Studio Project Settings

---

i The relative pathnames in this procedure (for example, `..\..\include`) are relative to the project *.dsp* file in the *./examples/f18* directory of the default VR-Link installation. The full path would be similar to *c:\Program Files\MAK Technologies\VR-Link\include*.

---

4. Select the Link tab.
5. In the space provided for object/library modules, enter the appropriate RTI, VR-Link and Visual C++ library files, for example:

```
rti13.lib fedtime.lib (or vrlFedtimeMD.lib) vlutilMD.lib mtlMD.lib
          matrixMD.lib vlRPRMD.lib netapi32.lib wsock32.lib comctl32.lib
```

   Also, make sure you list the appropriate paths to these files in the “Additional library path” box, for example:

```
..\..\lib, c:\Program Files\Mak Technologies\MakRti1.3\lib
```

When Compiling for DIS 2.0.4, 2.1.4, or IEEE 1278.1 and IEEE 1278.1a

1. On the C/C++ page under Project Settings, enter the following #define symbol in the space provided for preprocessor definitions:

```
DtDISVERS=5
```

2. In the space provided for additional include libraries, enter your VR-Link include path, for example:

```
..\..\include
```

3. On the Link page, enter the appropriate VR-Link and Visual C++library files, for example:

```
vlutilMD.lib mtlMD.lib matrixMD.lib vl5MD.lib netapi32.lib wsock32.lib
          comctl32.lib
```

   Also, make sure you list the appropriate paths to these files in the “Additional library path” box, for example:

```
..\..\lib
```

When Compiling for DIS 2.0.3

1. On the C/C++ page under Project Settings, enter the following #define symbol in the space provided for preprocessor definitions:

```
DtDISVERS=3
```

2. In the space provided for additional include libraries, enter your VR-Link include path, for example:

```
..\..\include
```

3. On the Link page, enter the appropriate VR-Link and Visual C++library files, for example:

```
vlutilMD.lib mtlMD.lib matrixMD.lib vl3MD.lib netapi32.lib wsock32.lib
          comctl32.lib
```

   Also, make sure you list the appropriate paths to these files in the “Additional library path” box, for example:

```
..\..\lib
```

4.1.5. Selecting an Example Application's Protocol under Windows

In Microsoft Developer Studio, you can choose a protocol for an example application from the Set Active Project Configuration dialog box.

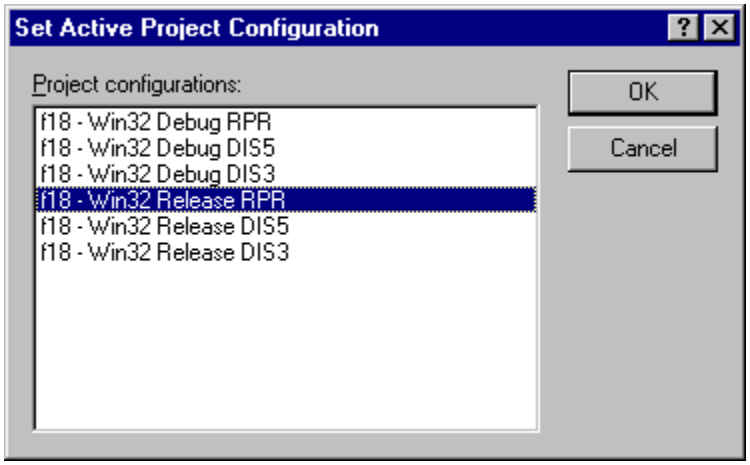


Figure 4-2. Selecting a protocol in the Microsoft C++ IDE.

> To open the Set Active Project Configuration dialog box, choose Build →Set Active Configuration.

If you are building for HLA, make sure to point to the correct location of the RTI include and library directories. The default is to point to the default installation directories for the MÄK RTI.

4.2. Using VR-Link from C

The directory *./examples/simpleC* contains an example of using VR-Link with an application written in C. The example is a C version of the listen-only program described in "A Listen-Only Example," on page 3-19. See the header files *interface.h* and *simpleC.h* for details.

Calling C++ from C

Since VR-Link is a C++ toolkit, any code that directly calls a VR-Link function must be compiled by a C++ compiler. You can compile this code in such a way that it may be called from C. To do so, you need at least one C++ file from which all calls to VR-Link functions can be made. In our example, the file *interface.cxx* serves this purpose. It contains several functions that interact with VR- Link code: initVRLink(), tickVR-Link(), and firstEntityPosition(). To alert the C++ compiler that these functions should be compiled to be callable from C, their declarations are surrounded by the following keywords, as seen by the C++ compiler:

```
extern "C" {
}
```

The file *listenC.c* represents the rest of the application. It is compiled by a C compiler, and does not directly call any VR-Link functions. It does, however, call the functions defined in *interface.cxx* that were compiled to be called from C.

Calling C from C++

You can create a C++ function that interfaces with VR-Link code and can also call C functions. For example, in *interface.cxx*, the callback on Fire PDUs (fireCb) is a C++ function which calls the C function firePrint().

To alert the C++ compiler that an external function is a C function, its declaration is surrounded by the following keywords, as seen by the C++ compiler:

```
extern "C" {
}
```

Header Files that are Included by C and C++

When C and C++ code interact, the declarations of many of the same functions must be included by both C and C++ code. As described above, these declarations need to be surrounded by

```
extern "C" {
}
```

when presented to the C++ compiler. However, these keywords are meaningless to a C compiler, so that they should always be surrounded by:

```
#ifdef __cplusplus
#endif
```

For example, *listenC.h* reads as follows:

```
#ifdef __cplusplus
extern "C" {
#endif

void firePrint(const char *attacker);

#ifdef __cplusplus
}
#endif
```

The Protocol-Independent Interface

You can access most of VR-Link's features through its protocol-independent API, described in this chapter. Protocol-specific portions of the API are described in Chapter 6, *The HLA-Specific Interface* and Chapter 8, *The DIS-Specific Interface*.

Topics covered in this chapter include:

5.1. Introduction to the Protocol-Independent Interface

The protocol-independent interface is a set of classes that encompass most of the features of VR-Link and allow you to create applications that will work in both DIS and HLA without significant modification. In addition to its multi-protocol capability, and perhaps more importantly, the protocol-independent interface shields you from many of the intricacies of the underlying protocol-specific classes and functions. At the same time, the protocol-specific classes are available to you if you want to use them.

When you create a protocol-independent application, you do not create one executable that works with both protocols; you write one application and compile it for each protocol by specifying the protocol at compile time. In cases where a class uses different constructors or functions depending on the protocol, such as creating an exercise connection, you need to use #ifdef(s) so that your application compiles using the correct constructors.

5.2. Connecting to Exercises

In general, an HLA federation or DIS exercise can be defined as a collection of simulation applications communicating with each other through the exchange of FOM-defined data (for HLA) or PDUs (for DIS). A simulation application (called a federate in HLA) can, in turn, be defined as a participant in an exercise. Such an application may simulate one or more entities.

VR-Link's class *DtExerciseConn* is a simulation application's interface to an exercise. The application sends data to other applications, and receives data from other applications, through its *DtExerciseConn*. Virtually all applications based on VR-Link make use of a *DtExerciseConn*.

There are HLA and DIS versions of *DtExerciseConn*. For HLA, the *DtExerciseConn* exchanges data with other applications by making calls to the RTI. For DIS, data is usually exchanged through the sockets-related system calls sendTo() and recvFrom().

The HLA version of *DtExerciseConn* is defined in *hExerciseConn.h*. The DIS version is defined in *dExerciseConn.h*. If you are writing an application for use with both HLA and DIS, you do not have to include both header files. You can include *exerciseConn.h*, which includes the appropriate version depending on whether or not you have included the *DtHLA*=1 definition in your compile line.

Although the two versions of *DtExerciseConn* are defined separately, most public member function names and signatures are shared by the two versions, so that you can make most of these calls regardless of which protocol you are using. Much of this commonality is enforced because both versions are derived from *DtBaseExerciseConn*, an abstract class that defines much of the *DtExerciseConn* interface.

Creating a *DtExerciseConn* must be done differently for HLA and DIS because they require different initialization parameters.

In this chapter, we will note other instances where you must write different code depending on the simulation standard you are using.

5.2.1. Creating an Exercise Connection for HLA

In order to create a *DtExerciseConn* in HLA, you must provide a federation execution name and a federate name. All *DtExerciseConn* constructors require these two arguments. One of *DtExerciseConn*'s constructors allows you to specify the name of the FED file that you want to use. The FED file provides information about the FOM to the RTI and to VR-Link. If you do not provide the name of a FED file to the *DtExerciseConn* constructor, we assume that you are using a file with the same name as the federation execution, but with a *.fed* extension. For example, if the federation execution name is VR-Link, we assume the FED file is *VR-Link.fed*.

VR-Link and the MÄK RTI first look for the FED file in the directory from which you run your application, then in the directory specified by the RTI_CONFIG environment variable. The DMSO RTI looks only in the directory specified by RTI_CONFIG, which defaults to the *config* directory under the main DMSO RTI installation directory.

! Make sure that the FED file that you are using with your *DtExerciseConn* appears in one of these places before running your application.

The default FED file supported by VR-Link is *VR-Link.fed*. It is in your VR-Link home directory. If you are using the FOM described by this file, you must copy *VR-Link.fed* to the RTI_CONFIG directory, or to the directory that you're running from.

All applications that want to interact in the same HLA world must use the same federation execution name, and the exact same FED file.

The federate name is a name you give to your application. It is used for RTI diagnostics. Federate names do not have to be unique in a federation.

The *DtExerciseConn* constructors:

- Initialize an application's RTI ambassadors (which causes the FED file to be read)
- Create the named federation execution if it does not already exist
- Join the federation execution.

If you are using the DMSO RTI, make sure that their *rtiexec* application is running, otherwise you will not be able to create or join the federation execution. If you use the MÄK RTI, running its *rtiexec* is optional. (See "Obtaining and Installing a Runtime Infrastructure (RTI)," on page 2-13.)

5.2.2. Creating an Exercise Connection for DIS

To create an exercise connection for DIS, you need:

- A UDP port number
- An exercise ID number
- A site number
- An application number.

All applications that want to interact in a DIS exercise must use the same UDP port number and exercise ID number.

If a packet server is running on the specified UDP port, the *DtExerciseConn* will connect to that packet server, rather than connecting directly to the network. (For information about the packet server and its benefits, see "The MÄK Packet Server (UNIX Only)," on page 10-13.)

By default, DIS PDUs sent through the *DtExerciseConn* are sent to the broadcast address of the machine's primary network interface. The *DtExerciseConn* receives any unicast or broadcast packets destined for the machine on the specified port. Additional constructors and functions of the DIS version of *DtExerciseConn* allow greater flexibility in setting up your network connection, and are discussed in "Configuring Your Connection to the DIS Network," on page 8-23.

5.2.3. Compiling Protocol-Independent Applications

Applications that can be built for both DIS and HLA typically use conditional compilation when code differs between the two domains. For example, the following code uses different constructors for *DtExerciseConn* depending on the simulation standard chosen at compile time:

```
#if DtHLA
    char *execName= "VR-Link";
    char *fedName= "myApp";
    DtExerciseConn exConn(execName, fedName);
#else
    int port   = 3000;
    int exerciseId= 1;
    int siteId = 1;
    int applicationNum= 15;
    DtExerciseConn exConn(port, exerciseId, siteId, applicationNum);
#endif
```

Most exercise connections described in this chapter work the same way in both HLA and DIS and do not need conditional compilation.

5.2.4. *DtExerciseConn* Member Functions

Table 5-1 lists and describes the most commonly used protocol-independent member functions for *DtExerciseConn*.

Table 5-1: DtExerciseConn member functions

Function	Description
addPostDrainCallback()	Allows you to register functions (with the *DtExerciseConn*) that you want to have called automatically each time drainInput() is called. See "Post-Drain Callbacks," on page 5-8.
removePostDrain Callback()	Allows you to unregister functions that you no longer want to have called each time drainInput() is called. See "Post-Drain Callbacks," on page 5-8.
drainInput()	Causes VR-Link to read and process input from the exercise. Most user callbacks are executed from within drainInput(). See "Receiving Data from the Exercise," on page 5-8.
nextEventID()	Generates consecutive event IDs for use in fire, detonate, or collision interactions. For DIS, the event ID is composed of the site ID, application ID, and event number based on the exercise connection's site and host. For HLA, the event ID is composed of the object ID (from setObjectId()) and event number. You can choose your own event IDs in both DIS and HLA. Event IDs are not managed by the RTI as Object IDs are. Therefore it is up to the application to ensure that event IDs are unique.
send() sendStamped()	Sends interaction messages, object updates, and DIS PDUs to the exercise (See"Sending Data to the Exercise," on page 5-8.) sendStamped() includes a time stamp with outgoing messages.
setTimeStampType()	Allows you to specify the type of time stamp being sent. The options are DtTimeStampAbsolute and DtTimeStampRelative. See "Time Stamp Type," on page 5-9.
timeStampType()	Returns the time stamp type.
vrlinkVersion()	This static function returns the version of VR-Link that you are running.

The following sections contain more detailed information about these functions.

Sending Data to the Exercise

send() or sendStamped() are used to send interactions to the exercise. See "Sending Interactions," on page 5-11 for details.

Sending of object state information is typically handled using higher-level object-management classes, such as *DtEntityPublisher*, rather than by sending individual update message directly using *DtExerciseConn*. See "Working with Locally Simulated Entities," on page 5-14 for details.

Receiving Data from the Exercise

The DtExerciseConn::drainInput() function receives and processes information from the RTI or the DIS network. From within DtExerciseConn::drainInput(), VR-Link executes any callbacks that you have registered on *DtInteractions* (for either HLA or DIS), *DtPdus* (for DIS), or various RTI services (for HLA).

Your application should call DtExerciseConn::drainInput() periodically, usually once per simulation loop. This is true for HLA even if your application doesn't care about remotely simulated objects or remotely generated interactions, because the RTI requires RTI::tick() to be called periodically.

HLA only

drainInput() repeatedly calls RTI::tick() until it fails to return RTI::RTI_TRUE (meaning there is no more data waiting to be read).

DIS only

drainInput() reads PDUs from the network directly, or from the Packet Server, depending on the system configuration.

The optional **timeout** argument to drainInput() lets you specify a time (in seconds) after which drainInput() should return even if there is more data left to be read. A value of -1 (the default) is to never return if there is still data to be read. A value of 0 means return immediately. The second argument to drainInput() specifies the number of packets to be read or the number of times to call RTI::tick() in between tests of whether the timeout has been exceeded.

Post-Drain Callbacks

Post-drain callbacks are callback functions that you can register with a *DtExerciseConn*, and have called automatically by DtExerciseConn::drainInput() immediately before it returns. Post-drain callbacks are called after all interaction, PDU, and RTI service callbacks have been made. For more information about callbacks in general, see "Using Callbacks," on page 3-14.

Post-drain callbacks have the following function signature:

```
void myCallback(void* usr);
```

They are registered with a *DtExerciseConn* using addPostDrainCallback(), and unregistered using removePostDrainCallback(). The value passed to your callback function as the **usr** pointer is the value that was passed as the **usr** argument to addPostDrainCallback().

The most common use of post-drain callbacks is to get around an HLA restriction. RTI services cannot be invoked from within user callbacks on other RTI services. For example, at the time that you are processing an incoming interaction within an interaction callback, you cannot send an interaction in response. But you can register a response-sending post-drain callback that is executed after all RTI service callbacks have been invoked, when it is safe to send interactions. See "Making RTI Calls in Response to Interactions," on page 5-13.

Time Stamp Type

DtExerciseConn's setTimeStampType() function allows you to set the time stamp type of your application to either DtTimeStampAbsolute or DtTimeStampRelative. Use DtTimeStampAbsolute if your local clock is synchronized to a global exercise clock, and you want to use the values of time stamps on messages from other synchronized hosts in dead-reckoning calculations. The default, DtTimeStampRelative, indicates that your local clock is not necessarily synchronized to anyone else's clock, and therefore time of receipt is used in dead-reckoning rather than incoming time stamp values. See "Timestamps," on page 3-18 for more information.

When using absolute timestamping, make sure to keep VR-Link simulation time in step with your system clock by passing the current time to *DtSetSimTime* each frame.

5.3. Working with Interactions

Within VR-Link, we use the term interaction to refer to the data that is exchanged among simulation applications in an exercise to describe events, such as the firing of a munition, detonation of a munition, or collision of entities. While the term is from HLA (rather than DIS), we use it in a protocol-independent sense to refer to an HLA interaction or a DIS PDU that describes an event.

Each type of interaction has its own class, for example, fire, detonate, and collision interactions are represented by *DtFireInteraction*, *DtDetonationInteraction*, and *DtCollisionInteraction* respectively. For both DIS and HLA, the individual interaction classes are derived from *DtInteraction*. *DtInteraction* is defined in *hInteraction.h* (HLA) and *dInteraction.h* (DIS). If you include *interaction.h*, an appropriate version is included.

There are DIS and HLA versions of each of the interaction classes. Although the two versions of each of these classes are defined separately, most of the public member function names and signatures are used in both versions, so you can use most of them in the same way regardless of whether you are using DIS or HLA.

A similar naming scheme exists for individual interaction classes and their header files. Table 5-2 lists some interaction classes and their associated header files for HLA, DIS, and protocol-independent programming. The protocol-independent header files include the appropriate version of the classes depending on the protocol for which you are building.

Table 5-2: Interaction class header file naming conventions

Interaction	HLA	DIS	Protocol-independent
DtFireInteraction	*hFireInter.h*	*firePdu.h*	*fireInter.h*
DtDetonationInteraction	*hDetInter.h*	*detonatePdu.h*	*detonateInter.h*
DtCollisionInteraction	*hCollideInter.h*	*collidePdu.h*	*collideInter.h*

DIS only

For DIS applications, *DtFireInteraction*, *DtDetonationInteraction*, *DtCollisionInteraction*, and so on are equivalent to *DtFirePdu*, *DtDetonationPdu*, and *DtCollisionPdu*. That is, one is typedefed to the other. You can use the *DtPdu* class names if your application is only for DIS. However, you must use the *DtInteraction* names if you are writing code for both DIS and HLA, and this is what we recommend.

In DIS, *DtInteraction* is derived from *DtPdu*. The *DtInteraction* class itself has no real functionality, but the extra level gives our class hierarchy some structure. For example, *DtFirePdus* are (and can be implicitly cast to) *DtInteractions* and *DtPdus*. Entity state PDUs are not interactions, and thus *DtEntityStatePdu* is derived directly from *DtPdu*, not from *DtInteraction*.

5.3.1. Sending Interactions

To send interactions:

1. Create an instance of the appropriate interaction class (derived from *DtInteraction*).
2. Use its mutator functions to set values for the different parameters.
3. Send it using *DtExerciseConn*'s sendStamped() function.

The following code is an example of creating and sending a fire interaction, which informs the exercise that you have fired a munition. The example assumes that *myId*, *targetId*, and *missileId* are existing objects.

```
DtExerciseConn exerciseConn(...);
    ...
// Create a DtFireInteraction
DtFireInteraction fireInter;
// Fill the DtFireInteraction with data
fireInter.setAttackerID(myId);
fireInter.setTargetID(targetId);
fireInter.setMunitionId(missileId);
fireInter.setEventId(exerciseConn.nextEventId());
    ...
// Send to the exercise
exerciseConn.sendStamped(fireInter);
```

In both DIS and HLA, sendStamped() includes a time stamp with outgoing interactions. Because this is usually desired, we recommend that you use sendStamped() to send interactions, rather then send().

The value used for the time stamp is the value returned by *DtExerciseConn*'s currentTimeForStamping() function. By default, this function returns the current value of the system clock when using relative timestamping, or the current value of VR-Link simulation time when using absolute timestamping. If you would like sendStamped() to compute timestamps for outgoing messages differently, you can subclass *DtExerciseConn* and override currentTimeForStamping(). Timestamps are marked as either "relative" or "absolute" depending on the *DtExerciseConn*'s current time stamp type. For more information, see "Timestamps," on page 3-18.

For a description of the mutator functions for a particular interaction class, see the appropriate header files.

DIS only

For DIS, sendStamped() also puts the *DtExerciseConn*'s exercise ID into the header of outgoing PDUs.

5.3.2. Receiving Interactions

Incoming interactions are handled through interaction callbacks. See "Using Callbacks," on page 3-14 for more information about callbacks.

Application developers write and register callback functions for a particular interaction type with VR-Link, and VR-Link calls those functions when it receives an interaction of that type from within DtExerciseConn::drainInput(). The received interaction is passed to the callback function, allowing you to process the interaction as it arrives.

Each interaction class has the static member functions addCallback() and removeCallback(). These functions allow you to register and unregister your callback functions for particular interaction classes.

There is a different callback function type for each interaction class. For example, only callback functions that take a *DtFireInteraction* pointer as an argument can be registered with the *DtFireInteraction* class. A *DtFireInteraction* callback might look like this:

```
void myFireCallback(DtFireInteraction* inter, void* usr)
{
   printf("Got a Fire Interaction from %s\n",
      inter->attackerId().string());
}
```

It would be registered with VR-Link's *DtInteraction* class like this:

```
DtFireInteraction::addCallback(exerciseConn, myFireCallback, NULL);
```

Since addCallback() is a static member of *DtFireInteraction*, it can be called using the qualified name, rather than calling it on a particular instance of *DtFireInteraction*.

The value passed to your callback function as **usr** is the value that was passed as the **usr** argument to addCallback(). Often, this **usr** argument passes a pointer to an object on which you would like to call a member function from within your callback. Simply cast **usr** back to the object's type within the callback. See "Using Callbacks," on page 3-14 for examples of passing objects through the usr pointer.

When an interaction is received for which there are multiple callbacks registered, the callbacks are called in the opposite order from that in which they were registered.

DtInteractions are transient, that is, they are deleted by VR-Link immediately after the last callback function registered for the interaction is called. Therefore, do not attempt to delete or save a pointer to the *DtInteraction* passed to your callback function. If you need to save the data, make a copy of the *DtInteraction*.

Making RTI Calls in Response to Interactions

In HLA applications, interaction callbacks are called from within RTI::tick() (which is called by DtExerciseConn::drainInput()). The RTI does not allow calls to RTI services from within other RTI service calls (including RTI::tick()). This means that you cannot directly or indirectly make any RTI calls from within your interaction callbacks.

If you need to make RTI calls based on the receipt of a *DtInteraction*, you can accomplish this as follows:

> From within your interaction callback, use DtExerciseConn::addPostDrainCallback() to register a function with VR-Link that will get executed right before drainInput() returns, when it is safe to make RTI calls.

For example, the following code will ensure that your application sends out a fire interaction each time it receives a fire interaction:

```
void sendFire(void *usr)
{
    // Cast the usr pointer back to a DtExerciseConn
    DtExerciseConn* exConn = (DtExerciseConn*) usr;
    // Create and send a DtFireInteraction
    DtFireInteraction fireInter;
    fireInter.setAttackerId(...);
    ...
    exConn->sendStamped(fireInter);
    // Remember to deregister ourselves, so that we don't
    // get called again the next time drainInput is called.
    exConn->removePostDrainCallback(sendFire, exConn);
}
void fireCb(DtFireInteraction* inter, void *usr)
{
    // Cast the usr pointer back to a DtExerciseConn
    DtExerciseConn* exConn = (DtExerciseConn*) usr;
    // Can't send a fire interaction here, since that
    // would involve an RTI call from within an RTI-service
    // callback. So instead, register a fire-sending
    // function (sendFire) as a postDrainCallback.
    exConn->addPostDrainCallback(sendFire, exConn);
}
main()
{
    DtExerciseConn exerciseConn(...);
    ...
    // Register fireCb as a fireInteraction callback,
    // passing a pointer to the DtExerciseConn as the usr
    // pointer, so that it will be available to us within
    // the callback.
    DtFireInteraction::addCallback(&exerciseConn, fireCb,
       &exerciseConn);
    ...
}
```

For more information, see "Post-Drain Callbacks," on page 5-8.

5.4. Working with Entities

VR-Link provides many classes that facilitate informing the exercise about the state of your locally simulated entities and other objects, and informing your application about the state of remotely simulated entities and other objects. These classes include:

- *DtObjectPublisher* and its subclasses, for example, *DtEntityPublisher,* send updates to the exercise.
- *DtReflectedObjectList* and its subclasses, for example, *DtReflectedEntityList,* keep track of remote objects.
- *DtReflectedObject* and its subclasses, for example, *DtReflectedEntity,* represent a single remote object.
- *DtStateRepository* and its subclasses, for example, *DtEntityStateRepository,* encapsulate an object's state.

In discussing how to manage objects using publishers, reflected object lists, reflected objects, and state repositories, we will focus on the classes used to manage entities - the most common kind of object in most exercises. Our concept of entities reflects the DIS and RPR FOM concept. Entities include vehicles, life forms, and most other types of real-world objects that have a position in space.

Publishers, reflected object lists, reflected objects, and state repositories exist in VR-Link for other types of objects as well, including aggregates, emitters, transmitters, designators and receivers. For information about managing these other types of objects, see "Working with Other Types of Objects," on page 5-34.

5.5. Working with Locally Simulated Entities

To inform other simulation applications about the state of locally-simulated entities, you must:

1. Create a *DtEntityPublisher* for each locally-simulated entity.
2. Update its current state during every frame through its *DtEntityStateRepository.*
3. Call DtEntityPublisher::tick().

tick() ensures that any needed information is sent to the other participants through the exercise connection.

5.5.1. Creating a *DtEntityPublisher*

There are both HLA and DIS versions of *DtEntityPublisher*, defined in *hEntityPub.h* and *dEntityPub.h* respectively. Rather than include both, you can include *entityPub.h*, which will include the appropriate version based on whether or not you have included the *DtHLA*=1 definition in your compile line.

Although the two versions of *DtEntityPublisher* are defined separately, most of the public member function names and prototypes have the same names, so you can make most of these calls regardless of which protocol you are using.

An instance of *DtEntityPublisher* can be created in a protocol-independent manner using a constructor that takes a *DtExerciseConn* and *DtEntityType* as arguments. For example:

```
DtExerciseConn exConn(...);
DtEntityType tankType(1, 1, 225, 1, 1, 0, 0);
DtEntityPublisher tankPub(tankType, &exConn);
```

The *DtExerciseConn* is the connection through which the publisher sends its updates.

The *DtEntityType* (defined in *hostStructs.h*) represents a seven-component enumeration defined by the DIS protocol and reused by DIS-based FOMs like the RPR FOM. It is used in outgoing entity state PDUs and in HLA updates (if you are using a FOM that includes this concept), and also determines which HLA object class is used to represent the entity in an HLA exercise. In the previous example, the entity type is that of an M1A1 tank. Please see the DIS Enumerations Document for a comprehensive list of valid entity types.

Other *DtEntityPublisher* Constructors

A second protocol-independent constructor takes as arguments three attributes of an entity that are not likely to change during a simulation:

- Dead-reckoning algorithm
- Guise (alternate entity type)
- Force ID.

DtEntityPublisher has protocol-specific constructors that allow you to specify parameters such as the HLA object class to represent the object. These are described in the chapters on the protocol-specific interfaces.

Choosing Entity Identifiers

All versions of the *DtEntityPublisher* constructor have an optional final argument that allows you to choose an identifier for the entity. Object identification is handled differently in DIS and HLA. Please see "Identifying Objects," on page 5-32 for more information about the different ways of identifying objects.

HLA only

In HLA, the identifier that you are allowed to choose is the object name - an arbitrary character string that can be used to refer to an entity in inter-federate communications. If you omit the name argument to the *DtEntityPublisher* constructor, as in the example on page 5-15, the RTI chooses one for you. The object handle, another form of object identification is always chosen by the RTI.

DIS only

In DIS, the identifier that you are choosing is a *DtEntityIdentifier*, the three-component identifier that is used both within an application and between applications to identify an entity. If you omit the identifier argument to the *DtEntityPublisher*, as in the example on page 5-15, VR-Link chooses one for you, using *DtExerciseConn*'s nextId() function.

Example:

```
#if DtHLA
DtGlobalObjectDesignator id = "Object1";
#else
DtGlobalObjectDesignator id = DtEntityIdentifier(1, 2, 3);
#endif

DtExerciseConn exConn(...);
DtEntityType tankType(1, 1, 225, 1, 1, 0, 0);
DtEntityPublisher tankPub(tankType, &exConn, id);
```

If you do not choose your own identifier for your object when you create the *DtEntityPublisher*, you can find out what identifier was chosen using the following *DtEntityPublisher* inspector functions:

- globalId() returns the global identifier of the object - the object name in HLA, or the entity identifier triplet in HLA
- localId() or id() returns the local identifier of the object - the object handle in HLA, or the entity identifier triplet again in DIS.

5.5.2. Setting an Entity's State

You can set a *DtEntityPublisher*'s state through its *DtEntityStateRepository* – a container for an entity's state. It is defined in *entitySR.h*.

DtEntityStateRepository is derived from *DtBaseEntityStateRepository*, so some inherited member functions appear only in the base class.

In each iteration of your simulation, update the *DtEntityStateRepository* of each entity you are simulating locally so that it contains the current state of the entity before calling DtEntityPublisher::tick(). You can obtain a pointer to a *DtEntityPublisher*'s *DtEntityStateRepository* using the member function DtEntityPublisher::entityStateRep(). (You can use esr() as shorthand for entityStateRep().)

DtEntityStateRepository has mutator functions that allow you to set the various components of an entity's state. Time, space, position information can be set with:

- setLocation()
- setVelocity()
- setAcceleration()
- setOrientation()
- setRotationalVelocity().

These functions (and the corresponding inspector functions) take an optional argument called **timeNow**. **timeNow** specifies the value of simulation time for which you would like to set or inspect the data. The default is the current value of VR-Link simulation time as last set with *DtSetSimTime*.

Components that affect the outward appearance of an entity can be set with functions such as:

- setDamageState()
- setFlamesPresent()
- setEngineSmokeOn()
- setHatchState().

See *entitySR.h* for a complete list.

Corresponding inspector functions also exist in *DtEntityStateRepository*, but these are more typically used to inspect a remote entity's state, rather than that of a locally simulated entity that you have set yourself.

Some mutator functions take enumerations as arguments. Values for these enumerations are in *disEnums.h*. Functions that take *DtBooleans* should be passed either DtTrue or DtFalse.

5.5.3. Coordinates

Positions, velocities, and accelerations must be set in geocentric coordinates, as specified in the DIS Standard and DIS-based FOMs like the RPR FOM. In this coordinate system, the origin represents the center of the earth, the X-Y plane goes through the equator, with the X axis passing through the prime meridian, the Y axis passing through 90 degree east longitude, and the Z axis pointing through the north pole. Use the *DtVector* class to set coordinates. Coordinates are specified in meters.

Orientation is specified as a *DtTaitBryan*, a class that represents three angles – successive rotations needed to transform from the geocentric coordinate system to the entity coordinate system (origin at the center of mass, X points out the front, Y points out the right side, Z points out the bottom.) The specific sequence of rotations is known as the Tait-Bryan sequence, in which you first rotate about Z, then about the new Y, then about the newest X axis. The angles are specified in radians.

See "Coordinate Conversions," on page 9-7 for illustrations of the different coordinate systems, information about converting between rotation matrices and Euler angles, and information about converting among the many different coordinate systems supported by VR-Link.

Alternatively, VR-Link provides various View classes, which provide the ability to set data in a *DtEntityStateRepository* in other coordinate systems without performing explicit conversions. These are described in "Coordinate Views," on page 5-37.

5.5.4. Example of Setting the State of An Entity

The following code sample is an example of using *DtEntityStateRepository*'s mutator functions to set the current state of an entity you are simulating:

```
DtEntityPublisher tankPub(...);
...
// Grab a pointer to our entity publisher's ESR
DtEntityStateRepository *esr = tankPub.entityStateRep();
// Set location to somewhere near Ft. Hunter Liggett, CA,
// in geocentric coordinates.
esr->setLocation(DtVector(-2696545.0, -4430407.0,
           3701906.0));
// Velocity and acceleration are also in geocentric
// coordinates
esr->setVelocity(DtVector(100.0, 100.0, 100.0));
esr->setAcceleration(DtVector(0.0, 0.0, 0.0));
// Set orientation as three Euler angles in radians
esr->setOrientation(DtTaitBryan(-2.11, 0.948, 2.469);
// Set angular velocity vector in body coordinates
esr->setRotationalVelocity(DtVector(0.0, 0.10, -0.125);
// Indicate that the entity is on fire
esr->setFlamesPresent(DtTrue);
// Indicate that the entity is slightly damaged
// The DtDamageState enumeration is in disEnums.h
esr->setDamageState(DtDamageSlight);
...
```

You can use DtEntityStateRepository::printData() to print the current contents of a *DtEntityStateRepository* in human readable form.

5.5.5. Using the DtEntityPublisher::tick() Function

DtEntityPublisher::tick() is the function that does most of the *DtEntityPublisher*'s work. Call it for each *DtEntityPublisher* once per simulation frame, after the entity's state has been updated using the *DtEntityStateRepository*'s mutator functions.

The tick() function decides what data needs to be sent to the other exercise participants and formats and sends that data through the *DtEntityPublisher*'s exercise connection.

How the tick() Function Knows When to Send Data

DtEntityPublisher internally maintains a second *DtEntityStateRepository* representing the way remote applications currently see the entity, based on prior entity state updates and dead-reckoning calculations. VR-Link compares the current state with this lower fidelity representation of state. If position or orientation differences exceed the *DtEntityPublisher*'s thresholds, or if any other attributes have changed at all, data needs to be sent so that remote applications will have the most current information.

DIS only

Current data is sent whenever:

- Position or orientation thresholds are exceeded
- Other state data changes
- The time since the last update was sent exceeds a threshold.

In the third case, data is sent even if nothing has changed. (This DIS rule is sometimes known as an entity heartbeat.) If data needs to be sent, an Entity State PDU is sent to the exercise containing current values for all fields.

HLA only

An attribute update is sent containing values for only those attributes whose update conditions (as specified by the FOM) have been met. The update condition for most attributes is any change since the attribute was last sent. However, in the RPR FOM, position, velocity, acceleration, orientation, and angular velocity are always updated together based on a single update condition - when the current position or orientation differs from the value that would be computed by dead-reckoning exceeds thresholds, as described earlier in this section.

5.5.6. Setting Position and Orientation Thresholds

Position and orientation thresholds are used to determine when position and orientation data needs to be sent to other exercise participants (as described in "How the tick() Function Knows When to Send Data," on page 5-19). The *DtThresholder* class (defined in *thresholder.h*) holds threshold values. *DtThresholder* has a set of global threshold values that are shared among all entities that have not overridden them. To change the global thresholds values, use *DtThresholder*'s static mutator functions.

You can override global thresholds on a per-entity basis through the mutator functions of each entity's thresholder object, which can be obtained using DtBaseEntityStateRepository::thresholder(). *DtEntityPublisher* has setDfltThreshold() functions for compatibility with older releases of VR-Link, but the *DtThresholder* functions should be used instead. A *DtThresholder* has a *DtArtPartThresholder* (available through the artPartThresholder() function) through which you can set and inspect articulated parts thresholds. *DtArtPartThresholder* is defined in *artPartThresh.h*.

Translation threshold is specified in meters, and defaults to 1.0. Rotation threshold is specified in radians and defaults to the radian equivalent of 3.0 degrees.

DIS only

For DIS, there is an additional time threshold that defaults to 5.0 seconds. Time is specified in seconds.

5.5.7. Removing a Locally-Simulated Entity

Deleting a *DtEntityPublisher*'s removes an entity from the exercise.

HLA only

In HLA, the *DtEntityPublisher* destructor calls the deleteObjectInstance RTI service.

DIS only

In DIS, the *DtEntityPublisher* destructor sends a final Entity State PDU with the FinalPdu appearance bit (bit 23) set. If you do not want a *DtEntityPublisher* to send this final PDU on destruction, pass an argument of DtFalse to its sendFinalPduOnDestruction() function at some point before deleting it.

5.6. Working with Remote Entities

A *DtReflectedEntityList* maintains the current state of entities that VR-Link learns about through updates received from other participants in an exercise.

DIS only

For DIS, an application receives entity state PDUs sent by itself along with those sent by other applications. Therefore, in DIS the *DtReflectedEntityList* contains a representation of locally simulated entities as well.

Each entity in the Reflected Entity List is represented by an instance of *DtReflectedEntity.* The list provides member functions that let you look up a *DtReflectedEntity* either local ID or global ID, or iterate through all of the entities in the list.

VR-Link provides both HLA and DIS versions of *DtReflectedEntityList* and *DtReflectedEntity*, which are defined in *hReflEntList.h*, *hReflectedEnt.h*, *dReflEntList.h*, and *dReflectedEnt.h*. Rather than include all of these header files, you can include *reflEntList.h* and *reflectedEnt.h*, which include the appropriate versions based on whether or not you have included the *DtHLA*=1 definition in your compile line.

Although the two versions of these classes are defined separately, most of the public member function names and prototypes are shared by the two versions, so that you can make most of these calls regardless of which protocol you are using.

5.6.1. Creating Reflected Entity Lists

A *DtReflectedEntityList* is created on an exercise connection as follows:

```
DtExerciseConn exConn(...);
   ...
DtReflectedEntityList(&exConn);
```

Whenever the *DtReflectedEntityList* detects a new entity, it automatically creates a new *DtReflectedEntity* to represent it. Whenever it receives a state update, it updates the corresponding *DtReflectedEntity* to reflect the current state. These two events will occur automatically as long as DtExerciseConn::drainInput() is called periodically within your application. No further action is required before inspecting data on remote entities.

5.6.2. Iterating Through A *DtReflectedEntityList*

DtReflectedEntityList has first() and last() member functions, which return the first and last entities in the list. *DtReflectedEntity* has next() and prev() member functions, which return the next and previous entity in the list. These two functions return NULL if you try to read past the end or beginning of the list. You can iterate through all entities in the list as follows:

```
DtReflectedEntityList rel(...);
...
for (DtReflectedEntity* ent = rel.first();
    ent;
    ent = ent->next())
{
    ...
}
```

DtReflectedEntity also has wrapNext() and wrapPrev() member functions that are similar to next() and prev(), but which loop back to the first or last entity in the list when you try to move past the end or beginning of the list.

You can obtain the total number of entities in the *DtReflectedEntityList* with the count() member function.

If you want to look up a *DtReflectedEntity* by its ID, use the *DtReflectedEntityList* member function lookup(). Lookup can accept either a global ID (object name in HLA, or Entity Identifier triplet in DIS), or a local ID (object handle in HLA, or the Entity Identifier triplet, again, in DIS). For example:

```
DtGlobalObjectDesignator id = fireInteraction.targetId();
DtReflectedEntity* ent = reflectedEntityList.lookup(id);
```

or

```
#if DtHLA
DtObjectId id = 15;
#else
DtObjectId id = DtEntityIdentifier(1, 2, 3);
#endif
DtReflectedEntity* ent = reflectedEntityList.lookup(id);
```

See "Identifying Objects," on page 5-32 for more information about the different ways of identifying objects.

5.6.3. Inspecting an Entity's State

DtReflectedEntity uses a *DtEntityStateRepository* to store the current state of the entity. This is the same class used by *DtEntityPublisher* to store the state of a locally simulated entity. See "Setting an Entity's State," on page 5-17 for more details. Individual state data items are inspected through *DtEntityStateRepository* inspector functions rather than directly through the *DtReflectedEntity*.

You can get a pointer to a *DtReflectedEntity*'s *DtEntityStateRepository* using DtReflectedEntity::entityStateRep() or DtReflectedEntity::esr().

DtEntityStateRepository has inspector functions that allow you to look at various components of an entity's state. Time, space, position information can be obtained using:

- location()
- velocity()
- acceleration()
- orientation()
- bodyToGeoc()
- rotationalVelocity().

You can examine components that affect the outward appearance of an entity with functions such as:

- damageState()
- flamesPresent()
- engineSmokeOn()
- hatchState().

DtEntityStateRepository has inspector functions that return the last value that was passed to the respective mutator function. These functions make it easier to obtain non-dead-reckoned data from an *DtEntityStateRepository* that is, in general, doing dead-reckoning. The functions are:

- lastSetLocation()
- lastSetVelocity()
- lastSetAcceleration()
- lastSetOrientation()
- lastSetRotationalVelocity().

See *entitySR.h* for the complete list. *DtEntityStateRepository* is derived from *DtBaseEntityStateRepository*, so inherited functions are in *baseEntitySR.h*.

HLA only

In HLA, when you create a *DtReflectedEntityList*, VR-Link requests an update of all attributes. However, because of the way HLA works, you can't be absolutely certain that all the attributes have been updated at the point that you inspect them. For information about how to resolve this problem, see page 5-28.

Mutator functions also exist within *DtEntityStateRepository*, but these are usually used to set a locally simulated entity's state, rather than that of a reflected entity, which is set automatically from data in state updates received from the exercise.

Positions, velocities, and accelerations returned by *DtEntityStateRepository* are in geocentric coordinates, as specified in the DIS Standard and DIS-based FOMs. (See "Coordinates," on page 5-18.) Orientation is available in one of the following forms:

- The orientation() member function returns a DtTaitBryan.
- The bodyToGeoc() member function returns a rotation matrix that will rotate from the body to geocentric coordinate system.

See "Coordinate Conversions," on page 9-7 for information about how to convert rotation matrices to Euler angles, and how to convert among the many different coordinate systems supported by VR-Link. In addition, "Coordinates," on page 5-18 provides more information about the coordinate systems mentioned above.

VR-Link also provides various View classes, described in "Coordinate Views," on page 5-37, which provide the ability to inspect data in a *DtEntityStateRepository* in other coordinate systems without explicitly performing coordinate conversions.

Some of the inspector functions return enumerations. Values for these enumerations are in *disEnums.h*. When a *DtBoolean* is returned, it will be either DtTrue or DtFalse.

The following example shows how to use *DtEntityStateRepository*'s inspector functions to examine the current state of a reflected entity, within a function that prints part of the state of the first entity in the list:

```
void printStateOfFirstEnt(DtReflectedEntityList* rel)
{
    // Grab a pointer to the first entity
    DtReflectedEntity* firstEnt = rel.first();
    // Make sure the list isn't empty
    if (!firstEnt)
    {
        return;
    }
    // Print the entity's ID
    printf("ID: %s\n", firstEnt->entityId().string());
    // Grab a pointer to the entity's ESR
    DtEntityStateRepository *esr = firstEnt->entityStateRep();
    // DtVector and DtTaitBryan's string member functions
    // return a string representation of their data
    printf("Loc: %s\n",    esr->location().string());
    printf("Vel: %s\n",    esr->velocity().string());
    printf("Accel: %s\n",  esr->acceleration().string());
    printf("Orient: %s\n", esr->orientation().string());
    printf("AngVel:%s\n",  esr->rotationalVelocity().string());
    if(esr->flamesPresent == DtTrue)
    {
        printf("Flaming!\n");
    }
```

```
        // The DtDamageState enumeration is in disEnums.h
        if (esr->damageState() == DtDamageDestroyed)
        {
            printf("Destroyed!\n");
        }
    }
```

DtEntityStateRepository::printData() works in a similar way, and you can use it to print the current contents of a *DtEntityStateRepository* in human readable form.

5.6.4. Dead-Reckoning

By default, the position, velocity and orientation returned by a *DtReflectedEntity*'s *DtEntityStateRepository*'s member functions (location(), velocity(), orientation(), and bodyToGeoc()), are dead-reckoned values. That is, they are not necessarily the values last received via state updates from the exercise. They are extrapolated forward to the current value of VR-Link simulation time from acceleration, velocity, and angular velocity based on the entity's current dead-reckoning algorithm.

The current value of VR-Link simulation time is the last value passed to *DtSetSimTime*, which should be called once per frame by an application. In this way, all entities are dead-reckoned to the same time within that frame, regardless of the order in which the locations are inspected. Time of validity of the inputs to the dead-reckoning equation is the value of VR-Link simulation time when the data was received from the exercise.

Notes:

- There is no performance penalty for calling the inspectors for dead-reckoned values more than once during a particular frame. As long as the value of VR-Link simulation time has not changed, the dead-reckoning code is not executed again; a cached value is used. Dead-reckoning is not performed for entities whose positions and orientations are never inspected.
- Dead-reckoning applies to entities and aggregates, but not to the other objects.

! Dead-reckoning of orientation as a set of Euler angles is a more expensive operation than dead-reckoning orientation as a rotation matrix. So use bodyToGeoc() rather than orientation() in cases where performance is an issue and you do not have an explicit need for the Euler angle representation.

For non-dead-reckoned values, use the values returned by *DtEntityStateRepository's* functions lastSetLocation(), lastSetVelocity(), and so on.

Dead-Reckoning Details

A *DtEntityStateRepository* uses a *DtDeadReckoner* to perform dead-reckoning calculations, or other types of extrapolation of position and orientation. When *DtEntityStateRepository's* mutator functions are used to set any position-related values, those values are passed to the *DtDeadReckoner's* mutators. Then, when *DtEntityStateRepository*'s inspectors are used to ask for current values, the *DtEntityStateRepository* obtains extrapolated values from the dead-reckoner using its inspectors.

If you would like to change the way dead-reckoning is handled by a *DtEntityStateRepository*, you can create a subclass of *DtDeadReckoner*, override the functions you need, and tell a *DtEntityStateRepository* to use an instance of your subclass through its useDeadReckoner() function (which is inherited from *DtBaseEntityStateRepository*).

Similarly, if you would like a *DtEntityStateRepository* not to perform dead-reckoning at all, pass NULL to useDeadReckoner(). In fact, this is what a *DtEntityPublisher* does with its *DtEntityStateRepository*, since you typically do not want dead-reckoned values when inspecting the *DtEntityStateRepository* of a locally simulated entity.

To restore the *DtEntityStateRepository*'s original default dead-reckoner, call useDeadReckoner() with no arguments.

5.6.5. Using Smoothing

VR-Link can smooth out the jumps in entity position that would otherwise occur when new HLA or DIS state data arrives. The *DtSmoother* class (defined in *smoother.h*) implements this functionality. *DtSmoother* is derived from *DtDeadReckoner*, so a *DtSmoother* can be used by *DtReflectedEntity*'s *DtEntityStateRepository* as its deadReckoner. If a *DtEntityStateRepository* is using a *DtSmoother*, the values returned by location(), velocity(), orientation() and bodyToGeoc() are smoothed values.

The *DtReflectedEntityList* constructors have an optional boolean argument indicating whether its reflected entities should use smoothers as their dead-reckoners or not (default is no). Alternatively, you can instruct an individual reflected entity's *DtEntityStateRepository* to use a *DtSmoother* using *DtEntityStateRepository*'s useSmoother() call.

To set the global default time over which smoothing takes place, use *DtSmoother*'s static function setDfltSmoothPeriod(). To override the default in individual *DtSmoothers*, use setSmoothPeriod().

Smoothing applies to entities and aggregates, but not to other objects.

Figure 3-3 illustrates smoothing.

5.6.6. Learning when Entities Join or Leave An Exercise

Often, an application will want to be notified when an entity joins or leaves the exercise. This is achieved by registering entity-addition and entity-removal callbacks with a *DtReflectedEntityList*. Entity-addition callbacks are called by VR-Link just after an entity is added to the *DtReflectedEntityList*.[1] Entity-removal callbacks are called just before an entity is removed from the *DtReflectedEntityList*. Both occur within DtExerciseConn::drainInput() when we receive news from the exercise that an entity has joined or left.

Entity-addition and entity-removal callbacks both must have the following function signature:

```
void func(DtReflectedEntity* ent, void* userData);
```

They are registered with a *DtReflectedEntityList* using its addEntityAdditionCallback() and addEntityRemovalCallback() functions. Similarly, these callbacks can be unregistered with removeEntityAdditionCallback() and removeEntityRemovalCallback().

In the following example, the application prints HELLO and GOODBYE when an entity comes or goes, along with the ID of the entity:

```
void printHello(DtReflectedEntity* ent, void* userData)
{
   printf("HELLO %s\n", ent->id().string());
}

void printGoodbye(DtReflectedEntity* ent, void* userData)
{
   printf("GOODBYE %s\n", ent->id().string());
}

main()
{
   ...
   rel->addEntityAdditionCallback(printHello, NULL);
   rel->addEntityRemovalCallback(printGoodbye, NULL);
   ...
}
```

An alternate method of receiving notification that an entity has joined or left the exercise is by subclassing *DtReflectedEntityList*, and overriding the virtual functions entityAdded() and removeAndDelete(). entityAdded() is called just after an entity is added to a *DtReflectedEntityList*,[1] while removeAndDelete() is called in order to remove an entity from the *DtReflectedEntityList*. If you do override removeAndDelete(), be sure to call down to the base version of this function from within your implementation, since this is what actually accomplishes the removal of the entity from the list.

1. An entity state repository is not filled out at this point in HLA.

HLA only

An entity is added to the *DtReflectedEntityList* as soon as VR-Link receives a discoverObject() service call from the RTI. This occurs before the first attribute update is presented to us by the RTI. For this reason, the *DtReflectedEntity's DtEntityStateRepository* will not contain any data at the time that the entity is passed to your entity addition callback or the overridden version of entityAdded(). Only the entity's ID will have been set at this point. Therefore, within your callback or entityAdded() you should only be doing things like saving a pointer to the entity, rather than trying to inspect any data.

Typically, the first update immediately follows the discoverObject() invocation for a new entity. Therefore, by the time drainInput() returns, most data will be valid for your application to inspect. If you want to be notified immediately after the first update arrives, see *"Notifying An Application When State Updates Arrive,"* on page 5-28.

Another consideration is that these callbacks as well as entityAdded() and removeAndDelete() are called from within RTI callbacks. RTI rules prohibit making RTI calls from within an RTI callback, so do not make any RTI calls (or call any functions that make RTI calls) from within entityAdded() or removeAndDelete().

DIS only

Entity addition callbacks and the virtual function entityAdded() are called after the first entity state PDU for the entity has been processed. Therefore, the state of the entity is available within entityAdded(), through the entity's *DtEntityStateRepository.* However in the interest of writing protocol-independent code (to take into account the HLA issues discussed in previous paragraphs), you might not want to inspect the data at this point in the code for DIS applications.

5.6.7. Notifying An Application When State Updates Arrive

Some applications might want to be notified when new state data is received from the application that is simulating a particular entity. You can do this by registering a post-update callback function with a *DtReflectedEntity.* These functions must have the following signature:

```
void func(DtReflectedEntity* ent, void* userData);
```

They are registered using *DtReflectedEntity's* addPostUpdateCallback(). Similarly, to unregister your callback function, use removePostUpdateCallback().

Post-update callbacks are called by VR-Link from within DtExerciseConn::drainInput() immediately after a state update message has been decoded into the reflected entity's *DtEntityStateRepository.* That is, when your callback is called, the *DtEntityStateRepository* will already reflect the new values contained in the update.

As an example of when you might want to use this functionality, suppose you have an application that needs to create different types of graphic icons to represent different types of entities. You would like to be notified when a new entity arrives, so that you can create a new icon, but entity type information is not available to an entity-addition callback in HLA. The way to accomplish this, is to register a post-update callback from within your entity-addition callback. When your post-update callback is invoked, you can check to see whether you have received a value for the entity type yet (since in HLA, that data might not arrive in the first update):

```
void myEntityAdditionCb(DtReflectedEntity* ent, void* usr)
{
        // A new entity has arrived, but its ESR is empty. Request
        //to be notified when an update has been processed.
   ent->addPostUpdateCb(myPostUpdateCb, usr);
}

void myPostUpdateCb(DtReflectedEntity* ent, void* usr)
{
   // A state update for the entity has just been processed,
        // but there's no guarantee that entity type was included.
   DtEntityStateRepository* esr = ent->esr();

   if (esr->entityType() != DtEntityType(0,0,0,0,0,0,0))
   {
        // We've received entity type info, and can now use it
        // to create an appropriate icon.
      addIcon(esr->entityType());

      // We probably no longer need the post-update callback.
      ent->removePostUpdateCb(myPostUpdateCb, usr);
   }
}

main()
{
   ...
   // Register the entity addition callback with a reflected
        // entity list.
   rel->addEntityAdditionCallback(myEntityAdditionCb, someObj);
   ...
}
```

Because the reception of state data works differently in DIS and HLA, and because we wanted the post-drain callback mechanism to work in a protocol-independent manner, you do not have access to the update message itself from within the post-drain callback.

If you want to actually intercept an incoming state update message in either HLA or DIS, please see the appropriate protocol-specific sections.

5.6.8. Timeouts

DtReflectedEntityList can time entities out, removing them from the list if an update has not been received within some period of time.

DIS only

This capability is on by default in DIS, where the rules state that an entity state PDU (heartbeat) must be sent periodically (usually once every 5 seconds), even if no data has changed. Therefore, if we have not received a heartbeat in a reasonable amount of time (usually 12 seconds), we can safely assume that the entity has unexpectedly left the exercise.

HLA only

In HLA, there is no heartbeat rule; it is perfectly valid for an entity to go several minutes or more without updating attributes, as long as nothing has changed. For this reason, timeouts are off by default. When timeout processing is off, a *DtReflectedEntity* is not removed from the *DtReflectedEntityList* until we are notified by the RTI that the entity has left the exercise.

You can control timeout processing on a per-reflected-entity-list basis. For both DIS and HLA, timeouts can be turned on and off using DtReflectedEntityList::setTimeout-Processing(). (This function is inherited from the *DtReflectedObjectList* base class.) The argument is a *DtBoolean*, either DtTrue or DtFalse.

If timeout processing is on, the *DtReflectedEntityList* checks to see if any entities need to be timed out each time DtExerciseConn::drainInput() is called. The timeout interval (the amount of time that can elapse since the last update before an entity is timed out) defaults to 12.0 seconds, but can be configured with DtReflectedObjectList::setTime-outInterval().

5.6.9. Subclassing *DtReflectedEntity*

Some visually-oriented applications might want to associate additional data or functionality with a *DtReflectedEntity.* One way to achieve this is by subclassing *DtReflected-Entity.* For example, you could use a subclass of *DtReflectedEntity* to associate graphics data with the entity.

If you subclass *DtReflectedEntity* (rather than associating data through composition, for example), you need to subclass *DtReflectedEntityList* as well. The reason for this is simple: *DtReflectedEntities* are created by the *DtReflectedEntityList*, and the *DtReflected-EntityList* only knows how to create *DtReflectedEntities*. You need to create a derived *DtReflectedEntityList* that knows how to create your derived *DtReflectedEntities.*

A *DtReflectedEntityList* uses the virtual function newReflectedEntity() to create *DtRe-flectedEntities*. It is this function that you must override with a definition that returns a new instance of your derived *DtReflectedEntity.*

The constructors for DIS and HLA versions of *DtReflectedEntity* take different arguments; therefore the two versions of DtReflectedEntityList::newReflectedEntity(), (which basically just passes its arguments to the *DtReflectedEntity* constructor) take different arguments as well.

The following example shows how to subclass the two classes:

```
class myReflectedEntity : public DtReflectedEntity
{
public:
    // Constructor
#if DtHLA
    myReflectedEntity(DtHlaObject* obj,
       DtExerciseConn* conn) :
        DtReflectedEntity(obj, conn)
#else
    myReflectedEntity(DtExerciseConn* conn,
        const DtEntityIdentifier& id,
        const DtEntityType& type) :
    DtReflectedEntity(conn, id, type)
#endif
    {
        // The two versions may be able to share a body
        ...
    }
    // Specifics of myReflectedEntity
    ...
};
class myREL : public DtReflectedEntityList
{
public:
    // Constructor (same for both DIS and HLA)
    myREL(DtExerciseConn* exConn) :
            DtReflectedEntityList(exConn) {}
#if DtHLA
    virtual DtReflectedEntity*
       newReflectedEntity(DtHlaObject* obj) const
    {
        return new myReflectedEntity(obj, exerciseConn());
    }
#else
    virtual DtReflectedEntity* newReflectedEntity(
        const DtEntityIdentifier& id,
        const DtEntityType& type) const
    {
        return new myReflectedEntity(exerciseConn(), id, type);
    }
#endif
};
```

5.7. Identifying Objects

Object identification is handled fairly differently in HLA and DIS.

5.7.1. Identifying Objects in DIS

In DIS, entities are identified by a triplet (site:application:entity) known as an Entity Identifier. This structure can be represented in VR-Link using the *DtEntityIdentifier* class, defined in *hostStructs.h*. Other, non-entity objects are identified in various ways. For example, emitter systems are identified using the Entity Identifier of the host entity plus an addition emitter number.

5.7.2. Identifying Objects in HLA

In HLA, an object can be identified in several ways:

- All objects have an object handle - an integer that an application uses to identify a particular object in RTI service calls. An object handle is meaningful only to a particular federate. The same object can be known to different federates by different object handles. Because of this, handles cannot be used within attribute updates or interaction messages that are exchanged among federates. If I say I'm attacking object #1, a remote federate would not know who I'm talking about. Object handles are assigned by the RTI when you register a locally simulated object with the RTI, or when a remote object is discovered by the RTI. They cannot be assigned by users. Object handles are represented by an RTI::ObjectHandle, which is typedefed to an unsigned long.
- All objects also have an object name - a character string that can be used to identify an object. The object name is known to the RTI, and the RTI provides functions to find out an object's name, given its handle, and vice versa. Object names can be chosen by applications that register the objects with the RTI, however if you do not want to choose names for objects, the RTI will assign names for you. Object names are represented using `char*`, or VR-Link's *DtString* class.
- FOMs may choose to define other ways of identifying particular kinds of objects. These other identifiers are merely attributes of the object, and do not have any special identifier characteristics as far as the RTI is concerned. For example, the *BaseEntity* class in the RPR FOM contains an attribute called EntityID that is represented by a DIS-style Entity Identifier triplet - the *DtEntityIdentifier* class in VR-Link.[1]

1. The RPR FOM chose to maintain the Entity Identifier attribute because it contains more information than just an object name or handle. It tells you the site and application that own the entity.

5.7.3. How VR-Link Identifies Objects

In the interests of protocol independence, VR-Link provides two types that can be useful in passing around the various types of identifiers.

A *DtObjectId* (or local ID) can be used regardless of protocol, to identify an object within an application, but not to communicate the identity of an object to other federates in PDUs, interactions, or attribute updates. In HLA, *DtObjectId* is a class that serves as a wrapper around an RTI::ObjectHandle. In DIS, *DtObjectId* is typedefed to *DtEntityIdentifier.*

A *DtGlobalObjectDesignator* (or global ID) can be used regardless of protocol to identify an entity in inter-application communication within PDUs, interactions, and attribute updates. In HLA, *DtGlobalObjectDesignator* is typedefed to *DtString* - a VR-Link wrapper around `char*` that can be used to store object names. In DIS, *DtGlobalObjectDesignator* is still typedefed to *DtEntityIdentifier*, since in DIS that same type of ID is used to identify an object both within an application, and between applications.

Reflected Object Lists allow you to look up reflected objects by local ID or by global ID. *DtObjectPublishers* and *DtReflectedObjects* have functions to return both the object's local ID (id() or objectId()) and global ID (globalId()). PDU, interaction, and state repository classes have mutator functions that expect global IDs and inspector functions that return global IDs.

For example, if you have a *DtEntityPublisher* representing an entity that you are simulating, and you are sending a fire interaction, indicating that your entity is firing a munition, you might say:

```
fireInter.setAttackerId(entityPub.globalId());
```

If you receive a fire interaction from a particular entity, and would like to find out more about its state, you can look up the entity in a reflected entity list like this:

```
DtReflectedEntity* ent = rel.lookup(fireInter.attackerId());
```

When constructing an HLA publisher, you can either choose a name for your object, or pass a NULL name (the default), and let the RTI choose for you. See "Working with Locally Simulated Entities," on page 5-14 for examples.

5.8. Working with Other Types of Objects

As we mentioned in "Working with Entities," on page 5-14, much of the discussion about managing entities applies to managing other types of objects. But rather than using *DtEntityPublisher*, *DtReflectedEntityList*, *DtReflectedEntity* and *DtEntityStateRepository*, you use the other subclasses of *DtObjectPublisher*, *DtReflectedObjectList*, *DtReflectedObject* and *DtStateRepository* to manage the appropriate type of objects, as described in this section. To use any of the following objects, read the description for managing an entity and substitute the appropriate object class for those used in the descriptions and examples:

- aggregate
  - *DtAggregatePublisher* (*hAggPub.h*)
  - *DtReflectedAggregateList* (*hReflAggList.h*)
  - *DtReflectedAggreate* (*hReflectedAgg.h*)
  - *DtAggregateStateRepository* (*aggregateSR.h*)
- designator
  - *DtDesignatorPublisher* (*hDesignatrPub.h*)
  - *DtReflectedDesignatorList* (*hRefDesigList.h*)
  - *DtReflectedDesignator* (*hReflDesigntr.h*)
  - *DtDesignatorRepository* (*designatorSR.h*)
- emitter
  - *DtEmitterSystemPublisher* (*hEmittrSysPub.h*)
  - *DtReflectedEmitterSystemList* (*hRefEmSysList.h*)
  - *DtReflectedEmitterSystem* (*hReflEmitrSys.h*)
  - *DtEmitterSystemRepository* (*emitterSysSR.h*)
- radio receiver
  - *DtRadioReceiverPublisher* (*hRadioRcvrPub.h*)
  - *DtReflectedRadioReceiverList* (*hRefRadRvList.h*)
  - *DtReflectedRadioReceiver* (*hReflRadRecvr.h*)
  - *DtRadioReceiverRepository* (*radioRecvrSR.h*)
- radio transmitter
  - *DtRadioTransmitterPublisher* (*hRadioXmitPub.h*)
  - *DtReflectedRadioTransmitterList* (*hRefRadXmList.h*)
  - *DtReflectedRadioTransmitter* (*hReflRadXmitr.h*)
  - *DtRadioTransmitterRepository* (*radioXmitSR.h*).

Each kind of state repository has accessors for the data associated with the corresponding kind of object, and each kind of publisher and reflected object use the appropriate kind of *DtStateRepository.* Each kind of reflected object list manages instances of the appropriate kind of reflected object.

5.8.1. Managing Emitters

Emitters are a bit of a special case, particularly since the RPR FOM implements emitter systems and their constituent emitter beams using two different FOM object classes. We have tried to hide this FOM detail as much as possible, so that we can work with FOMs that do not break things up this way, and with DIS.

VR-Link contains *DtEmitterBeamPublisher, DtReflectedEmitterBeamList*, and *DtReflectedEmitterBeam* classes to manage individual emitter beam objects, but application code normally should not use these directly. Instead, instances of these classes are used by the corresponding emitter system classes to manage a system's component beams, and you can get or provide information about a system's beams through the classes that manage emitter system information.

The state of an emitter beam is represented with a *DtEmitterBeamRepository* (*emitterBeamSR.h*), a class that you will need to use. A *DtEmitterSystemRepository* consists of some system wide state, then a list of instances of *DtEmitterBeamRepository* - one for each constituent beam.

When publishing an emitter system, create the *DtEmitterSystemPublisher*, obtain a pointer to its *DtEmitterSystemRepository* using its emitterSystemRep() or esr() member. Then add beams using *DtEmitterSystemRepository*'s addBeam() function. A *DtEmitterBeamRepository* to be used to set the state of the new beam is returned by addBeam(). After adding a beam to a system repository, make sure to set the beam's parent system ID to your system's ID using *DtEmitterBeamRepository*'s setEmitterSystemId().

You can remove beams using *DtEmitterSystemRepository*'s removeBeam() function. The list of all beam's repositories is available through *DtEmitterSystemRepository*'s beamList() member.

Here's an example:

```
// Create the system publisher
DtEmitterSystemPublisher sysPub(&conn);

// Get a pointer to its system state repository
DtEmitterSystemRepository* esr = sysPub.esr();

// Add a beam, and keep a pointer to its beam repository
// We'll give the beam a beam ID of 10.
DtEmitterBeamRepository* bsr = esr->addBeam(10);

// Set the beam's parent system ID
bsr->setEmittingSystemId(sysPub.globalId());

while (...)
{
   // Each frame, you you can set various attributes of the system's
```

```
   // state using esr, and of the beam's state using bsr.
   ...

   // Then just tick the system publisher.  This will cause any
   // necessary system and beam data to be sent.
   sysPub.tick();
   ...
}
```

On the receiving side, you can create an instance of *DtReflectedEmitterSystemList*, and iterate through or look up systems as with any other type of reflected object list.

When you inspect a *DtEmitterSystemRepository*, it contains a list of *DtEmitterBeamRepositories* for the system's beams, available through beamList(). Because beams are separate objects in the RPR FOM, beams can come and go while the system exists in HLA. If you've saved a pointer to a *DtEmitterBeamRepository*, you can make sure that beam is still valid by looking it up in the system's list using DtEmitterBeamRepository::isMember().

Here's an example:

```
DtReflectedEmitterSystemList sysList(&conn);

while (...)
{
   ...

   // Get a pointer to the first system in the list.
   DtReflectedEmitterSystem* sys = sysList.first();

   // Get a pointer to the system's state repository.
   DtEmitterSystemRepository* esr = sys->esr();

   // Get a pointer to the system's first beam's state repository
   DtEmitterBeamRepository* bsr =
      (DtEmitterBeamRepository*) esr->beamList()->first();

   // Now you can use esr and bsr to inspect the state of the system
   // and its first beam.
}
```

If VR-Link obtains HLA updates for a beam without having first discovered its parent system, *DtReflectedEmitterSystemList* creates a phantom system with the object name indicated by the beam's **emittingSystemId** attribute, and adds a beam state repository representing the beam to that phantom system's repository. If we later discover the system object, its updates are decoded directly into the existing system repository.

5.9. Coordinate Views

DtEntityStateRepository's mutators and inspectors expect and return position and orientation and their derivatives in geocentric coordinates. If you would like to use a different coordinate system in your application, you can use VR-Link's explicit coordinate conversion routines to convert outgoing data to a geocentric form before setting values in a *DtEntityPublisher*'s *DtEntityStateRepository*, and to convert incoming data obtained from a *DtReflectedEntity*'s *DtEntityStateRepository* from geocentric to your coordinate system of choice. However, there is an easier way.

VR-Link provides several View classes that can be created on a *DtEntityStateRepository*, providing access to the data in a coordinate system other than geocentric. After creating the appropriate view, simply use its mutators and inspectors, rather than directly calling the *DtEntityStateRepository*'s functions. The View's functions call down to the *DtEntityStateRepository*'s functions before or after doing the conversions, so that the *DtEntityStateRepository* always stores geocentric data.

VR-Link has three View classes:

- *DtUtmView* supports UTM coordinates.
- *DtCartesianView* supports any arbitrary Cartesian coordinate system.
- *DtTopoView* is derived from *DtCartesianView* and supports topographic coordinate systems (Cartesian systems that have their X-Y plane tangent to the Earth's surface at their origin, with the X axis pointing north, the Y axis pointing east, and the Z axis pointing down).

These classes are defined in *utmView.h*, *cartView.h*, and *topoView.h*. See "Coordinate Conversions," on page 9-7 for more information about these coordinate systems.

The *DtEntityStateRepository* inspector and mutator functions that can be circumvented using Views are:

- location()
- setLocation()
- velocity()
- setVelocity()
- acceleration()
- setAcceleration()
- orientation()
- setOrientation()
- bodyToGeoc()
- rotationalVelocity()
- setRotationalVelocity().

However, since rotationalVelocity() is always specified in body rather than geocentric coordinates, the Views rotationalVelocity() and setRotationalVelocity() functions just pass the *DtEntityStateRepository* values through without any conversion.

5.9.1. Topographic Coordinate Views

The *DtTopoView* constructor takes a pointer to the *DtEntityStateRepository* that we will be accessing, and a latitude and longitude (in radians) that define the particular topographic coordinate system we'd like to work in, for example:

```
DtEntityPublisher entPub(...);
DtEntityStateRepository* esr = entPub.entityStateRep();
DtTopoView topoView(esr, DtDeg2Rad(36.0), DtDeg2Rad(-121.0));
```

You can use *DtTopoView*'s mutators to set positions and orientations specified with respect to a topographic coordinate system whose origin is at a lat/long of {36.0, -121.0}.

Topographic Euler angles correspond to heading, pitch and roll.

```
// Topographic coordinates
topoView.setLocation(DtVector(100.0, 100.0, 0.0));
topoView.setVelocity(DtVector(10.0, 10.0, 0.0));
topoView.setAcceleration(DtVector(1.0, 1.0, 0.0));
// Topographic Euler angles - heading, pitch and roll
topoView.setOrientation(DtTaitBryan(0.0, DtDeg2Rad(10.0), 0.0);
// Rotational velocity is always in body coordinates
topoView.setRotationalVelocity(DtVector(0.0, 0.10, -0.125);
```

Going the other way, we see how a *DtTopoView* is used to inspect a remote entity's data in topographic coordinates:

```
DtReflectedEntityList rel(...);
...
DtReflectedEntity* ent = rel.first();
DtEntityStateRepository* esr = ent->entityStateRep();
DtTopoView topoView(esr, DtDeg2Rad(36.0), DtDeg2Rad(-121.0));
```

We can now use *DtTopoView*'s inspectors to get positions and orientations specified with respect to the topographic coordinate system.

```
// Topographic coordinates
DtVector topoLoc = topoView.location();
DtVector topoVel = topoView.velocity();
DtVector topoAccel = topoView.acceleration();
// Topographic Euler angles - heading, pitch and roll
DtTaitBryan topoOrient = topoView.orientation();
// Or choose a matrix representation of orientation
DtDcm bodyToLocal = topoView.bodyToLocal();
// Rotational velocity is always in body coordinates
DtVector angVel = topoView.rotationalVelocity();
```

5.9.2. UTM Coordinate Views

DtUtmView is a View that allows you to work in UTM coordinates. Its constructor takes only a *DtEntityStateRepository*, but relies on the fact that you've already initialized VR-Link's UTM coordinate system using *DtUtmInit*. (See "Coordinate Conversions," on page 9-7). Once you've chosen a particular UTM coordinate system in this way, you can use the inspectors and mutators like you do with *DtTopoView*.

Locations and their derivatives are in UTM coordinates, and are expressed as *DtUtmCoords*, defined in *utmCoord.h*. UTM orientations are treated in the same way as topographic orientations. The three Euler angles represent heading, pitch and roll, while bodyToLocal() returns a rotation matrix that can rotate a vector from the body frame to a topographic frame centered at the entity's current location. As always, rotational velocity is represented in body coordinates.

5.9.3. Cartesian Coordinate Views

DtCartesianView allows the creation of a view defined by any Cartesian coordinate system. Along with a *DtEntityStateRepository*, its constructor takes a *DtVector* indicating the origin of the new system, expressed in geocentric coordinates, and a *DtDcm* representing a rotation matrix that can rotate a vector from the geocentric frame to the frame we are defining.

Alternatively, you can use the constructor that takes only a *DtEntityStateRepository*, and initialize the View by passing the two parameters to the setOffset() and setRotation() member functions.

Since *DtTopoView*, inherits its inspectors and mutators from *DtCartesianView*, the descriptions of these functions in "Topographic Coordinate Views," on page 5-38 applies here, except that Euler angles do not correspond to heading, pitch and roll, when expressed with respect to an arbitrary Cartesian system.

5.10. Articulated Parts

DIS and DIS-based FOMs like the RPR FOM define articulated parts as movable parts of an entity, such as a turret, gun, or landing gear. The state of an entity's articulated parts is part of an entity's state, and is accessible through a *DtEntityStateRepository*.

A *DtEntityStateRepository* uses a *DtArtPartList*, defined in *artPartList.h*, to store the articulated parts portion of its state. You can obtain a pointer to a *DtEntityStateRepository*'s *DtArtPartList* with its artPartList() member function.

5.10.1. Inspecting Articulated Parts Data

DtArtPartList provides many inspector functions for examining the state of an entity's articulated parts. The numArtParts() member function returns the number of articulated parts the entity has. For example, if a tank had a turret and a gun, numArtParts() returns 2. The numArtParams() member function returns the number of articulation parameters the entity is using to convey articulated parts information to the exercise. For example, if the tank is publishing information on its turret's azimuth and azimuth rate, as well as its gun's elevation, numArtParams() returns 3.

DtArtPartList::getArtParamVal() allows you to query the list for the value of a particular articulation parameter. The part type should be a member of the DtArtPartType enumeration, while the parameter type should be a member of the DtArtParamType enumeration, both defined in *disEnums.h*. The **val** argument is the variable you would like filled with the value being inspected, passed by reference. The function returns DtTrue if successful, and DtFalse if the part-type/param-type combination is not valid for the entity.

If a rate for the parameter in question has been included in the parameter list for the entity, the value returned is dead-reckoned to the current value of VR-Link simulation time. See "Disabling Dead-Reckoning," on page 5-41.

For example, here's how to inspect the value of the turret azimuth of a particular entity:

```
DtReflectedEntityList rel(...);
....
DtReflectedEntity* firstEnt = rel.first();
DtEntityStateRepository* esr = firstEnt->entityStateRep();
DtArtPartList* artParts = esr->artPartsList();
float turAz = 0.0;
if (artParts->getArtParamVal(DtPrimaryTurret1, DtApAzimuth,
        turAz) != DtTrue)
{
    printf("Entity has no turret azimuth.\n");
}
```

To obtain more detailed information about articulated parts, including obtaining non-dead-reckoned values, you must look at the individual *DtArtPart* objects that a *DtArtPartList* uses to represent individual parts. *DtArtPart* is defined in *artPart.h*.

The *DtArtPart* Class

You can obtain a pointer to a particular *DtArtPart* by part type or by index (in the range of 1 to numArtParts), through the *DtArtPartList* functions artPartFromPartType() or artPartFromPartNum(). Both return NULL if there is no part identified by the given part type or part number.

You can iterate through the list of *DtArtParts* as follows:

```
DtArtPartList *parts = ...;
for (int i = 1; DtArtPart* part =
           parts->artPartFromPartNum(i); i++)
```

A *DtArtPart*'s part type (a member of the DtArtPartType enumeration) and part number (its index in a part list) can be obtained with partType() and partNum(). partAttachedTo() returns the type of the part to which the current part is attached. For example, a gun is often attached to a turret, whereas a turret is usually attached to the base entity (indicated by a return value of DtAttachedToBase).

The number of parameters used for this part is found using numParams(), while firstValidParamType() and nextValidParamType() are used to iterate through the set of parameters as follows:

```
DtArtPart* part = ...;
for (int param = part->firstValidParamType();
    param;
    param = part->nextValidParamType(param));
```

DtArtPart::getVal() lets you inspect the value of a parameter by parameter type, filling in a variable passed by reference, and returning DtFalse if the parameter is not valid for the part. DtArtPart::val() doesn't do the validity checking, so should be called only when you know the parameter is valid.

Disabling Dead-Reckoning

By default, values returned by these functions are dead-reckoned to the current time if the parameter's rate is also a valid parameter for the part. However, dead-reckoning can be disabled using the setApproximating() member function, passing DtFalse as an argument.

If, rather than inspect individual parameters, you would like to obtain the state of the part as a whole, expressed as a vector of X, Y, and Z translations, or as a set of three angles representing azimuth, elevation, and rotation, then use the functions translations() and angles().

Both *DtArtPart* and *DtArtPartList* have print() functions that print their data in human readable form.

5.10.2. Setting Articulated Parts Data

To set articulated parts data:

1. Use *DtEntityStateRepository*'s artPartList() function to access a pointer to a *DtEntityPublisher*'s *DtEntityStateRepository*'s articulated part list.
2. Use *DtArtPartList* and *DtArtPart* mutator functions to set current data for an entity you are simulating.

A *DtArtPartList* is initialized with the function init(), which takes an array of *DtArtPartSpecs*, or articulated part specifications. Before you try to set values for any of the parameters, you must call init() to indicate what parts and parameters are valid for the entity.

A *DtArtPartSpec* has the following definition, which is in *apSpec.h*:

```
typedef struct DtArtPartSpec
{
    int partType;
    int attachedType;
    int params[DtMAX_PARAMS_PER_PART];
} DtArtPartSpec;
```

where:

- **partType** should be a member of the DtArtPartType enumeration found in *disEnums.h*, and should specify the type of the part.
- **attachedType** is the **partType** of the part to which the articulated part is attached. If the part is attached to the base entity, rather than to another part, the attached type should be DtAttachedToBase. Figure 5-1 illustrates the relationship of **partType** and **attachedType** for an articulated part. It corresponds to the example on page 5-43.
- **params** is a zero-terminated array indicating which parameters will be used to communicate information about this part to the exercise. Elements of the array should be members of the DtArtParamType enumeration.

The array should have:

- One *DtArtPartSpec* for each of the entity's articulated parts for which you will be communicating information to the exercise
- A terminating *DtArtSpec* that has a part type of zero.

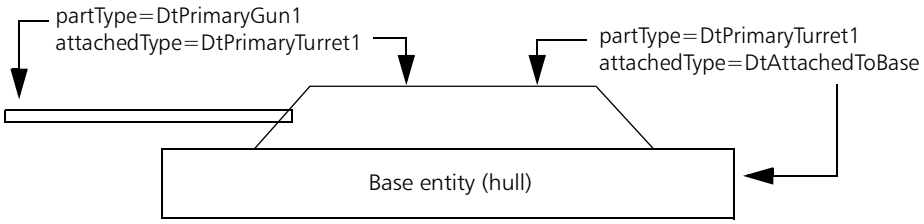


Figure 5-1. Illustration of partType and attachedType

The following example shows how to statically create an array of *DtArtPartSpecs* for an entity with a turret and a gun, and use it to initialize a *DtArtPartList*. We will use two parameters (azimuth and azimuth rate) for the turret, and one parameter (elevation) for the gun. Note the use of the DtNullArtPartSpec macro to create the terminating NULL *DtArtPartSpec*.

```
DtArtPartSpec* specList =
{
    {
        DtPrimaryTurret1,
        DtAttachedToBase,
        {DtApAzimuth, DtApAzimuthRate, 0}
    },
    {
        DtPrimaryGun1,
        DtPrimaryTurret1,
        {DtApElevation, 0}
    },
    {
    DtNullArtPartSpec
    }
};
DtEntityPublisher entityPub(...);
DtEntityStateRepository* esr = entityPub.entityStateRep();
DtArtPartList* artParts = esr->artPartList();
artParts->init(specList);
```

An equivalent way to initialize a *DtEntityStateRepository*'s articulated parts is to pass the array of *DtArtPartSpecs* to DtEntityStateRepository::initArtParts().

Once the *DtArtPartList* is initialized you can use its setArtParamVal() member function to set values for the parameters. This function works like its complementary inspector artParamVal():

```
artParts->setArtParamVal(DtPrimaryTurret1, DtApAzimiuth,
           DtDeg2Rad(90.0));
```

Alternatively, you can obtain a pointer to an individual *DtArtPart*, and use the setVal() function:

```
DtArtPart* turret = artParts->artPartFromPartType(DtPrimaryTurret1);
if (turret)
{
    turret->setVal(DtApAzimuth, DtDeg2Rad(90.0));
}
```

If you want to change the set of parts or parameters used for the entity, call init() again with a new array of *DtArtPartSpecs*.

5.11. Attached Parts

An attached part is an entity that is attached to another entity. For example, a missile could be attached to a launcher. *DtEntityStateRepository* has a member function, attPartList(), that returns a pointer to a protocol-independent *DtAttachedPartList* (defined in *attPartList.h*.) You can use *DtAttachedPartList*'s inspectors to find out the current state of a reflected entity's attached parts. You can use its mutators to set the state of a locally simulated entity's attached parts.

For example,

```
// Create the publisher, and get its ESR's attached parts list
DtEntityPublisher pub(...);
DtEntityStateRepository* esr = pub.esr();
DtAttachedPartList* attList = esr->attPartList();

// Add a single attached part, a sidewinder missile attached to
// station number 15.  (Station numbers are model-specific).
attList->setNumAttachedParts(1);
attList->setStationNumber(0, 15);
attList->setEntityTypeAtStation(15, DtEntityType(2:1:225:1:1:0:0));
```

5.12. Controlling MÄK Products Remotely

VR-Link has two sets of classes that allow you to control other MÄK applications. View Control messages allow you to control the view state of the MÄK Stealth. Logger Control messages allow you to control the behavior of the MÄK Logger.

5.12.1. Controlling the MÄK Stealth

You can control the view state of the MÄK Stealth with View Control Interactions in HLA and View Control PDUs in DIS. For example, use *DtMimicVCPdu* to instruct the Stealth to change its view state to mimic mode.

DIS only

For DIS, VR-Link defines 10 types of View Control PDUs that the MÄK Stealth accepts - one each for nine of its View Modes, and one to indicate a change of attached entity or view group without setting the current mode. (Each view mode requires different parameters to describe the view state).

The View Control PDU classes are all derived from *DtViewControlPdu* (defined in *vcPdu.h)*, which is in turn derived from *DtPdu*. The subclasses are:

- *DtAbsoluteVCPdu*
- *DtTetherVCPdu*
- *DtCompassVCPdu*
- *DtMimicVCPdu*
- *DtOrbitVCPdu*
- *DtTrackVCPdu*
- *DtTetherTrackVCPdu*
- *DtMimicTrackVCPdu*
- *DtGroupVCPdu*
- *DtModelessVCPdu*.

HLA only

For HLA, we have defined an HLA interaction class in our *VR-Link.fed* file called *ViewControl*. It contains a single parameter called **ViewControlPdu**. The data representation of this parameter is the same as the network representation of our DIS View Control PDUs.

The HLA class *DtViewControlInteraction* (defined in *hVcInter.h*) represents this HLA interaction.

Sending View Control Messages

To send a View Control message in DIS:

1. Create and fill out an instance of the appropriate View Control PDU class.
2. Send it like any other PDU, using *DtExerciseConn*'s sendStamped() function.

To send a View Control message in HLA:

1. Create and fill out the View Control PDU instance just like in DIS.
2. Pass it to *DtViewControlInteraction*'s setVcPdu() member function.
3. Send the interaction using sendStamped().

For example, to send a PDU telling the Stealth to enter absolute mode and go to the indicated location and orientation (in geocentric coordinates):

```
DtAbsoluteVcPdu vcPdu;
pdu.setLoc(DtVector(-960122.075, 5445122.868, 3170873.735));
pdu.setOrientation(DtTaitBryan(0.1, 0.2, 0.3));

#if DtHLA
DtViewControlInteraction inter;
inter.setVcPdu(vcPdu);
exerciseConn.sendStamped(inter);
#else
exerciseConn.sendStamped(vcPdu);
#endif
```

The example uses conditional compilation.

5.12.2. Controlling the MÄK Logger

You can control many of the features of the MÄK Logger remotely via Logger Control Interactions in HLA and Logger Control PDUs in DIS.

DIS only

For DIS, VR-Link defines 17 types of Logger Control PDUs - one for each Logger feature that can be activated remotely. The various Logger Control PDU classes are all derived from *DtLgrControlPdu* (defined in *lcPdu.h*), which is in turn derived from DtPdu. The subclasses are all defined in header files that begin with "lc". The classes are:

- *DtLgrEndPdu*
- *DtLgrEofActionPdu*
- *DtLgrHeartbeatPdu*
- *DtLgrPausePdu*
- *DtLgrPlayPdu*
- *DtLgrPlayActionPdu*
- *DtLgrPbFilePdu*
- *DtLgrQuitPdu*
- *DtLgrRecordActionPdu*
- *DtLgrRecFilePdu*
- *DtLgrRecordPdu*
- *DtLgrSetTimePdu*
- *DtLgrSkipTimePdu*
- *DtLgrStartPdu*
- *DtLgrStopPdu*
- *DtLgrTimeScalePdu*
- *DtLgrTimeoutPdu.*

HLA only

For HLA, we have defined an HLA interaction class in our *VR-Link.fed* file called *LgrControl*. It contains a single parameter called **LgrControlPdu**. The data representation of this parameter is the same as the network representation of our DIS Logger Control PDUs.

The HLA class *DtLgrControlInteraction* (defined in *hLcInter.h*) represents this HLA interaction.

Sending Logger Control Messages

To send a Logger Control message in DIS:

1. Create and fill out an instance of the appropriate Logger Control PDU class.
2. Send it like any other PDU, using *DtExerciseConn's* sendStamped() function.

To send a Logger Control message in HLA:

1. Create and fill out the Logger Control PDU instance just like in DIS.
2. Pass it to *DtLgrControlInteraction's* setLcPdu() member function.
3. Send the interaction using sendStamped().

For example, to send a PDU telling the Logger to jump to a point 10 seconds into the current logger file:

```
DtLgrSetTimePdu lcPdu;
lcPdu.setTime(10.0);

#if DtHLA
DtLgrControlInteraction inter;
inter.setLcPdu(lcPdu);
exerciseConn.sendStamped(inter);
#else
exerciseConn.sendStamped(lcPdu);
#endif
#endif
```

The example uses conditional compilation.

The HLA-Specific Interface

This chapter describes VR-Link's HLA-specific parameters and features:

If your application is intended for both DIS and HLA, or if you plan to add this support in the future, enclose your HLA-specific code within a preprocessor directive pair to ensure that it will not be compiled when you try to build for DIS:

```
#if DtHLA
   // HLA-specific code
   #endif
```

6.1. Interacting Directly With The RTI

Although most applications based on VR-Link will never need to access the RTI directly (VR-Link functions make the RTI calls for you), some applications will want the flexibility of making their own calls to the RTI. Often, you will want to mix and match, letting VR-Link handle most interaction with the RTI, while handling some of it directly within the same application.

The RTI's API contains two main classes: the *RTIambassador* and the *FederateAmbassador*. When a *DtExerciseConn* is constructed, it creates instances of both of these classes.

6.1.1. Federate-Initiated Services

Federate-initiated services, such as publishObjectClass(), updateAttributeValues(), and sendInteraction(), are invoked by making calls to the RTI ambassador's member functions.

A *DtExerciseConn*'s RTI ambassador is available through the rtiAmb() member function. Use this object to invoke any federate-initiated RTI services. The object returned by rtiAmb() is an instance of the class *DtVrlRtiAmbassador*, which serves as a wrapper around RTI::RTIambassador. (*DtVrlRtiAmbassador* has the same member functions as RTI::RTIambassador, but the functions are declared virtual, so that they can be overwritten by application code.)

For example, to call the RTI's requestPause() service:

```
DtExerciseConn exConn(...);
...
DtVrlRtiAmbassador* rtiAmb = exConn.rtiAmb();
rtiAmb.requestPause(...);
```

See the RTI documentation provided by DMSO for information about the RTI's functions.

6.1.2. RTI-Initiated Services

RTI-initiated services, such as discoverObject(), reflectAttributeValues(), and receiveInteraction(), are invoked when the RTI calls virtual member functions of RTI::FederateAmbassador() from within RTI::RTIambassador::tick(). Federates are responsible for deriving a class from *RTI::FederateAmbassador* and providing definitions for these pure virtual functions.

Normally, this responsibility is satisfied through VR-Link's *DtVrlFederateAmbassador* class, defined in *vrlFedAmb.h*. This class is derived from *RTI::FederateAmbassador*, and includes definitions for all virtual functions (though some are empty definitions).

Subclassing *DtVrlFederateAmbassador*

For direct access to the data the RTI provides through RTI-initiated services, you can derive your own class from *DtVrlFederateAmbassador*, and implement new definitions for the virtual functions in your subclass. In general, you should call the *DtVrlFederateAmbassador* versions of these functions from your versions, to insure that any VR-Link functionality that may depend on this continues to work.

The following example shows a subclass of *DtVrlFederateAmbassador*'s that uses the RTI-initiated service initiatePause. The implementation of *DtVrlFederateAmbassador*'s initiatePause() member function is empty. VR-Link does not take any action by default when this service is invoked by the RTI. In order for your application to be able to handle this service invocation, you can provide your own definition for the initiatePause() virtual function.

```
class MyFedAmb : public DtVrlFederateAmbassador
{
public:
    virtual void initiatePause(const RTI::PauseLabel label)
        throw (RTI::FederateAlreadyPaused,
               RTI::FederateInternalError);
};
void MyFedAmb::initiatePause(const RTI::PauseLabel label)
        throw (RTI::FederateAlreadyPaused,
               RTI::FederateInternalError)
{
    // Your code to handle the pause
    ...
    // Call superclass version in case it's
    // doing anything important
    DtVrlFederateAmbassador::initiatePause(label);
}
```

Telling *DtExerciseConn* About Your Derived Class

After you create a subclass of *DtVrlFederateAmbassador*, you need to tell *DtExerciseConn* that it should use an instance of your class rather than the default *DtVrlFederateAmbassador* as its federate ambassador. To tell *DtExerciseConn* about your subclass, use the static member function DtExerciseConn::setFedAmbCreator(). This function expects a DtFedAmbCreator() function as an argument – a function that returns a new instance of a *DtVrlFederateAmbassador* or a subclass of it. The current DtFedAmbCreator() is called in the *DtExerciseConn* constructor, so you need to call setFedAmbCreator() before the *DtExerciseConn* is constructed.

A FedAmbCreator() function for our *MyFedAmb* class might look like this:

```
DtVrlFederateAmbassador* MyFedAmbCreator()
{
    return new MyFedAmb();
}
```

Tell the *DtExerciseConn* class about this function before creating a *DtExerciseConn* instance as follows:

```
DtExerciseConn::setFedAmbCreator(MyFedAmbCreator);
```

```
...
// The DtExerciseConn we create here will use a MyFedAmb as
// its federate ambassador
DtExerciseConn exConn(...);
```

A *DtExerciseConn*'s *DtVrlFederateAmbassador* is available through its fedAmb() member function.

6.2. Getting Information About The FOM

When an HLA *DtExerciseConn* is constructed, VR-Link reads the appropriate FED file, and builds a database of information about the FOM that the application code can query. This FOM information is stored in a *DtFom* (defined in *fom.h*). You can obtain a pointer to a *DtExerciseConn*'s *DtFom* using the member function fom().

A *DtFom* contains a list of object class and interaction class descriptors. These descriptors are represented by the VR-Link classes *DtObjClassDesc* (defined in *objClassDesc.h*) and *DtInterClassDesc* (defined in *interClassDesc.h*) respectively. Class descriptors contain information about FOM classes, including:

- Class handle and name
- A pointer to the base class's descriptor (if any)
- The set of valid attributes or parameters
- Whether the class has been published or subscribed
- The attributes of an object class that have been published or subscribed
- Whether the class and its various attributes are needed by other federates.

The class descriptors also allow you to subscribe and publish, if you would like to override the default way these are handled.

DtFom's functions interClassByName(), interClassByHandle(), objClassByName() and objClassByHandle() allow you to obtain the class descriptor for a particular class. Alternatively, you can iterate through all of the classes in a FOM, using *DtFom*'s firstObjClass(), and nextObjClass(), or firstInterClass() and nextInterClass(). For example:

```
// Print out the names of all object classes in the current fom.
DtExerciseConn conn(...);
...
DtFom* fom = conn.fom();
for (DtObjClassDesc* desc = fom->firstObjClass();
   desc; desc = fom->nextObjClass(desc))
{
   printf("ClassName: %s\n", desc->name());
}
```

DtFom has a print() member function that is similar to the example, but it prints out more information, such as attribute and parameter names and handles.

6.3. Publishing and Subscribing to FOM Classes and Attributes

Before they send any data, HLA federates must tell the RTI the set of FOM classes and the set of attributes of object classes for which it is capable of sending data. The HLA specification refers to this initialization activity as "publishing" a class, or "publishing" a class with a set of attributes. (Though some people refer to the act of sending interactions or sending attribute value updates as publishing, that is not the meaning of the word as defined by the RTI or HLA specification.)

Similarly, an HLA federate must indicate to the RTI the set of FOM classes and the set of attributes of object classes for which it is interested in receiving data from other federates. This initialization activity is called "subscribing".

Most VR-Link applications do not need to do anything special to fulfill the publishing and subscribing requirement; VR-Link takes care of this for you. However, if you want to alter the set of classes or attributes that get subscribed to or published, you can do so through VR-Link.

6.3.1. Publishing Classes and Attributes

When you create an instance of a particular kind of object publisher (for example, a *DtEntityPublisher*), VR-Link determines which object class from the FOM should be used to represent the object (typically by either using the class handle passed to the publisher's constructor, or by asking the FOM Mapper). Before registering the new HLA object with the RTI, we publish the object class if the federate has not already published the class or explicitly unpublished it. By default, the class's full set of valid attributes is published.

To publish only a subset of an object class's attributes, you must explicitly publish the object class, along with the desired attribute set, before creating the object publisher. This is done using *DtObjClassDesc's* publish() member function. When the object publisher is being constructed, it will see that the class has already been published, and will refrain from performing the default publication.

Do not publish by directly using RTI services. The RTI API does not provide a way to find out which classes and attributes you have published. So if you publish directly through the RTI API, VR-Link cannot know that you have done so, and will perform the default publish with all attributes included.

For example, to publish only the EntityType and Position attributes of the MilitaryAirLandPlatform class:

```
DtExerciseConn conn(...);
...
// Get the appropriate class descriptor from the DtFom, using the
// fully qualified class name
DtObjClassDesc* desc = conn.fom()->objClassByName(
   "BaseEntity.PhysicalEntity.MilitaryEntity.MilitaryPlatform."
   "MilitaryAirLandPlatform");
```

```
RTI::ObjectClassHandle classHand = desc->handle();

// Create the desired attribute handle set, represented by the RTI's
// RTI::AttributeHandleSet class
RTI::AttributeHandleSet* hSet =
           RTI::AttributeHandleSetFactory::create(2);
hSet->add(conn.rtiAmb()->getAttributeHandle("EntityType", classHand));
hSet->add(conn.rtiAmb()->getAttributeHandle("Position", classHand));

// Call the object class descriptor's publish() function
desc->publish(*hSet);

// Delete the set; it is no longer needed
delete hSet;

// Now create an entity publisher
DtEntityPublisher pub(&conn, DtEntityType("1:2:225:1:9:0:0"));
```

Attribute publications apply only to the class that you specify, not to subclasses or superclasses. So you will need to call publish() on multiple class descriptors to change the set of attributes published for a whole portion of a class hierarchy.

Interactions work a little differently. You cannot publish only a subset of the parameters of an interaction. You either publish an interaction class with all of its parameters, or not at all. When you send an interaction of a particular class using DtExercise-Conn::send() or sendStamped(), VR-Link publishes the class for you immediately before sending if the class has not yet been published or is explicitly unpublished by application code.

6.3.2. Subscribing to Classes and Attributes

Subscription to object classes normally is performed by a reflected object list's constructor. A particular kind of reflected object list (for example, a *DtReflectedEntity-List*) determines which FOM classes it is interested in, typically by either using the class handles passed to the constructor, or by asking the FOM Mapper. It then subscribes to each of those classes if the federate has not already subscribed or explicitly unsubscribed to the class.

Most reflected object lists in VR-Link take an optional **handleList** argument, which allows you to specify exactly which FOM classes should be subscribed to (and managed by the reflected object list). The **handleList** is a *DtList* of RTI::ObjectClassHandles, cast to `void*`, (not pointers to RTI::ObjectClassHandles). If this argument to the *DtReflectedEntityList* constructor is omitted, it gets the set of classes to subscribe to from the FOM Mapper. (The default FOM Mapper says that *BaseEntity* and all of its subclasses except *AggregateEntity* should be managed by *DtReflectedEntityList*.)

The following example demonstrates the procedure for telling a *DtReflectedEntityList* to subscribe to and manage only the MilitaryAirLandPlatform and Munition FOM classes:

```
DtExerciseConn conn(...);
...
// Create the desired handle list
```

```
DtList handleList;
handleList.add((void*) conn.rtiAmb()->getObjectClassHandle(
   "BaseEntity.PhysicalEntity.MilitaryEntity.MilitaryPlatform."
   "MilitaryAirLandPlatform"));
handleList.add((void*) conn.rtiAmb()->getObjectClassHandle(
   "BaseEntity.PhysicalEntity.MilitaryEntity.Munition"));

// Create the reflected entity list
DtReflectedEntityList(&conn, &handleList);
```

By default, when VR-Link subscribes to an object class, we subscribe to all attributes of the class. To subscribe to only a subset of an object class's attributes, follow the procedure described in "Publishing Classes and Attributes," on page 6-6 for publishing a subset of an object class's attributes above, but use the DtObjClassDesc() member function subscribe() instead of publish().

Do not subscribe by directly using RTI services. The RTI API does not provide a way to find out which classes and attributes you have subscribed to. If you subscribe directly through the RTI API, VR-Link will not know that you have done so, and will perform the default subscribe with all attributes included.

Subscription to interaction classes normally takes place when you register a callback function with a VR-Link interaction class. That action tells VR-Link that you are interested in a certain set of interaction classes, and we pass that information on to the RTI. Just as with publication, the RTI API does not let you subscribe to a subset of an interaction's parameters. It's all or nothing.

6.4. Managing HLA Objects

VR-Link uses instances of the *DtHlaObject* class (defined in *hlaObj.h*) to represent both local and reflected HLA objects. Because most entity management functionality is available through VR-Link's protocol-independent layer, most applications do not need to interact with *DtHlaObjects* (or even know they exist). However, there are always *DtHlaObjects* behind the scenes.

On the outgoing side, a *DtObjectPublisher* creates and uses a *DtHlaObject* to help with the sending of attribute updates. On the incoming side, VR-Link creates a *DtHlaObject* instance for each object that we discover through the RTI. If the object is to be managed by a *DtReflectedObject*, then that *DtReflectedObject* stores a pointer to its *DtHlaObject*, and uses it to help manage the reception of incoming attribute updates.

Both *DtReflectedObject* and *DtObjectPublisher* have an hlaObject() member function that returns a pointer to the *DtHlaObject* that it is managing.

VR-Link maintains two lists consisting of all reflected and locally simulated HLA objects, without regard to what kind of reflected object or object publisher is being used to manage the objects (if any). These lists are accessible through the *DtReflectedHlaObjectManager* and *DtLocalHlaObjectManager* classes (defined in *refObjMgr.h* and *locObjMgr.h*) respectively. Instances of these two classes are created by *DtExerciseConn*, and are available through *DtExerciseConn*'s reflectedObjectManager() and localObjectManager() functions.

Each of these HLA object manager classes has a function called allHlaObjects(), which returns a *DtList* of pointers to all current reflected or local *DtHlaObjects*. The lists are automatically updated whenever a new object is discovered or removed by a RTI-initiated service invocation, or when you create a *DtHlaObject* to represent a locally simulated object (which happens when you create a *DtObjectPublisher*).

For example, to print the IDs of all objects currently being locally simulated:

```
DtExerciseConn exConn(...);
 ...
 // Subscribe to various object classes, process discoverObjects and
 // reflectAttributeValues calls. (This is usually achieved simply by
 // creating a DtReflectedObjectList).
 ...
 DtReflectedHlaObjectManager* refObjMgr =
           exConn.reflectedObjectManager();
 DtList* allObjects = refObjMgr->allHlaObjects();
 for (DtListItem* item = allObjects->first(); item;
    item = item->next())
 {
    // Cast the generic void* to a DtHlaObject*
    DtHlaObject* obj = (DtHlaObject*) item->data();
    printf("ID: %ld\n", obj->objectId());
 }
```

The HLA manager classes also allow you to look up HLA objects by ID, using their hlaObject() member functions. These functions return NULL if no object with the given ID exists.

Besides the expected functions that give basic information about an HLA object (objectId(), name(), classDesc()), *DtHlaObject* has additional member functions that allow you to obtain more specialized information. Table 6-1 describes these functions.

Table 6-1: Additional DtHlaObject member functions

Function	Description
attributesNeededByFederation()	Returns the set of attributes of this object that have been subscribed to by remote federates, as told to us by the RTI.
classNeededByFederation()	Tells you whether the class has been subscribed to at all by remote federates.
requestedAttributes()	Returns the set of attributes for which updates have been requested by another federate through the RTI, since the time that we last sent updates for those attributes.
lastSimTimeUpdateReceived()	Returns the time that the last attribute update was received for this object, if it is a reflected object.

6.4.1. Finding out When HLA Objects are Discovered and Removed

"Learning when Entities Join or Leave An Exercise," on page 5-27 described protocol-independent methods of receiving notification about the discovery and removal of reflected objects. These methods involved either registering "objectAddition" and "objectRemoval" callbacks with a *DtReflectedObjectList*, or further deriving from a subclass of *DtReflectedObjectList*, and providing new definitions for the "objectAdded" and removeAndDelete() virtual functions.

Although these are the preferred methods (due to protocol-independence, and ability to access the *DtReflectedObject* associated with an object), this section describes an alternate method, which works at the level of *DtHlaObject*. By using this method, you can be notified when any HLA object joins or leaves, rather than just one particular kind of object (entities, for example).

When a new object is discovered, the *DtExerciseConn* creates a *DtHlaObject* to represent it, and adds it to the *DtReflectedObjectManager*'s list of all reflected HLA objects. It is then passed to any "discoverObject" callback functions that have been registered with the *DtExerciseConn* for the object's class. You can register discoverObject callback functions for any object class using *DtExerciseConn*'s addDiscoverObjectCallback() member function. Callbacks can be unregistered using removeDiscoverObjectCallback().

If you would like to be notified when any object is discovered regardless of class, you can pass an RTI::ObjectClassHandle of 0 to addDiscoverObjectCallback().

Callback functions that you provide should match the signature defined in *dscvObjCbInfo.h*, for example:

```
void myDiscoverObjectCb(DtHlaObject* obj, void* usr);
```

If you would like to be notified when an object is removed, you can use DtHlaObject::addRemoveObjectCb() to register a removal callback directly with the *DtHlaObject* of interest. Such functions can be unregistered with removeRemoveObjectCb().

Alternatively, you can use *DtExerciseConn*'s addRemoveObjectCallback() and removeRemoveObjectCallback(), in which case you need to pass an object ID to specify which object you are interested in.

In either case, your callback functions should match the signature defined in *remObjCbInfo.h*. For example:

```
void myRemoveObjectCb(DtHlaObject* obj, void* usr);
```

When the removeObject() service is invoked by the RTI, the corresponding *DtHlaObject* is removed from VR-Link's internal list of reflected HLA objects, passed to any callback functions registered on the removal of the object, then deleted.

As an example, these callback mechanisms are how *DtReflectedObjectLists* are notified of the arrival and removal of reflected objects. When the reflected object list's discoverObject() callback is invoked, it creates a new instance of the appropriate *DtReflectedObject* subclass to manage the object. When its removeObject callback is invoked (reflected object lists register one for each object as it is discovered), the *DtReflectedObject* is removed from the list and deleted.

6.4.2. Intercepting Reflected Attribute Values

In "Notifying An Application When State Updates Arrive," on page 5-28, we described a protocol-independent way of being notified when an attribute update arrives for a particular object. This was accomplished by registering a postUpdate callback with a *DtReflectedObject*.

Here we describe an alternate mechanism that works at the level of *DtHlaObjects*. This method is useful when there are some types of objects that are not managed by a *DtReflectedObjectList*, or when you want to be notified when an object is updated, regardless of what type of object it is.

In HLA, VR-Link routes attribute updates directly to the *DtHlaObject* whose attributes the update describes, which may process it however it desires. (Typically, the *DtHlaObject* decodes the message into its *DtReflectedObject*'s *DtStateRepository*). If you would like to be notified when an update message has been processed by a *DtHlaObject*, you can register a callback function with the *DtHlaObject* using its addPostReflectCb() member function. Use removePostReflectCb() to unregister the callback.

These callback functions should look like the following, as specified in *objRefCbInfo.h*:

```
void postReflectCb(const DtStateMsg& msg, DtHlaObject* obj, void* usr);
```

Your function is called immediately after the *DtHlaObject* has finished processing the message. Along with the *DtHlaObject* (and the **usr** pointer, of course), the message itself is passed to your function.

DtStateMsg, defined in *hStateMsg.h*, is basically a wrapper that contains the ID of an object whose state the message is updating, and the message itself. The information is presented as an RTI::AttributeHandleValuePairSet, the form in which we get it from the RTI. These two components of a *DtStateMsg* are available through its objectId() and ahvps() functions respectively.

An example of intercepting reflected attribute values is our *hlaNetdump* application, which works as follows:

1. It registers a discoverObject callback with the *DtExerciseConn* for all objects of all classes.
2. Within the discoverObject callback, *hlaNetdump* registers a postReflect callback with each *DtHlaObject* that is discovered.
3. Within that postReflect callback, we print the contents of the message.

See *./examples/test/hlaNetdump.cxx* for details.

6.4.3. Forcing Attribute Updates

When you call a *DtObjectPublisher*'s tick() function, it calls down to its *DtHlaObject*'s update() function, which decides which attributes need to be sent, and then sends them.

Several factors go into the decision of which attributes need to be sent:

- Only those attributes that are both published by us, and subscribed to by some remote federates are eligible to be sent.
- Of these eligible attributes, we only actually send those whose update conditions have been met, plus any attributes for which updates have been explicitly requested by a remote federate (as indicated by the RTI through provideAttributeUpdate() service invocations).

If you want certain eligible attributes to be sent during the next call to a *DtObjectPublisher*'s tick() function, even if their update conditions have not been met, and they have not been specifically requested by a remote federate, you need to pass the desired set of attributes to the appropriate *DtObjectPublisher*'s forceUpdate() member function. If you omit the attribute set argument to forceUpdate(), then an update for all eligible attributes is sent during the next call to tick().

6.5. Using the DtInteraction Class

The HLA version of the *DtInteraction* class is defined in *hInteraction.h*. Various HLA-specific data are available through a *DtInteraction*, much as HLA-specific object data is available through a *DtHlaObject*.

Much of this data is not available (unable to be determined by a *DtInteraction*) until the *DtInteraction* is told which FOM interaction class is being used to represent it, and which *DtExerciseConn* it is associated with. For example, we cannot determine if the *DtInteraction*'s FOM interaction class has been subscribed to by remote federates unless we know what FOM class we wish to use and which *DtExerciseConn* to ask for subscription information.

DtInteraction has a member function called setExConn() that allows you to set the *DtExerciseConn* and *DtInterClassDesc* (interaction class descriptor) that are currently associated with a particular instance. setExConn() must be called on an interaction before many of *DtInteraction*'s member functions can return meaningful results. If the class descriptor argument to setExConn() is omitted, the *DtInteraction* subclass will choose a reasonable default (see "Choosing An Interaction Class to Publish," on page 6-25 for details).

It is rare that application code will need to call setExConn() explicitly. VR-Link typically calls it for you. When sending a *DtInteraction*, *DtExerciseConn*'s send() or sendStamped() functions call setExConn() on the *DtInteraction* before trying to actually send. That *DtExerciseConn* passes itself as the **exConn** argument, and passes NULL as the **classDesc** argument, allowing the *DtInteraction* to choose a FOM class to use for sending. On the incoming side, after creating the *DtInteraction* instance to represent a received interaction, but before passing it to user callbacks, VR-Link calls setExConn() on the interaction, passing the *DtExerciseConn* on which the interaction was received, along with the interaction class of the received interaction.

Interaction classes typically store their parameter values in their own native representation, but you can obtain this data in the RTI's representation (an RTI::ParameterHandleValuePairSet) using DtInteraction::phvps(). In fact, this is what *DtExerciseConn*'s send() and sendStamped() functions use to obtain the object to pass to the RTI for sending.

A *DtInteraction* can be assigned from an RTI representation using the member function setFromPhvps(). *DtExerciseConn* uses this function to initialize a *DtInteraction* as a result of a receiveInteraction() RTI service invocation.

Besides these two important functions, Table 6-2 describes functions that require that a *DtInteraction* instance have a valid current exercise connection and class descriptor.

Table 6-2: Additional DtInteraction functions

Function	Description
interactionClassHandle()	Returns the current interaction class's handle.
interactionClassName()	Returns the current interaction class's name.
numParameters()	Returns the number of parameters in the current interaction class.
neededByFederation()	Returns whether the class has been subscribed to by remote federates.

6.5.1. *DtInteraction* Print-Related Functions

DtInteraction has four different virtual printing functions. tab describes these functions.

Table 6-3: DtInteraction print functions

Function	Description
print()	The standard printing function, used by VR-Link's hlaNetdump utility. It uses the virtual name() function to print the name of the interaction, then calls printParams() and printData().
printHeader()	Prints information about the FOM class being used for the object.
printParams()	first calls printHeader(), then prints the set of parameters included in the interaction message, along with raw parameter values if its withHex argument is set to DtTrue.
printData()	Is a pure virtual function that is implemented by derived classes to print the specific data contained in the interaction, in the form that is available through inspector functions.

Both printHeader() and printParams() require that the interaction have a current exercise connection and class descriptor to fully do their jobs.

If you would like to print the data in a *DtInteraction* that you create before you send it through a *DtExerciseConn*, you can explicitly call setExConn() before printing, to insure that all data can be printed. For example:

```
DtExerciseConn exConn(...);
...
DtFireInteraction inter;
inter.setAttacker(...);
...
inter.setExConn(&exConn);
inter.print();
```

6.6. General HLA Issues

The following sections discuss a variety of general HLA issues.

6.6.1. Reflecting Locally Generated Data

In HLA, the RTI does not reflect locally generated interactions or attribute updates back to us. Therefore, reflected object lists typically do not contain *DtReflectedObjects* representing objects that we are simulating locally. Similarly, callback functions registered on interactions will not be called for interactions that the local federate sends.

However, VR-Link has an option that causes locally generated data to be reflected back to a sending federate, as if the RTI was doing so. The option can be enabled using *DtExerciseConn*'s setReflecting() function, passing DtTrue as an argument. To turn this option off (the default), pass DtFalse to setReflecting(). The function DtExerciseConn::reflecting() tells you what the current state of this flag is.

When we are doing this kind of local reflecting, separate *DtHlaObjects* exist in your federate to represent the local object and its reflected counterpart, but both will have the same RTI::ObjectHandle and name.

This option will only work if there are other federates in the exercise that have subscribed to your data. If not, the data will never be sent at all.

6.6.2. Time Stamps

You can set and inspect an HLA *DtStateMsg* or *DtInteraction*'s time stamp with the setTimeStamp() and timeStamp() functions. These work the same way as the DIS *DtPdu*'s function - taking a time, and a time stamp type (either *DtTimeStampRelative* or *DtTimeStampAbsolute*).

DtExerciseConn::sendStamped() sets the time stamp of an outgoing HLA message to the current time, with the type returned by DtExerciseConn::timeStampType().

Consistent with the RPR FOM convention, our sending functions encode the time in the “tag” string argument to the RTI's sendInteraction() and updateAttributeValues() calls. The time is encoded as the 8-byte ASCII representation of the hex number that would have been sent in a DIS time stamp.

If you do not want us to use this convention, use DtInteraction::setTagIsAsciiTime(DtFalse); and DtStateMsg::setTagIsAsciiTime(DtFalse). If you do this, your time stamps will not be sent, since the RTI does not provide a mechanism for sending time stamps along with your messages unless you are using RTI time management, which real-time simulations typically do not do.

6.6.3. Subclassing RTI::FedTime

As per the HLA specification, RTI users can choose their own representation of federation time for use by time management services. Unfortunately, even if you are not using RTI time-management services, (and most VR-Link-based applications do not), RTI users must tell the RTI how they would like federation time to be represented in case they use a time management service.

To communicate this information to the RTI, RTI users must subclass the abstract base class RTI::FedTime, and tell the RTI to use the subclass by providing implementations for the functions RTI::FedTimeFactory::decode() and RTI::FedTimeFactory::makeZero(). (These functions are declared, but not defined by the RTI's API.)

Both the DMSO RTI and MÄK RTI provide a library called *libfedtime.so* (or *libfedtime.dll*) that satisfies this RTI requirement. However, to avoid forcing all applications to link with this extra library, and to satisfy cases where RTI implementations do not come with sample libfedtime libraries, we have provided an RTI::FedTime subclass, and definitions to the required functions, within VR-Link. The subclass is called *DtVrlFedTime*, and is defined in *vrlFedTime.h*.

In UNIX releases, this class is part of the *libvlRPR* library. However, on PCs, the RTI requires that RTI::FedTime subclass implementations be provided in DLLs, and not in archives. So on PCs, the *DtVrlFedTime* class is contained in a separate VR-Link library called *vrlFedTimeMD.dll*, rather than in *libvlRPR*.

6.6.4. VR-Link Calls to RTI Services

This section lists the RTI services that may be called by VR-Link.

Table 6-4 lists Federate-initiated services that may be called by VR-Link. Table 6-5 lists RTI-initiated services for which VR-Link provides non-NULL definitions.

Table 6-4: Federate-initiated service called by VR-Link (Page 1 of 3)

RTI Service	How it is invoked
createFederationExecution	Invoked from within DtExerciseConn constructor.
destroyFederationExecution	Invoked from within DtExerciseConn destructor, if the DtExerciseConn's destroyFedExec flag is true (the default).
joinFederationExecution	Invoked from within DtExerciseConn constructor.
resignFederationExecution	Invoked from within DtExerciseConn destructor.
publishInteractionClass	Invoked from within DtExerciseConn's send or sendStamped whenever it is called with an interaction whose class has not been published yet.

Table 6-4: Federate-initiated service called by VR-Link (Page 2 of 3)

RTI Service	How it is invoked
publishObjectClass	Invoked from within DtEntityPublisher's (or any other object publisher's) constructor if its object class has not already been published.
subscribeInteractionClass	Invoked when registering a callback function with VR-Link on a particular interaction class using a DtInteraction subclass's addCallback function.
subscribeObjectClassAttributes	Invoked from within DtReflectedEntityList's (or any other reflected object list's) constructor.
unsubscribeInteractionClass	Called by DtObjClassDesc::deregisterInterest, but that function is not called by other VR-Link code. So if application code does not call it, this service is never invoked.
unsubscribeObjectClass	Called by DtInterClassDesc::deregisterInterest, but that function is not called by other VR-Link code. So if application code does not call it, this service is never invoked.
deleteObjectInstance	Invoked from the DtEntityPublisher's (or any other object publisher's) destructor.
localDeleteObjectInstance	Invoked only from DtReflectedObjectList::removeObject, which is not typically called by applications.
registerObjectInstance	Invoked from DtEntityPublisher's (or any other object publisher's) constructor.
requestClassAttributeValueUpdate	Invoked from DtReflectedEntityList's (or any other reflected object list's) constructor, if the list's requestClassUpdate flag is true (the default).
requestObjectAttributeValueUpdate	If DtReflectedEntityList's (or any other reflected object list's) requestObjectUpdate flag is true (the default) then this service is invoked shortly after each DtReflectedEntity (or other reflected object) is discovered and created. Since we can't call the service from within our discover object callback (it would be a concurrent access violation), the service is not called until just before DtExerciseConn::drainInput returns.
sendInteraction	Invoked from within DtExerciseConn's send or sendStamped.

Table 6-4: Federate-initiated service called by VR-Link (Page 3 of 3)

RTI Service	How it is invoked
updateAttributeValues	Invoked from within DtEntityPublisher's (or any other object publisher's) tick function, if we judge that data needs to be sent based on update conditions and remote subscriptions and requests.
RTIambassador	DtExerciseConn's constructor constructs an RTI ambassador.
~RTIambassador	DtExerciseConn's destructor deletes its RTI ambassador.
enableAttributeRelevance AdvisorySwitch	Invoked by DtExerciseConn constructor.
enableClassRelevance AdvisorySwitch	Invoked by DtExerciseConn constructor.
enableInteractionRelevance AdvisorySwitch	Invoked by DtExerciseConn constructor.
tick	Called from within DtExerciseConn's drainInput.

Table 6-5: RTI-initiated services with non-NULL definitions (Page 1 of 2)

RTI Service	VR-Link Response
startRegistrationForObjectClass	VR-Link maintains information about which object classes are needed by the federation.
stopRegistrationForObjectClass	VR-Link maintains information about which object classes are needed by the federation.
turnInteractionsOff	VR-Link maintains information about which interaction classes are needed by the federation.
turnInteractionsOn	VR-Link maintains information about which interaction classes are needed by the federation.
discoverObjectInstance	If any reflected object list has registered interest in the object's class or a superclass, VR-Link creates a new reflected object and adds it to the list.
provideAttributeValueUpdate	VR-Link notes that the attribute set has been requested. The next time the object's publisher's tick is called, the request attribute updates are sent.

Table 6-5: RTI-initiated services with non-NULL definitions (Page 2 of 2)

RTI Service	VR-Link Response
receiveInteraction	If any user callbacks have been registered with VR-Link for the received interaction type, those callbacks are invoked.
reflectAttributeValues	The relevant reflected object's state repository is updated based on the contents of the received attribute update.
removeObjectInstance	The relevant reflected object is removed from the reflected object list, and deleted.
turnUpdatesOffForObjectInstance	VR-Link maintains information about which attributes are needed by the federation.
turnUpdatesOnForObjectInstance	VR-Link maintains information about which attributes are needed by the federation.

The following ancillary services are called at various points within VR-Link:

- getAttributeHandle
- getAttributeName
- getInteractionClassHandle
- getObjectClassHandle
- getObjectInstanceName
- getParameterHandle
- getParameterName.

6.7. FOM Agility

VR-Link has been designed with FOM flexibility in mind. VR-Link comes with built-in support for different versions of the RPR FOM[1], but it can be extended or configured in many different ways to work with changes or extensions to the FOM, or to work with entirely new FOMs.

This section, along with examples in the *./examples/extend* directory, explains how VR-Link's architecture enables FOM agility and how you can extend or configure VR-Link for new FOMs.

6.7.1. FOM Mapping Overview

VR-Link's basic architecture for FOM flexibility looks like this:

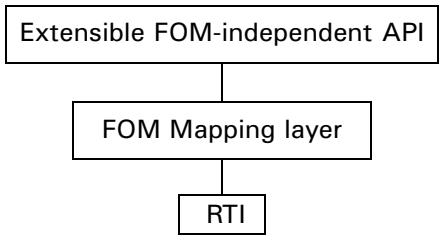


Figure 6-1. FOM mapping architecture

The top-level FOM-independent API[2] allows you to set and inspect the state of local and remote objects, and the data contained in locally generated or received interactions, without regard to how the data is actually represented in a FOM. This top-level API includes the object publishers, reflected objects, reflected object lists, state repositories and interaction classes.

The FOM Mapping layer routes and converts data between the FOM-independent API and the FOM representations you are using for a particular federation execution.

There are two broad categories of FOM modifications that developers might implement:

- FOMs that contain only those concepts that are already present in VR-Link's top-level API, but which may represent the concepts differently from the way they are represented in the VR-Link API.
- FOMs that include concepts that are not present in VR-Link's API.

1. See the release documentation for your version of VR-Link for a list of the versions of the RPR FOM supported.
2. The FOM-independent API is also referred to as the protocol-independent API because it can support DIS in addition to various HLA FOMs.

If you are implementing FOMs that have the same concepts as the VR-Link API, you can switch from FOM to FOM just by modifying the FOM Mapping layer, which you can do without affecting the top-level API. This means that application code written to the top-level API does not need to change when switching from FOM to FOM. (We will see in "Passing the Name of A Shared Library," on page 6-33, that because FOM Mapping code can be loaded from shared libraries, applications do not even need to be recompiled when switching among such FOMs.)

For example, in one FOM (such as the RPR FOM), position data may be represented in geocentric coordinates in an attribute named "Position". In another, it might be represented in topographic coordinates in an attribute named "Location". In either case, position data is available to application code through VR-Link in the manner dictated by its FOM-independent API - as geocentric data obtainable using *DtEntity-StateRepository*'s location() member function. FOM Mapping code takes care of converting from the various FOMs' representations to the representation chosen by VR-Link.

If you are using FOMs that contain concepts not found in the VR-Link API, you must follow a two-step FOM-configuration process.

1. Extend the top-level API, by deriving new classes from our base state repository, publisher, reflected object, or reflected object list classes to include your new concepts. Once you have done this, your code is part of the FOM-independent API.
2. Create mappings from arbitrary FOM representations of these concepts to your new API extensions, just as if your extensions were part of the standard VR-Link API. (Typically, you will choose representations for your API that closely match the representation in a particular FOM, meaning that the mapping code will be fairly straightforward).

6.7.2. FOM Mapping Information Required By VR-Link's API

The HLA implementations of the classes that make up VR-Link's top-level API require the following pieces of FOM Mapping information:

- Class mappings
  - Publishers need to know which FOM object class to choose to represent a locally simulated object.
  - Reflected object lists need to know the set of FOM classes for which they should register discoverObject callbacks, that is, which types of objects should be managed by the list.
  - Interaction instances need to know which FOM class to use to represent themselves when they are being sent.
  - Interaction classes need to know which FOM classes to subscribe to when their static addCallback functions are called.
  - *DtExerciseConn* needs to know which *DtInteraction* subclass to instantiate when an interaction of each class in the FOM is received.
- Attribute and parameter encoding and decoding
  - Publishers need to know how to decide whether each attribute needs to be sent during a call to tick(), and how to encode values from its state repository into outgoing attribute updates.
  - Reflected objects need to know how to decode each attribute of a FOM class into its state repository.
  - Interaction instances need to know how to encode and decode each parameter.

Architecturally, it is the responsibility of the these top-level objects to choose how they obtain this FOM mapping information. For example, a particular subclass of *DtInteraction* might hard-code the name of the FOM class it should be associated with, or it might check some global variable.

However, all of the FOM-independent classes in VR-Link get their FOM mapping information from a central place called a *DtFomMapper*. (Actually, many of these classes allow application code to override the FOM mapper's choice, but asking the FOM mapper is the default behavior.)

The FOM mapper that these top-level objects use, is obtained from their *DtExerciseConn*, through *DtExerciseConn*'s fomMapper() member function. *DtExerciseConn* creates, or is passed, a *DtFomMapper* on construction. See "Choosing A DtFomMapper," on page 6-32 for details.

6.7.3. FOM Mapping Details

A *DtFomMapper* (defined in *fomMapper.h)* is a central repository for FOM mapping information. Top-level API classes call *DtFomMapper* functions to obtain FOM mapping information. For example, to find out which FOM object class to use to represent an object being managed by a particular type of *DtObjectPublisher*, the publisher calls *DtFomMapper*'s chooseObjectClass() member function.

To set or change a FOM Mapper's mappings, use *DtFomMapper*'s mutator functions to configure a FOM Mapper so that its functions return the data you want.

For example, you can use *DtFomMapper*'s setObjectClassToChoose() to set up a mapping between a particular subclass of *DtObjectPublisher* and a particular FOM object class.

In order to have VR-Link use new FOM mappings that you provide, you can either allow the *DtExerciseConn* to create its *DtFomMapper*, then add your own mappings to this FOM mapper later, or you can create your own instance of a *DtFomMapper*, and tell the *DtExerciseConn* to use it. See "Choosing A DtFomMapper," on page 6-32 for more details. The following sections describe how a FOM mapper is configured, regardless of whether you choose to do it before or after *DtExerciseConn* construction.

6.7.4. Class Mappings

As described in "FOM Mapping Information Required By VR-Link's API," on page 6-22, a FOM mapper contains class mapping information as well as attribute and parameter encoding and decoding information. This section describes how to set up a FOM mapper's class mappings.

Choosing An Object Class to Publish

A *DtObjectPublisher* asks the FOM mapper for the name of the FOM object class to use to represent its locally simulated object by calling chooseObjectClass(). The publisher passes the name of its own VR-Link C++ publisher class, for example, *DtEntityPublisher* or *DtAggregatePublisher.*

When configuring a FOM mapper, you can dictate what FOM class chooseObject-Class() will return in one of two ways:

- Map a single class to a publisher
- Map multiple classes to a publisher

Mapping A Single FOM Class to A Publisher

If one FOM class should always be chosen for a particular kind of publisher, create this association using *DtFomMapper*'s setObjectClassToChoose(). For example, in the RPR FOM, *DtAggregatePublishers* should always use the FOM class *BaseEntity.AggregateEntity*:

```
fomMapper->setObjectClassToChoose("DtAggregatePublisher",
          "BaseEntity.AggregateEntity");
```

Mapping Multiple FOM Classes to A Publisher

It might be possible to map several FOM classes to a publisher. In these cases, some instance-specific data might be necessary in order to make a choice. In this case, you can associate a class choosing function with the name of a VR-Link class, rather than just a single object class, using setObjectClassChooser().

A class choosing function looks like this:

```
char* myClassChooser(const char* vrlinkClassName, DtExerciseConn* conn,
          void* usr);
```

You can associate a class choosing function with a VR-Link publisher class name as follows:

```
fomMapper->setObjectClassChooser("DtEntityPublisher", myClassChooser);
```

The VR-Link class may pass instance-specific data as the usr argument to chooseObjectClass(), and this data will be passed to your chooser function as usr. For example, *DtEntityPublisher* passes a pointer to the entity's *DtEntityType*, and in the RPR FOM mapper, our class chooser function picks a FOM class based on the entity type.

Subscribing to Object Classes

A *DtReflectedObjectList* asks the FOM mapper for the names of the FOM classes to subscribe to and manage, by calling objectClassNames(). Besides subscribing to the returned classes, the reflected object list registers discoverObjectInstance callbacks with lower layers of VR-Link for these classes, so that it can be made aware of new instances, and can create *DtReflectedObject* instances to represent them. The reflected object list passes the name of its own VR-Link C++ class, for example, *DtReflectedEntityList*.

When configuring a FOM mapper, you can specify which FOM classes objectClassNames() returns in one of two ways:

- If a particular *DtReflectedObjectList* subclass should manage only a single FOM class, create this association using setObjectClass(). For example:

  ```
  fomMapper->setObjectClass("DtReflectedAggregateList",
            "BaseEntity.AggregateEntity");
  ```

- If you want a particular kind of list to manage more than one FOM object class, use setObjectClassNames() to pass a *DtList* of names of FOM classes. For example:

  ```
  DtList entityClasses;
  entityClasses.add("BaseEntity");
  entityClasses.add("PhysicalEntity");
  ...
  fomMapper->setObjectClasses("DtReflectedEntityList", entityClasses);
  ```

Choosing An Interaction Class to Publish

Choosing an interaction class works very similarly to choosing an object class. When you call setExConn() on an interaction (*DtExerciseConn*'s send() and sendStamped() call this function), the *DtInteraction* instance asks the FOM mapper for the name of a FOM class to use to represent the interaction using chooseInteractionClass(). The interaction instance passes a reference to itself. Based on the name of the VR-Link C++ interaction class (which can be obtained from an interaction instance using its name() function) the FOM mapper must choose an appropriate FOM class to use.

As with objects, you can specify which FOM class chooseInteractionClass() returns in one of two ways:

- If one single FOM class should always be chosen for a particular kind of *DtInteraction*, create this association using *DtFomMapper*'s setInteractionClassToChoose(). For example, in the RPR FOM, *DtFireInteraction*s should always use the FOM class *WeaponFire*:

  ```
  fomMapper->setInteractionClassToChoose("DtFireInteraction",
            "WeaponFire");
  ```

- When more than one mapping is possible for a single kind of *DtInteraction*, associate an interaction class chooser function with the VR-Link class name using setInteractionClassChooser(). Choosers look like this:

  ```
  char* myChooser(DtExerciseConn* exConn, const DtInteraction& inter);
  ```

  and are registered like this:

  ```
  fomMapper->setInteractionClassChooser("DtFireInteraction", myChooser);
  ```

  The actual interaction for which a choice is being requested is passed to your chooser, in case any instance-specific data is needed to make the choice.

Subscribing to Interaction Classes

When you call the static addCallback() function on a particular *DtInteraction* subclass, you are saying that you are interested in interactions of that type. The *DtInteraction* subclass maps that request to subscriptions to FOM classes. It obtains the set of FOM classes to subscribe to from the FOM mapper using its interactionClasses() member function.

You can determine what this function will return using setInteractionClass() or setInteractionClasses(). Use setInteractionClass() to map to a single FOM class:

```
fomMapper->setInteractionClass("DtFireInteraction", "WeaponFire");
```

and setInteractionClasses() for multiple FOM classes:

```
DtList classNames;
classNames.add(RadioSignal.ApplicationSpecific");
classNames.add(RadioSignal.EncodedAudio");
...
fomMapper->setInteractionClasses(classNames);
```

6.7.5. Creating Instances of *DtInteraction*

For interactions, the final class mapping component is the mapping between FOM classes and functions that create instances of the appropriate *DtInteraction* subclasses. *DtFomMapper* uses a *DtInteractionFactory* (defined in *hInterFactory.h*) to manage this kind of mapping.

When an interaction is received from the RTI, the *DtExerciseConn* asks the FOM mapper's *DtInteractionFactory* to create an instance of the appropriate kind of *DtInteraction* subclass, using its createInteraction() function. The FOM mapper's interaction factory is obtained using *DtFomMapper*'s interactionFactory() function.

You can add or change an interaction factory's mappings using its addCreator() function, passing a FOM class name and an interaction creation function such as a *DtInteraction* subclass's static create() function. For example, when we receive an interaction of FOM class *WeaponFire*, we want to create an instance of *DtFireInteraction*:

```
fomMapper()->interactionFactory()->addCreator("WeaponFire",
           DtFireInteraction::create);
```

You can replace a *DtFomMapper*'s interaction factory with a new one using its setInteractionFactory() member. For example, you might want to create a subclass of *DtInteractionFactory* for your FOM that registers all of the desired creators from within its constructor. You would then create an instance of this *DtInteractionFactory* subclass, and instruct the FOM mapper to use it as its interaction factory.

No analogous mapping step is required for objects, since instances of appropriate *DtReflectedObject* subclasses are created not by a central class like *DtExerciseConn*, as interactions are, but rather by *DtReflectedObjectList* subclasses. And each *DtReflectedObjectList* knows what kind of *DtReflectedObject* subclass to instantiate.

6.7.6. Attribute and Parameter Encoding and Decoding

Besides the class mappings described in "Class Mappings," on page 6-23, a *DtFomMapper* also contains information about how to encode and decode individual attributes and parameters, that is how to go between an RTI message in FOM representation, and VR-Link's *DtStateRepositories* and *DtInteractions*. This section describes how to configure that aspect of a FOM mapper.

Here's an overview of the classes we will be discussing, and how they fit together: Encoders and decoders are tables of encoding and decoding functions - one for each attribute. Encoder factories and decoder factories are tables of encoders and decoders - one for each object or interaction class. A *DtFomMapper* has an encoder factory and a decoder factory for both objects and interactions. Top-level VR-Link classes request encoders and decoders from a FOM mapper's encoder and decoder factories, and use those encoders and decoders to generate or decode RTI representations of state updates or interactions.

You can change the way that attributes or parameters are encoded or decoded, or provide methods for encoding or decoding new attributes or parameters, by adding your own functions to the encoders' and decoders' tables. You can get a pointer to the encoder or decoder you want to change from the appropriate factory. To add whole new FOM classes, you can create new encoders and decoders and add them to the factories' tables.

Encoders and Decoders

At the heart of VR-Link's FOM Mapping functionality are our encoder and decoder classes:

- *DtHlaStateEncoder* (*hStateEncoder.h*)
- *DtHlaStateDecoder* (*hStateDecoder.h*)
- *DtInteractionEncoder* (*interEncoder.h*)
- *DtInteractionDecoder* (*interDecoder.h*)
- And their subclasses.

A *DtReflectedObject* uses a state decoder's decode() function to unpack data that arrives in incoming state updates or interactions (which are represented as dictated by the FOM), convert it to the representation expected by the top-level API (mutator functions in state repositories), and pass the converted data to those mutator functions.

Similarly, a *DtInteraction* uses an interaction decoder's decode() function to unpack the data contained in an interaction message received from the RTI, convert, and pass that data to the *DtInteraction* subclass instance's appropriate mutator functions.

DtObjectPublisher's tick() function uses a state encoder's encode() function to decide which attributes need to be sent at the present time, and to generate an update for those attributes. The encoder evaluates update conditions, and generates outgoing update message by obtaining data from the publisher's state repository using its inspector functions, converting to the representations dictated by the FOM, and adding these values to an outgoing message for sending through the RTI.

Similarly, a *DtInteraction* uses an interaction encoder's encode() function during sending, to convert values obtained from inspector functions to outgoing interaction messages that can be sent through the RTI.

VR-Link's encoder and decoder classes implement an efficient, table-driven approach to attribute encoding and decoding. You are free to subclass encoders and decoders and to override the encode() and decode() functions, but this is rarely necessary. Usually, you will just configure encoders and decoders, adding individual attribute or parameter encoding or decoding functions to their tables.

Functions for Encoding and Decoding

A decoder is implemented as a table of decoding functions, one per FOM attribute or parameter. Each function is responsible for decoding the attribute or parameter it has been associated with. Similarly, an encoder contains a table of encoding functions. For objects, an encoder contains an additional table of checking functions - functions that evaluate the update conditions for individual attributes, indicating whether the attribute currently needs to be sent to the exercise.

Encoders and decoders have addEncoder() and addDecoder() functions, and *DtHlaStateEncoder* has an additional addChecker() function, that allow you to register your own encoding, decoding, and checking functions for a particular attribute or parameter with an encoder or decoder. These add functions return the function that had previously been registered for the attribute or parameter, if any. The add functions create associations only for attributes or parameters of a single FOM class. If you would like a function to be applied to an attribute or parameter of a class and all of its subclasses, you can call addEncoder(), addDecoder() and addChecker() on multiple encoder or decoder objects, or use the more convenient multi-class registration functions in the encoder or decoder factory classes. See "Encoder and Decoder Factories," on page 6-31 for details.

We now describe the form and function of encoding, decoding, and checking functions.

Decoding Interactions

DtInteractionDecoder's decode() function takes an RTI representation of an interaction message (an RTI::ParameterHandleValuePairSet, or PHVPS), and a pointer to a *DtInteraction* object to fill out based on the RTI data. It goes through the parameters in the PHVPS, and for each, calls the parameter decoding function that has been associated with the parameter.

Below is an example of the parameter decoding function used for decoding the "RateOfFire" attribute of the *WeaponFire* RPR FOM class into a *DtFireInteraction*. The general decode() function passes it a pointer to the interaction, the PHVPS containing the parameter values, and the index into the set of the parameters this decoding function should be decoding. Within the function, we pull out the index'th value, convert it to the representation expected by the appropriate *DtFireInteraction* mutator function (in this case, setRate()), and pass it to that mutator function. In this case, the conversion is fairly trivial - from a short to an int. In addition, when VR-Link's "Net" types are used byte swapping occurs when converting to a native type if necessary:

```
void decodeRateOfFire(DtFireInteraction* inter,
   const RTI::ParameterHandleValuePairSet& pvlist,
   int index)
{
   RTI::ULong length = 0;
   DtNetU16* netVal = (DtNetU16*) params.getValuePointer(index, length);
   int nativeVal = (DtU16) *netVal;
   inter->setRate(nativeVal);
}
```

Decoding Object State Updates

On the object side, *DtHlaStateDecoder*'s decode() function takes a *DtStateMsg* (a VR-Link wrapper around an RTI::AttributeHandleValuePairSet, or AHVPS), and a pointer to a *DtStateRepository* to decoder the message into. It walks the list of attributes in the message, and calls the attribute decoding function that has been associated with each.

Here's an example of a decoding function for an attribute. It's a function we might use for decoding the **Position** attribute of the *BaseEntity* RPR FOM object class into a *DtEntityStateRepository.* Again, index indicates which attribute in the set to decode. Here, the conversion from FOM representation to VR-Link representation looks trivial, since a *DtNet64Vector* can be implicitly converted to a *DtVector.* Clearly, more complex conversions, such as coordinate conversions, could be added if a FOM so dictates.

```
void decodePosition(
   DtEntityStateRepository* stateRep,
   const RTI::AttributeHandleValuePairSet& attrs,
   int index)
{
   RTI::ULong length = 0;
   DtNet64Vector* netVal = (DtNet64Vector*)
      attrs.getValuePointer(index, length);
   stateRep->setLocation((DtVector) *netVal);
}
```

Encoding Interactions

Encoders work like decoders, but in reverse. *DtInteractionEncoder*'s encode() function takes a *DtInteraction* instance containing interaction data, and a PHVPS to fill out with the FOM representation of this data. For each parameter in the *DtInteraction*'s FOM interaction class, encode() calls the encoding function that has been registered with the encoder for that parameter. Parameter encoding functions look like the following example. The **paramHandle** argument indicates the handle (not index) of the parameter to encode. Again, the byte swapping occurs if necessary when converting from an int to a *DtNetU16*.

```
void encodeRateOfFire(
   const DtFireInteraction& inter,
   RTI::ParameterHandleValuePairSet* params,
   RTI::ParameterHandle paramHandle)
{
   DtNetU16 netVal = (DtNetU16) inter.rate();
   params->add(paramHandle, (char*) &netVal, sizeof(DtNetU16));
}
```

Encoding Objects

Encoding objects is a little bit more complicated than encoding interactions, because the encoder has to first decide which attributes currently need to be sent, and then encode those attributes. A *DtHlaStateEncoder* contains both a table of encoding functions and a table of checking functions that determine whether an attribute needs to be sent now, based on current values, and values sent in previous updates. Update conditions are specified in a FOM. For example, many attributes have an update condition of ON_CHANGE, meaning that they need to be sent every time their value changes.

When a publisher's tick function calls a *DtHlaStateEncoder*'s encode() function, it passes five arguments:

- A *DtStateRepository* containing the current state of the object to be encoded.
- A second *DtStateRepository* containing the state of the object as it would be seen by remote federates based on attribute updates that we have already sent. This state repository is maintained by the publisher by using a decoder to decode updates that it sends into its as-seen-by-remote state repository after it sends them.
- The set of attributes that are currently subscribed to by remote federates.
- The set of attributes that should be sent now, even if their update conditions have not been met. This includes any attributes that have been specifically requested by a remote federate through a requestAttributeUpdate RTI service invocation, and any attributes for which the user has called DtObjectPublisher::forceUpdate().
- A pointer to the *DtStateMessage* (wrapper around AHVPS) to fill out.

For each published attribute of the FOM class, encode() determines whether the attribute needs to be included in the update message or not. This may require calling the checker function that has been registered for the attribute. If the attribute does need to be sent, encode() calls the encoding function that has been registered for the attribute to encode it.

Encoding functions look similar to interaction encoding functions described previously. They take the state repository containing current state, the AHVPS to encode into, and the attribute handle of the attribute to be encoded.

```
void encodePosition(
   const DtEntityStateRepository& stateRep,
   RTI::AttributeHandleValuePairSet* avList,
   RTI::AttributeHandle attrHandle)
{
   DtNet64Vector netVal = (DtNet64Vector) stateRep.location();
   avList->add(attrHandle, (char *) &netVal, sizeof(DtNet64Vector));
}
```

Checking functions take two state repositories as arguments: one holding the current state of the object, and one holding the state of the object as it would be seen by remote federates based on updates that we have sent. (The publisher decodes its outgoing updates into this as-seen-by-remote state repository in order to keep it up to date). Many checking functions usually need to compare the current value to the as-seen-by-remote value to determine whether an attribute needs to be sent. Here's an example of a checker used for the **DamageState** attribute of the *PhysicalEntity* RPR FOM class. It returns true if the value has changed since it was last sent:

```
DtBoolean needDamageState(
   const DtEntityStateRepository& stateRep,
   const DtEntityStateRepository& asSeenByRemote)
{
   return (stateRep.damageState() != asSeenByRemote.damageState()) ?
      DtTrue : DtFalse;
}
```

Encoder and Decoder Factories

When a publisher, reflected object, or incoming interaction is created, or when an outgoing interaction is sent, it obtains from the FOM mapper an instance of an encoder and/or decoder that has been configured for use with its FOM class. These are the instances that they will later use to perform the encoding and decoding of FOM data.

DtFomMapper uses four factory classes to help with the job of returning appropriate encoders and decoders:

- *DtStateDecoderFactory* (*decFactory.h*)
- *DtStateEncoderFactory* (*encFactory.h*)
- *DtInteractionDecoderFactory* (*intDecFact.h*)
- *DtInteractionEncoderFactory* (*intEncFact.h*).

Pointers to a *DtFomMapper*'s various encoder and decoder factories can be obtained using the following member functions:

- stateDecoderFactory()
- stateEncoderFactory()
- interactionDecoderFactory()
- interactionEncoderFactory().

These factories have addEncoder() or addDecoder() functions that allow you to associate a properly configured instance of an encoder or decoder with each FOM object or interaction class. When a publisher, reflected object, or interaction needs an instance of an encoder or decoder for its FOM class, it requests one by calling a factory's createEncoder() or createDecoder(). These functions return a new'ed encoder or decoder that is a clone of the instance you have associated with the class using addEncoder() or addDecoder(). Clones are created using their the encoder or decoder's virtual clone() function.

The factory classes also have encoder() and decoder() functions that return pointers to the encoder or decoder that is currently registered for a particular FOM class. The optional second argument to these functions allows you to indicate what you want to happen when there is no encoder or decoder associated with a class. Use DtTrue to indicate that you would like the encoder or decoder associated with the most specific superclass for which one has been registered. Use DtFalse to indicate that you want NULL to be returned when there is no encoder or decoder registered for exactly the FOM class you specified.

When you want to register your own encoding or decoding functions for a particular FOM class, use a factory's encoder() or decoder() function to get a pointer to the object containing the function table for that class. Then use the encoder or decoder's mutators to register your own functions:

```
RTI::InteractionClassHandle handle =
   rtiAmb->getInteractionClassHandle("WeaponFire");
DtInteractionDecoder* fireDecoder =
   fomMapper->interactionDecoderFactory()->decoder(handle);
fireDecoder->addDecoder("RateOfFire", myFunc);
```

When you use this method to register functions, you are associating new encoding, decoding, or checking functions only with a single FOM class at a time. If you would like to add such functions to the encoders and decoders for a class and all of its subclasses, you can use one of the following member functions of one of the factory classes:

- addAttributeEncoder()
- addAttributeDecoder()
- addAttributeChecker()
- addParameterEncoder()
- addParameterDecoder().

These functions takes the name (or handle) of a FOM class, the name (or handle) of an attribute or parameter, a function to register, and a boolean flag indicating whether the function should apply to subclasses of the indicated FOM class (default is DtTrue).

For example, to register a new encoding function for the **Position** attribute of the *BaseEntity* RPR FOM class and all of its subclasses:

```
fomMapper->stateEncoderFactory()->addAttributeEncoder("BaseEntity",
           "Position", myFunc, DtTrue);
```

You can replace any of the factories used by a FOM mapper using the set-X-Factory functions: For example, if you are creating a FOM mapper for a new FOM, you might create a subclass of *DtHlaStateEncoderFactory* whose constructor registers all of the desired encoder prototypes. You can tell the FOM mapper to use an instance of this subclass like this:

```
MyEncoderFactory* factory = new MyEncoderFactory(...);
fomMapper->setStateEncoderFactory(factory);
```

Alternatively, you can start with the empty factories, or the factories contained in some existing FOM mapper (like VR-Link's RPR FOM mapper), and just add your own encoders and decoders to them without ever creating new factory instances.

6.7.7. Choosing A *DtFomMapper*

When you construct a *DtExerciseConn*, you must choose a FOM Mapper for it to use. You can do this in one of several different ways:

- Pass a *DtFomMapper* instance to a *DtExerciseConn* constructor
- Pass the name of a shared library containing a function that creates the desired FOM mapper to a *DtExerciseConn* constructor
- Set the fallback FOM mapper creation function, using *DtExerciseConn*'s static setFomMapperCreator()
- Allow *DtExerciseConn* to create an empty FOM mapper and configure it after the *DtExerciseConn* constructor returns. (In fact, regardless of how you choose a FOM mapper, you can always modify or reconfigure it after the *DtExerciseConn* constructor returns.)

Passing A DtFomMapper Instance

The following *DtExerciseConn* constructor takes an instance of a *DtFomMapper.*

```
DtExerciseConn(const char* execName, const char* federateName,
           DtFomMapper* mapper = DtRprFomMapper::create(), const char*
           fedFileName = NULL);
```

For example, if you create a FOM mapper subclass called *MyFomMapper*, you might instruct the *DtExerciseConn* to use your FOM mapper as follows:

```
MyFomMapper mapper();
DtExerciseConn("VR-Link", "MyAppName", &mapper);
```

Note that the mapper argument has a default value, which is a pointer to a new'ed *DtRprFomMapper* - a FOM mapper implemented in VR-Link that maps to the Real-time Platform Reference FOM (RPR FOM). If you omit the FOM mapper argument, as in the examples in the protocol-independent section, the RPR FOM mapper is used.

If you pass a value of NULL as the mapper argument to this constructor, VR-Link uses the FOM mapper creation function last passed to setFomMapperCreator() to create a FOM mapper. The default is a function that creates a *DtEmptyFomMapper.*

Passing the Name of A Shared Library

The following *DtExerciseConn* constructor takes the name of a shared library (DSO or DLL).

```
DtExerciseConn(const char* execName, const char* federateName,
   const char* dsoName, const char* fedFileName = NULL,
   void* fomMapInitData = NULL);
```

When you use this constructor, `dsoName` must be the name of a shared library (DLL or DSO) that contains definitions for the following functions:

```
DtFomMapper* DtCreateFomMapper(void* usr);
void DtDeleteFomMapper(DtFomMapper* mapper);
```

DtCreateFomMapper() is called by the *DtExerciseConn* after it opens the shared library. It should return a pointer to an instance of a *DtFomMapper* (or a subclass) that you want to use. DtDeleteFomMapper() is a function that will be called from within the *DtExerciseConn* destructor, which should delete the FOM mapper instance that you provided.

When you specify the name of your shared library, you can omit the filename extension (.so or .dll) to provide platform independence.

These two function definitions that you provide in a shared library should have C linkage. You can insure this, by wrapping `extern "C" {}` around your function definitions:

For example:

```
extern "C"
{

DtFomMapper* DtCreateFomMapper(void* usr)
{
   return new MyFomMapper();
}

void DtDeleteFomMapper(DtFomMapper* mapper)
{
   delete mapper;
}

}
```

You can specify the full path to your shared library within the `dsoName` argument to the *DtExerciseConn* constructor, or make sure that it is in your current shared library search path.

The final argument to the *DtExerciseConn* constructor that takes a shared library name, is a `void*` that can point to any arbitrary data that is required by the DtCreateFomMapper() function defined in your FOM mapper shared library. The pointer that is passed to the *DtExerciseConn* constructor as fomMapInitData will be passed as the usr argument to the DtCreateFomMapper() function defined in your shared library. In the example above, we simply ignored this usr data.

If VR-Link cannot successfully open the specified shared library, or if VR-Link cannot find the symbols DtCreateFomMapper() and DtDeleteFomMapper() within it, *DtExerciseConn* acts as though you explicitly passed a NULL FOM mapper: VR-Link uses the function last passed to DtExerciseConn::setFomMapperCreator() to create a FOM mapper. The default is a function that creates a *DtEmptyFomMapper.*

Please see the *myFomMap* example under *./examples/extend/myFomMap* for a demonstration of the process of creating a FOM mapper shared library.

The Fallback FOM Mapper Creation Function

DtExerciseConn has a fallback FOM Mapper creation function that is used when you pass NULL as the mapper argument to the *DtExerciseConn* constructor, or when an invalid FOM mapper shared library is passed as the dsoName argument.

You can set the fallback FOM Mapper creation function by using *DtExerciseConn*'s static setFomMapperCreator() member function before creating a *DtExerciseConn* instance. A FOM mapper creation function has the same prototype as the DtCreateFomMapper() function shown above. The default fallback function is DtEmptyFomMapper::create() - which creates an instance of *DtEmptyFomMapper* (defined in *emptyFomMap.h*). A *DtEmptyFomMapper* is a clean slate - a FOM mapper that contains no mappings between VR-Link classes and FOM classes.

Configuring A FOM Mapper After Constructing A *DtExerciseConn*

A *DtFomMapper* can be configured using its mutator functions after the *DtExercise-Conn* constructor returns. You can use this capability to add a few extra mappings to a FOM mapper that has been passed to the *DtExerciseConn*, or constructed from a shared library.

Alternatively, you can do all of your FOM configuration this way. Pass NULL as the mapper argument to the *DtExerciseConn* constructor, and start with the default *DtEmp-tyFomMapper* that will be created in this case (assuming that you have not set the fall-back FOM mapper creation function to something other than DtEmptyFomMapper::create()).

Using Different RPR FOM Versions

As we mentioned earlier, VR-Link contains built-in support for the Real-Time Platform Reference FOM (RPR FOM). We accomplished this by building a class called *DtRpr-FomMapper* - a *DtFomMapper* which self-registers mappings for RPR FOM classes, attributes and parameters. *DtRprFomMapper* is defined in *rprFomMap.h*.

The *DtRprFomMapper* constructor takes an optional version argument that indicates which version of the RPR FOM you would like to use. See release documentation for currently supported versions.

For example, to tell VR-Link to use a FOM mapper configured for RPR FOM version 0.7, do the following:

```
DtExerciseConn conn("VR-Link", "MyAppName", new DtRprFomMapper(0.7));
```

Because *DtRprFomMapper* contains information about all RPR FOM classes, attributes, and parameters, using it within your application can make your executable fairly large. To help alleviate this problem, we have created an alternate FOM mapper called *DtSimpleRprFomMapper*, defined in *simRprFomMap.h*. *DtSimpleRprFomMapper* has mappings for only a subset of the RPR FOM, namely those classes that are needed by VR-Link's examples and other MÄK products such as the MÄK Stealth and MÄK PVD:

- *BaseEntity* and all of its many subclasses
- *EmbeddedSystem.EmitterSystem*
- *EmitterBeam* and its subclasses
- *WeaponFire*
- *MunitionDetonation*
- *Collision*
- *ViewControl*
- *LgrControl*.

If you are using only these classes, passing an instance of *DtSimpleRprFomMapper* to your *DtExerciseConn* constructor instead of the default *DtRprFomMapper* will signifi-cantly reduce executable size.

If you are using just a few additional classes, you can create a *DtSimpleRprFomMapper*, and manually add mappings for the other classes that you need.

Deriving Your Own DtFomMapper

We mentioned in "Choosing A DtFomMapper," on page 6-32, that one way that you might configure your *DtFomMapper*, is by creating a subclass of *DtFomMapper* that self-registers the desired mapping information, and telling the *DtExerciseConn* to use an instance of your subclass. This is a way to group your configuration code all in one place. In fact, this what we have done in VR-Link with our *DtRprFomMapper* class for the RPR FOM.

When you create a *DtFomMapper* subclass, you should usually derive from *DtEmptyFomMapper*, rather than directly from *DtFomMapper*. *DtEmptyFomMapper* initializes all factories and lists for you, so that you only need to add mappings within your subclass.

Because we wanted to allow the possibility of creating a *DtFomMapper* before a *DtExerciseConn* (and passing the *DtFomMapper* to the *DtExerciseConn*), the *DtFomMapper* constructor does not take a *DtExerciseConn* as an argument. However, a *DtFomMapper* needs a *DtExerciseConn* in order to fully initialize itself.

For this reason, the *DtFomMapper* constructor does very little. Most initialization occurs within the virtual init() function, which is called by the *DtExerciseConn* constructor after it has read the FED file. The *DtExerciseConn* passes a pointer to itself to init(), so that it is available to the *DtFomMapper* during its real initialization work.

In your derived *DtFomMappers*, you will need to break things up this way as well. Your constructor should be basically empty. All of your registration of mapping code belongs in your implementation of the virtual init() function. From within your init() function, first call down to the base class's (usually *DtEmptyFomMapper's*) version of init. Then perform your specific initialization, such as addition of encoders, decoders, and class mappings. See *./examples/extend/myFomMap* for an example.

FOM Agility Examples

This chapter describes examples of FOM mapping and extending RPR FOM classes:

7.1. VR-Link FOM Mapping and Extension Examples

VR-Link provides examples of how to do various FOM mapping tasks, and of how to extend the top-level API to work with new FOM concepts. These examples are in several subdirectories of *./examples/extend*.

- The *myFomMap* example shows how to create a FOM mapper for a new FOM, but assumes that the new FOM contains concepts already covered by VR-Link's API (so no extensions are necessary). It also demonstrates how to put the new FOM mapper into a shared library that can be loaded by *DtExerciseConn*, and by MÄK applications like MÄK Stealth, MÄK Logger, and MÄK Plan View Display. The shared library works with the FED file called *MyFomMap.fed* that is in the VR-Link root directory.
- The *addAttr* example demonstrates how to extend the RPR FOM by adding new attributes or parameters to existing classes, particularly when this requires extending VR-Link's API to work with them.
- The *testInter* and *testObj* examples demonstrate how to add a new interaction or object class to your FOM, including extending VR-Link's API to work with them.
- The *testSimpInter* example demonstrates a method of working with new interaction classes that does not involve VR-Link's encoder and decoder classes. This implementation is simpler in some cases, but less extensible and less efficient. We do not provide a walk-through of this example. See comments in the source files for details.

All the examples except *myFomMap* create executables that work with the *VrlExtend.fed* FED file that is in the VR-Link root directory.

In the following sections, we walk through the examples, explaining the code we've written.

7.2. Creating Mappings from New FOMs to VR-Link Classes

The *myFomMap* example shows how to create a FOM mapper for a new FOM, but assumes that the new FOM contains concepts already covered by VR-Link's API (so no extensions are necessary). It also demonstrates how to put the new FOM mapper into a shared library that can be loaded by *DtExerciseConn*, and by MÄK applications like MÄK Stealth, MÄK Logger, and MÄK PVD. The shared library works with the FED file called *MyFomMap.fed* that is in the VR-Link root directory.

For this example, we created a new FED file called *MyFomMap.fed*, representing a new, simple FOM. The MyFomMap FOM contains one object class called *Vehicle*, and one interaction class called *Shoot*. The FED file is in the VR-Link root directory.

Vehicle has two attributes:

- VehicleType - an integer that represents a vehicle's type as either 0 for an M1A1 tank, and 1 for a T72
- GeocLoc - an array of three doubles that represents a vehicle's position in geocentric coordinates.

These attributes are mapped to *DtEntityStateRepository*'s concepts of entity type and location.

Shoot has two parameters: **Shooter** and **Shootee**. Both are character strings representing the HLA object names of the attacker and target respectively. These parameters will be mapped to *DtFireInteraction*'s concepts of attacker ID and target ID respectively.

This FOM uses different class, attribute, and parameter names than the RPR FOM, but uses similar data representations to keep our type conversion examples simple. (Though in one case - the VehicleType attribute, our data representation is very different to demonstrate that this can be done).

For each of these FOM classes, we:

1. Create an encoder class and a decoder class whose constructors self-register the necessary encoding and decoding functions for each attribute and parameter (and checking functions for each attribute as well).
2. Register these encoders and decoders with a FOM mapper along with other class mapping information.
3. Implement the two global functions required for FOM mapper shared libraries.

7.2.1. Writing the Encoder and Decoder Classes

ShootEncoder is defined in *shootEnc.h* and *shootEnc.cxx* as follows:

```
class ShootEncoder : public DtInteractionEncoder
{
public:

   // Constructor
   ShootEncoder(DtExerciseConn* exConn, DtInterClassDesc* classDesc);
```

```
   // Destructor
   virtual ~ShootEncoder();

protected:

   // Individual parameter encoding functions - one for each parameter.
   // These macros expand to look like this:
   // static void encodeShooter(const DtFireInteraction& inter,
   //    RTI::ParameterHandleValuePairSet* params,
   //    RTI::ParameterHandle paramHandle);
   DtDECLARE_SHOOT_PARAM_ENCODER(Shooter);
   DtDECLARE_SHOOT_PARAM_ENCODER(Shootee);
};

ShootEncoder::ShootEncoder(
   DtExerciseConn* exConn,
   DtInterClassDesc* classDesc) :
   DtInteractionEncoder(exConn, classDesc)
{
   // Register the individual encoding functions for Test's parameters with
   // object. The encoding functions have names like encodeShooter and
   // encodeShootee. The macro expands to look like this:
   // addEncoder("Shooter", (DtParameterEncoder) encodeShooter);
   // addEncoder("Shootee", (DtParameterEncoder) encodeShootee);
   DtADD_PARAM_ENCODER(Shooter);
   DtADD_PARAM_ENCODER(Shootee);
}

// Encoding functions for the parameters

void ShootEncoder::encodeShooter(const DtFireInteraction& inter,
   RTI::ParameterHandleValuePairSet* params,
   RTI::ParameterHandle paramHandle)
{
   // Get the value from the inspector function, convert to FOM representation
   // (in this case, a trivial conversion), and add it to the phvps.
   const char* val = inter.attackerId().string();
   params->add(paramHandle, val, strlen(val));
}

void ShootEncoder::encodeShootee(const DtFireInteraction& inter,
   RTI::ParameterHandleValuePairSet* params,
   RTI::ParameterHandle paramHandle)
{
   // Get the value from the inspector function, convert to FOM representation
   // (in this case, a trivial conversion), and add it to the phvps.
   const char* val = inter.targetId().string();
   params->add(paramHandle, val, strlen(val));
}
```

Defining the *ShootDecoder* Class

ShootDecoder is defined in *shootDec.h* and *shootDec.cxx* as follows:

```
class ShootDecoder : public DtInteractionDecoder
{
public:

   // Constructor
   ShootDecoder(DtExerciseConn* exConn, DtInterClassDesc* classDesc);

   // Destructor
   virtual ~ShootDecoder();

protected:

   // Individual parameter decoding functions - one for each parameter.
   // These macros expand to look like this:
   // static void decodeNumber(DtFireInteraction* inter,
   //    const RTI::ParameterHandleValuePairSet& params, int index);
   DtDECLARE_SHOOT_PARAM_DECODER(Shooter);
   DtDECLARE_SHOOT_PARAM_DECODER(Shootee);
};

ShootDecoder::ShootDecoder(
   DtExerciseConn* exConn,
   DtInterClassDesc* classDesc) :
   DtInteractionDecoder(exConn, classDesc)
{
   // Register the individual decoding functions for Shoot's parameters with
   // object. The decoding functions have names like decodeShooter and
   // decodeShootee. The macro expands to look like this:
   // addDecoder("Shooter", (DtParameterDecoder) decodeShooter);
   DtADD_PARAM_DECODER(Shooter);
   DtADD_PARAM_DECODER(Shootee);
}

// Decoding functions

void ShootDecoder::decodeShooter(DtFireInteraction* inter,
   const RTI::ParameterHandleValuePairSet& params,
   int index)
{
   RTI::ULong length = 0;
   // Get value in RTI representation into netVal.
   char* netVal = (char*)
      params.getValuePointer(index, length);

   // Must copy to a buffer, because string returned by getValuePointer may
   // not be NULL terminated.
   static char buff[100];
   strncpy(buff, netVal, length);
   buff[length] = 0;

   // In this case, we can implicitly convert from FOM representation (char*)
   // to VR-Link representation (DtGlobalObjectDesignator). No explicit
   // conversion is necessary. Just pass the value to the mutator function.
   inter->setAttackerId(buff);
}
```

```
void ShootDecoder::decodeShootee(DtFireInteraction* inter,
   const RTI::ParameterHandleValuePairSet& params,
   int index)
{
   RTI::ULong length = 0;
   // Get value in RTI representation into netVal.
   char* netVal = (char*)
      params.getValuePointer(index, length);

   // Must copy to a buffer, because string returned by getValuePointer may
   // not be NULL terminated.
   static char buff[100];
   strncpy(buff, netVal, length);
   buff[length] = 0;

   // In this case, we can implicitly convert from FOM representation (char*)
   // to VR-Link representation (DtGlobalObjectDesignator). No explicit
   // conversion is necessary. Just pass the value to the mutator function.
   inter->setTargetId(buff);
}
```

Defining the *VehicleDecoder* Class

On the object side, the *VehicleDecoder* class is in *vehicleDec.h* and *vehicleDec.cxx*:

```
class VehicleDecoder : public DtHlaStateDecoder
{
public:

   // default constructor
   VehicleDecoder(DtExerciseConn* exConn, DtObjClassDesc* classDesc);

   // destructor
   virtual ~VehicleDecoder();

protected:

   // Individual attribute decoding functions - one for each attribute.
   // These macros expand to look like this:
   // static void decodeVehicleType(DtEntityStateRepository* stateRep,
   //    const RTI::AttributeHandleValuePairSet& attrs, int pairSetIndex)

   DtDECLARE_VEHICLE_ATTR_DECODER(VehicleType);
   DtDECLARE_VEHICLE_ATTR_DECODER(GeocLoc);
};

VehicleDecoder::VehicleDecoder(
   DtExerciseConn* exConn,
   DtObjClassDesc* classDesc) :
   DtHlaStateDecoder(exConn, classDesc)
{
   // Register the individual decoding functions for Test's attributes with
   // the object. The decoding functions have names like decodeVehicleType
   // and decodeGeocLoc. The macro expands to look like this:
   // addDecoder("VehicleType", (DtAttributeDecoder) decodeVehicleType);
   DtADD_ATTR_DECODER(VehicleType);
   DtADD_ATTR_DECODER(GeocLoc);
}

// Decoding functions
```

```
void VehicleDecoder::decodeVehicleType(DtEntityStateRepository* stateRep,
   const RTI::AttributeHandleValuePairSet& attrs,
   int index)
{
   RTI::ULong length = 0;
   // Get value from RTI representation into netVal.
   DtNetInt32* netVal = (DtNetInt32*) attrs.getValuePointer(index, length);

   // Convert from FOM representation to VR-Link representation.

   // Byte swapping occurs if necessary, when converting from "Net" types to
   // native types.
   int nativeVal = (DtInt32) *netVal;
   DtEntityType entityType;
   if (nativeVal == 0)
   {
      // It's an M1.
      entityType = DtEntityType(1, 1, 225, 1, 1, 0, 0);
   }
   else if (nativeVal == 1)
   {
      // It's a T72.
      entityType = DtEntityType(1, 1, 222, 1, 2, 0, 0);
   }

   // pass the value to the mutator function
   stateRep->setEntityType(entityType);
}

void VehicleDecoder::decodeGeocLoc(DtEntityStateRepository* stateRep,
   const RTI::AttributeHandleValuePairSet& attrs,
   int index)
{
   RTI::ULong length = 0;
   // Get value from RTI representation into netVal.
   DtNet64Vector* netVal =
      (DtNet64Vector*) attrs.getValuePointer(index, length);

   // In this case, the FOM representation (DtNet64Vector) can be implicitly
   // converted to VR-Link representation (DtVector)
   stateRep->setLocation(*netVal);
}
```

Defining the *VehicleEncoder* Class

The *VehicleEncoder* class is defined in *vehicleEnc.h* and *vehicleEnc.cxx*:

```
class VehicleEncoder : public DtHlaStateEncoder
{
public:

   // default constructor
   VehicleEncoder(DtExerciseConn* exConn, DtObjClassDesc* classDesc);

   // destructor
   virtual ~VehicleEncoder();

protected:
```

```
   // Individual attribute encoding functions - one for each attribute.
   // This macro expands to look like this:
   // static void encodeVehicleType(const DtEntityStateRepository& rep,
   //    RTI::AttributeHandleValuePairSet* attrs,
   //    RTI::AttributeHandle attrHandle)
   DtDECLARE_TEST_ATTR_ENCODER(VehicleType);
   DtDECLARE_TEST_ATTR_ENCODER(GeocLoc);

   // Individual attribute checking functions - one for each attribute.
   // This macro expands to look like this:
   // static DtBoolean needNumber(const DtEntityStateRepository& rep,
   //    const DtEntityStateRepository& asSeenByRemote);

   DtDECLARE_TEST_ATTR_CHECKER(VehicleType);
   DtDECLARE_TEST_ATTR_CHECKER(GeocLoc);
};

VehicleEncoder::VehicleEncoder(
   DtExerciseConn* exConn,
   DtObjClassDesc* classDesc) :
   DtHlaStateEncoder(exConn, classDesc)
{
   // Register the individual encoding and checking functions for Vehicle's
   // attributes with the object. The encoding functions have names like
   // encodeGeocLoc, while checking functions have names like needGeocLoc.
   // The macros expands to look like this:
   // addEncoder("VehicleType", (DtAttributeEncoder) encodeVehicleType);
   // addChecker("VehicleType", (DtAttributeChecker) needVehicleType);
   DtADD_ATTR_ENCODER(VehicleType);
   DtADD_ATTR_ENCODER(GeocLoc);

   DtADD_ATTR_CHECKER(VehicleType);
   DtADD_ATTR_CHECKER(GeocLoc);
}

// Encoding and checking functions

DtBoolean VehicleEncoder::needVehicleType(
   const DtEntityStateRepository& stateRep,
   const DtEntityStateRepository& asSeenByRemote)
{
   return (DtBoolean) !(stateRep.entityType() ==
               asSeenByRemote.entityType());
}

DtBoolean VehicleEncoder::needGeocLoc(
   const DtEntityStateRepository& stateRep,
   const DtEntityStateRepository& asSeenByRemote)
{
   return (DtBoolean) !(stateRep.location() == asSeenByRemote.location());
}

void VehicleEncoder::encodeVehicleType(const DtEntityStateRepository& rep,
   RTI::AttributeHandleValuePairSet* attrs,
   RTI::AttributeHandle attrHandle)
{
   // Get value using inspector
   DtEntityType type = rep.entityType();

   // Convert to FOM representation:
```

```
   DtNetInt32 netVal = 0;
   if (type == DtEntityType(1, 1, 225, 1, 1, 0, 0))
   {
      // M1A1
      netVal = 0;
   }
   else if (type == DtEntityType(1, 1, 222, 1, 2, 0, 0))
   {
      // T72
      netVal = 1;
   }

   // Add to ahvps
   attrs->add(attrHandle, (char*) &netVal, sizeof(netVal));
}

void VehicleEncoder::encodeGeocLoc(const DtEntityStateRepository& rep,
   RTI::AttributeHandleValuePairSet* attrs,
   RTI::AttributeHandle attrHandle)
{
   // Get the value from the inspector function, convert to FOM representation
   // (in this case, a trivial conversion), and add it to the phvps.
   DtVector vec = rep.location();
   DtNet64Vector netVal(vec);

   // Add to ahvps
   attrs->add(attrHandle, (char*) &netVal, sizeof(netVal));
}
```

7.2.2. Configuring the FOM Mapper

There are several different ways that we can configure a FOM mapper to deal with this new FOM. For example, we could allow the *DtExerciseConn* constructor to create an empty FOM mapper, and then add mappings to that FOM mapper later.

However, what we will do here is create a subclass of *DtFomMapper* called *MyFom-Mapper*, whose implementation of the virtual init() function self registers all of the mappings we want. We can then either create an instance of this class and pass it to the *DtExerciseConn* constructor, or create a shared library containing a function called DtCreateFomMapper() that returns a new'ed instance of *MyFomMapper*, and pass the name of the shared library to the *DtExerciseConn* constructor. In this example, we demonstrate the shared library option.

Regardless of how an instance of a *DtFomMapper* is passed to a *DtExerciseConn* constructor, *DtExerciseConn* calls init() on that FOM mapper after reading the FED file and initializing the RTI.

The *MyFomMapper* class is defined in *myFomMap.h* and *myFomMap.cxx*. The code is included below. Notice that *MyFomMapper* is derived from *DtEmptyFomMapper*. Within MyFomMapper::init(), we call down to the DtEmptyFomMapper::init(), which initializes the FOM mapper with empty factories and tables. We then add our own mappings to the tables, including adding instances of our encoders and decoders to the FOM mapper's encoder and decoder factories.

```
class MyFomMapper : public DtEmptyFomMapper
```

```
{
public:

   // Default constructor - most initialization is delayed until init() is
   // called.
   MyFomMapper();

   // Destructor
   virtual ~MyFomMapper();

   // Virtual function override. init actually initializes most data. It is
   // called from DtExerciseConn constructor. Required.
   virtual void init(DtExerciseConn* conn);
};

MyFomMapper::MyFomMapper() :
   DtEmptyFomMapper()
{
   // Real initialization takes place in virtual init() function,
   // when a DtExerciseConn is available.
}

void MyFomMapper::init(DtExerciseConn* exConn)
{
   // Call base class init. This sets myExConn, and initializes all factories
   // to default or empty factories (no encoders or decoders registered, no
   // mappings between FOM interaction classes and VR-Link interaction
   // classes.
   DtEmptyFomMapper::init(exConn);

   // Add encoders and decoders for our Vehicle object class and Shoot
   // interaction class. When DtEntityPublisher, DtReflectedEntity, and
   // DtFireInteraction ask the FOM mapper for encoders and decoders to use,
   // this FOM mapper will return clones of the encoders and decoders we're
   // registering here.
   DtObjClassDesc* objClass = exConn->fom()->objClassByName("Vehicle");
   if (objClass)
   {
      stateEncoderFactory()->addEncoder(
         objClass->handle(), new VehicleEncoder(exConn, objClass));
      stateDecoderFactory()->addDecoder(
         objClass->handle(), new VehicleDecoder(exConn, objClass));
   }

   DtInterClassDesc* interClass = exConn->fom()->interClassByName("Shoot");
   if (interClass)
   {
      interactionEncoderFactory()->addEncoder(
         interClass->handle(), new ShootEncoder(exConn, interClass));
      interactionDecoderFactory()->addDecoder(
         interClass->handle(), new ShootDecoder(exConn, interClass));
   }

   // Set up mappings for publishing. Indicate the FOM class to use for each
   // kind of DtObjectPublisher, and each kind of DtInteraction.
   setObjectClassToChoose("DtEntityPublisher", "Vehicle");
   setInteractionClassToChoose("DtFireInteraction", "Shoot");

   // Set up mappings for subscribing. Indicate which FOM class each kind of
   // DtReflectedObjectList should subscribe to, and which FOM class each kind
```

```
   // of DtInteraction's addCallback function should subscribe to.
   setObjectClass("DtReflectedEntityList", "Vehicle");
   setInteractionClass("DtFireInteraction", "Shoot");

   // Set up mappings for creating DtInteraction instances. Indicate what
   // kind of DtInteraction to create to represent each FOM interaction
   // class.
   interactionFactory()->addCreator("Shoot", DtFireInteraction::create);
}
```

The file *createFomMap.cxx* contains definitions for the functions that are required by VR-Link if you are creating a FOM mapper shared library. When you pass the name of a shared library to a *DtExerciseConn* constructor, it opens the shared library and looks for the functions DtCreateFomMapper() and DtDeleteFomMapper(). Here, DtCreateFomMapper() just returns a new instance of our *MyFomMapper* class. These two functions must have C linkage, so we enclose their definitions within `extern "C" {}`.

```
extern "C"
{

DtFomMapper* DtCreateFomMapper(void* usr)
{
   return new MyFomMapper();
}

void DtDeleteFomMapper(DtFomMapper* mapper)
{
   delete mapper;
}

}
```

When you do a “make” in this example directory, you will build a shared library called *myFomMap.so* or *myFomMap.dll*. You can test this library with any VR-Link example that accepts a path to a FOM mapping shared library through the `-f` option. You can omit the .so or .dll, so that the same command can be used on a PC or in UNIX. Remember to also specify a federation execution name of MyFomMap, using the `-x` option, so that the *MyFomMap.fed* file is used.

For example, from the *./binRPR* directory, you can run:

```
f18 -f ../examples/extend/myFomMap/myFomMap -x MyFomMap
```

Alternatively, you can write your own application that passes the name of the shared library to the *DtExerciseConn*. Again, make sure that you use the right FED file:

```
DtExerciseConn conn("MyFomMap", "MyApp", "myFomMap");
```

7.3. Adding Attributes and Parameters to RPR FOM Classes

The code in the *addAttr* example explains the steps you need to take when you add a new attribute or parameter to existing classes in your FOM. In this example, we assume that the new attribute and parameter represent concepts not present in the VR-Link top-level API. Therefore, we must first extend the API to provide accessors for the new concepts, then configure the FOM Mapper so that it can map between the new FOM elements, and our API extensions.

The FOM elements that we have added for this example are:

- A new attribute called **Mass** has been added to the *BaseEntity* object class.
- A new parameter called **Temperature** has been added to the *WeaponFire* interaction class.

These additions are in the file *VrlExtend.fed*. To work with these additions do the following:

1. Create a subclass of *DtEntityStateRepository* with accessors for the new Mass attribute. (See *myEsr.h.*)
2. Create a subclass of *DtFireInteraction* with accessors for the new Temperature parameter. (See *myFireInter.h.*)
3. Add mappings for the new attribute and parameter to the FOM mapper, and tell the FOM mapper to instantiate *MyFireInteraction* when an interaction of class *WeaponFire* is received. (See *configFomMap.h* and *configFomMap.cxx.*)
4. Tell *DtReflectedEntity* and *DtEntityPublisher* to use *MyEntityStateRepository* instead of the standard *DtEntityStateRepository.* (See *main.cxx*.)

7.3.1. Creating A Subclass of *DtEntityStateRepository*

Applications typically get state information about entities from a *DtEntityStateRepository.* This class has mutators and inspectors that provide access to many components of an entity's state, but the concept of mass is not one of these components. So in the file *myEsr.h*, we extend the *DtEntityStateRepository* API by deriving a class called *MyEntityStateRepository*:

```
class MyEntityStateRep : public DtEntityStateRepository
{
public:

   // Set/Get the value of mass
   virtual void setMass(float mass);
   virtual float mass() const;

   // Virtual function override: print the state rep's data.
   virtual void printData() const;

public:

   // Create a MyEntityStateRep. Caller is reponsible for deletion.
   static DtEntityStateRepository* create();
```

```
protected:

   float myMass;
};
```

We added an inspector and mutator function for the concept of mass, and overrode the virtual printData() function, so that mass will be printed along with the other state data. Finally, we provided a static create() function that you can register with other VR-Link classes, so that they can create an instance of our new state repository.

These functions have straightforward implementations:

```
inline void MyEntityStateRep::setMass(float temp)
{
   myMass = temp;
}

inline float MyEntityStateRep::mass() const
{
   return myMass;
}

inline void MyEntityStateRep::printData() const
{
   DtEntityStateRepository::printData();
   printf("Mass: %lf\n", mass());
}

inline DtEntityStateRepository* MyEntityStateRep::create()
{
   return new MyEntityStateRep();
}
```

7.3.2. Creating A Subclass of *DtFireInteraction*

In *myFireInter.h*, we extend the API provided by the *DtFireInteraction* class, to add accessor functions for the concept of temperature. Again, we override printData(), and provide a static create() function:

```
class MyFireInteraction : public DtFireInteraction
{
public:

   // Set/Get the value of temperature
   virtual void setTemperature(float temp);
   virtual float temperature() const;

   // Virtual function override: print the interaction's data.
   virtual void printData() const;

public:

   // Create a MyFireInteraction. Caller is reponsible for deletion.
   static DtInteraction* create();

protected:
```

```
   float myTemperature;
};
```

7.3.3. Adding Attribute and Parameter Mappings to the FOM Mapper

Next, we need to provide code that can map between the FOM attribute and parameter, and our API extensions. This code is in *configFomMap.cxx*. The function configFomMapper() takes a pointer to a FOM mapper as an argument and configures that FOM Mapper so that it contains our new mappings. The assumption is that the FOM Mapper that will be passed to us already contains the standard RPR FOM mappings, so that we only need to add our new mappings. We don't have to worry about adding mappings for the rest of the FOM.

In configFomMapper(), listed below, we first register our *MyFireInteraction* class's create function with the FOM mapper's interaction factory, so that it knows that this is the *DtInteraction* subclass that it should instantiate to represent incoming interactions of the FOM class *WeaponFire*. We don't need to do anything similar for objects here, but we will see later that we need to tell our *DtEntityPublisher* and *DtReflectedEntity* classes that they should be using our *MyEntityStateRepository* class to store the state of the objects that they represent.

The rest of configFomMapper() shows the registration of encoding, decoding, and checking functions for our new attribute and parameter. Decoding functions take data from FOM representation, and pass it to interaction or state repository mutator functions (after performing any necessary format conversions). Encoding functions obtain data using interaction or state repository inspector functions, and add them to outgoing state updates in FOM representation (again after performing any necessary conversions). Checking functions check whether an attribute's update condition has been met, usually by comparing the data in a current state repository, and a state repository that represents the state that would be seen by remote federates based on updates that we have sent.

configFomMapper() looks like this:

```
void configFomMapper(DtFomMapper* mapper)
{
   // Register MyFireInteraction's creator function with the FOM Mapper's
   // interaction factory, so that VR-Link creates an instance of
   // MyFireInteraction instead of DtFireInteraction to represent an incoming
   // WeaponFire interaction.
   mapper->interactionFactory()->addCreator(
      "WeaponFire", MyFireInteraction::create);

   // Add your encoding function for the "Temperature" parameter to the
   // prototype encoders for WeaponFire and any subclasses if they existed.
   mapper->interactionEncoderFactory()->addParameterEncoder(
      "WeaponFire", "Temperature",
      (DtInteractionEncoder::DtParameterEncoder) encodeTemperature, DtTrue);

   // Add your decoding function for the "Temperature" parameter to the
   // prototype decoders for WeaponFire and any subclasses if they existed.
   mapper->interactionDecoderFactory()->addParameterDecoder(
      "WeaponFire", "Temperature",
```

```
      (DtInteractionDecoder::DtParameterDecoder) decodeTemperature, DtTrue);

   // Add your encoding function for the "Mass" attribute to the prototype
   // encoders for BaseEntity and all of its subclasses.
   mapper->stateEncoderFactory()->addAttributeEncoder(
      "BaseEntity", "Mass",
      (DtHlaStateEncoder::DtAttributeEncoder) encodeMass, DtTrue);

   // Add your decoding function for the "Mass" attribute to the prototype
   // decoders for BaseEntity and all of its subclasses.
   mapper->stateDecoderFactory()->addAttributeDecoder(
      "BaseEntity", "Mass",
      (DtHlaStateDecoder::DtAttributeDecoder) decodeMass, DtTrue);

   // Add your checking function for the "Mass" attribute to the prototype
   // encoders for BaseEntity and all of its subclasses.
   mapper->stateEncoderFactory()->addAttributeChecker(
      "BaseEntity", "Mass",
      (DtHlaStateEncoder::DtAttributeChecker) needMass, DtTrue);
}
```

Adding Encoding, Decoding, and Checking Functions

The encoding, decoding, and checking functions that we register with the FOM Mapper are defined as follows. By using VR-Link's "Net" types in our functions, byte swapping occurs automatically during conversions to and from native types.

```
// Encoding function to be used for the "Mass" attribute.
void encodeMass(const MyEntityStateRep& stateRep,
   RTI::AttributeHandleValuePairSet* avList,
   RTI::AttributeHandle handle)
{
   // Get the value using  stateRep's mass(), and add it to the avList.
   double mass = stateRep.mass();
   DtNetFloat64 netVal(mass);
   avList->add(handle, (char*) &netVal, sizeof(DtNetFloat64));
}

// Decoding function to be used for the "Mass" attribute.
void decodeMass(MyEntityStateRep* stateRep,
   const RTI::AttributeHandleValuePairSet& avlist,
   int pairSetIndex)
{
   // Get the value from the avList, and pass it to stateRep's setMass().
   DtNetFloat64 netVal;
   RTI::ULong length = 0;
   avlist.getValue(pairSetIndex, (char* )&netVal, length);
   stateRep->setMass((double) netVal);
}

// Checking function to be used for the "Mass" attribute (checks whether
// update condition has been met).
DtBoolean needMass(
   const MyEntityStateRep& stateRep,
   const MyEntityStateRep& asSeenByRemote)
{
   // Just compare the masses in the two state repositories.
   return (DtBoolean) (stateRep.mass() != asSeenByRemote.mass());
}
```

```
// Encoding function to be used for the "Temperature" parameter.
void encodeTemperature(const MyFireInteraction& fire,
   RTI::ParameterHandleValuePairSet* pvList,
   RTI::ParameterHandle handle)
{
   // Get the value using the interaction's temperature() and add it to the
   // avList.
   double temp = fire.temperature();
   DtNetFloat64 netVal(temp);
   pvList->add(handle, (char*) &netVal, sizeof(netVal));
}

// Decoding function to be used for the "Temperature" parameter.
void decodeTemperature(MyFireInteraction* fire,
   const RTI::ParameterHandleValuePairSet& pvlist,
   int pairSetIndex)
{
   // Get the value from the pvlist, and pass it to the interaction's
   // setTemperature().
   DtNetFloat64 netVal;
   RTI::ULong length = 0;
   pvlist.getValue(pairSetIndex, (char* )&netVal, length);
   fire->setTemperature((float) netVal);
}
```

In *main.cxx*, we construct a *DtExerciseConn*, telling it to use our modified FED file, *VrlExtend.fed*. By not passing a *DtFomMapper* argument, we are indicating that the default RPR FOM Mapper should be used:

```
DtExerciseConn conn("VrlExtend", "addAttr");
```

After the *DtExerciseConn* is created, we pass its FOM mapper to the configFomMapper() function that we wrote in *configFomMap.cxx*. This adds our new mappings:

```
configFomMapper(conn.fomMapper());
```

Telling VR-Link to Use the New Classes

The last thing that we need to do is to get our *DtEntityStateRepository* extensions plugged into VR-Link. We tell the *DtReflectedEntity* and *DtEntityPublisher* classes that they should use instances of *MyEntityStateRepository*, rather than the default *DtEntityStateRepository*, to store their objects' state. We do this by using those classes' static member function setStateRepCreator(), as follows:

```
DtReflectedEntity::setStateRepCreator(MyEntityStateRep::create);
DtEntityPublisher::setStateRepCreator(MyEntityStateRep::create);
```

Later, when we create a *DtEntityPublisher* to manage sending updates for a locally simulated entity, we can cast the state repository returned by its esr() function to a *MyEntityStateRepository*, and use its setMass() function to set the entity's mass:

```
// Create an entity publisher
DtEntityPublisher pub(DtEntityType(1, 1, 225, 1, 1, 0, 0),  &conn);

// We've told the publisher to use a MyEntityStateRep as its state
// repository, so this cast should be safe.
MyEntityStateRep* esr = (MyEntityStateRep*) pub.esr();
```

```
esr->setMass(215.0);
```

On the receiving side, we can cast a *DtReflectedEntity*'s state repository to a *MyEntityStateRepository* as well, and inspect the entity's mass using its mass() member. In the example, we just call its printData() virtual function, for which the cast isn't really necessary:

```
// We've told DtReflectedEntity to use a MyEntityStateRep as its
// state repository, so this cast should be safe.
MyEntityStateRep* esr = (MyEntityStateRep*) ent->esr();
esr->printData();
```

Using the New Parameters

To send a fire interaction that includes our temperature parameter, just create an instance of *MyFireInteraction* and send it:

```
MyFireInteraction inter;
inter.setTemperature(150.0);
conn.sendStamped(inter);
```

To receive it, register a callback function with *DtFireInteraction* as usual.

```
// Register the callback on incoming Fire Interactions.
DtFireInteraction::addCallback(&conn, fireCb, NULL);
```

Within the callback, we can cast the *DtFireInteraction* pointer to a pointer to *MyFireInteraction*, and inspect the temperature using its temperature() member, or just use its virtual printData() function, which should print the temperature along with the other parameters:

```
// Callback function to be called when a Fire Interaction is received.
void fireCb(DtFireInteraction* inter, void*)
{
   printf("Received Fire Interaction!\n");
   inter->printData();
   printf("\n");
}
```

In this example, we did not provide static addCallback() and removeCallback() functions in the *MyFireInteraction* class, as most VR-Link interaction classes do. Had we done so, they would have allowed callback functions that take a *MyFireInteraction* pointer rather than a *DtFireInteraction* pointer. If we were able to use such specific callbacks, we could avoid a cast to *MyFireInteraction* within the callback, which is otherwise necessary to inspect subclass-specific data.

After you build the *addAttr* example, run two copies of it. You should see them communicating with each other. Each should print state and interaction information received from the other, including values for the new attribute and parameter.

7.4. Creating New Interaction Classes

The code in the *testInter* example explains the steps you need to take when adding a new interaction class to your FOM. We need to extend the VR-Link top-level API by adding a new kind of *DtInteraction*. Then we create mappings between the new FOM class and the new *DtInteraction* subclass.

For this example, we added an interaction class called *Test* to the *VrlExtend.fed* FED File. It has two parameters:

- **Number**, which is represented as a 32-bit float
- **Vector**, which is represented as an array of three 64-bit doubles.

To work with these additions we need to:

1. Create a new subclass of *DtInteraction* to represent the new kind of interaction. (See *testInter.h* and *testInter.cxx*.)
2. Create *TestEncoder* and *TestDecoder* classes, subclasses of *DtInteractionEncoder* and *DtInteractionDecoder* respectively. These classes contain encoding and decoding functions for the attributes of the *Test* interaction. (See *testDec.h*, *testDec.cxx*, *testEnc.h* and *testEnc.cxx*.)

7.4.1. Creating a New Interaction

The files *testInter.h* and *testInter.cxx* contain definitions for a new class called *TestInteraction*, and its member functions.

The class definition appears below. Like all interaction classes in VR-Link, it is derived from *DtInteractionWithEncDec*, rather than directly from *DtInteraction*. *DtInteraction* is rather general, and allows for a wide variety of subclass implementations, including those that do not involve our concept of encoders and decoders. *DtInteractionWithEncDec* is set up to deal with encoders and decoders, and we will be taking advantage of this.

TestInteraction includes inspector and mutator functions to access the values of the Number and Parameter attributes. We represent them in the same way as in the FOM, but this is not strictly necessary. We could perform a non-trivial conversion in FOM mapping code instead.

The *TestInteraction* class hard-codes its own FOM mapping information, rather than relying on the default behavior, which is to obtain it from the FOM mapper. We do this here, because it make things simpler, by saving us the step of configuring the FOM mapper with mappings for our new class. But this choice means that it would be more difficult to change the way this kind of interaction is represented in the FOM.

This choice manifests itself in the fact that we override the following virtual functions: interactionClassToUse(), createEncoder() and createDecoder(). The base class implementation of interactionClassToUse() asks the FOM mapper for the name of a FOM class to use. But *TestInteraction*'s implementation just returns the class name *Test*. The base versions of createEncoder() and createDecoder() ask the FOM mapper for instances of the kind of encoder and decoder that have been registered with it for use with the *Test* interaction class. But *TestInteraction*'s implementations of these functions just return a new'ed *TestEncoder* and *TestDecoder* instance respectively.

In addition, within TestInteraction::addCallback(), we inform the *DtExerciseConn*'s interaction factory that it should create an instance of the *TestInteraction* class to represent incoming interactions of FOM class *Test*. This is achieved using *DtInteractionFactory*'s addCreator() function.

```
class TestInteraction : public DtInteractionWithEncDec
{
public:

   // Blank Constructor - constructs a blank TestInteraction
   // ready to be filled out and sent by application code.
   // Required.
   TestInteraction();

   // Destructor
   // Not required by VR-Link, but should clearly be written if
   // this class has any dynamic data that needs to be cleaned up.
   virtual ~TestInteraction();

   // Copy constructor.
   // Not required by VR-Link, but needed if you will ever want to
   // copy TestInteractions.
   TestInteraction(const TestInteraction& orig);

   // Assignment operator.
   // Not required by VR-Link, but needed if you will ever want to
   // copy TestInteractions.
   TestInteraction& operator=(const TestInteraction& orig);

   // Virtual function override. Should return the name of this C++ class,
   // which is not necessarily the Interaction Class name from the FOM.
   // Required.
   virtual const char* name() const;

   // Virtual function override. Prints the data values in the interaction.
   // Not strictly required, but encouraged, since otherwise calling
   // print() on an instance of this class with merely print raw hex data.
   virtual void printData() const;

   // ************************************************************
   // Inspector and mutator functions for the interaction class' data.
   // These will look different for each DtInteraction subclass.
   // None are required by VR-Link, but the class will not be
   // particularly useful without a way to get data in and out in a
   // typesafe manner.
   // ************************************************************

   // Set/Get the parameter named "Number"
```

```
   virtual void setNumber(int num);
   virtual int number() const;

   // Set/Get the parameter named "Vector"
   virtual void setVector(const DtVector& vec);
   virtual const DtVector& vector() const;

public:

   // Create and return a new TestInteraction. Basically a wrapper
   // around the constructor. It is necessary because pointers to
   // static functions like this can be registered with a
   // DtInteractionFactory, but constructors can not.
   // Required.
   static DtInteraction* create();

   // Wrappers around DtExerciseConn's add/removeCallback functions. Not
   // required since the DtExerciseConn functions may be used instead, but
   // strongly encouraged, so that the non-typesafe cast can be limited to
   // this one place. In addition, within add-callback, we choose to register
   // the TestInteraction class with VR-Link, letting VR-Link know to create
   // one of these to represent the right FOM class.
   static void addCallback(DtExerciseConn* conn,
      TestInteractionCb cb, void* usr);
   static void removeCallback(DtExerciseConn* conn,
      TestInteractionCb cb, void* usr);

protected:

   // Virtual function override. Returns the name of the FOM class to use to
   // represent this interaction when sending. It is called from within
   // setExConn, when that function is called by DtExerciseConn::send.
   // Implementing this function is required unless you would like to rely on
   // the default behavior - which is to obtain the class to use from the FOM
   // Mapper. If you choose this option, you must configure the FomMapper so
   // that it can correctly provide this information.
   virtual char* interactionClassToUse(DtExerciseConn* exConn) const;

   // Virtual function override. Creates and returns a decoder to be used by
   // setFromPhvps to decode a ParameterHandleValuePairSet into this
   // interaction. Implementing this function is required, unless you would
   // like to rely on the default behavior - which is to obtain a new instance
   // of the right decoder from the FOM Mapper. If you choose this option,
   // you must configure the FOM Mapper so that it can correctly provide this
   // object.
   virtual DtInteractionDecoder* createDecoder(DtExerciseConn* exConn) const;


   // Virtual function override. Creates and returns an encoder to be used by
   // phvps to encode the data in this interaction into a
   // ParameterHandleValuePairSet. Implementing this function is required,
   // unless you would like to rely on the default behavior - which is to
   // obtain a new instance of the right encoder from the FOM Mapper. If you
   // choose this option, you must configure the FOM Mapper so that it can
   // correctly provide this object.
   virtual DtInteractionEncoder* createEncoder(DtExerciseConn* exConn) const;

protected:

   // Here, we store the actual data values, in native representation.
```

```
   int myNum;
   DtVector myVec;
};
```

Defining Member Functions

Some of the relevant member function definitions are provided below. See *testInter.cxx* for the full listing:

```
// Accessor functions are typically pretty straightforward.

void TestInteraction::setNumber(int num)
{
   myNum = num;
}

int TestInteraction::number() const
{
   return myNum;
}

void TestInteraction::setVector(const DtVector& vec)
{
   myVec = vec;
}

const DtVector& TestInteraction::vector() const
{
   return myVec;
}

void TestInteraction::addCallback(DtExerciseConn* conn,
   TestInteractionCb cb, void* usr)
{
   // Cast the TestInteractionCb to the more general DtReceiveInteractionCb
   // and pass it along with the FOM class name to the DtExerciseConn who
   // actually maintains the callback lists for all interaction classes.
   conn->addInteractionCallbackByName(
      "Test", (DtReceiveInteractionCb) cb, usr);

   // In addition, we choose to register the TestInteraction class with
   // VR-Link here, by associating TestInteraction's create function with the
   // right FOM class. By doing this here, we avoid making the user perform
   // this registration step.
   conn->interactionFactory()->addCreator("Test",
      TestInteraction::create);
}
char* TestInteraction::interactionClassToUse(DtExerciseConn* exConn) const
{
   return "Test";
}

DtInteractionDecoder* TestInteraction::createDecoder(
   DtExerciseConn* exConn) const
{
   return new TestDecoder(exConn, classDesc());
}

DtInteractionEncoder* TestInteraction::createEncoder(
```

```
   DtExerciseConn* exConn) const
{
   return new TestEncoder(exConn, classDesc());
}
```

Notice that the *TestInteraction* has decided that it will use instances of the classes *TestEncoder* and *TestDecoder* to map between its representation of its parameters, and the FOM representation.

7.4.2. Creating New Encoder and Decoder Functions

TestDecoder, defined in *testDec.cxx*, is derived from *DtInteractionDecoder*. Macros help out in the declaration and definition of the decoding functions for individual parameters. The "simple" implementation provided by the macros for a decoding function is to get a pointer to the parameter data from an RTI::ParameterHandleValuePairSet, cast it to a pointer to an appropriate "Net" type, then pass it to one of the interaction class's mutator functions. The implicit cast from the "Net" type to the type expected by the mutator functions will perform byte swapping if necessary.

The *TestDecoder* constructor just adds its decoding functions to the base class's table using addDecoder().

Note that while decoding functions are implemented as static member functions of the *TestDecoder* class, this was not strictly necessary - we could have written them as ordinary global function instead. But by implementing them as static members of the class, we can take advantage of some VR-Link macros.

```
class TestDecoder : public DtInteractionDecoder
{
public:

   // Constructor
   TestDecoder(DtExerciseConn* exConn,
      DtInterClassDesc* classDesc);

   // Destructor
   virtual ~TestDecoder();

protected:

   // Individual parameter decoding functions - one for each parameter.
   // These macros expand to look like this:
   // static void decodeNumber(TestInteraction* inter,
   //    const RTI::ParameterHandleValuePairSet& params, int index);
   DtDECLARE_TEST_PARAM_DECODER(Number);
   DtDECLARE_TEST_PARAM_DECODER(Vector);
};

// Constructor.
TestDecoder::TestDecoder(
   DtExerciseConn* exConn,
   DtInterClassDesc* classDesc) :
   DtInteractionDecoder(exConn, classDesc)
{
   // Register the individual decoding functions for Test's parameters with
   // object. The decoding functions have names like decodeNumber and
```

```
   // decodeVector. The macro expands to look like this:
   // addDecoder("Number", (DtParameterDecoder) decodeNumber);
   DtADD_PARAM_DECODER(Number);
   DtADD_PARAM_DECODER(Vector);
}

// The DtDEFINE_SIMPLE_PARAM_DECODER macro can be used to define a "simple"
// decoding function. Expanded, a decoding function looks like this:
//
// void TestDecoder::decodeNumber(TestInteraction* inter,
//    const RTI::ParameterHandleValuePairSet& params,
//    int index)
// {
//    RTI::ULong length = 0;
//    // Get value from RTI representation into netVal.
//    DtNetInt32* netVal = (DtNetInt32*) params.getValuePointer(index,
//              length);
//    // Check for size mismatch
//    if (length > sizeof(DtNetInt32))
//    {
//       DtWarn("Size of value decoded for parameter %s (%d)\n",
//          "Number", length);
//       DtWarn(" is larger than size of %s (%d)\n",
//          "DtNetInt32", sizeof(DtNetInt32));
//    }
//    // Pass the value to the mutator function, assuming the "net" type can
//    // be implicitly cast to the native type expected by the mutator.
//    inter->setNumber(*netVal);
// }

DtDEFINE_SIMPLE_TEST_PARAM_DECODER(Number, DtNetInt32, setNumber);

DtDEFINE_SIMPLE_TEST_PARAM_DECODER(Vector, DtNet64Vector, setVector);
```

Creating the Encoder Function

TestEncoder, defined in *testEnc.cxx*, is derived from *DtInteractionEncoder*. Macros help out in the declaration and definition of the encoding functions for individual parameters. (Again, encoding functions are provided as static members of the *TestEncoder* class).

The “simple” implementation provided by the macros for an encoding function is to obtain a value from a particular interaction class inspector function, cast it to the appropriate “Net” type, which performs byte swapping if necessary, and adds the value to an RTI::ParameterHandleValuePairSet for sending.

The *TestEncoder* constructor just adds its encoding functions to the base class’s table using addEncoder().

```
class TestEncoder : public DtInteractionEncoder
{
public:

   // Constructor
   TestEncoder(DtExerciseConn* exConn,
      DtInterClassDesc* classDesc);

   // Destructor
```

```
   virtual ~TestEncoder();

protected:

   // Individual parameter encoding functions - one for each parameter.
   // These macros expand to look like this:
   // static void encodeNumber(const TestInteraction& inter,
   //    RTI::ParameterHandleValuePairSet* params,
   //    RTI::ParameterHandle paramHandle);
   DtDECLARE_TEST_PARAM_ENCODER(Number);
   DtDECLARE_TEST_PARAM_ENCODER(Vector);
};

// Constructor.
TestEncoder::TestEncoder(
   DtExerciseConn* exConn,
   DtInterClassDesc* classDesc) :
   DtInteractionEncoder(exConn, classDesc)
{
   // Register the individual encoding functions for Test's parameters with
   // object. The encoding functions have names like encodeNumber and
   // encodeVector. The macro expands to look like this:
   // addEncoder("Number", (DtParameterEncoder) encodeNumber);
   // addEncoder("Vector", (DtParameterEncoder) encodeVector);
   DtADD_PARAM_ENCODER(Number);
   DtADD_PARAM_ENCODER(Vector);
}

// The DtDEFINE_SIMPLE_PARAM_ENCODER macro can be used to define a "simple"
// encoding function. Expanded, a encoding function looks like this:
// void TestEncoder::encodeNumber(const TestInteraction& inter,
//    RTI::ParameterHandleValuePairSet* params,
//    RTI::ParameterHandle paramHandle)
// {
//    DtNetInt32 netVal;
//    netVal = inter.number();
//    params->add(paramHandle, (char*) &netVal, sizeof(DtNetInt32));
// }

DtDEFINE_SIMPLE_TEST_PARAM_ENCODER(Number, DtNetInt32, number);

DtDEFINE_SIMPLE_TEST_PARAM_ENCODER(Vector, DtNet64Vector, vector);
```

7.4.3. Using the New Class

Once we build the *TestInteraction* class, and outfit it with a *TestEncoder* and *TestDecoder*, we can use it in an application just like any other *DtInteraction* subclass. In *main.cxx*, we create a *TestInteraction*, fill it out, and send it. We also register a callback on incoming *TestInteractions*, and print their contents using the virtual printData() function.

```
// Callback function to be called when a TestInteraction is received.
void testCb(TestInteraction* inter, void*)
{
   printf("Received Test Interaction!\n");
   inter->print();
   printf("\n");
}
```

```
main()
{
   DtExerciseConn conn("VrlExtend", "TestInter", (DtFomMapper*) NULL);
   DtTimeInit();

   // Register the callback on incoming TestInteractions.
   TestInteraction::addCallback(&conn, testCb, NULL);

   while (1)
   {
      DtSetSimTime(DtAbsRealTime());
      conn.drainInput();

      // Send a TestInteraction.
      TestInteraction inter;
      inter.setNumber(10);
      inter.setVector(DtVector(1.0, 2.0, 3.0));
      conn.sendStamped(inter);

      DtSleep(1.0);
   }
}
```

After you build the *testInter* example, run two copies of it. You should see them communicating with each other. Each should print *TestInteractions* received from the other.

7.5. Creating New Object Classes

The code in the *testObj* example explains the steps you need to take when you add a new object class to your FOM. You need to extend the VR-Link top-level API by adding a new kind of state repository, publisher, reflected object, and reflected object list. You also need to create mappings between the new FOM class and the API extensions.

For this example, we added an object class called *Test* to the FED file *VrlExtend.fed*. It has two attributes:

- **Number**, which is represented as a 32-bit float
- **Vector**, which is represented as an array of three 64-bit doubles.

To work with these additions we need to:

1. Create a new subclass of *DtStateRepository*, called *TestStateRepository*, to hold the state of a new object of this kind. (See *testSR.h* and *testSR.cxx*.)
2. Create new subclasses of *DtObjectPublisher*, *DtReflectedObject*, and *DtReflectedObjectList*. Most of their functionality is in their base classes, but the derived classes provide an important element of type safety by localizing casts from base *DtStateRepositories* to our new *TestStateRepository* subclass. (See *testPub.h*, *testPub.cxx*, *refTest.h*, *refTest.cxx*, *refTestList.h* and *refTestList.cxx*.)
3. Create *TestEncoder* and *TestDecoder* classes, subclasses of *DtHlaStateEncoder* and *DtHlaStateDecoder* respectively. These classes contain encoding and decoding functions for the attributes of the *Test* object class. (See *testDec.h*, *testDec.cxx*, *testEnc.h* and *testEnc.cxx*.)

While this may seem like a lot of classes to create, most of them are fairly trivial because they inherit most of their behavior from base classes, and you can use the files in *./examples/testObj* as templates from which to create your own classes.

7.5.1. Creating a New State Repository

We first create a new subclass of *DtStateRepository* called in *TestStateRepository*, in *testSR.h* and *testSR.cxx*. This class is used to store the state of both local and reflected objects of our *Test* class.

The class definition is listed below. We provide inspector and mutator functions to access the state data, an override of the virtual printData() function that prints the current state data, and an override of the virtual clone() function that returned a new'ed instance of this class. We represent state data in the same way as in the FOM, but this is not strictly necessary. We could perform a non-trivial conversion in FOM mapping code instead.

```
class TestStateRepository : public DtStateRepository
{
public:

   // Default constructor
```

```
   TestStateRepository();

   // Destructor
   virtual ~TestStateRepository();

   // Copy constructor
   TestStateRepository(const TestStateRepository& orig);

   // Assignment operator
   TestStateRepository& operator=(const TestStateRepository& orig);

   // Return an empty object of the same type as this. Required.
   virtual DtStateRepository* clone() const;

   // Virtual function override. Prints the data values in the state
   // repository. Not strictly required, but encouraged, since otherwise
   // calling print() on an instance of this class with merely print raw hex
   // data.
   virtual void printData() const;

   // ************************************************************
   // Inspector and mutator functions for the object's state data.
   // These will look different for each DtStateRepository subclass.
   // None are required by VR-Link, but the class will not be
   // particularly useful without a way to get data in and out in a
   // typesafe manner.
   // ************************************************************

   // Set/Get the attribute named "Number"
   virtual void setNumber(int num);
   virtual int number() const;

   // Set/Get the attribute named "Vector"
   virtual void setVector(const DtVector& vec);
   virtual const DtVector& vector() const;

protected:

   // Here, we store the actual data values, in native representation.
   int myNum;
   DtVector myVec;
};
```

Here are the definitions of the various member functions:

```
void TestStateRepository::setNumber(int num)
{
   myNum = num;
}

int TestStateRepository::number() const
{
   return myNum;
}

void TestStateRepository::setVector(const DtVector& vec)
{
   myVec = vec;
}
```

```
const DtVector& TestStateRepository::vector() const
{
   return myVec;
}

DtStateRepository* TestStateRepository::clone() const
{
   return new TestStateRepository();
}

void TestStateRepository::printData() const
{
   printf("Number: %d\n", number());
   // DtVector has a string() member function to get a printable string.
   printf("Vector: %s\n", vector().string());
}
```

7.5.2. Creating New Publisher, Reflected Object, and Object List Classes

Next, we need to create subclasses of *DtObjectPublisher*, *DtReflectedObject DtReflectedObjectList* for our *Test* objects.

Technically, these classes are not strictly necessary. We can do just about everything we need to using the base classes, but then we would need to configure each instance of those base classes with the appropriate FOM classes, state repositories, encoders, and decoders every time we create one. By subclassing, we encapsulate the configuration of each base class for our new object type into one place. In addition, we hide the casting from base *DtStateRepositories* to the derived *TestStateRepository*, so that application code that uses our derived classes will be able to automatically obtain pointers to *TestStateRepositories*.

We derive *TestPublisher* from *DtObjectPublisher* in *testPub.h* and *testPub.cxx*:

```
class TestPublisher : public DtObjectPublisher
{
public:

   // Constructor - configure base class with proper FOM class, state
   // repositories, encoder and decoder.
   TestPublisher(DtExerciseConn* conn, const char* objName = NULL);

   // destructor
   virtual ~TestPublisher();

   // Returns a pointer to the TestStateRepository - through
   // which you can set current state information.
   virtual TestStateRepository* testStateRep();

   // Synonym for testStateRep()
   virtual TestStateRepository* tsr();
};
```

As you can see, this class just configures its base class properly in its constructor, and provides type safety in accessing its state repository:

```
TestPublisher::TestPublisher(
   DtExerciseConn* conn, const char* objName) :
```

```
   DtObjectPublisher(
      conn, new TestStateRepository(), new TestStateRepository())
{
   // setHlaObject is defined in the base DtObjectPublisher. Pass the FOM
   // class name, and an encoder and decoder. These objects will be deleted
   // by the base class on destruction.
   const char* className = "Test";
   DtObjClassDesc* desc = conn->fom()->objClassByName(className);
   setHlaObject(className,  new TestEncoder(conn, desc),
      new TestDecoder(conn, desc), objName);
}

TestStateRepository* TestPublisher::testStateRep()
{
   return (TestStateRepository*) stateRep();
}

TestStateRepository* TestPublisher::tsr()
{
   return (TestStateRepository*) stateRep();
}
```

Creating the Reflected Object Class

ReflectedTest performs similar tasks on the incoming side. In addition to the testStateRep() and tsr() functions, the class provides type-specific versions of the list management functions next(), prev(), wrapNext(), and wrapPrev(), and of the postUpdate callback registration functions.

This class is derived from *DtReflectedObject* in *refTest.h* and *refTest.cxx*:

```
class ReflectedTest : public DtReflectedObject
{
public:

   // constructor
   ReflectedTest(DtHlaObject* obj, DtExerciseConn* conn);

   // destructor
   virtual ~ReflectedTest();

   // Returns a pointer to the TestStateRepository, through which you can
   // inspect current state information.
   virtual TestStateRepository* testStateRep() const;

   // Synonym for testStateRep()
   virtual TestStateRepository* tsr() const;

   // Functions to get the next and previous objects in the list. The "wrap"
   // functions cycle to the beginning/end of the list.
   virtual ReflectedTest* next() const;
   virtual ReflectedTest* prev() const;
   virtual ReflectedTest* wrapNext() const;
   virtual ReflectedTest* wrapPrev() const;

   // Registration/deregistration of postUpdate callbacks.
   virtual void addPostUpdateCallback(
      TestCallbackFunction cb, void* userData);
   virtual void removePostUpdateCallback(
```

```
      TestCallbackFunction cb, void* userData);
};
```

The implementations of these functions are trivial, in most cases calling down to base class versions, and performing a cast between a specific and a general type:

```
ReflectedTest::ReflectedTest(DtHlaObject* obj, DtExerciseConn* conn) :
   DtReflectedObject(conn, new TestStateRepository())
{
   setHlaObject(obj, new TestDecoder(conn, obj->classDesc()));
}

TestStateRepository* ReflectedTest::testStateRep() const
{
   return (TestStateRepository*) myStateRep;
}

TestStateRepository* ReflectedTest::tsr() const
{
   return (TestStateRepository*) myStateRep;
}

ReflectedTest* ReflectedTest::next() const
{
   return (ReflectedTest*) DtListedObj::next();
}

ReflectedTest* ReflectedTest::prev() const
{
   return (ReflectedTest*) DtListedObj::prev();
}

ReflectedTest* ReflectedTest::wrapNext() const
{
   return (ReflectedTest*) DtListedObj::wrapNext();
}

ReflectedTest* ReflectedTest::wrapPrev() const
{
   return (ReflectedTest*) DtListedObj::wrapPrev();
}

void ReflectedTest::addPostUpdateCallback(
   TestCallbackFunction cb, void* userData)
{
   baseAddPostUpdateCallback(
      (DtReflectedObjectCallbackFunction) cb, userData);
}

void ReflectedTest::removePostUpdateCallback(
   TestCallbackFunction cb, void* userData)
{
   baseRemovePostUpdateCallback(
      (DtReflectedObjectCallbackFunction) cb, userData);
}
```

Creating a Reflected Object List Class

After we create a *ReflectedTest* class, we need to create a *ReflectedTestList* class that knows how to create and manage *ReflectedTests*. This class is derived from *DtReflectedObjectList* in *refTestList.h* and *refTestList.cxx*:

```
class ReflectedTestList : public DtReflectedObjectList
{
public:

   // Constructor
   ReflectedTestList(DtExerciseConn* exConn);

   // destructor
   virtual ~ReflectedTestList();

   // The following public functions are not required by VR-Link, but they are
   // strongly recommended. The functions call down to base class versions,
   // and cast to and from more generic types.

   // Look up object by identifier.
   virtual ReflectedTest* lookup(RTI::ObjectHandle id);

   // Look up object by name (global object designator).
   virtual ReflectedTest* lookup(const DtGlobalObjectDesignator& objName);

   // Get the first/last object in the list.
   virtual ReflectedTest* first() const;
   virtual ReflectedTest* last() const;

   // Functions that allow a user to register callbacks that will be called
   // when "Test" objects are added and removed from the list.

   // Registration/deregistration of object addition callbacks
   virtual void addTestAdditionCallback(
      TestCallbackFunction cb, void* userData);
   virtual void removeTestAdditionCallback(
      TestCallbackFunction cb, void* userData);

   // Registration/deregistration of object removal callbacks
   virtual void addTestRemovalCallback(
      TestCallbackFunction cb, void* userData);
   virtual void removeTestRemovalCallback(
      TestCallbackFunction cb, void* userData);

protected:

   // Override of virtual function that the base class uses to create the
   // appropriate kind of DtReflectedObject. Here, we create a ReflectedTest.
   virtual DtReflectedObject* newReflectedObject(DtHlaObject* obj) const;
};
```

Defining Functions

Definitions of these functions are very straightforward, and rely mostly on the base class versions, as follows:

```
ReflectedTestList::ReflectedTestList(
   DtExerciseConn* exConn) :
```

```
   DtReflectedObjectList(exConn)
{
   // Tell the base class that we will be managing the FOM class "Test".
   addObjectClassByName("Test");
}

ReflectedTest* ReflectedTestList::lookup(RTI::ObjectHandle id)
{
   return (ReflectedTest*) lookupObj(id);
}

ReflectedTest* ReflectedTestList::lookup(
   const DtGlobalObjectDesignator& objName)
{
   return (ReflectedTest*) lookupObj(objName);
}

ReflectedTest* ReflectedTestList::first() const
{
   return (ReflectedTest*) firstObj();
}

ReflectedTest* ReflectedTestList::last() const
{
   return (ReflectedTest*) lastObj();
}

void ReflectedTestList::addTestAdditionCallback(
   TestCallbackFunction cb, void* userData)
{
   addObjectAdditionCallback(
      (DtReflectedObjectCallbackFunction) cb, userData);
}

void ReflectedTestList::removeTestAdditionCallback(
   TestCallbackFunction cb, void* userData)
{
   removeObjectAdditionCallback(
      (DtReflectedObjectCallbackFunction) cb, userData);
}

void ReflectedTestList::addTestRemovalCallback(
   TestCallbackFunction cb, void* userData)
{
   addObjectRemovalCallback(
      (DtReflectedObjectCallbackFunction) cb, userData);
}

void ReflectedTestList::removeTestRemovalCallback(
   TestCallbackFunction cb, void* userData)
{
   removeObjectRemovalCallback(
      (DtReflectedObjectCallbackFunction) cb, userData);
}

DtReflectedObject* ReflectedTestList::newReflectedObject(
   DtHlaObject* obj) const
{
   return new ReflectedTest(obj, myExConn);
}
```

TestPublisher, *ReflectedTest*, and *ReflectedTestList* hard-code their own FOM mapping information, rather than relying on the default behavior, which is to obtain it from the FOM mapper. We do this because it make things simpler, by saving us the step of configuring the FOM mapper with mappings for our new class. But this choice means that it would be more difficult to change the way this kind of interaction is represented in the FOM.

Specifically, the *TestPublisher* and *ReflectedTest* constructors tell their base classes what kinds of encoders and decoders to use. The *TestPublisher* constructor chooses its own class to use to represent its objects, and the *ReflectedTestList* constructor chooses what FOM classes to manage.

7.5.3. Creating Encoder and Decoder Classes

We now turn to the *TestEncoder* and *TestDecoder* classes, the encoder and decoder whose use is dictated by *TestPublisher* and *ReflectedTest*.

Creating the Decoder Class

TestDecoder is defined in *testDec.h* and *testDec.cxx*, as shown below. Macros help out in the declaration and definition of decoding functions, which are implemented as static member functions of *TestDecoder*. The *TestDecoder* constructor self-registers its decoding functions.

```
class TestDecoder : public DtHlaStateDecoder
{
public:

   // default constructor
   TestDecoder(
      DtExerciseConn* exConn,
      DtObjClassDesc* classDesc);

   // destructor
   virtual ~TestDecoder();

protected:

   // Individual attribute decoding functions - one for each attribute.
   // These macros expand to look like this:
   // static void decodeNumber(TestStateRepository* stateRep,
   //    const RTI::AttributeHandleValuePairSet& attrs, int pairSetIndex)

   DtDECLARE_TEST_ATTR_DECODER(Number);
   DtDECLARE_TEST_ATTR_DECODER(Vector);
};

// Default constructor
TestDecoder::TestDecoder(
   DtExerciseConn* exConn,
   DtObjClassDesc* classDesc) :
   DtHlaStateDecoder(exConn, classDesc)
{
   // Register the individual decoding functions for Test's attributes with
```

```
   // the object. The decoding functions have names like decodeNumber and
   // decodeVector. The macro expands to look like this:
   // addDecoder("Number", (DtAttributeDecoder) decodeNumber);
   DtADD_ATTR_DECODER(Number);
   DtADD_ATTR_DECODER(Vector);
}

// Decoding functions

// The DtDEFINE_SIMPLE_ATTR_DECODER macro can be used to define a "simple"
// decoding function. Expanded, a decoding function looks like this:
//
// void TestDecoder::decodeNumber(TestStateRepository* stateRep,
//    const RTI::AttributeHandleValuePairSet& attrs,
//    int index)
// {
//    RTI::ULong length = 0;
//    // Get value from RTI representation into netVal.
//    DtNetInt32* netVal = (DtNetInt32*) attrs.getValuePointer(index,
//             length);
//    // Check for size mismatch
//    if (length > sizeof(DtNetInt32))
//    {
//       DtWarn("Size of value decoded for attribute %s (%d)\n",
//          "Number", length);
//       DtWarn(" is larger than size of %s (%d)\n",
//          "DtNetInt32", sizeof(DtNetInt32));
//    }
//    // Pass the value to the mutator function, assuming the "net" type can
//    // be implicitly cast to the native type expected by the mutator.
//    stateRep->setNumber(*netVal);
// }

DtDEFINE_SIMPLE_TEST_ATTR_DECODER(Number, DtNetU32, setNumber);

DtDEFINE_SIMPLE_TEST_ATTR_DECODER(Vector, DtNet64Vector, setVector);
```

Creating The Encoder Class

The *TestEncoder* class is implemented similarly, in *testEnc.h* and *testEnc.cxx*, but it contains checking functions in addition to encoding functions:

```
class TestEncoder : public DtHlaStateEncoder
{
public:

   // default constructor
   TestEncoder(
      DtExerciseConn* exConn,
      DtObjClassDesc* classDesc);

   // destructor
   virtual ~TestEncoder();

protected:

   // Individual attribute encoding functions - one for each attribute.
   // This macro expands to look like this:
   // static void encodeNumber(const TestStateRepository& rep,
```

```
   //    RTI::AttributeHandleValuePairSet* attrs,
   //    RTI::AttributeHandle attrHandle)
   DtDECLARE_TEST_ATTR_ENCODER(Number);
   DtDECLARE_TEST_ATTR_ENCODER(Vector);

   // Individual attribute checking functions - one for each attribute.
   // This macro expands to look like this:
   // static DtBoolean needNumber(const TestStateRepository& rep,
   //    const TestStateRepository& asSeenByRemote);

   DtDECLARE_TEST_ATTR_CHECKER(Number);
   DtDECLARE_TEST_ATTR_CHECKER(Vector);
};
TestEncoder::TestEncoder(
   DtExerciseConn* exConn,
   DtObjClassDesc* classDesc) :
   DtHlaStateEncoder(exConn, classDesc)
{
   // Register the individual encoding and checking functions for Test's
   // attributes with the object. The encoding functions have names like
   // encodeNumber, while checking functions have names like needNumber. The
   // macros expands to look like this:
   // addEncoder("Number", (DtAttributeEncoder) encodeNumber);
   // addChecker("Number", (DtAttributeChecker) needNumber);
   DtADD_ATTR_ENCODER(Number);
   DtADD_ATTR_ENCODER(Vector);

   DtADD_ATTR_CHECKER(Number);
   DtADD_ATTR_CHECKER(Vector);
}

// Encoding functions

DtDEFINE_SIMPLE_TEST_ATTR_ENCODER(
   Number, DtNetU32, number);

DtDEFINE_SIMPLE_TEST_ATTR_ENCODER(
   Vector, DtNet64Vector, vector);

// Checking functions

DtDEFINE_SIMPLE_TEST_ATTR_CHECKER(
   Number, number);

DtDEFINE_SIMPLE_TEST_ATTR_CHECKER(
   Vector, vector);
```

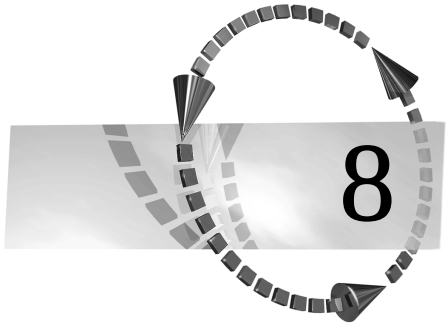
7.5.4. Using the New Classes in An Application

Once all of these classes are set up, they can be used in applications just like VR-Link's normal publishers, reflected object lists, and reflected objects, as in *main.cxx*:

```
main()
{
   DtExerciseConn conn("VrlExtend", "TestObj", (DtFomMapper*) NULL);
   DtTimeInit();

   // Create a reflected test list
   ReflectedTestList rtl(&conn);
```

```
   // Create a test publisher
   TestPublisher pub(&conn);

   // Grab a pointer to its state repository
   TestStateRepository* tsr = pub.tsr();

   // Set some data
   tsr->setVector(DtVector(1,2,3));

   // Run the main loop for 20 seconds
   DtTime beginTime = DtAbsRealTime();
   while (DtAbsRealTime() < beginTime + 20.0)
   {
      DtSetSimTime(DtAbsRealTime());
      conn.drainInput();

      // Print the state of the first object in the list:
      if (rtl.first())
      {
         rtl.first()->tsr()->printData();
      }

      // Change the value of "Number" once a second
      tsr->setNumber((int) DtGetSimTime());

      // Tick the publisher, causing data to be sent when necessary.
      pub.tick();

      // Sleep till next iteration
      DtSleep(0.2);
   }
}
```

After you build the *testObj* example, run two copies of it. You should see each print out state data that it receives from the other.



The DIS-Specific Interface

VR-Link's protocol-independent interface provides the functionality most simulations need to interact with a DIS exercise. However, some applications need greater control over DIS parameters. This chapter describes VR-Link's DIS-Specific interface.

This chapter covers the following topics:

If your application is intended for both DIS and HLA, or if you plan to add this support in the future, enclose your DIS-specific code within a preprocessor directive pair to ensure that it will not be compiled when you try to build for HLA:

```
#if !DtHLA
// DIS-specific code
#endif
```

8.1. Working with PDUs

Protocol Data Units (PDUs) are the mechanism through which DIS applications send data. VR-Link's *DtPdu* class, defined in *pdu.h*, is used to represent DIS PDUs. *DtPdu* is an abstract base class. All PDU classes are derived from *DtPdu*. Every interaction supported by VR-Link has an interaction PDU. For example fire interactions are represented by *DtFirePdu*. Interaction PDUs that have HLA counterparts are derived from *DtInteraction*, an essentially empty class, which is in turn derived from *DtPdu*. We have provided typedefs so that you can refer to, for example, a *DtFirePdu* as a *DtFireInteraction* in protocol-independent code.

All DIS 2.0.3, 2.0.4 (IEEE1278.1), and 2.1.4 (IEEE1278.1a) PDUs have PDU classes implemented in VR-Link. Since there are nearly 70 PDUs, we do not describe all of their inspectors and mutators in this manual.

---

i You can find descriptive information about many of the PDU classes in *VR-Link Reference Manual*.

The derived PDU classes provide mutator and inspector functions to set and examine the values of PDU fields. Header information is accessed through *DtPdu* member functions. In general, an inspector function has the same names as the field being inspected, and returns the value it is inspecting, for example DtEntityStatePdu::entityType(). Mutator functions have the same names, preceded by the word "set", for example, DtEntityStatePdu::setEntityType().

Please see the header files in *VR-Link Reference Manual* for the exact prototypes of all PDU class member functions. In this manual, Appendix A, *Guide to VR-Link Header Files* contains tables that associate class names with their header files.

8.1.1. Sending PDUs

The procedure for sending a PDU is identical to the procedure for sending interactions, described in "Sending Interactions," on page 5-11. This section contains additional DIS-specific information.

You send PDUs using *DtExerciseConn*'s send() or sendStamped() functions. sendStamped() sends a copy of the PDU that has the exercise ID and timestamp fields in the PDU header set to the *DtExerciseConn*'s exercise ID, and the current time. send() leaves the PDU header unmodified.

The following example shows how to create and send a Transmitter PDU:

```
DtExerciseConn exchange(...);
...
// Create a DtTransmitterPdu
DtTransmitterPdu pdu;
// Fill the PDU with data
pdu.setEntityId(DtEntityIdentifier(1,2,3));
pdu.setRadioId(4);
pdu.setTransmitState(DtOn);
```

```
...
// Send to the exercise
exConn.sendStamped(pdu);
```

Normally, send() and sendStamped() send the PDU to the *DtExerciseConn*'s default destination IP address (see "Creating an Exercise Connection for DIS," on page 5-5 for details). However, the functions have an optional argument that specifies a different destination address. For example, to send a PDU to the machine with address 207.86.232.1, use:

```
exConn.sendStamped(pdu, DtStringToInetAddr("207.86.232.1"));
```

Applications usually send interactions PDUs as described in this section. They usually use *DtEntityPublisher* to manage the sending of entity state PDUs.

8.1.2. Receiving PDUs

Applications handle incoming PDUs through callbacks. The process for receiving PDUs is virtually identical to that described in "Receiving Interactions," on page 5-12. This section has additional DIS-specific information.

As described in the Chapter 5, *The Protocol-Independent Interface*, a callback is registered with VR-Link using a PDU class's static addCallback() function, and within the callback function, you can use the PDU class's inspector functions to examine the fields of the PDU.

The DIS version of *DtExerciseConn* has callback registration and unregistration functions called addPduCallback() and removePduCallback(). However, these functions are meant to be used only by a PDU class's static addCallback() function, and not directly by application code. If an application uses the *DtExerciseConn* functions directly, then the PDU class never gets registered with the *DtPduFactory* (this occurs within a PDU class's addCallback() function), and VR-Link will not be able to correctly create instances of the class to pass to your callback functions. See "DtPduFactory," on page 8-10 for more information.

More Control Over Receiving PDUs

Most applications use VR-Link's callback mechanism to handle incoming PDUs, leaving the job of actually reading the PDUs and dispatching them to the callbacks to *DtExerciseConn*'s drainInput(). However, to give you more flexibility and control over reading PDUs from the network, *DtExerciseConn* provides access to the functions used by drainInput().

drainInput() works by repeatedly calling readAndProcess(). readAndProcess() tries to read a single PDU from the network using netRead(), then passes the PDU to processPdu(), from which all callbacks are invoked. Any of these functions can be used by application code, instead of calling drainInput().

Using the netRead() Function

netRead() tries to read a packet from the network, constructs an instance of a PDU class from the network representation, and returns it.

If you use netRead() directly, note the following considerations:

- For efficiency reasons, successive calls to netRead() return *DtPdus* that all use the same buffer for their network representations. Applications must delete the *DtPdu* returned by netRead() before another call to netRead() is made. Otherwise, the previous *DtPdus* are trashed. Using readAndProcess() (or drainInput()) ensures that each *DtPdu* is deleted before the next call to netRead(), which is why it is safer to use these two functions.
- Because PDU filtering based on PDU kind is done at the *DtSocket* level (before the packets get to netRead()), if filtering is enabled, netRead() returns only those kinds of PDUs for which interest has been registered. If you are using netRead(), you should either turn filtering off with DtExerciseConn::disableFiltering(), or use DtExerciseConn::addInterestInPduKind() to explicitly register interest in the kinds of PDUs you would like to receive.
- When a typical application calls a PDU's addCallback() function, the PDU class registers itself with *DtExerciseConn*. Registration allows *DtExerciseConn* to know which type of derived *DtPdu* to create to represent an incoming PDU of a particular kind. If you are using netRead() and not using addCallBack(), make sure that all PDU classes you want to use are registered with *DtPduFactory.* For more information, see "DtPduFactory," on page 8-10.

The optional return code argument to netRead() is a pointer to a return code that is set depending on the results of the netRead() operation as follows:

- If a PDU was successfully read, **retCode** is set to DtNET_READ_SUCCESS. If there are no PDUs to be read, **retCode** is set to DtNET_READ_NO_PACKETS, and NULL is returned.
- If the *DtExerciseConn* is connected to a Packet Server through a *DtClientSocket*, and a packet is received that represents a control message from the Packet Server, rather than a DIS PDU, **retCode** is set to DtNET_READ_SERVER_MSG.
- If data is read, but it is not a PDU of the correct DIS version, DtNET_READ_BAD_PACKET is indicated.
- If the PDU length field conflicts with the actual size of the packet, the packet is rejected with status DtNET_READ_SIZE_MISMATCH.

So, an application can use netRead() as follows:

```
DtExerciseConn exConn(...);
...
int retCode;
while (1)
{
    DtPdu* pdu = exConn.netRead(&retCode);
    if (retCode == DtNET_READ_SUCCESS)
    {
        // do something with the PDU
        ...
        delete pdu;
    }
}
```

Using the readUntil() Function

readUntil() is similar to drainInput() in that it repeatedly reads and processes PDUs, but it returns when predicate() returns true when called on the PDU being processed, or after **timeout** seconds, whichever happens first. If **timeout** seconds elapse, it returns *DtOtherPduKind*. Otherwise, it returns the kind of the PDU that caused the **predicate** to return true. predicate() is a function you provide that has the following signature:

```
int predicate(DtPdu *pdu, void *arg);
```

readUntil()'s signature is as follows:

```
DtPduKind readUntil(DtPredicate predicate, void* arg,
           DtTime timeout, DtTime sleepTime);
```

The **arg** argument passed to readUntil() is passed as the **arg** argument to predicate() each time it is called.

If no PDU is available to be read, then readUntil() will sleep for **sleepTime** seconds before polling the socket again. A timeout less than 0 causes readUntil() to poll continuously, without sleeping.

8.1.3. Inspecting and Setting Data in A PDU Header

The *DtPdu* class provides several functions to inspect and set the data in a DIS PDU's header. Other fields of the various PDUs are accessed through member functions of the classes derived from *DtPdu*.

The following *DtPdu* member functions return the values of their respective fields in the PDU's header:

- protocolVersion()
- exerciseId()
- kind()
- protocolFamily()
- length().

protocolFamily() is not available when linking with *libvl3*, because it was not a field in DIS 2.0.3 headers. The *DtPduKind* and *DtProtocolFamily* enumerations are in *disEnums.h*.

Other functions for setting and returning data are described in Table 8-1:

Table 8-1: DtPdu member functions

Function	Description
timeStamp()	Returns a floating point number representing the time (in seconds past the hour) that is encoded in the PDU's time stamp. For more information, see "Timestamps," on page 3-18.
timeStampType()	Returns either DtTimeStampRelative or DtTimeStampAbsolute, depending on the type of the PDU's timestamp. For more information, see "Timestamps," on page 3-18.
setExerciseId()	Sets the value of the exercise ID field of the PDU's header to id.
setTimeStamp()	Sets the PDU's timestamp to **time** modulo 3600, and the timestamp type to **relOrAbs**. Time is the time in seconds at which the PDU's data is to be considered valid. **relOrAbs** must be either DtTimeStampRelative or DtTimeStampAbsolute. The default is DtTimeStampRelative. For more information, see "Timestamps," on page 3-18.
setVersion()	Sets the value of the Protocol Version field.
print()	Prints the entire contents of a PDU in human readable form, including header and data, by calling printHeader() followed by printData(). This is the function used by the netdump utility to print the contents of PDUs. printHeader() prints the information contained in the PDU's header, while printData() is a virtual function that prints the data below the header. The base class version of printData() prints the data in hexadecimal format.

Other header information is set at the time the *DtPdu* is constructed, and cannot be set later. For details, see "DtPdu Constructors," on page 8-8.

8.1.4. Getting a PDU's Network Representation

DtPdu::packet() returns a pointer to a buffer containing the network representation of the PDU – the exact bytes that would be sent onto the network if you were to send a *DtPdu* at any given time. This is a pointer to an internal buffer which, in most circumstances, you should not try to modify. Use member functions to set values for particular fields.

However, if you want to directly change the data that will go out to the network, you can modify a network representation. If you choose to do so, it is your responsibility to keep the data consistent and to ensure that the buffer represents a valid PDU.

This will only work for those *DtPdu* classes that store their current data in their network representation buffer, (as opposed to storing it in some other representation and merely converting to network representation when packet() is called). While this has been the chosen implementation for all PDU classes so far, we might choose other implementations in the future.

8.1.5. *DtPdu* Constructors

Since *DtPdu* is an abstract class, you cannot directly create an instance of a *DtPdu*. You can only create instances of classes derived from *DtPdu*.

All PDU classes have at least two constructors, a blank PDU constructor, and a from-network-representation constructor.

Constructing a Blank PDU

The blank PDU constructor constructs a blank, minimal PDU. By minimal, we mean that for variable length PDUs the size is equal to the smallest size that a PDU of that type can legally have. For example, a *DtDataPdu* created with the blank PDU constructor will have zero fixed datum fields and zero variable datum fields; a *DtEntityStatePdu* so created will contain zero articulated parts. By blank, we mean that all bytes of the PDU below the header typically contain zeros (although in some cases other values more accurately represent the concept of a field being blank).

PDU Header Information

When a blank PDU is constructed, the protocol version is set to the value of the VR-Link global variable **DtProtocolVersionToSend**, (declared in *pdu.h)*. **DtProtocolVersionToSend** defaults to 5 (IEEE 1278.1) if you're building with DtDISVERS=5 and to 3 (DIS 2.0.3) if you're building with DtDISVERS=3.

If you want to indicate 4 (DIS 2.0.4) set **DtProtocolVersionToSend** to 4; to indicate DIS 2.1.4/IEEE 1278.1a, set it to 6.

PDU kind and protocol family depend on what type of PDU class you are creating (based on the value returned by the virtual function internalGetPduKind() which must be defined for each derived PDU class). The length is the PDU's minimal length, as defined above. The exerciseId and time stamp are zero.

There is an optional buffer argument to the blank PDU constructor. In most cases, you can ignore this argument and use the default, DtUSE_INTERNAL_BUFFER. Blank *DtPdu*'s are constructed as follows:

```
DtFirePdu pdu();
```

or simply

```
DtFirePdu pdu;
```

Alternate uses of the buffer argument are outlined in "Using External Buffers in a PDU Class," on page 8-11.

The blank PDU constructor is typically used when creating PDUs that you are sending. After creating the PDU, you can use its member functions to set the various data, then pass it to DtExerciseConn::sendStamped() for sending.

Constructing a PDU from a Network Representation

The from-network-representation constructor takes a pointer to a network representation of a PDU as an argument, and constructs a *DtPdu* object, which represents this PDU. By network representation, we mean a buffer containing the exact bytes that would be used to represent the PDU on the network.

It is rare that an application will need to directly use the from-network-representation constructor for a *DtPdu*. It is most commonly used only by a *DtPduFactory*, which is used by *DtExerciseConn* after receiving a packet from the network in order to construct the PDU class object that is passed to application callbacks.

Like the blank PDU constructor, the from-network-representation PDU class constructor has an optional buffer argument. This argument is used in the same way as the buffer argument to the blank PDU constructor, and is discussed in "Using External Buffers in a PDU Class," on page 8-11.

Each of the PDU class from-network-representation constructors requires as an argument a network representation of specifically that type of PDU. For instance, the *DtFirePdu* constructor specifically requires a *DtNetFirePdu*.

When constructing a *DtPdu* from a network representation through a PDU class constructor, the argument passed must be a valid network representation of a PDU. The status() member function may be used to determine whether a *DtPdu* represents valid data after construction. If status returns DtSTATUS_OK, it represents valid data. Otherwise, results of using the *DtPdu* are undefined. The meaning of the value returned by status depends on the type of PDU, and is meant to give some indication of why the PDU representation is invalid.

Currently, status is not verified by all *DtPdu* constructors. Therefore, a status of DtSTATUS_OK does not necessarily mean that a PDU is valid. However, a status other than that means that the PDU is not valid.

To construct a *DtPdu* object from a network representation of a PDU whose type you do not know, you can use a *DtPduFactory* (defined in *pduFactory.h*) – specifically, its createPdu() member function.

8.1.6. DtPduFactory

A *DtPduFactory* implements a kind of "virtual construction", by maintaining a table of associations between PDU kinds and functions that create a corresponding PDU class object.

The createPdu() function checks the PDU kind in the packet, then passes it and the buffer argument to the right creator function from the table. The creator in turn, is just a wrapper around the appropriate PDU class from-network-representation constructor (a constructor itself can't be added to a table). A pointer to the newly constructed object is returned by createPdu().

If the PDU kind in a packet does not have an individual PDU class implemented (that is, there's no creator registered with the *DtPduFactory* for the PDU kind), a *DtUnknownPdu* is created. See "Using DtUnknownPdu," on page 8-14.

A pointer to the PDU factory that a *DtExerciseConn* uses to create PDU class objects can be obtained using DtExerciseConn::pduFactory().

Given a buffer containing the network representation of a PDU (perhaps you read it from a file), you can use this *DtPduFactory* to construct the right type of PDU class as follows:

```
DtExerciseConn exConn(...);
...
// buffer contains the network representation of some PDU
char *buffer = ...;
// Grab a pointer to our DtExerciseConn's DtPduFactory
DtPduFactory* fact = exConn.pduFactory();
// Cast the buffer to a DtNetPacket* before passing it to
// createPdu to indicate you are treating it as the network
// representation of a PDU.
DtPdu* pdu = fact->createPdu((DtNetPacket*) buffer);
```

createPdu() rejects packets (returns NULL) that have a protocol version less than DtProtocolVersionToRecvMin or greater than DtProtocolVersionToRecvMax. These are global variables declared in *pdu.h*. When using DtDISVERS=5, these default to 4 and 6 respectively so that DIS 2.0.4 (IEEE 1278.1) and 2.1.4 (IEEE 1278.1a) PDUs are accepted. When using DtDISVERS=3, both default to 3 so that only DIS 2.0.3 PDUs are accepted.

By default, a *DtExerciseConn* creates an empty *DtPduFactory*, one that is not aware of any associations between PDU kinds and PDU classes. However, when you call a PDU class's addCallback() function, that class registers itself with the exercise connection's PDU factory. If you are using *DtPduFactory*, and are not using VR-Link's PDU callback mechanism, you must register creator functions with the *DtPduFactory* for all PDU classes you might want to work with.

Each PDU class has a static member function called create(), which can serve as that PDU's creator function. These creator functions have the following signature:

```
DtPdu *create(const DtNetPacket *initial,
           DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
```

So, for example, to change a *DtExerciseConn*'s *DtPduFactory* so that it will create a *DtEntityStatePdu* to represent PDU kind number 10, you can do this:

```
DtExerciseConn exConn(...);
...
exConn.pduFactory()->addCreator(DtPduKind(10),
           DtEntityStatePdu::create);
```

A NULL creator function can be added for a PDU kind to indicate that there is no PDU class for that kind and *DtUnknownPdu* should be used instead.

8.1.7. Using External Buffers in a PDU Class

Each *DtPdu* object stores a network representation of the PDU it represents. It is a pointer to this network representation that is returned by packet(). Normally, the memory used for this network representation is allocated internally by the PDU class constructor and deleted by the destructor.

In some cases, you might want a specific area of memory to be used by the *DtPdu* class for storage of its network representation (for example, memory from the stack, or from a shared memory pool. For this purpose, all PDU class constructors have an optional buffer argument. Users can cast a pointer to a particular area of memory to a DtBufferPtr and pass it as the buffer argument to indicate that a *DtPdu* should store its network representation there:

```
char buffer[144];
// construct blank PDU, but use buffer to store network
// representation
DtEntityStatePdu pdu(DtBufferPtr(buffer));
```

However, the most common reason why we would want to have a *DtPdu* use an external buffer for storage of its network representation is increased efficiency. When constructing a *DtPdu* from a network representation of a PDU, if you specify that the memory already occupied by the network representation be used by the PDU to store its network representation, a buffer copy may be avoided, for example:

```
DtNetFirePdu* netPacket = ......;
DtFirePdu(netPacket, DtBufferPtr(netPacket));
```

In fact, this is what *DtExerciseConn* does when it creates a PDU class object. It reads a packet from the network into a buffer, then instructs the PDU class object to use this buffer for its network representation. The data does not need to be copied into a buffer internal to the *DtPdu*.

There are four considerations to keep in mind when passing a pointer to external memory to the blank *DtPdu* constructor:

- It is your responsibility to insure that this external buffer is large enough to accommodate the largest network representation that this particular *DtPdu* object will need.
- It is your responsibility to insure that the lifetime of the buffer memory extends for the lifetime of the *DtPdu*.
- Once you pass a buffer pointer to a *DtPdu* constructor, it belongs to the *DtPdu* object for the duration of its lifetime. Modifying the buffer will result in undefined behavior when later using the *DtPdu*'s mutator or inspector functions.
- This memory will NOT be freed by the *DtPdu* object; since it was allocated by the application, it is the application's responsibility to free it.

8.1.8. Copying PDUs

DtPdu has the copy constructor and assignment operator overloaded. When you do an assignment, such as:

```
DtFirePdu pdu1, pdu2;
pdu1 = pdu2;
```

pdu1 now contains the same data as *pdu2*. The two objects do not, however, share any data objects.

As with the standard PDU class constructors, the *DtPdu* constructor has an optional buffer argument. (See "Using External Buffers in a PDU Class," on page 8-11.)

In all cases,

```
DtFirePdu pdu1;
DtFirePdu pdu2(pdu1, buffer);
```

has the same effect as

```
DtFirePdu pdu1;
DtFirePdu pdu2(buffer);
pdu2 = pdu1;
```

8.1.9. Obtaining An Object ID

In addition to the functions described in "DtExerciseConn Member Functions," on page 5-7 there is DIS-specific function - nextId(). nextId() obtains an ID for use with an entity (or other object) that you are simulating. nextId() returns an object of the type *DtObjectId*, defined in *hostStructs.h*.

You can choose your own entity IDs rather than obtain them using nextId().

DtObjectId is typedefed to *DtEntityIdentifier* (defined in *hostStructs.h*) – a class that represents the DIS identification triplet of site ID, application number, and entity number. *DtObjectID* is used rather than *DtEntityIdentifier* or RTI::ObjectID() when we want to refer to an identifier in a protocol-independent fashion.

The DIS version returns an identifier consisting of the *DtExerciseConn*'s site and host, and an entity number that is one greater than the one returned in the previous call to nextId(). There is no guarantee that the entity ID is unique in the DIS exercise.

8.2. Working With Non-Standard PDUs

Many VR-Link users need to use PDUs that are not part of the DIS standard, and thus not implemented in VR-Link. However, VR-Link can work with user-defined PDUs just like it treats PDUs that are implemented in VR-Link.

There are two ways to work with user-defined PDUs:

- Use *DtUnknownPdu* to represent your type of PDU
- Derive a class from *DtPdu*.

Using *DtUnknownPdu* is sometimes easier for simple PDUs that can be represented by C-style structures. In essence, the *DtUnknownPdu* just serves as a wrapper around such a structure. Deriving your own *DtPdu* class requires some extra work up front, but it is easier to use in your application, especially if it's a variable-length PDU, or one that has several variants.

8.2.1. Using *DtUnknownPdu*

A *DtUnknownPdu* (defined in *unknownPdu.h*) can be used to represent a PDU for which there is no PDU class. This includes new or experimental PDUs that you develop.

To send a PDU using the *DtUnknownPdu* class:

1. Fill a buffer with the exact bytes you want to send on the network (probably by filling in a C-structure). Make sure this structure includes the DIS PDU header whose structure is defined in *pduHeader.h* (in the *include/packets* directory).

   By using our "Net" types, such as *DtNetInt32* (defined in *NetTypes.h*), you insure platform independence. When you are on little endian machines, byte swapping is performed when assigning to a Net type, and when a Net type is implicitly cast to a native type.

2. Pass this network representation to the *DtUnknownPdu* from-network-representation constructor.

3. Send the PDU using the normal DtExerciseConn::sendStamped().

For example, suppose you create a Test PDU with two fields called *a* and *b*. We will choose PDU kind number 220 to represent this PDU:

```
// Define the structure of your network representation
typedef struct NetTestPdu
{
    DtNetPduHeader header;
    DtNetInt32 a;
    DtNetInt32 b;
} NetTestPdu;
// Create an instance of the structure and fill in the net
// representation
NetTestPdu netPdu;
netPdu.header.version = 5;
netPdu.header.DtExercise = 1;
netPdu.header.DtKind = 220
...
netPdu.a = 10;
netPdu.b = 11;
// Create a DtUnknownPdu from this network representation
DtUnknownPdu pdu((DtNetPduHeader*) &netPdu);
// Send the PDU as you normally would
exConn.sendStamped(pdu);
```

On the receiving side, *DtUnknownPdu* does not have addCallback() or removeCallback() functions, since a *DtUnknownPdu* does not know what PDU kind you are interested in. But you can still register callbacks with the *DtExerciseConn* on a particular PDU kind, in our case 220. Note the cast of the integer 220 to a *DtPduKind* enum here.

```
exConn.addPduCallback(DtPduKind(220), testCallback, NULL);
```

Since 220 is not a PDU kind that VR-Link knows about, it will create a *DtUnknownPdu* when it receives a packet of that kind and pass that to your callback function. You can cast the *DtPdu** to a *DtUnknownPdu** within your callback if you want to, but it's not necessary since the only useful thing you can do with a *DtUnknownPdu* is get its network representation, which is done through a *DtPdu* member function.

```
void testCallback(DtPdu* pdu, void *usr)
{
    // Get the PDU's network representation, and cast it to a
    // NetTestPdu structure, so that you can access the data
    NetTestPdu* netPdu = pdu->packet();
    int a = netPdu->a;
    int b = netPdu->b;
}
```

8.2.2. Deriving Classes from *DtPdu*

VR-Link's DtExerciseConn::send() and DtExerciseConn::sendStamped() can send any type of PDU, as long as the object passed to it is of a type derived from *DtPdu*. So, in order to send user-defined PDUs, you can write a derived class, after which you can immediately use your derived class with these functions.

When *DtExerciseConn* receives a packet from the network, it creates an object of the appropriate class derived from *DtPdu*, based on the PDU type in the packet. For instance, a *DtEntityStatePdu* is created if a packet representing an Entity State PDU is received. For this reason, VR-Link needs to know that your derived class exists. Registration of your class with VR-Link is usually done in your PDU class addCallback() function.

The following example creates a variable length PDU. The *TestPdu* class represents a hypothetical variable-length "Test PDU" - one that has three fields: an integer named "a", a variable-length array of integers named "b" whose cardinality is the value of "a", and a float called c. The code for this example is in *vrlink/examples/extend/testPdu.*

To derive a class from *DtPdu*:

1. Create the PDU layout, a structure that is used for the network representation of the PDU. (Network representations for all of the VR-Link PDUs, are in the *include/packets* directory.)

   By using our "Net" types, such as *DtNetInt32* (defined in *NetTypes.h*), you ensure platform independence. When you are on little endian machines, byte swapping is performed when assigning to a Net type, and when a Net type is implicitly cast to a native type.

   The first field in all network representations should be a *DtNetPduHeader.*

   If you are creating a fixed-length PDU, the structure should describe the entire layout of the PDU. Since this PDU is variable length, it is not possible to fully describe the PDU layout.

   Define as much of the network representation structure as is possible with a C-style structure. The cardinality of "b" in the structure doesn't matter, since we'll only be casting buffers to pointers to this structure, and not creating instances of the structure or relying on its size. We won't be able to access "c" at all through the structure, since it comes after the variable length array "b". We will have to use byte-arithmetic to reach "c".

```
typedef struct NetTestPdu
{
   DtNetPduHeader header;
   DtNetInt32 a;
   DtNetInt32 b[1];
} NetTestPdu;
```

2. Extend the DtPduKind enumeration, adding our chosen value. This is the number that ends up in the PDU header's kind field.

   The value you choose for PDU kind should not be one that is used by any other PDU. We recommend a value between 220 and 255.

   ```
   const DtPduKind TestPdu Kind = DtPduKind(220);
   ```

3. Create a type called *TestPduCb*. Functions of this type can be registered as callbacks to be called on receipt of *TestPDUs*. Forward declaration of *TestPdu* is necessary first.

   ```
   class TestPdu;
   typedef void (*TestPduCb) (TestPdu* pdu, void* usr);
   ```

4. Create the TestPdu class definition.

   – Each class derived from *DtPdu* must have the two standard constructors possessed by all PDU classes (see "DtPdu Constructors," on page 8-8). Your *DtPdu* can have additional constructors, but it must have the two standard ones.

   – PDU classes need to provide a definition for the virtual function internalGet-PduKind(), which returns the PDU kind value being used for this type of PDU.

   – Your class should provide the static member functions addCallback() and removeCallback() for callback management, and the static member function create(), that returns a new instance of the class.

   – Most *DtPdu* sub-classes provide inspector and mutator functions to examine and set the fields of your PDU, but VR-Link does not require that you do so.

The definition for the TestPdu class might look like this:

```
// Define the TestPdu class.
class TestPdu : public DtPdu
{
public:

// Blank constructor - constructs a blank TestPdu ready to
// be filled out and sent by application code. Required.

   TestPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

// From-network-representation constructor. Constructs a PDU from an
// already-filled-out buffer containing the exact bytes that represent this
// type of PDU. This constructor is used by VR-Link (through the static
// create function) to create an instance of this class from
// PDU data received from the network. Required.

   TestPdu(const NetTestPdu *initial,
      DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

// Destructor - Not required by VR-Link, but should be written if this class
// has any dynamic data that needs to be cleaned up.

   virtual ~TestPdu();

// Copy constructor. Not required by VR-Link, but needed if
// you will ever want to copy TestPdus.
```

```
   TestPdu(const TestPdu& orig);

// Assignment operator. Not required by VR-Link, but needed if
// you will ever want to copy TestPdus.

   TestPdu& operator=(const TestPdu& orig);

// Virtual function override. Prints the data values in the PDU. Not strictly
// required, but encouraged, since otherwise calling print() on an instance of
// this class will merely print raw hex data.

   virtual void printData() const;

// Inspector and mutator functions for the PDU class's data. These will look
// different for each DtPdu subclass. None are required by VR-Link, but the
// class will not be particularly useful without a way to get data in and out
// in a typesafe manner.

// Set/Get the value of a.

   virtual void setA(int val);
   virtual int a() const;

// Set/Get the index'th element of the "b" array.
// (Alternatively, we could have created setB() and b()
// functions that set or returned the whole array all at once.)

   virtual void setB(int index, int val);
   virtual int b(int index) const;

// Set/Get the value of c.

   virtual void setC(float val);
   virtual float c() const;

// Returns a pointer to the PDU's network representation by
// casting the value returned by the base class' packet()
// function to a NetTestPdu. We include both const and non-const versions.
// These functions are optional, but they make it easier to write
// inspectors and mutators.

   virtual NetTestPdu* netTestPdu();
   virtual const NetTestPdu* netTestPdu() const;

public:

// Create and return a new TestPdu. Basically a wrapper
// around the constructor. It is necessary because pointers to
// static functions like this can be registered with a
// DtPduFactory (which tells VR-Link how to construct an
// instance of a particular kind of PDU), but constructors can not. Required.

   static DtPdu* create(const DtNetPacket *initial,
      DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

// Wrappers around DtExerciseConn's add/removeCallback
// functions. Not required since the DtExerciseConn functions
// may be used instead, but strongly encouraged, so that the
// non-typesafe cast can be limited to this one place. In
```

```
// addition, within addCallback, we choose to register this
// class's create function with VR-Link, letting VR-Link know
// to create one of these to represent the right PDU kind. If
// this is not done within addCallback, it must be done
// explicitly by application code that uses this class.

   static void addCallback(DtExerciseConn *conn, TestPduCb cb, void *usr);
   static void removeCallback(DtExerciseConn *conn, TestPduCb cb, void *usr);

protected:

// Virtual function override. Returns the PDU kind associated
// with this PDU class. This value goes in the header of all
// instances of this PDU class. Required.

   virtual DtPduKind internalGetPduKind() const;
};
```

Defining the Blank PDU Constructor

Define the blank PDU constructor as follows:

```
TestPdu::TestPdu(DtBufferPtr buffer) :
   DtPdu()
{
int minimalSize = sizeof(DtNetPduHeader) + sizeof(DtNetInt32) +
      sizeof(DtFloat32);
   initPdu(minimalSize, buffer);
}
TestPdu::TestPdu(DtBufferPtr buffer) :
   DtPdu()
{
int minimalSize = sizeof(DtNetPduHeader) + sizeof(DtNetInt32) +
           sizeof(DtFloat32);
   initPdu(minimalSize, buffer);
}
```

The size passed to initPdu() is the size of the smallest legal *TestPdu*; that is, one with zero elements in the array *b*. The size of the PDU is therefore the size of the header plus the size of the DtNetInt32 field "a" plus the size of the DtNetFloat32 field "c".

initPdu() is a member of the base class *DtPdu*, and it handles allocating memory for the network representation of the PDU, and other related items.

---

i If this was a fixed-length PDU, the size would be the size of the NetTestPdu structure.

Defining the From-Network-Representation Constructor

The from-network-representation constructor should look like this:

```
TestPdu::TestPdu(const NetTestPdu *initial, DtBufferPtr buffer)
{
       initPdu(initial, buffer);
}
```

Defining internalGetPduKind()

Our definition of internalGetPduKind() should return the value 220, which we are using for our TestPdu.

```
DtPduKind TestPdu::internalGetPduKind() const
{
        //We defined TestPduKind in testPdu.h as DtPduKind(220).
   return TestPduKind;
}
```

Implementing the create() Function

You can implement the create() function as follows:

```
DtPdu* TestPdu::create(const DtNetPacket *initial, DtBufferPtr buffer)
{
// Return a new'ed instance of this class.
   return new TestPdu((NetTestPdu *) initial, buffer);
}
```

Implementing Callback Registration Functions

The callback-related functions should look like this:

```
void TestPdu::addCallback(DtExerciseConn *conn, TestPduCb cb, void *usr)
{
  // Register this class' create function with the DtExerciseConn's
  // pduFactory so that the DtExerciseConn will know to create an instance of
  // this class to represent incoming PDUs of the appropriate kind. If this
  // is not done here, the user of this class will have to explictly register
  // the class with the PDU factory himself.
  conn->pduFactory()->addCreator(TestPduKind, TestPdu::create);

  // Cast the TestPduCb to the more general DtPduCallbackFcn and pass it
  // along with the PDU kind to the DtExerciseConn who actually maintains the
  // callback lists for all PDU kinds.
  conn->addPduCallback(TestPduKind, (DtPduCallbackFcn) cb, usr);
}

void TestPdu::removeCallback(DtExerciseConn *conn, TestPduCb cb, void *usr)
{
  // Cast the TestPduCb to the more general DtPduCallbackFcn and pass it
  // along with the PDU kind to the DtExerciseConn who actually maintains the
  // callback lists for all PDU kinds.
  conn->removePduCallback(TestPduKind, (DtPduCallbackFcn) cb, usr);
}
```

Writing Mutator and Inspector Functions

For a fixed-length PDU, writing mutator and inspector functions is fairly straightforward. A pointer to the PDU's network representation is stored in a *DtPdu* member and can be obtained using DtPdu::packet(). The void* returned must be cast to a pointer to your network representation structure before fields are examined and set within your accessors. We define the function netTestPdu() to perform this cast. In our test PDU example, we can then use this function to obtain a pointer to the PDU's network representation structure.

When you write mutators for variable length PDUs, like the testPdu example, you need to call one of the following *DtPdu* member functions every time you change the size or layout of the PDU:

- insertBytes()
- deleteBytes().

In addition, you may need to do some pointer arithmetic to access certain fields of the PDU that cannot be accessed through members of the static structure.

insertBytes() inserts the indicated number of blank (zeroed) bytes at the desired offset into the PDU's network representation.

deleteBytes() deletes the desired number of bytes, starting with the indicated offset.

These functions handle reallocating the memory for the network representation if necessary, and will also make sure the size field in the header is updated to reflect the new size.

Since *a* represents the number of elements in the array *b*, we must insert or delete bytes from the PDU layout whenever we change *a* through its mutator.

Mutator for A

The code does the following:

1. Gets the old value of "a".
2. Either push down or pull up the rest of the PDU data depending on whether the value of "a" is increasing or decreasing.
3. Set the value of the "a" field.

```
void TestPdu::setA(int val)
{
    int oldA = a();

       if (val > oldA)
       {
            insertBytes(&(netTestPdu()->b[oldA]),
                (val - oldA)* sizeof(DtNetInt32));
       }
       else if (val < oldA)
       {
            deleteBytes(&(netTestPdu()->b[val]), (oldA - val) *
                sizeof(DtNetInt32));
       }
```

```
    netTestPdu()->a = val;
    }
```

Implement the Inspector for A

Implement the inspector:

```
int TestPdu::a() const
{
   return netTestPdu()->a;
}
```

Implement the Inspector and Mutator for B

1. Our *b* and setB() functions should take an index into the array, so that we know which element to inspect or set. We can do some bounds checking if we want:
2. Either do bounds checking, then set the index'th element of the "b" array to val, or do bounds checking, then return the index'th element of the "b" array.

```
void TestPdu::setB(int index, int val)
{
if (index < 0 || index ≥ a())
   {
      printf("TestPdu::setB(): index out of range.\n");
      return;
   }
   netTestPdu()->b[index] = val;
}

int TestPdu::b(int index) const
{
if (index < 0 || index ≥ a())
   {
      printf("TestPdu::b(): index out of range.\n");
      return 0;
   }
   return netTestPdu()->b[index];
}
```

Implement the Inspector and Mutator for C

The inspector and mutator for the *c* field must do some pointer arithmetic to find where the data is stored, since it can't be accessed through the structure.

```
void TestPdu::setC(float val)
{
   // We can find "c" after the last element of "b".
   DtNetFloat32* netC = (DtNetFloat32*) &(netTestPdu()->b[a()]);

   // val will be implicitly cast to a DtNetFloat32.
   *netC = val;
}

float TestPdu::c() const
{
   // We can find "c" after the last element of "b".
   DtNetFloat32* netC = (DtNetFloat32*) &(netTestPdu()->b[a()]);
```

```
    // netC will be implicitly cast to a native float.
    return *netC;
}
```

8.3. Configuring Your Connection to the DIS Network

As described in "Connecting to Exercises," on page 5-4, a *DtExerciseConn* serves as an application's connection to a DIS or HLA exercise. The DIS version is defined in *dExerciseConn.h*.

A DIS *DtExerciseConn* uses a *DtSocket*, defined in *Socket.h*, as its low-level interface to a virtual network. *DtSocket* is an abstract class whose virtual functions DtSend() and DtRecv() allow you to send and receive a packet of data among applications without regard to the actual communication mechanism used. Subclasses of *DtSocket* implement different methods of communication and take different configuration parameters.

For example, a *DtNetSocket* (defined in *NetSocket.h*) is used when a *DtExerciseConn* wants to communicate directly with an IP/UDP network. A *DtClientSocket* (defined in *ClientSocket.h*) is used when we wish to communicate through the VR-Link packet server.

8.3.1. *DtExerciseConn* Constructors

DtExerciseConn has several constructors. Two of them create a *DtSocket* based on their arguments, and the third takes an existing *DtSocket* as an argument.

All constructors take an exercise ID as an argument. The exercise ID is copied into the header of outgoing PDUs when sendStamped() is used, and is used for filtering of incoming PDUs as well. Only those PDUs on the matching exercise are passed to the application by the exercise connection. If a *DtExerciseConn*'s exercise ID is set to zero, it will read and process PDUs from all exercises. However, PDUs should never be sent to exercise zero, so exercise ID zero should usually be used only in applications that are listen-only. Once a *DtExerciseConn* is constructed, you can change or examine the current exercise ID with setExerciseId() and exerciseId().

Standard Constructor

The standard *DtExerciseConn* constructor (described in "Creating an Exercise Connection for DIS," on page 5-5):

- Checks to see if a packet server is running on the specified UDP port
- Creates a *DtClientSocket* to communicate with it if there is one
- Creates a *DtNetSocket* to communicate directly to the network if there isn't.

In either case (when the packet server is not in internal mode), the *DtExerciseConn* receives any unicast or broadcast packets destined for the machine on the specified port, and, by default, sends packets to the broadcast address of the machine's primary network interface (when no address is specified in a call to send() or sendStamped()).

A Constructor that Takes A Destination Address

The second *DtExerciseConn* constructor is similar to the first, but in addition to the four arguments expected by the standard constructor, it takes a destination address. The destination address is used as the default IP address to which outgoing packets are sent. It can be a unicast, multicast or broadcast address.

For example, to indicate that packets should be sent only to the machine with the address 207.86.232.1:

```
DtExerciseConn(3000, DtStringToInetAddr("207.86.232.1"));
```

In this case, we will still receive packets destined for our broadcast and unicast addresses.

Using A Secondary Network Device

This constructor is also useful when you want to use a device other than a machine's primary network device. Pass the broadcast address associated with the desired device as the destination address.

The two devices must be on different networks. There is no way for a UDP socket to distinguish between two devices on the same network.

A Constructor that Takes A Pointer to A *DtSocket*

The third *DtExerciseConn* constructor takes a pointer to a *DtSocket*, and thus gives you full control of how the *DtSocket* is created. You can even derive your own type of *DtSocket* and pass an instance of it to a *DtExerciseConn*. (Perhaps you would like to create a SharedMemorySocket.) In this case, the socket is not deleted by the *DtExercise-Conn* object. Because it is allocated by the user, it is the user's responsibility to delete it. However, the user must not delete the *DtSocket* until after the *DtExerciseConn* has been destroyed.

See the following sections for more configuration information. Once a *DtExerciseConn* is constructed, its socket() member function will return a pointer to the *DtSocket* it is using.

8.3.2. Configuring a DtNetSocket

If you create your own *DtSocket* before passing it to a *DtExerciseConn*, you have several additional configuration choices.

If you are constructing a *DtNetSocket*, (defined in *NetSocket.h*), that class's constructors take a flags argument in addition to the port and destination address arguments described above. Flags can be a bitwise OR of zero or more of the following:

- DtRead
- DtWrite
- DtReadWrite
- DtQuiet
- DtBindToDestAddr.

By default, a *DtSocket*, and therefore a *DtExerciseConn*, can be used for both reading and writing to the network. But by creating a *DtNetSocket* using either the DtRead or DtWrite flag alone, you can create a read-only or write only connection. For example:

```
DtNetSocket sock(3000, DtWrite);
DtExerciseConn conn(&sock, 1, 1, 1);
```

The DtQuiet flag keeps the constructor from printing its startup diagnostics.

The DtBindToDestAddr flag configures the socket to listen for only those packets destined for exactly the address specified as the destination address. For example, if you want to receive packets on a certain multicast address only, but not broadcast packets or unicast packets directed to your machine, you can specify the desired address as the destination address, and use DtBindToDestAddr:

```
DtNetSocket sock(3000, DtStringToInetAddr("225.5.6.7"),
           DtBindToDestAddr | DtReadWrite);
           DtExerciseConn conn(&sock, 1, 1, 1);
```

8.3.3. Configuring a DtClientSocket

A *DtClientSocket*, defined in *ClientSocket.h*, is a *DtSocket* that sends and receives packets with the help of the VR-Link packet server. Depending on the state of the packet server, packets may be sent to and received from the network or may be passed only among the server's clients on the local host.

The constructor tries to establish communication with the VR-Link Packet Server running on the specified port. If there is no server running on the port, there is a fatal error. To avoid a fatal error, an application can check whether a server is alive before attempting to create a *DtClientSocket* using the C-style function DtPacketServer-IsAlive(), declared as follows:

```
int DtPacketServerIsAlive(int port);
```

For example:

```
DtSocket *skt;
if (DtPacketServerIsAlive(disPort))
```

```
{
    skt = new DtClientSocket(disPort);
}
else
{
    skt = new DtNetSocket(disPort);
}
```

In addition to the port and destination address arguments described in the *DtExercise-Conn* section, *DtClientSocket* takes two additional optional arguments:

- A size (in bytes) for the queue through which the packet server passes incoming messages
- A flags argument, which is a bitwise OR of zero or more configuration flags. See *ClientSocket.h* for the list of valid flags.

When using a *DtClientSocket*, you can use the function isInternal() to find out if the packet server you are connected to is in internal mode or not.

Communicating Across Networks

Normally, DIS PDUs are sent using broadcast or multicast UDP. When using broadcast, a PDU is sent once with a network's broadcast address as its destination address, and all machines on that network can receive the message. Similarly, when a PDU is sent using multicast, all machines on the network that are subscribed to that multicast address can receive the PDU.

Most routers, bridges, and gateways filter out multicast and broadcast traffic. This is usually true even for routers between subnets on the same LAN. So using ordinary broadcast or multicast, applications on different LANs, or different subnets cannot communicate.

If you are using multicast, and your applications are on different subnets on the same LAN, configuring your network hardware so that it will forward multicast packets from subnet to subnet will usually be enough to get DIS applications on the different subnets to communicate with each other. If you have applications on different LANs on a WAN, you would have to configure all routers separating the two LANs, so that none of them filter out multicast packets. This is usually not feasible, especially when your WANs are part of the Internet, and not connected only to each other.

When using broadcast, the physical blocking of packets from flowing between networks is only half of the problem. Even if your packets could physically get from one network to another, they would likely be ignored by the intended receivers. The reason is that different networks (even different subnets) usually have different broadcast addresses. So a packet that contains network A's broadcast address as its destination address, will not be interpreted by machines on network B as a broadcast packet, and will therefore be ignored.

For these reasons, multicast and broadcast are usually not feasible when you want to communicate across networks. The alternative is unicast, or point-to-point, that is, sending a packet directly to the address of the recipient. (Unfortunately, this requires knowing the addresses of all participants in an exercise, which is not necessary when using broadcast or multicast.)

VR-Link contains a kind of *DtSocket* called *DtGroupSocket* (*groupSocket.h*) that allows you to send PDUs to many different unicast addresses with a single call to its sending functions. To communicate with exercise participants on other networks:

1. Create a *DtGroupSocket*.
2. Configure the *DtGroupSocket* with the IP addresses of all participants (including yourself).
3. Instruct the *DtExerciseConn* to use this *DtGroupSocket* instance. You can add IP addresses to a *DtGroupSocket* using its addRemoteSite() member function.

For example, if you are running an exercise over a WAN with participants at the following IP addresses: 123.123.123.123, 124.124.124.124, and 125.125.125.125, configure your connection to the DIS network as follows:

```
// Port number is optional.  Default is 3000
DtGroupSocket sock(3000);
socket.addRemoteSite(DtStringToInetAddr("123.123.123.123");
socket.addRemoteSite(DtStringToInetAddr("124.124.124.124");
socket.addRemoteSite(DtStringToInetAddr("125.125.125.125");

// Now tell the DtExerciseConn to use this as its socket
DtExerciseConn conn(&sock, 1, 1, 1);
```

The addresses that you add are really just addresses that packets will get sent to. Broadcast addresses can be used here, if desired. If you have several applications on each of several different LANs, you can save bandwidth on each LAN by replacing the addRemoteSite() calls for local machines in each application with a single call, specifying the local LAN's broadcast address. A single broadcast packet will be sent for the local LAN instead of a copy being sent for each local machine. Truly remote addresses still have to be added individually.

8.3.4. Subscribing to Multicast Addresses

DtExerciseConn allows subscription to one or more multicast addresses. Valid multicast addresses are those that lie between 224.0.0.0 and 239.255.255.255, but you should probably not use any that start with 224.0.0, in order to avoid potential conflict with addresses used by operating systems and network services.

---

i Not all platforms support multicast.

Interest in a particular multicast address is indicated using DtExerciseConn::addInterestInMcastAddr(). When a multicast group is no longer of interest, call subtractInterestInMcastAddr().

Different code modules may add and subtract interest in a particular group independently. As long as the number of calls to addInterest() outnumbers calls to subtractInterest() for a particular group, packets on that multicast address are received.

> To send to a particular multicast address, include that address as the **destAddr** argument to send() or sendStamped().

You do not have to be subscribed to a multicast address in order to send to it.

If the destination address passed to a *DtExerciseConn* constructor is a multicast address, you do not need to call addInterest() for that address. You will automatically be subscribed.

---

i On many platforms, multicast subscription works on a per-host basis, rather than on a per-application basis. This means, that if you have several applications listening to the same port on the same machine, and one of them subscribes to a particular multicast address, all of the applications will receive packets on that multicast address, even if they did not explicitly subscribe.

If there are several routers on your network between machines that are exchanging PDUs using multicast, it may be necessary to increase the Time To Live (TTL) for outgoing multicast packets, to insure that the packets are actually forwarded across the routers. This can be achieved using DtSocket::setMcastTtl().

8.3.5. Filtering PDUs

A *DtExerciseConn* can instruct its *DtSocket* to filter out PDUs based on certain criteria. In this way, unwanted PDUs require no processing by the *DtExerciseConn*. This is especially valuable if the *DtExerciseConn* is using a *DtClientSocket* connected to a packet server. Filtering is then done in the asynchronous packet server, which may be running on a different processor than your application.

By default, packets that are not of an expected protocol version and packets whose exercise IDs do not match the exercise ID of the *DtExerciseConn* are filtered out. Expected protocol versions are those between the values of the global variables DtProtocolVersionToRecvMin and DtProtocolVersionToRecvMax, declared in *pdu.h*. These parameters default to 4 and 6 respectively, if you are building with DtDISVERS=5 (so that we accept DIS 2.0.4 (IEEE 1278.1) and 2.1.4 (IEEE 1278.1a) PDUs), and to 3 and 3 if you are building with DtDISVERS=3 (so that we accept DIS 2.0.3 PDUs only).

In addition, the *DtExerciseConn* maintains a list of all PDU kinds in which it is interested. This includes all PDU kinds on which callbacks have been registered, as well as any PDU kinds in which an interest has been explicitly added using DtExerciseConn::addInterestInPduKind(). (subtractInterestInPduKind() removes interest in a PDU kind). The *DtExerciseConn*'s *DtSocket* filters out all PDU kinds in which it has no interest. The *DtPduKind* enumeration is in *disEnums.h*.

Filtering can be turned on and off with *DtExerciseConn*'s functions disableFiltering() and enableFiltering().

8.3.6. Bundling and Unbundling PDU Packets

VR-Link supports the bundling of multiple PDUs into a single network packet, and the unbundling of such packets into their constituent PDUs.

Bundling is off by default, but can be turned on using *DtSocket*'s setBundling() function. Pass to it the desired maximum size for a packet. When bundling is on, a sequence of packets passed to a *DtSocket*'s DtSend() function (for example, from DtExerciseConn::sendStamped()), are concatenated together. The bundle is passed to DtSocket::sendTo() whenever a packet would push the bundle size beyond **maxSize**.

A good choice for **maxSize** is the maximum size of the data portion of an ethernet packet: 1464 bytes. If you are using a networking scheme other than ethernet, use the maximum packet size for that scheme. Bundling is turned off by passing a **maxSize** of 0 to setBundling().

If you turn on bundling, other participants in your simulation must be able to unbundle the packes.

You can force sending of the current bundle with DtSocket::flush(). If you are using bundling, flush() should be called at the end of each simulation frame, to ensure that all packets generated during a frame get sent before the next frame begins.

isBundling() returns 0 if bundling is off; otherwise it returns the maximum bundle size.

Unbundling is enabled by default, but it can be turned on or off by passing 0 or 1 to DtSocket::setUnbundling(). With unbundling on, when a packet arrives containing more than one PDU, successive calls to DtRecv() (the call made by DtExercise-Conn::netRead()) return successive PDUs from the packet. On the next call to DtRecv() after the last PDU from a packet has been returned, the first PDU from the next packet is returned.

If unbundling is disabled, but a bundled packet is received, all but the first PDU in the packet is ignored.

When you are using the packet server, unbundling takes place in the server. So for *DtClientSockets*, setUnbundling() merely instructs the packet server to turn unbundling on. Since another *DtClientSocket* might turn unbundling on or off in your client's packet server unbeknownst to you, it is possible that your *DtClientSocket* will unbundle when you have turned unbundling off with setUnbundling() or vice versa.

8.3.7. Sending and Receiving Packets Using *DtSocket*

A *DtSocket* is an object through which packets can be sent to and received from other applications, without regard to the actual communications mechanism used.

Sending Packets

There are two functions involved in the sending of packets: DtSend() and sendTo(). sendTo() is a virtual function that, in derived classes, does the actual sending of the data. DtSend() performs any processing common to all *DtSockets* (such as bundling) before passing data on to sendTo() for actual sending.

A buffer, the number of bytes to send, and an optional address are passed to these functions. If the address is omitted, the *DtSocket*'s default destination address is used. If the packet cannot be sent, -1 is returned. Otherwise, the number of bytes actually sent is returned. DtExerciseConn::send() and DtExerciseConn::sendStamped() use DtSocket::DtSend() to send DIS PDUs.

DtSockets don't know anything about byte swapping or PDU layout (with the exception of some knowledge of PDU header layout used for filtering). The data contained in the buffer passed to *DtSocket*'s sending functions are the exact bytes that will go out onto the network.

Receiving Packets

DtRecv() is used to receive packets through a *DtSocket*. This is the function called by DtExerciseConn::netRead() to read a PDU from the exercise. There are three different versions of DtRecv(). The first version reads the next packet available, and copies up to *size* bytes into buffer *buffer*. The number of bytes read from the packet is returned. If there is no packet available, 0 is returned:

```
int DtRecv(caddr_t buffer, size_t size);
```

The second version sets *buffptr* to point to an internal buffer containing the packet read, and returns the size of the packet. If there is no packet available, *buffptr* is set to NULL, and size 0 is returned:

```
int DtRecv(caddr_t *buffptr, int *status);
```

If status is non-NULL, it is set to indicate a return status. The status is DtSOCKET_PACKET if a packet was successfully received, and DtSOCKET_NO_PACKET if no packets are available. If a packet is received but is filtered out, the return code is set to DtSOCKET_FILTERED_PACKET, *buffptr* is set to NULL, and 0 is returned. Return codes other than these may be used when DtRecv() returns because it received a control message that does not represent an actual packet on the virtual network.

For callers uninterested in the packet size, the third version returns a pointer to an internal buffer containing the packet read, or NULL if none is available:

```
caddr_t DtRecv();
```

DtSocket Functions

Table 8-2 describes *DtSocket* member functions other than send and receive functions.

Table 8-2: DtSocket member functions (Sheet 1 of 2)

Function	Description
lastSourceInetAddr()	Returns the IP address of the sender of the packet received in the last call to any of the DtRecv() functions. If an implementation does not support reporting of source IP addresses, an address of 0 is returned.
DtReportPktsRcvd()	Returns the total number of packets read through the *DtSocket* since it was constructed.
DtBecomeNonBlocking()	Ensures that reads and writes are non-blocking. This is called by all *DtExerciseConn* constructors, even for sockets which are passed to *DtExerciseConn* already constructed. If this function is not called, *DtNetSocket* reads and writes and *DtClientSocket* reads will block. *DtClientSocket* writes are always non-blocking.
readFd()	Returns the read file descriptors. Those types of *DtSockets* that do not have associated file descriptors return -1.

Table 8-2: DtSocket member functions (Sheet 2 of 2)

Function	Description
writeFd()	Returns the write file descriptors. Those types of *DtSockets* that do not have associated file descriptors return -1.
notifyMe()	Lets the *DtSocket* know that the caller is using the presence of data on the *DtSocket's* read file descriptor as an indicator that there are packets available on the virtual network. This mechanism is useful for applications that are event-driven in nature, and wish to wait via a DtSelect() for input from one of various devices. The *DtSocket* ensures that there is data on the read file descriptor whenever there are packets available, even if the underlying implementation does not cause this to be true automatically.
listenToMulticastGroup() dropMulticastGroup()	Do what their names imply, but rarely need to be called directly, since they are called by the preferred DtExerciseConn::addInterestInMcastAddr() and subtractInterestInMcastAddr(). Do not use these functions if you are using the *DtExerciseConn* versions in the same application.
setMcastTtl()	Sets the IP time-to-live (TTL) field for multicast packets to ttl hops (or seconds). The TTL field controls the number of routers a packet is forwarded through. The *DtSocket* constructor does not initialize the multicast TTL to any particular value. Instead, the system default is used. Rather than calling setMcastTtl() in an application, you may want to change the system-wide default for multicast TTL (on platforms that support this). For example, under Solaris2, the following sets the system default to 64 hops: `ndd -set /dev/ip ip_broadcast_ttl 64` setMcastTtl() returns 1 on success and 0 on error (for example, on a platform that does not support multicast).
isOk()	Returns 1 if the *DtSocket* is currently able to send and receive packets, 0 otherwise (for example, if a *DtClientSocket's* packet server is down).
exerciseId() protocolVersion() pduKind() ipAddr()	Return pointers to *DtFilterTests* (defined in *Socket.h*) that control filtering on various attributes of PDUs. However, most of these features are better accessed through the relevant *DtExerciseConn* functions.

8.4. Intercepting Incoming Entity State PDUs

In Chapter 5, *The Protocol-Independent Interface*, we mention that processing incoming state updates for a particular entity must account for the protocol, because the concept of an HLA attribute update is fairly different from the concept of a DIS entity state PDU.

In DIS, there are two ways to intercept incoming entity state PDUs:

- Register a callback on entity state PDUs, just as you would on any other kind of PDU.
- Override DtReflectedEntity::processEntityState().

When an entity state PDU arrives, it is dispatched to a callback that *DtReflectedEntityList* registers with the *DtExerciseConn*. Within this callback, the REL calls the appropriate *DtReflectedEntity*'s processEntityState(), depending on the entity ID in the PDU.

DtReflectedEntity's default implementation of this virtual function is to update its own *DtEntityStateRepository* based on the contents of the PDU. But by subclassing *DtReflectedEntity*, you can provide an alternate implementation. Your version of this function should probably call down to DtReflectedEntity::processEntityState() to make sure that the ESR is properly updated even if there is additional work you want to perform here.

Remember that when subclassing *DtReflectedEntity*, you must also subclass *DtReflectedEntityList* so that it knows to create your new type of *DtReflectedEntity*. See "Creating Reflected Entity Lists," on page 5-21 for examples of subclassing these two classes.

Utilities

VR-Link provides several utilities to help make it easier to write a simulation application. These utilities include coordinate conversion routines, matrix and vector classes and functions, time-related functions, and others.

This chapter describes the utilities as follows:

9.1. Vector and Matrix Classes

This section describes the classes that you use for vectors and matrices, *DtVector* and *DtDcm*.

9.1.1. Using the DtVector Class

You can use the *DtVector* class, found in *vlVector.h*, to represent vectors in three dimensional space.

You can initialize *DtVectors* with their three components, as in:

```
DtVector vec(10.0, 20.0, 30.0);
```

The default constructor, which takes no arguments, constructs a vector containing three zeros.

The copy constructor, assignment operator, equivalence and subscripting operator are defined, so you can do things like:

```
DtVector vec(10.0, 20.0, 30.0);
DtVector vec2(vec);
DtVector vec3 = vec;
DtVector vec4;
vec4 = vec;
if (vec == vec2) {...}
```

and

```
double x = vec[0];
double y = vec[1];
double z = vec[2];
```

or

```
vec[0] = x;
vec[1] = y;
vec[2] = z;
```

DtVector also has a string() member function that returns a text representation of the contents of the vector, for example:

```
{          10.000,          20.000,          30.000}
```

The following static member functions of *DtVector* will return some commonly used vectors:

```
zero()     {0,0,0}
i()        {1,0,0}
j()        {0,1,0}
k()        {0,0,1}
ones()     {1,1,1}
```

9.1.2. Using the DtDcm Class

DtDcm, found in *dcm.h*, is used to represent 3x3 matrices. The name DCM stands for direction cosine matrix, and reflects the fact that we often use the class to store that particular type of matrix in order to represent an orientation. (Alternatively, orientation can be represented as a set of three Euler angles in the Tait Bryan sequence, using the *DtTaitBryan* class. For more information, see "Orientation, Euler Angles, and DtTait-Bryan," on page 9-5.)

A *DtDcm* can be constructed from it's nine components, as in:

```
DtDcm mat(1.0, 2.0, 3.0,
          4.0, 5.0, 6.0,
          7.0, 8.0, 9.0);
```

or from three component vectors:

```
DtDcm ident(DtVector::i(),
          DtVector::j(),
          DtVector::k());
```

If the constructor is used with no arguments, a matrix containing nine zeros is constructed.

The copy constructor and assignment operators are defined, so you can do things like:

```
DtDcm mat(1,2,3,4,5,6,7,8,9);
DtDcm mat2(mat);
DtDcm mat3 = mat;
DtDcm mat4;
mat4 = mat;
```

The subscripting operator is defined as well, so you can obtain any of the three vectors that make up the matrix:

```
DtVector firstRow = mat[0];
```

In conjunction with *DtVector*'s subscripting operator, the familiar double subscript syntax is supported:

```
double upperLeft = mat[0][0];
double lowerRight = mat[2][2];
```

or

```
mat[0][0] = 10.0;
mat[2][2] = 20.0;
```

DtDcm has static member functions, zero() and identity(), that return the zero matrix and identity matrix respectively.

9.1.3. *DtVector* and *DtDcm* Semantics

Although VR-Link functions usually employ pass-by-pointer semantics when expecting objects that they will modify, functions that operate on *DtVectors* and *DtDcms* are exceptions. These objects are usually passed to VR-Link functions by reference, so that the subscripting operator can be used without the awkwardness of having to dereference a pointer. In functions that do not change *DtVectors* and *DtDcms*, they are passed by const reference, just like other VR-Link classes.

We've created some typedefs to make this explicit. They are listed in Table 9-1.

Table 9-1: DtVector and DtDcm typedefs

Function	Equivalent function
DtVectorRef	DtVector&
DtDcmRef	DtDcm&
DtConstVector	const DtVector&
DtConstDcm	const DtDcm&

DtVector can be passed to a function that takes a DtVectorRef or DtConstVector, while a DtDcm can be passed to a function that takes a DtDcmRef or DtConstDcm.

9.1.4. Vector and Matrix Manipulation Functions

LibMatrix.h contains a variety of C-style functions that operate on *DtVector* and *DtDcm*. The standard operations are included:

- addition
- subtraction
- multiplication
- scaling
- dot products
- cross products
- negation
- normalization
- determinant calculation
- transpose
- inverse.

And so on.

9.2. Orientation, Euler Angles, and DtTaitBryan

Although direction cosine matrices are often used in VR-Link to represent orientation, an alternative representation is a set of three Euler angles. Euler angles represent the orientation of a body in space with respect to a certain coordinate system as three ordered rotations needed to go from the reference coordinate system to the body coordinate system.

DIS and the RPR FOM in HLA use Euler angles to represent the orientation of an entity with respect to the geocentric coordinate axes. (The body coordinate system is defined by X out the front, Y out the right side, Z out the bottom of the entity.)

There are many possible ways to order the three rotations, but the one used by DIS and the RPR FOM, and hence VR-Link, is known as the Tait-Bryan sequence. The first rotation is about the Z axis, the second about the new Y axis, and the third about the newest X axis. The three angles are usually denoted by psi, theta, and phi respectively.

These three angles do not necessarily correspond to the familiar heading, pitch and roll of an entity. Euler angles correspond to heading, pitch and roll only when the reference coordinate system is a topographic system. Specifically, the Euler angles sent in DIS entity state PDUs or RPR FOM entity updates use the geocentric coordinate system as a reference, and thus do not correspond to heading, pitch and roll. (See "Coordinate Conversions," on page 9-7 for information on converting between geocentric-reference Euler angles and topographic-reference Euler angles.)

VR-Link's *DtTaitBryan* class, defined in *taitBryan.h*, is used to represent a set of Euler angles. A *DtTaitBryan* can be constructed from its three component angles, specified in radians, for example:

```
DtTaitBryan angles(3.14, 0.0, 1.57);
```

The class has the following inspector and mutator functions that provide access to the individual angles:

- psi()
- setPsi()
- theta()
- setTheta()
- phi()
- setPhi().

The assignment operator, equality operators, and copy constructor are implemented for *DtTaitBryan* as well. The string() member function provides a string representation of the contents of the object, in a similar fashion to DtVector::string().

9.2.1. Converting Between Euler Angles and Matrix Representation

Euler.h contains functions that convert between the two representations of orientation.

These functions are not converting orientations from one coordinate system to another; they are converting between two ways of expressing an orientation within a given coordinate system.

Given a set of Euler angles that represent the orientation of a body with respect to some reference frame (say, the geocentric frame, or a topographic frame), DtEuler_to_BodyToRef() produces a rotation matrix (or *DtDcm*) that can rotate from body coordinates to reference coordinates, while DtEuler_to_RefToBody() produces its inverse – a matrix that can rotate from reference to body coordinates.

For example:

```
DtTaitBryan geocEulerAngles(...);
DtDcm bodyToGeoc;
DtEuler_to_BodyToRef(geocEulerAngles, bodyToGeoc);
```

The functions DtBodyToRef_to_Euler(), and DtRefToBody_to_Euler() go the other way – producing Euler angles from the matrix representation.

9.3. Coordinate Conversions

VR-Link supports several coordinate systems and provides classes and functions for converting locations, vectors, and orientations from one system to another.

9.3.1. Geocentric Coordinates

DIS and the RPR FOM in HLA specify that world locations, velocities, accelerations and orientations be represented with respect to a right-hand geocentric Cartesian coordinate system. The origin of this geocentric coordinate system is the center of the earth. The positive X-axis passes through the prime meridian at the equator; the positive Y-axis passes through 90 degrees east longitude at the equator; and the positive Z-axis passes though the north pole (Figure 9-1).

The *DtVector* class is typically used to represent locations and vectors in geocentric coordinates, while *DtTaitBryan* or *DtDcm* is used to represent orientation.

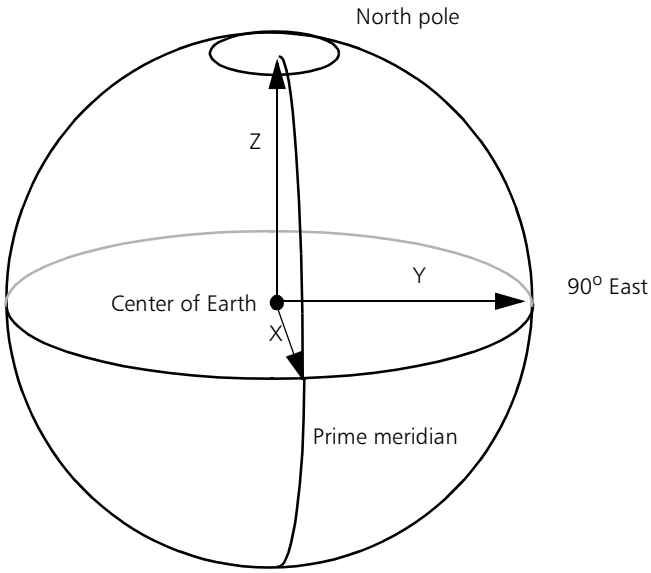


Figure 9-1. Geocentric coordinate system

9.3.2. Geodetic Coordinates

Geodetic coordinates are another means of specifying location in the world. VR-Link geodetic coordinates consist of latitude and longitude in radians, and altitude in meters above a certain reference ellipsoid, for example, the surface of the earth at sea level.

VR-Link uses the class *DtGeodeticCoord* (defined in *geodCoord.h*) to represent geodetic coordinates. A *DtGeodeticCoord* can be initialized from a latitude, longitude, and altitude. Negative latitudes are south latitudes; negative longitudes are west longitudes.

```
DtGeodeticCoord myLoc(DtDeg2Rad(45.0), DtDeg2Rad(30.0), 1000.0);
```

If the default constructor is used, the resulting object represents a latitude, longitude, and altitude of zero.

DtGeodeticCoord has the following inspectors and mutators to get and set values for the individual components of a geodetic coordinate:

- lat()
- setLat()
- lon()
- setLon()
- alt()
- setAlt().

DtGeodeticCoord also has member functions that enable you to convert to and from geocentric coordinates. DtGeodeticCoord::geocentric() returns the geocentric equivalent of a geodetic coordinate. For example:

```
DtGeodeticCoord geod(DtDeg2Rad(30.0), DtDeg2Rad(100.0), 1000.0);
DtVector geoc = geod.geocentric();
```

After these lines of code, geoc will contain the coordinate:

{-960122.075, 5445122.868, 3170873.735}

which represents the same point in space as 30 degrees north latitude, 100 degrees east longitude, 1000 meters altitude.

A similar function is DtGeodeticCoord::getGeocentric(), which, rather than returning a geocentric coordinate, sets the value of an existing *DtVector*, for example,

```
DtGeodeticCoord geod(DtDeg2Rad(30.0), DtDeg2Rad(100.0), 1000.0);
DtVector geoc;
geod.getGeocentric(geoc);
```

Going the other direction, DtGeodeticCoord::setGeocentric() sets the value of a *DtGeodeticCoord* to the lat/long/alt equivalent of a given geocentric coordinate:

```
DtVector geoc(-960122.075, 5445122.868, 3170873.735);
DtGeodeticCoord geod;
geod.setGeocentric(geoc);
```

Also in *geodCoord.h*, are the C-style functions DtGeocToGeod() and DtGeocToGeod() that can be used instead of the member functions described previously to convert between geocentric and geodetic coordinates, although their use is discouraged in favor of the member functions.

Choosing a Reference Ellipsoid

The default reference ellipsoid used for geocentric to geodetic conversions is WGS84, but this is configurable through the function DtUseMapDatum(), declared in *geodCoord.h*. Any *DtMapDatum* can be passed to this function. *DtMapDatum* is defined as:

```
typedef struct DtSpheroid
{
DtFloat64 semiMajor; // semimajor axis of the ellipsoid,
               //in meters
DtFloat64 semiMinor; // semiminor axis of the ellipsoid,
               // in meters
} DtSpheroid;
typedef struct DtMapDatum
{
    DtSpheroid spheroid;
    DtFloat64 datumShift[3]; // added to wgs84 to get GCC
               //in this datum
} DtMapDatum;
```

VR-Link has the following *DtMapDatums* pre-defined in *geodCoord.h*:

```
DtWGS84, DtED50, DtNASANAD27, DtNASBNAD27, DtCONUSNAD27.
```

If you'd like to use the ED50 reference, for example, try

```
DtUseMapDatum(&DtED50);
```

9.3.3. Topographic Coordinates

VR-Link defines a topographic coordinate system as a right-handed Cartesian coordinate system whose X-Y plane is tangent to the earth's surface at the origin, with the positive X axis pointing north, the positive Y axis pointing east, and the positive Z axis pointing down (Figure 9-2). Obviously, there are an infinite number of topographic coordinate systems – one for each point on the earth's surface.

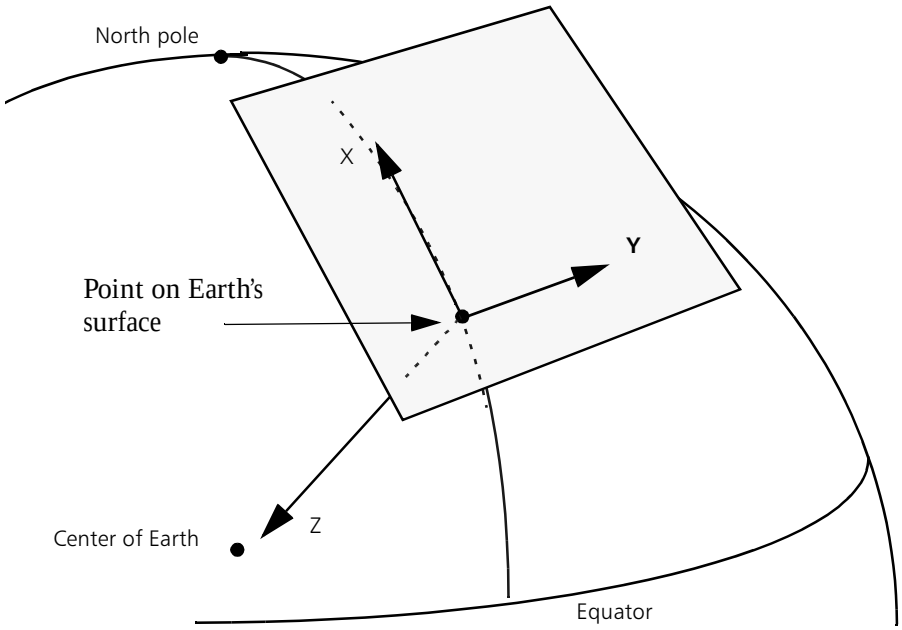


Figure 9-2. Topographic coordinate system

A *DtVector* is usually used in VR-Link to represent a topographic coordinate.

DtCoordTransform

The easiest way to convert between geocentric and topographic coordinates is by using a *DtCoordTransform* object, defined in *LibMatrix.h*. A *DtCoordTransform* is an object that can transform locations, vectors, or orientations from one Cartesian coordinate system to another.

In general, a *DtCoordTransform* can be constructed by passing to its constructor the origin of one system with respect to the other, and a rotation matrix that indicates the orientation of one system with respect to the other. However, in the case of a geocentric to topographic transformation, VR-Link provides the function DtGeocToTopoTransform() to initialize a *DtCoordTransform* for you. This function, declared in *topoCoord.h*, does the following:

1. Takes the latitude and longitude of a point on the earth and computes the rotation matrix that expresses the orientation of the topographic system defined by that point
2. Initializes a *DtCoordTransform* using this matrix along with the geocentric representation of the chosen point.

For example, to create a *DtCoordTransform* that can convert between geocentric coordinates and the topographic coordinate system whose origin is at 30 degrees north latitude, 100 degrees east longitude, you can do the following:

```
DtCoordTransform geocToTopo;
DtGeocToTopoTransform(DtDeg2Rad(30.0), DtDeg2Rad(100.0),
           &geocToTopo);
```

Now, this transform can be used to convert from geocentric to topographic coordinates. For example, if we have a geocentric location in a *DtEntityStateRepository*, and we wish to convert it to our topographic frame, we can use DtCoordTransform::coordTrans():

```
DtEntityStateRepository *esr = ...;
DtVector topoLocation;
geocToTopo.coordTrans(esr->location(), topoLocation);
```

If we want to do the reverse, we can initialize a second *DtCoordTransform* to convert from topographic to geocentric using the member function setByInverse():

```
DtCoordTransform topoToGeoc;
topoToGeoc.setByInverse(geocToTopo);
```

Now, we can use topoToGeoc() to convert to geocentric, say, for filling in a locally simulated entity's ESR:

```
DtEntityStateRepository *esr = ...;
DtVector geocLocation;
topoToGeoc.coordTrans(topoLocation, geocLocation);
esr->setLocation(geocLocation);
```

In addition to coordTrans(), *DtCoordTransform* also has vecTrans() and eulerTrans() member functions for converting vectors (such as velocity and acceleration), and orientations (expressed as Euler angles) between two different Cartesian coordinate systems.

Heading, Pitch and Roll

Converting Euler angles from geocentric to topographic coordinates (or vice versa) is often particularly useful, since topographic-referenced Euler angles correspond to heading, pitch and roll.

Using a *DtCoordTransform* that can convert from geocentric to topographic coordinates, we can obtain an entity's heading, pitch, and roll as follows:

```
DtTaitBryan topoEuler;
geocToTopo.eulerTrans(esr->orientation, &topoEuler);
double heading = topoEuler.psi();
double pitch = topoEuler.theta();
double roll = topoEuler.phi();
```

9.3.4. UTM Coordinates

Locations in the world can be specified using a UTM coordinate system. UTM coordinates are mapped to a reference ellipsoid approximating the surface of the earth using a Universal Transverse Mercator projection. The coordinates consist of easting and northing from an origin or reference location, and an altitude above a reference ellipsoid, all in meters.

In a true UTM coordinate system, the origin or reference point (0,0,0) is often a point on the equator at the center of a particular UTM zone. (The world is longitudinally divided into 60 UTM zones, each centered on an odd multiple of 3 degrees). However, VR-Link allows you to define an “offset UTM” coordinate system by passing any arbitrary reference point, to the function DtUtmInit().

DtUtmInit() must be called to establish the world location of the origin of your UTM attribute system before you use any of the UTM coordinate conversion functions. Currently, VR-Link supports the use of only one UTM reference point at a time. However, you can call DtUtmInit() multiple times in your application to define the UTM origin you want to use for the next set of conversions.

Using DtUtmInit()

The first two arguments to DtUtmInit() are **DtDegMinSec** structures, representing the latitude and longitude of the reference point. **DtDegMinSec** has the following definition:

```
enum DtPolarDirection { DtEast, DtWest, DtNorth, DtSouth };
typedef struct
{
    double deg, min, sec;
    DtPolarDirection direction;
} DtDegMinSec;
```

The following code initializes VR-Link's UTM conversion routines so that they use 35° north latitude and 122° west longitude as the reference point:

```
DtDegMinSec latRef = {35.0, 0.0, 0.0, DtNorth};
DtDegMinSec lonRef = {122.0, 0.0, 0.0, DtWest};
DtUtmInit(latRef, lonRef, 0);
```

The optional third argument to DtUtmInit() indicates that a particular convention should be used, whereby 500,000 meters are added to all easting values and 10,000,000 meters are added to all north values, in order to eliminate the use of negative numbers in UTM coordinates. Rare applications that need to use this convention should pass a value of 1 for this argument as opposed to the more common 0.

The optional fourth argument (not shown) controls the level of diagnostic information printed by the function.

A UTM coordinate is represented using *DtUtmCoord*, defined in *utmCoord.h*. A *DtUtmCoord* can be constructed by passing its three components (in meters) to the constructor:

```
DtUtmCoord utmLoc(50.0, 100.0, 70.0);
```

Assuming that DtUtmInit() was called as above, this *DtUtmCoord* would represent a point of altitude 70 meters, that is 50 meters east and 100 meters north of our reference point of 35 degrees north by 122 degrees west.

The class has a copy constructor and assignment operator defined, and has the following inspectors and mutators to access the various components: east, setEast, north, setNorth, up, setUp.

DtUtmCoord also has member functions that enable you to convert to and from geocentric or geodetic coordinates. DtUtmCoord::geocentric() and DtUtm-Coord::geodetic() return the geodetic and geocentric equivalents of a UTM coordinate respectively. For example:

```
DtUtmInit(...);
...
DtUtmCoord utm(50.0, 60.0, 70.0);
DtGeodeticCoord geod = utm.geodetic();
```

The member functions getGeocentric() and getGeodetic() work similarly, but fill in an existing object passed by reference to the function, rather than returning the desired object.

Alternatively, the C-style functions DtGeodToUtm(), DtUtmToGeod(), DtGeocToUtm(), and DtUtmToGeoc() can also convert locations in UTM coordinates to geocentric or geodetic coordinates, and vice-versa. However their use is discouraged in favor of the member functions.

9.3.5. Differences Between UTM and Topographic Coordinates

Trouble often arises when programmers do not understand the differences between a UTM coordinate system and a topographic coordinate system that share the same origin.

Although the coordinates of a point near the origin may be similar in the two systems, they are different in several ways.

- In a UTM system, Z represents altitude above the earth's surface. In the topographic frame, -Z represents height above a plane tangent to the earth's surface, which deviates from altitude above the earth's surface itself as one moves away from the origin (Figure 9-3).

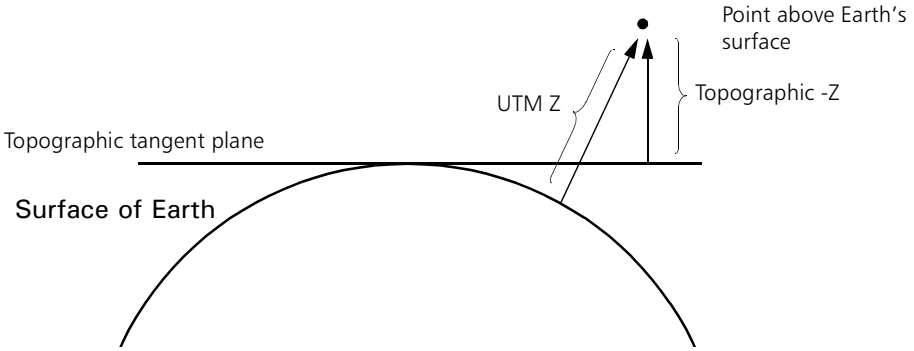


Figure 9-3. Comparison of UTM Z and topographic -Z

- Neglecting the small angle between northing and north, UTM coordinates X, Y, and Z correspond to east, north, and up, while the topographic X, Y, and Z correspond to north, east, and down.

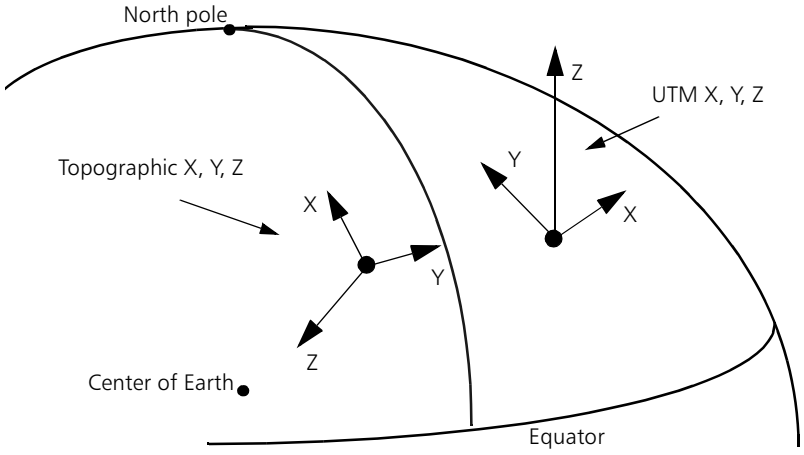


Figure 9-4. Topographic X, Y, and Z compared to UTM

- Although lines of constant easting in UTM coordinates run very close to north/south, they do not run truly north/south as lines of longitude do. This means that two points with equal easting values in their UTM coordinates, such as {100, 100, 0} and {200, 100, 0} do not lie exactly on the same line of longitude. The second point is not due north of the first. Depending on location within a UTM zone, the deviation can be plus or minus a couple of degrees. The function DtUtmCoord::gridDeclination() returns the magnitude (in radians) of the angle between northing and true north at a particular UTM location.

9.3.6. Lower Level Coordinate Conversion Functions

If you would like a greater level of control over your coordinate conversions than that offered by the *DtCoordTransform* class described in "DtCoordTransform," on page 9-11, there are several lower-level functions that can help.

If you have a rotation matrix that expresses a rotation between two coordinate systems, you can multiply that matrix by a vector, such as velocity, or acceleration using DtDcmVecMul() in order to convert that vector between the coordinate systems.

For geocentric/topographic conversions, the needed rotation matrix can be obtained using DtLatLon_to_GeocToTopo() or DtLatLon_to_TopoToGeoc(), declared in *topoCoord.h*. These functions take a latitude and longitude and return rotation matrices to convert between the geocentric frame and the topographic frame defined by that point on earth.

For example:

```
DtGeodeticCoord myOriginInGeod(35.0, -122.0, 0.0);
// Obtain a rotation matrix for the topo frame in question
DtDcm geoc2Topo;
DtLatLon_to_GeocToTopo(myOriginInGeod, geoc2Topo);
DtEntityStateRepository* esr = ...;
DtVector topoVel;
// Perform the vector rotation, using the matrix.
DtDcmVecMul(geoc2Topo, esr->velocity(), topoVel);
```

You can also use a rotation matrix to rotate a set of Euler angles from one coordinate system to another, using the function DtEulerToEuler(), declared in *Euler.h*, for example:

```
DtTaitBryan topoOrient;
DtEulerToEuler(esr->orientation(), geoc2Topo, topoOrient);
```

But if your goal is to obtain an entity's orientation as topographic Euler angles (heading, pitch, and roll), a more efficient way would be:

1. Use esr->bodyToGeoc() rather than orientation.
2. Multiply this returned matrix by geoc2Topo or by a geocentric to topographic rotation matrix obtained using DtLatLon_to_GeocToTopo as described in the example above.

3. Convert the result to Euler angles using DtBodyToRef_to_Euler() as follows:

```
DtDcm bodyToTopo;
DtDcmDcmMul(geoc2Topo, esr->bodyToGeoc(), bodyToTopo);
DtTaitBryan topoEuler;
DtBodyToRef_to_Euler(bodyToTopo, topoEuler);
```

Coordinate locations cannot be transformed using only a rotation matrix. The translational offset between the two coordinate systems is also required. To transform from one cartesian system to another:

1. Subtract the origin of the second system from the point being transformed.
2. Perform the rotation as above.

For example, to transform a location from geocentric to a topographic coordinate:

```
// Find the local origin expressed in geocentric coordinates
DtVector myOriginInGeoc = myOriginInGeod.geocentric();
// Subtract the local origin from the point being transformed
DtVector tmp;
DtVecSub(esr->location(), myOriginInGeoc, tmp);
// Rotate the result
DtVector topoLoc;
DtDcmVecMul(geoc2Topo, tmp, topoLoc);
```

9.4. Time-Related Utilities

VR-Link's time-related functions are declared in *vlTime.h*.

Before any of VR-Link's time-related functions are used, you need to call *DtTimeInit*. This applies to almost all applications, since *DtExerciseConn*, *DtReflectedEntityList*, and *DtEntityPublisher* all use time-related functions.

Usually, you will want to advance VR-Link simulation time each frame, in proportion to the amount of real time that has elapsed since the previous frame, so that it becomes a discrete approximation to (possibly scaled or offset) real time.

The following functions may be of assistance in this task:

- DtAbsRealTime() provides a machine-independent interface to obtaining the absolute current time, in seconds
- DtGetElapsedRealTime() returns the time, in seconds, since DtTimeInit() was called.

For more information, see “Managing Time,” on page 3-13.

9.5. Creating Linked Lists with DtList

DtList, defined in *vlList.h*, provides a general linked-list capability in VR-Link. To enable *DtList* to be used to represent a list of any type of object, pointers to the objects being listed are represented by void*, and must be explicitly cast back to the data type you are listing when you want to inspect data in the list.

Within VR-Link, a *DtList* is used to represent the list of articulated parts that an entity possesses, the list of HLA objects in the exercise, and other such lists. For these reasons, you may need to work with *DtLists* even if you do not choose to use them to represent your own lists.

A *DtList* is constructed as follows:

```
DtList list;
```

Elements can be added to a *DtList* using:

- addToStart()
- addToEnd()
- addBefore()
- addAfter()
- add() – which is equivalent to addToEnd().

These functions return a pointer to a *DtListItem* – an envelope in which your data gets wrapped before insertion into the list.

These functions take a pointer to the element to be added. (As an exception, we sometimes store simple data types that are <=32 bits directly in the list rather than by using pointers.) These pointers must remain valid for the duration of the time they are in the list. For example, the address of automatic variables should not be put in a list if the list remains beyond the scope of the automatic variables. As a result, in general, the pointers in a list should typically be from new'ed data.

DtList's first() and last() functions return pointers to *DtListItems* that hold the first and last elements of the list.

DtListItem's data() member function can be used to get back the pointer you added to the list.

first() and last() return NULL when the list is empty.

DtListItem has prev() and next() member functions that facilitate iterating through a list. These functions return NULL when the beginning and end of the list have been reached, respectively. Thus, you can iterate through the elements in a list as follows:

```
DtList list;
    ...
for (DtListItem* item = list.first(); item; item = item->next());
```

The following example adds several items to a list (in this case instances of class *A*), and pulls them out of the list later:

```
// Definition of class A
class A
{
public:
    A(int num) { a = num; }
    int a;
};
// Create the list
DtList list;
// Create three instances of class A
A* a1 = new A(1);
A* a2 = new A(2);
A* a3 = new A(3);
// Add the instances to the list
list.add(a1);
list.add(a2);
list.add(a3);
// Iterate through the list, inspect the data
for (DtListItem* item = list.first(); item; item = item->next())
{
    // Use DtListItem::data to get back the pointer you added.
    // You'll need to cast the pointer back to an A* from a void*
    A* current = (A*) item->data();
    printf("%d\n", current->a);
}
```

9.6. Diagnostic Utilities

The amount of diagnostic information printed by VR-Link can be controlled by setting the global variable **DtNotifyLevel**, declared in *vlprint.h*. **DtNotifyLevel** may be set to any of the following values:

DtNlFatal	Only fatal messages are printed.
DtNlWarn	Warning messages and fatal messages are printed.
DtNlInfo	Some diagnostic information is printed with warnings and fatal messages.
DtNlVerbose	Maximum information is printed.

The default is **DtNlInfo**.

You can use this notify level to control messages generated by your own application as well, if you use VR-Link's printing functions rather than printf() or cout() directly. These printing functions are also declared in *vlprint.h*, and have the same prototype as printf(). They include:

- DtFatalPerror() and DtWarnPerror(), which print perror information in addition to the information you pass to the function
- DtFatalError()
- DtWarn()
- DtInfo().

The fatal functions call *DtAbort* after printing their messages.

9.7. IP Address Manipulation Functions

vlutil.h, has a set of functions for obtaining and manipulating IP addresses. These functions are often useful for generating addresses to use in initializing a DIS exercise connection.

VR-Link represents IP addresses with the *DtInetAddr* type, which is equivalent to an unsigned 32 bit integer.

DtStringToInetAddr() constructs a *DtInetAddr* from a string containing the common dot notation for an IP address. For example:

```
DtInetAddr myAddr = DtStringToInetAddr("207.86.232.1");
```

DtInetAddrString() goes the other way, returning a string representation of a *DtInetAddr*.

DtInetAddrOfDevice() returns the IP address of a particular device, specified by its device name. Device names are typically something like le0 or ec0. Similarly, DtNetMaskOfDevice() returns the netmask of a device.

DtInetBroadcastOfDevice() computes and returns the broadcast address of a device, by using those portions of the device's IP address that are covered by the netmask, and setting the lower order bits to 1's. For example, if the IP address of a device is 207.86.232.1, and the netmask is 0xffffff00 (class C network), then the broadcast address will be 207.86.232.255.

On some UNIX platforms, this does not match the erroneous broadcast address reported by ifconfig.

DtFirstHostInetAddr() returns the IP address of the host's primary network device.

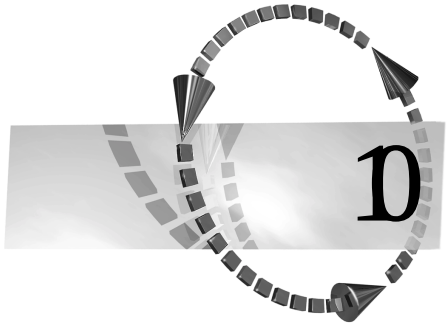
DtIsMulticastAddr() returns 1 if the address passed to it is a valid multicast address, that is, one between 224.0.0.0 and 239.255.255.255.

9.8. Miscellaneous Functions

Table 9-2 lists additional miscellaneous functions.

Table 9-2: Miscellaneous functions

Function	Description
DtSleep()	Provides a machine-independent way to halt program execution for some period of time. It's argument is the time, in seconds, for which to "sleep". The implementation of DtSleep() is machine-dependent, but it uses system calls such as UNIX's select, rather than a CPU-intensive busy loop. (See *ProcControl.h)*
DtSelect()	Provides a machine-independent interface to the UNIX select system call, which takes different arguments on different platforms. DtSelect()'s arguments are a number of file descriptors, three arrays of file descriptors to monitor for presence of data to read, availability for writing, and pending exceptional conditions, respectively, and a timeout in seconds. (See *vlutil.h*)
DtAbort()	Is called whenever VR-Link generates a fatal error. In turn, it makes the abort system call. If you need to modify VR-Link's abort behavior, you can write your own definition for DtAbort(). Because VR-Link's definition for DtAbort() resides in its own object file, your definition will merely supersede ours with no other effects. If your definition is linked into your application through a library that you create (rather than through an object file on your link line), make sure that your library appears before *libvlutil.a* on your link line. (See *vlprint.h*.)



Example and Utility Applications

VR-Link includes a set of supporting applications that can help you develop and troubleshoot your own applications.

The following example programs are included with VR-Link, with source code:

The following executables are included without source code:

10.1. About the Applications

VR-Link's example and utility applications are provided to make your development and troubleshooting tasks easier, and to demonstrate how VR-Link's classes and functions are used in practice.

The executables for these programs are in the *./bin* directories described in "VR-Link Files Installed," on page 2-4. You can find source code for the example applications in subdirectories under *./examples*.

The *buffgrep* program and *vrlinkd3i* packet server are provided only as utilities, and do not include source code. The *vrlinkdu* packet server is available as a separate option.

10.1.1. License Enforcement for the Example Applications

The prebuilt example applications in the *./bin* directories do not cause a VR-Link license to be checked out, so you can run them freely.

If you rebuild the example applications, be careful not to overwrite the original versions. Your new versions will require VR-Link licenses in order to run.

All VR-Link libraries for PCs were compiled with Microsoft Developer Studio 5.0.

10.2. The f18 Program

The *f18* program simulates a simple HLA or DIS vehicle. By default, that vehicle is an F/A18, flying in a circle, just off the coast of Fort Hunter-Liggett, but you can control the entity type, location and route.

f18 generates predictable HLA entity update messages (or DIS entity state PDUs), so it is a useful tool for debugging applications that receive and interpret these messages. *f18* can also fire munitions and react to detonations.

The *f18* syntax is:

```
f18 -h
f18 [options...]
```

> To exit *f18*, press "q" followed by Enter or Return.

Table 10-1 summarizes *f18*'s command line options.

Table 10-1: f18 Startup Options (Sheet 1 of 2)

Option	Description
	Protocol Independent
`-d secs`	Set the delay (in seconds) between the time *f18* receives a detonation interaction (or PDU), and the time it broadcasts its final message and exits.
`-D algo`	Set dead reckoning algorithm to `algo`, where `algo` is one of: 0 - Other 1 - Static 2 - Fixed rotation, positional DR, world coordinates 3 - Rotational DR, positional DR, world coordinates 4 - Rotational DR, velocital DR, world coordinates 5 - Fixed rotation, velocital DR, world coordinates
`-h`	Displays a summary of command-line options, then exits.
`-H deg`	Sets the initial heading of the f18 to `deg` degrees.
`-l file`	Loads the specified *.mtl* (MÄK Technologies Lisp) configuration file.
`-L n,e,u`	Set initial position, as `north,east,up` in topographic coordinates.
`-O lat,long`	Sets the reference latitude and longitude for the topographic coordinate system.
`-r radius`	Sets turn *radius* of the f18 to the specified number of meters. Setting the *radius* to 0.0 causes the f18 to move in a straight line.
`-s mps`	Sets initial speed of the f18 to `mps` meters/second.
`-T type`	Sets the entity type, specified as a string of the form: kind:domain:country:category:subcategory:specific:extra
`-W x,y,z`	Sets the initial position in geocentric coordinates.

Table 10-1: f18 Startup Options (Sheet 2 of 2)

Option	Description
`-x` *`ex-name`*	Sets the exercise ID for DIS (default is 1), or the federation execution name for HLA (default is VR-Link).
	DIS Only
`-a` *`ID`*	Sets the second component of the entity ID to *ID*. By default, the entity ID is 1:2:1, however if you want to run multiple *f18*'s at once, each needs a unique entity ID.
`-A` *`addr`*	Sets the default destination IP address for outgoing PDUs to *addr*. *addr* is in the form 123.123.123.123.
`-P` *`portnum`*	Specifies the UDP port. Defaults to 3000.
`-S` *`addr`*	Subscribes to the multicast address specified by *addr*. Multiple *`-S's`* can appear on a command line. Multicast is not supported on all platforms. *addr* is in the form 123.123.123.123.
`-V` *`ver`*	Sets the DIS protocol version contained in outgoing PDUs. Replace *ver* with an integer as follows: 3 - DIS 2.0.3 5 - IEEE 1278.1 4 - DIS 2.0.4 6 - DIS 2.1.4 (IEEE 1278.1a)

10.2.1. The f18 Configuration File

By default, *f18* loads the configuration file *f18.mtl*, if it exists in the current directory. This file sets many of the *f18*'s default parameters, including notify level, dead-reckoning algorithm, and initial heading, position, and speed.

To change default parameters:

- Edit the *f18.mtl*, file
- Use command-line options to override parameters
- Use the `-l` option to specify an alternate configuration file.

You can rebuild *f18*, or create a new application based on *f18* that does not use a configuration file.

> To write a variation of *f18* that does not use a configuration file, remove the initMtl() call.

You can also specify a different configuration file in your code, at the DtcLoadLispFile() function call.

10.2.2. Firing Munitions

> To cause the F18 to fire at another entity, press the Enter or Return key.

If other entities exist in the exercise, *f18* issues a fire message directed at the closest one. Though there is no tracked munition, *f18* issues a detonate message three seconds later to indicate that the target was hit.

> To change the delay, edit the **munitionFlightTime** parameter in the configuration file.

10.2.3. Reactions to Detonate Messages

The F18 reacts to detonation PDUs or RPR FOM Detonation Interactions. If a detonation occurs within the **lethalDetonationRange**, or if the detonation result is entity impact or entity proximate, the F18 is destroyed. Updates reflect a damage state of DamageDestroyed for a period of **destroyedToFinalDelay** seconds, at which point the entity leaves the exercise and *f18* exits.

LethalDetonationRange defaults to 20 meters, and **destroyedToFinalDelay** defaults to zero. You can change these parameters in *f18.mtl*.

10.2.4. Absolute timestamping

The value of the `timeStampType` flag in *f18.mtl* determines whether absolute or relative timestamps are used in *f18*'s outgoing state update messages. The flag defaults to relative (0), but can be set to absolute (1). When the `timeStampType` is absolute, *f18* dead-reckons remote entities based on their absolute timestamps, when applicable. Absolute timestamping should only be used when the local machine's clock is synchronized with the clock being used by other applications in the exercise for their absolute timestamps.

For more information about timestamps, see "Timestamps," on page 3-18.

10.2.5. Using A Modified FOM

The *f18* example has a source file called *config.cxx* that shows how you can configure VR-Link to use a modified FOM. It shows how you can instruct VR-Link to represent position on the network as Z, Y, X, rather than X, Y, Z. For this purpose, we use a new attribute of the *BaseEntity* object class called **ReversePosition** instead of the default **Position** attribute, and a new parameter of the *WeaponFire* interaction class called **ReverseFiringLocation** instead of the default **FiringLocation**.

By default, this functionality is #ifdefed out. But if you include the definition REVERSE=1 on your make line, when building *f18*, for example:

```
make REVERSE=1 f18
```

it will be compiled in. If you want to try this out, remember to modify the *.fed* file to reflect the expected changes before running the executable.

10.3. Calling VR-Link from Other Languages

The simple C example demonstrates how to use VR-Link from within a C language application. It is in *examples/simpleC*. Since most other languages can interact with C, this should also serve as a starting point for applications written in other languages that use VR-Link.

10.4. Launcher

The launcher example demonstrates use of articulated and attached parts in VR-Link for both DIS and HLA. It is in the *examples/test* directory.

10.5. netdump (DIS Only)

netdump is a DIS debugging tool that displays the contents of arriving DIS PDUs in an easy-to-read form. (Its HLA counterpart, *hlaNetdump*, is described on page 10-9.)

Netdump writes its output to stdout. For each PDU, netdump prints the number of PDUs it has received and the PDU's arrival time (relative to *netdump*'s initialization).

You can filter PDUs based on specific fields by piping *netdump*'s output through VR-Link's *buffgrep* utility.

netdump's usage is:

```
netdump -h
netdump [-c] [-e 0|1] [-P port] [-S address]
```

Table 10-2 describes the command-line options.

Table 10-2: netdump startup options

Option	Description
`-c`	Clears the screen between PDUs. This option is especially useful for debugging single-entity applications. The screen is cleared between PDUs, so corresponding fields of successive PDUs appear in the same place. This makes changes in various parameters easily visible.
`-e level`	Sets the level of aggressiveness in trying to print erroneous PDUs. If *level* is 0, *netdump* always tries to print each PDU. If `level` is 1, *netdump* doesn't try to print PDUs when the size indicated in the DIS header doesn't match the size of the packet received. Default: 1.
`-h`	Displays a summary of command-line options, then exits.
`-P portnum`	Specifies the UDP port. Defaults to 3000.
`-S addr`	Subscribes to the multicast address specified by `addr`. Multiple `-S's` can appear on a command line. Multicast is not supported on all platforms. `addr` is in the form 225.10.10.10.

Notes:

- If *netdump* has been linked with *libvl* in the *./bin5* directory, it prints all fields of DIS 2.0.4, IEEE 1278.1 or DIS 2.1.4 (IEEE 1278.1a) PDUs.
- If it has been linked with *libvl3* in the *./bin3* directory, it prints fields from DIS 2.0.3 PDUs.
- When it receives PDUs of the wrong DIS version, *netdump* prints the DIS header information and a hex dump of the rest of the PDU.

- *netdump* prints version information as described in Table 10-3:

Table 10-3: PDU versions and DIS protocol versions

PDU Version	DIS Protocol Version
3	2.0.3
4	2.0.4
5	IEEE 1278.1
6	2.1.4 (IEEE 1278.1a)

10.6. hlaNetdump (HLA only)

hlaNetdump listens to an HLA federation execution, subscribes to all object and interaction classes in the FOM, and prints out data whenever it receives an attribute update or interaction.

hlaNetdump handles interactions similarly to *netdump* (page 10-7) — it interprets and prints out the data in the interaction.

When *hlaNetdump* receives an attribute update message, it is decoded into a *DtEntityStateRepository*, then the state repository is printed.

hlaNetdump prints default values (usually zero or false) for attributes not contained in the update. A list appears above the state information, showing the names of attributes actually contained in the update.

The usage for *hlaNetdump* is:

```
hlaNetdump -h
hlaNetdump [-c] [-r] [-m] [-x exec-name]
```

Table 10-4 describes the command-line options.

Table 10-4: hlaNetdump Startup Options

Option	Description
`-c`	Clears the screen between messages (updates or interactions). This option is especially useful for debugging single entity applications. The screen is cleared between messages, so corresponding fields of successive messages appear in the same place. This makes changes in various parameters easily visible.
`-h`	Displays a summary of command-line options, then exits.
`-m`	Subscribes to MOM classes.
`-r`	Runs *hlaNetdump* in raw mode. For more information, see the paragraph following this table.
`-x exec-name`	Specifies the name of the federation execution to be monitored. Default is *VR-Link*.

When you run *hlaNetdump* with the `-r` option, it behaves like *nethex*. For each attribute update or interaction received, it prints the name of the attribute, the size of the value of that attribute, and a hexadecimal dump of the value of the attribute.

hlaNetdump can print incoming HLA data in a completely FOM-independent manner. It subscribes to every class in the *.fed* file (including MOM classes if you use the `-m` argument). If an update or interaction represents a class that VR-Link "knows" how to decode, (includes all BaseEntity subclasses, Aggregates, Fire, Detonate and Collision Interactions) the data is printed as the correct data types (unless you're using raw mode). If the message represents a class that VR-Link cannot decode, *hlaNetdump* prints the name of the class, and the name, size and hex value of all attributes and parameters.

10.7. nethex (DIS only)

nethex is a DIS debugging tool that prints out arriving DIS PDUs in raw, hexadecimal format. Using *nethex*, you can see the information contained in each byte of a PDU. *nethex* writes its output to stdout.

nethex's usage is:

```
nethex -h
nethex [-P portnum] [-S addr]
```

Table 10-5 describes the command-line options.

Table 10-5: nethex Startup Options

Option	Description
`-h`	Displays a summary of command-line options, then executes.
`-P portnum`	Specifies the UDP port. Defaults to 3000.
`-S addr`	Subscribes to the multicast address specified by *addr*. Multiple *-S's* can appear on a command line. Multicast is not supported on all platforms. *addr* is of the form 225.10.10.10.

10.8. talk and listen

talk and *listen* are simple send-only and receive-only programs. They demonstrate many VR-Link fundamentals.

talk simulates the flight of an F18 aircraft. *listen* repeatedly prints an entity's updated position, and if a fire PDU or interaction occurs, prints the entity ID of the attacker. Both programs are written for protocol-independence.

For detailed descriptions, see "A Listen-Only Example," on page 3-19 and "A Send-Only Example," on page 3-22.

10.9. vdc – the VR-Link Daemon Controller

vdc, the vrlink-daemon controller, lets you send messages that control the Packet Server (described in "The MÄK Packet Server (UNIX Only)," on page 10-13). Its source code is provided as a working example of how a VR-Link application can control the Packet Server at runtime.

vdc controls the Packet Server whether it is running in the background (as a daemon) or in the foreground. However, *vdc*'s terminate command is ignored if the Packet Server is running in the foreground.

To issue a *vdc* command, enter the following:

```
vdc port [cmd]
```

where *port* is the required UDP port number and *cmd* is a one-letter command from Table 10-6 below. If you omit the command, *vdc* sends an `a` command (test if alive).

Table 10-6: vdc Control Keys

Key	Purpose
`a`	Test if alive.
`c`	Print server client status.
`e`	Set Packet Server to external mode.
`i`	Set Packet Server to internal mode.
`n`	Turn unbundling off.
`p`	Cause server to ping clients.
`q`	Terminate Packet Server.
`s`	Print server status.
`t`	Quietly test if alive.
`u`	Turn unbundling on.

All commands except `a` and `t` have the same meaning as those used by the Packet Server itself.

The `a` and `t` commands are unique to *vdc*. You can use them to perform a conditional action depending on whether the Packet Server is running. See "vdc Examples," on page 10-12 for examples.

When you control the Packet Server through *vdc*, status messages and diagnostics are displayed by *vdc*, rather than by the Packet Server. A terminate request sent by *vdc* is ignored if the Packet Server is running in the foreground.

10.9.1. *vdc* Examples

vdc accepts multiple commands in a string, as in these examples:

`vdc 3000 is` Change to internal mode and then display server status.

`vdc 3000 pc` Ping clients and then display client status.

You can use the test commands `a` and `t` to perform a different action depending on whether the Packet Server is running:

`vdc 3000 tq` If the server does not exist, exit *vdc* quietly; otherwise terminate the server.

`vdc 3000 a || vrlinkd3i 3000` Start server if not already running.

`vdc 3000 t && echo hi || echo bye` Display “hi” if the server is running; otherwise display “bye”.

10.9.2. *vdc* Diagnostics

If you receive the message

`no vrlink server on port` *nnnn*

and you know that the Packet Server is running on the specified port, the server's AF-UNIX socket file may have become unlinked. See “The Server Socket File,” on page 10-22 for more information.

10.10. The MÄK Packet Server (UNIX Only)

The MÄK Packet Server, also called the VR-Link daemon, or *vrlinkd*, is an application with several purposes:

- On some UNIX platforms, multiple applications running on the same machine cannot all receive network packets from a single UDP port. On these platforms, the Packet Server acts as a necessary intermediary, receiving packets from the UDP port and forwarding them to all connected client applications through shared memory.
- In many circumstances, even on platforms that do allow multiple applications to read from the same UDP port, using the Packet Server can offer performance advantages and can limit the effect of dropped packets.
- The Packet Server allows multiple applications running on the same machine to communicate with each other, without generating network traffic.

Though typically used with DIS applications, the Packet Server can be used with HLA applications in conjunction with the MÄK RTI. For more information about using the Packet Server with HLA, see "Using the Packet Server in HLA," on page 10-15.

You need a VR-Link license for each VR-Link-based application that you run concurrently, whether or not you are using a Packet Server.

10.10.1. Packet Server Modes

The Packet Server operates in two modes, external and internal.

In external mode, the Packet Server serves as an intermediary between its client applications and a UDP socket. It asynchronously reads packets from a specified UDP port and forwards the packets to client applications through a shared memory interface. It also provides client-specific filtering based on exercise ID, DIS protocol version, and PDU kind.

In internal mode, the Packet Server allows client applications on a single machine to exchange packets among themselves, without interacting with the network.

The server can switch between internal and external mode, bringing its clients into and out of a larger exercise.

10.10.2. Packet Server Configurations

The Packet Server is available in two configurations, distinguished by the number of clients they support:

- *vrlinkd3i* supports three clients in internal mode, or one client in external mode. This version is included as a standard part of the VR-Link toolkit.
- *vrlinkdu* supports an unlimited number of clients in either internal or external mode. It is available as a separate product and requires a license.

You can start multiple Packet Servers to service more than one UDP port on a host.

10.10.3. How Applications Connect to the Packet Server

A DIS application based on VR-Link that uses the default exercise connection constructor automatically uses the Packet Server if one is running on its port. Otherwise, the application connects directly to the network.

If a Packet Server is running, but is already handling its maximum number of clients, further clients will terminate when they attempt to connect to the Packet Server.

10.10.4. Using the Packet Server

You can use the packet server with one client or multiple clients.

Using the Packet Server with Only One Client Application

If only one DIS application based on VR-Link is running per machine, the applications can communicate directly with the network and no Packet Server is necessary. However, the Packet Server can be beneficial even when there is only one client.

Because the asynchronous server can respond more quickly than a fixed-rate client to network traffic, the Packet Server may reduce the number of dropped packets. Moreover, if the client cannot drain packets from the Packet Server quickly enough, the Packet Server ensures that the oldest packets are dropped rather than the newest. The opposite would be true if the client were reading directly from the network. In addition, the Packet Server can enable a higher packet throughput when it can run on a separate processor.

When only one client needs to connect to the Packet Server, you can use *vrlinkd3i*, which comes standard with VR-Link.

Using the Packet Server with Multiple Clients per Host

On some UNIX platforms, you need a Packet Server to run more than one VR-Link DIS application on a single machine. You can use the *vrlinkd3i* version if you are running two or three applications and the applications communicate only among themselves. Otherwise, you need the *vrlinkdu* version.

Figure 10-7 depicts some common configurations of applications using a Packet Server. In scenario A, each machine hosts one application, so no Packet Server is necessary. However, Machine #3 uses one anyway for improved packet handling. In scenario B, one machine hosts two applications. On some platforms, the unlimited *vrlinkdu* version is required in such a case to connect multiple applications to an external network. In scenario C, the Packet Server is used in internal mode to connect three applications on the same machine.

10.10.5. Using the Packet Server in HLA

When you use the MÄK RTI, you might want to use the Packet Server for the same reasons you would in DIS, to:

- Get around a platform's one-application-per-port rule
- Offload network I/O to a separate process
- Run an exercise solely on a single machine without generating any network traffic.

Like VR-Link-based DIS applications, applications that use the MÄK RTI automatically connect to a packet server if one is running on the port you are using, otherwise they connect directly to the network.

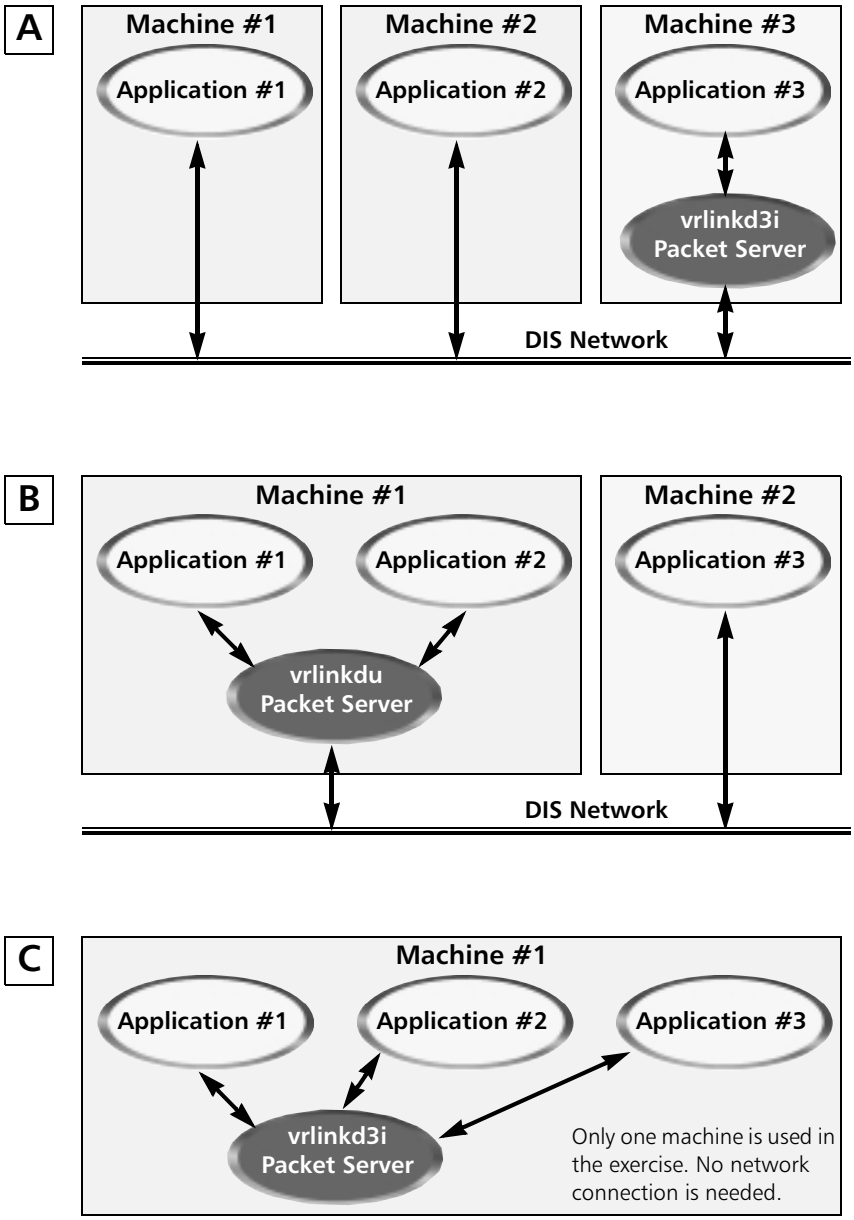


Figure 10-7. Scenarios using the Packet Server.

10.10.6. Starting the Packet Server

> To start the Packet Server, run *vrlinkd3i* or *vrlinkdu* with the number specifying the UDP port to be serviced.

The example below starts a Packet Server on port 3000, which is the default port for VR-Link applications:

```
vrlinkd3i 3000
```

By default, the Packet Server starts in external mode.

> To run the Packet Server in internal mode, start it with the `-I` option.

> To run the Packet Server in the background after initialization (as a daemon), start it with the `-B` option.

For example, the following command starts an internal Packer Server in the background on UDP port 3001.

```
vrlinkd3i -I -B 3001
```

Startup options appear before the port number on the command line. The options in Table 10-8 apply to both Packet Server configurations.

Table 10-8: Packet Server Startup Options (Sheet 1 of 2)

Option	Description
portnum	The UDP port number is the Packet Server's only required argument.
`-A` *addr*	Specifies where to send packets that are received from clients created with VR-Link versions prior to 2.4.2. *addr* is an IP address, and may be a broadcast, point-to-point or multicast address. This option does not apply to applications created with newer VR-Link versions, which choose their own destination address. Do not use this option with -D.
`-B`	Tells the Packet Server to execute in the background (as a daemon), and sets the notify level (option -N) to 2.
`-D` *device*	Tells the Packet Server to listen to the network on which *device* listens. *device* is the name of a network device, (for example, ec0 or le0). Use netstat -i to list available devices. The destination address for packets sent by pre-2.4.2 clients is set to the broadcast address of *device*. The -D and -A should not be used together.
`-I`	Starts the Packet Server in internal mode, ignoring the network.
`-n`	Tells the Packet Server to drop new packets rather than old ones when the queue is full and the drop mode is not specified by the client.

Table 10-8: Packet Server Startup Options (Sheet 2 of 2)

Option	Description
`-N level`	Sets notify level to one of the following: 0 FATAL 2 NOTIFY 1 WARNINGS 3 INFO (default)
`-r size`	Sets the size of the queue the Packet Server uses to receive packets from pre-2.4.2 clients for clients that do not specify a size. The default is 20 *KB*.
`-s size`	Sets the size of the queue the Packet Server uses to send packets to clients for clients that do not specify a size. The default is 200 *KB*.
`-u state`	Turns unbundling on or off depending on whether *state* is 1 or 0 respectively. Unbundling is on by default.
`-w`	Tells the Packet Server, when dropping old packets, not to wait for the queue lock when the client has the lock, and to drop the packet instead. The Packet Server may run slightly faster in this mode while dropping relatively few new packets.

10.10.7. Stopping the Packet Server

> If the Packet Server is running in the foreground, stop it by typing 'q', then Enter, or Ctrl-C.

> If the Packet Server is running in the background, stop it using the VR-Link daemon controller:

```
vdc server-port q
```

The Packet Server also can terminate in response to a SIGTERM or SIGINT signal, or a terminate message from another process.

When terminating, the Packet Server first notifies each client that it is closing the connection to the client. Typically, the next read or write by a client after such notification causes a fatal error in the client.

The Packet Server also closes connections if you change its mode and it is serving too many clients for the new mode. For example, if you connect three clients to *vrlinkd3i* running in internal mode, then to switch to external mode, the connections to all but the first client are closed.

10.10.8. Packet Server Controls

When the Packet Server is running in the foreground, you can control it using its keyboard controls. When it is running in the background, you can control it through messages from another process such as the *vdc* program (see "vdc – the VR-Link Daemon Controller," on page 10-11) or your own application.

The keyboard controls consist of the single-character commands summarized in Table 10-9. Each command is followed by the Enter key.

Table 10-9: Packet Server Control Keys

Key	Purpose
c	Print server client status.
e	Set Packet Server to external mode.
i	Set Packet Server to internal mode.
n	Turn unbundling off.
p	Cause server to ping clients.
q	Terminate Packet Server.
s	Print server status.
u	Turn unbundling on.

When you run the Packet Server in the background, the *vdc* daemon controller program uses these same single character commands and some additional ones. See "Background Operation," on page 10-21 for more information.

10.10.9. Pinging Clients

> To cause the Packet Server to ping its clients, press the "p" key.

It determines whether the clients are still alive, and then sends them a control message. Client applications then typically display a reception message, thus indicating that packet-server connections are intact.

If a client is no longer alive, the Packet Server removes it from its list of clients.

10.10.10. Unbundling Packets

The Packet Server can extract individual PDUs that are concatenated into the same IP packet. The PDUs are served to clients one at a time as if each had arrived in its own packet. Unbundling is on by default.

> To turn off packet bundling, use the `-u 0` startup option, or press the "n" key.

10.10.11. Status Commands

The server status command (the "s" key) displays the following information:

- The Packet Server's internal/external mode status
- The UDP port number
- The bundling mode
- The number of packet clients and control-only clients
- The maximum number of clients allowed for the current mode.

The client status command (the c key) displays the server status followed by this information for each client:

- Whether the client is a packet client or control client
- The client's ID (used for IPC keys) and process ID
- The number of messages sent and received
- Old messages and new messages dropped
- Size of the send and receive queues
- The list of mode flags currently in effect (Read, Write, Notify, and so on).

10.10.12. Queue Sizes

Packets are dropped when a client does not drain the buffers often enough for a given queue size. You may be able to compensate for this problem by increasing the size of the Packet Server's send queue.

---

i Shared memory is always allocated in full page sizes, typically 4kBytes. The Packet Server allocates a single shared memory segment equal to the combined size of the send and receive queues. Therefore, for the most efficient use of system resources, specify send and receive queue sizes that add up to a multiple of 4kB.

> To set the send and receive queue sizes, use the `-s` and `-r` startup options, respectively.

10.10.13. Technical Notes

This section contains additional technical information.

External-Mode Operation with a Single Client

vrlinkd3i supports one client in external mode. Using the packet server in this way moves filtering into a separate process, which is useful on multiprocessing machines. It also allows the application to vary the effective amount of network packet buffering, which can be especially helpful to applications that do not service the network often. Finally, it allows the network to be turned on and off from the application or from an external controller such as *vdc*.

Background Operation

Using the `-B` option to run the Packet Server as a daemon is similar to executing a shell background job, but differs in the following respects:

- Control is returned to the caller only after the port is tested for availability
- Termination by the control client is enabled
- Diagnostics are sent to *syslog*
- The notify level is set to ignore Info.

The server behaves as a polite daemon (own process group, no control terminal, and *cwd* set to */tmp*).

The Application's Interface to the Packet Server

Client applications use the *DtClientSocket* packet interface to send and receive packets through the server. The *DtClientSocket* is a *DtSocket* that sends and receives packets with the help of the Packet Server. For details, see

Constraints Imposed by System Resources

While *vrlinkdu* has no inherent limit to the number of clients it can support, it may be limited by the availability of System V IPC resources. Each client uses a separate semaphore id. Some UNIX systems are configured to allow only ten. You can modify this by reconfiguring the kernel. For large numbers of clients, you may also need to increase the total system-wide number of semaphores (which is different from the limit on semaphore ids) and the number of shared memory segments.

The allowable maximum queue size is also constrained by the system limit on the size of shared memory segments.

The Server Socket File

The server accepts control requests (for example, add new client) on an AF-UNIX socket file in */tmp* (for example, */tmp/vld-3000*). If this file is removed, then no further control requests will be accepted. In fact, control requests will return with the error:

```
no vrlink server on port nnnn
```

Without its socket file, the server will continue to function with existing clients, but will not be able to add new clients or change communication mode.

> To restore these capabilities, restart the server and reestablish connections for the clients.

> To terminate a background server that has lost its socket file, use `kill -15`.

10.11. buffgrep (UNIX only)

The *buffgrep* utility searches the standard input for buffers matching one or more patterns. It is typically used with *netdump* ("netdump (DIS Only)," on page 10-7) or *hlaNetdump* ("hlaNetdump (HLA only)," on page 10-9).

A buffer is defined as all the lines from the end of the last buffer (or the beginning of the stream) up to and including the first line that matches a specified ending pattern. Buffers that contain any of the search patterns specified are copied to the standard output. *buffgrep* patterns are regular expressions like those used by *grep* and *ed*.

The buffgrep syntax is:

```
buffgrep [-v] endpattern pat1 [pat2…]
```

The -v option causes *buffgrep* to print a buffer only if no line contains any of the specified patterns *pat1*, *pat2*, and so on.

buffgrep is often useful in conjunction with *netdump* (or *hlaNetdump*) and awk. When using *buffgrep* in this manner, use the following sequence as the *endpattern*:

```
'^$'
```

10.11.1. *buffgrep* Examples

Print out all DIS EntityState PDUs:

```
netdump | buffgrep '^$' EntityState
```

Print out all attribute updates from objects of the class MilitaryAirLandPlatform:

```
hlaNetdump | buffgrep '^$' MilitaryAirLandPlatform
```

Print out all Fire Interactions from object named MyEnt

```
hlaNetdump | buffgrep '^$' FireInteraction | buffgrep '^$' 'From.*MyEnt'
```

Print out all HLA fire interactions:

```
hlaNetdump | buffgrep '^$' FireInteraction
```

Print out all EntityState PDUs from entity 1:8:1:

```
netdump | buffgrep '^$' EntityState | buffgrep '^$' 'From.*1:8:1'
```

Print out all Fire Interactions from entity named "F18":

```
hlaNetdump | buffgrep '^$' FireInteraction | buffgrep -v '^$' 'From.*F18'
```

Print out only the PDU banner, TimeStamp, and Velocity for Collision PDUs:

```
netdump | buffgrep '^$' Collision | awk '/PDU/ /TimeStamp/ /Velocity/'
```

Print out the TimeStamp changes in updates from entity named F18:

```
hlaNetdump | buffgrep '^$' 'Name.*F18' |
           awk '/TimeStamp:/ {dt=$2-t; t=$2; print $0, " ", dt}'
```

Guide to VR-Link Header Files

This appendix covers the following topics:

A.1. Convenience Header Files

Some VR-Link header files exist simply for convenience or to provide protocol independence, and do not require further description in this manual.

The following four files each include the other header files for their respective libraries:

- *matrixInc.h*
- *vlInc.h*
- *mtlInc.h*
- *vlutilInc.h.*

For convenience, you can include just these general header files in your code. However, you might speed up compilation by including only the header files you need.

The following files group other related header files together for convenience:

- *mathTypes.h* *groups math-related headers, defines math macros*
- *pdus.h* *groups all PDU headers.*

A.1.1. Header Files for Protocol Independence

Table A-1 lists header files that are used for protocol-independence.

Table A-1: Header files for protocol independence (Page 1 of 2)

This file:	Provides protocol independence for:
ackInter.h	Acknowledge interactions
actReqInter.h	Action request interactions
actRespInter.h	Action response interactions
aggPub.h	Local aggregates
collideInter.h	Collision interactions
commentInter.h	Comment interactions
createEntInt.h	Entity interactions
dataInter.h	Data interactions
dataQInter.h	Data query interactions
detonateInter.h	Detonation interactions
entityPub.h	Local entities
eventRepInter.h	Event report interactions
exerciseConn.h	Exercise connections
fireInter.h	Fire interactions

Table A-1: Header files for protocol independence (Page 2 of 2)

This file:	Provides protocol independence for:
interaction.h	Base interactions
lcInter.h	Logger control interactions
refEmitLst.h	Reflected emitter lists
reflAggList.h	Reflected aggregate lists
reflectedAgg.h	Reflected aggregates
reflectedEnt.h	Reflected entities
reflEmittr.h	Reflected emitters
reflEntList.h	Reflected entity lists
remEntInter.h	Remove entity interactions
reflObjList.h	Reflected object lists
reflectedObj.h	Reflected objects
repCompInter.h	Repair complete interactions
repRespInter.h	Repair response interactions
resupplyInter.h	Resupply interactions
servReqInter.h	Service request interactions
setDataInter.h	Set data interactions
signalInter.h	Signal interactions
startInter.h	Start and resume interactions
stateMsg.h	State messages
stopInter.h	Stop and freeze interactions
vcInter.h	View control interactions

A.2. Class-to-Header Cross Reference

Table A-2 contains an alphabetical list of VR-Link classes along with the header file that contains each one. In the *VR-Link Reference Manual*, the The names of the header files are links to the header file code.

Table A-2: Classes and their Corresponding Header Files (Page 1 of 19)

Class	Header File
DtAbsNotifyObserver	absNotifyObs.h
DtAbsoluteVCPdu	vcPdu.h
DtAcknowledgeDecoder	hAckDec.h
DtAcknowledgeEncoder	hAckEnc.h
DtAcknowledgeInteraction	hAckInter.h
DtAcknowledgePdu	ackPdu.h
DtAcknowledgeRPdu	ackRPdu.h
DtAckPredOpt	smUtil.h
DtActionRequestDecoder	hActReqDec.h
DtActionRequestEncoder	hActReqEnc.h
DtActionRequestInteraction	hActReqInter.h
DtActionRequestPdu	actionPdu.h
DtActionRequestRPdu	actReqRPdu.h
DtActionRequestToObjDecoder	hActReqObjDec.h
DtActionRequestToObjEncoder	hActReqObjEnc.h
DtActionRequestToObjInter	hActReqObjInt.h
DtActionResponseDecoder	hActRespDec.h
DtActionResponseEncoder	hActRespEnc.h
DtActionResponseInteraction	hActRespInter.h
DtActionResponseFromObjDecoder	hActRspObjDec.h
DtActionResponseFromObjEncoder	hActRspObjEnc.h
DtActionResponseFromObjInter	hActRspObjInt.h
DtActionResponsePdu	actionPdu.h
DtActionResponseRPdu	actRespRPdu.h

Table A-2: Classes and their Corresponding Header Files (Page 2 of 19)

Class	Header File
DtAggregateEntityDecoder	aggEntDec.h
DtAggregateEntityEncoder	aggEntEnc.h
DtAggregatePublisher	dAggPub.h
DtAggregatePublisher	hAggPub.h
DtAggregateStateDecoder	dAggDec.h
DtAggregateStateEncoder	dAggEnc.h
DtAggregateStateRepository	aggregateSR.h
DtAggregateStatePdu	aggStatePdu.h
DtAntennaPatternRecord	antPatRec.h
DtApproximator	approximator.h
DtAppSpecificRadioSignalDecoder	hAppRadSigDec.h
DtAppSpecificRadioSignalEncoder	hAppRadSigEnc.h
DtArealObjectStatePdu	arealObjStPdu.h
DtArtPart	artPart.h
DtArtPartList	artPartList.h
DtArtPartThresholder	artPartThresh.h
DtAttachedPartList	attPartList.h
DtAttributeChangeReqDec	hAttChgReqDec.h
DtAttributeChangeReqEnc	hAttChgReqEnc.h
DtAttributeChangeReqInter	hAttChgReqInt.h
DtAttributeChangeResDec	hAttChgResDec.h
DtAttributeChangeResEnc	hAttChgResEnc.h
DtAttributeChangeResInter	hAttChgResInt.h
DtBaseEmbeddedSystemDecoder	baseEmbedDec.h
DtBaseEmbeddedSystemEncoder	baseEmbedEnc.h
DtBaseEmitterBeamDecoder	baseEmittrDec.h
DtBaseEmitterBeamEncoder	baseEmittrEnc.h
DtBaseEntityDecoder	baseEntDec.h

Table A-2: Classes and their Corresponding Header Files (Page 3 of 19)

Class	Header File
DtBaseEntityEncoder	baseEntEnc.h
DtBaseEntityStateRepository	baseEntitySR.h
DtBaseExerciseConn	baseExConn.h
DtBaseHashKey	baseHashKey.h
DtBaseMsg	baseMsg.h
DtBaseReflectedAggregate	baseReflAgg.h
DtBaseReflectedEntity	baseReflEnt.h
DtBasicFixedWingAircraftRecord	gedBasFixWing.h
DtBasicGroundCombatSoldierRecord	gedBasSoldier.h
DtBasicGroundCombatVehicleRecord	gedBasVehicle.h
DtBasicRotorWingAircraftRecord	gedBasRotWing.h
DtBeamAntennaPatternRecord	beamAntRec.h
DtBurstDescriptor	hostStructs.h
DtCartesianView	cartView.h
DtClientSocket	ClientSocket.h
DtCollisionDecoder	hCollideDec.h
DtCollisionEncoder	hCollideEnc.h
DtCollisionElasticPdu	collisElasPdu.h
DtCollisionInteraction	hCollideInter.h
DtCollisionPdu	collidePdu.h
DtCommentDecoder	hCommentDec.h
DtCommentEncoder	hCommentEnc.h
DtCommentInteraction	hCommentInter.h
DtCommentPdu	commentPdu.h
DtCommentRPdu	commentRPdu.h
DtCompassVCPdu	vcPdu.h
DtCoordTransform	LibMatrix.h
DtCoordTranslation	LibMatrix.h

Table A-2: Classes and their Corresponding Header Files (Page 4 of 19)

Class	Header File
DtCreateEntityDecoder	hCreatEntDec.h
DtCreateEntityEncoder	hCreatEntEnc.h
DtCreateEntityInteraction	hCreatEntInt.h
DtCreateEntityPdu	createPdu.h
DtCreateEntityRPdu	createEntRPdu.h
DtCreateObjReqDecoder	hCrtObjReqDec.h
DtCreateObjReqEncoder	hCrtObjReqEnc.h
DtCreateObjReqInter	hCrtObjReqInt.h
DtCreateObjectResultDecoder	hCrtObjResDec.h
DtCreateObjectResultEncoder	hCrtObjResEnc.h
DtCreateObjectResultInter	hCrtObjResInt.h
DtCsFilterTest	ClientSocket.h
DtDatabaseIndexRadioSignalDecoder	hDbiRadSigDec.h
DtDatabaseIndexRadioSignalEncoder	hDbiRadSigEnc.h
DtDataDecoder	hDataDec.h
DtDataEncoder	hDataEnc.h
DtDataInteration	hDataInter.h
DtDataPdu	dataPdu.h
DtDataQueryDecoder	hDataQDec.h
DtDataQueryEncoder	hDataQEnc.h
DtDataQueryInteraction	hDataQInter.h
DtDataQueryPdu	queryPdu.h
DtDataQueryRPdu	dataQueryRPdu.h
DtDataRPdu	dataRPdu.h
DtDatumMap	datumMap.h
DtDatumMapEntry	datumMap.h
DtDatumRecordSet DtVarDatumSpec	datumRecSet.h

Table A-2: Classes and their Corresponding Header Files (Page 5 of 19)

Class	Header File
DtDcm	dcm.h
DtDeadReckoner	deadReckoner.h
DtStateDecoderFactory	decFactory.h
DtDesignatorDecoder	designatorDec.h
DtDesignatorEncoder	designatorEnc.h
DtDesignatorPdu	designatePdu.h
DtDesignatorPublisher	hDesignatrPub.h
DtDesignatorRepository	designatorSR.h
DtDetonationDecoder	hDetDec.h
DtDetonationEncoder	hDetEnc.h
DtDetonationInteraction	hDetInter.h
DtDetonationPdu	detonatePdu.h
DtDirectedRotRate	directRotRate.h
DtDirectRtiAmbassador	dirRtiAmb.h
DtDiscoverObjectCbInfo	dscvObjCbInfo.h
DtEntityPublisher	dEntityPub.h
DtExerciseConn	dExerciseConn.h
DtDisPduFactory	disPduFactory.h
DtReflectedEntity	dReflectedEnt.h
DtReflectedEntityList	dReflEntList.h
DtDmeGlobalFcns	datumMap.h
DtElectromagneticEmissionPdu	elecEmitPdu.h
DtEmbeddedSystemRepository	embedSystemSR.h
DtEmissionPdu	emissionPdu.h
DtEmitterBeamPublisher	hEmitrBeamPub.h
DtEmitterBeamRepository	emittrBeamSR.h
DtEmitterSystemDecoder	emitterSysDec.h
DtEmitterSystemEncoder	emitterSysEnc.h

Table A-2: Classes and their Corresponding Header Files (Page 6 of 19)

Class	Header File
DtEmitterSystemPublisher	hEmittrSysPub.h
DtEmitterSystemRepository	emitterSysSR.h
DtEmptyFomMapper	emptyFomMap.h
DtEncodedAudioRadioSignalDecoder	hEncARSigDec.h
DtEncodedAudioRadioSignalEncoder	hEncARSigEnc.h
DtEnhancedFixedWingAircraftRecord	gedEnhFixWing.h
DtEnhancedGroundCombatSoldierRecord	gedEnhSoldier.h
DtEnhancedGroundCombatVehicleRecord	gedEnhVehicle.h
DtEnhancedRotorWingAircraftRecord	gedEnhRotWing.h
DtEnvironmentalEntityDecoder	envDec.h
DtEnvironmentalEntityEncoder	envEnc.h
DtEntityIdentifier	hostStructs.h
DtEntityPublisher	dEntityPub.h hEntityPub.h
DtEntityStateDecoder	dEntityDec.h
DtEntityStateEncoder	dEntityEnc.h
DtEntityStatePdu	esPdu.h
DtEntityStateRepository	entitySR.h
DtEntityStateUpdatePdu	esUpdatePdu.h
DtEntityType	hostStructs.h
DtEnvironmentalProcessPdu	envProcPdu.h
DtEnvironmentTypeRecord	enviroTypeRec.h
DtEventId	eventId.h
DtEventReportDecoder	hEventRepDec.h
DtEventReportEncoder	hEventRepEnc.h
DtEventReportInteraction	hEventRepInt.h
DtEventReportPdu	eventReptPdu.h
DtEventReptRPdu	eventReptRPdu.h

Table A-2: Classes and their Corresponding Header Files (Page 7 of 19)

Class	Header File
DtExerciseConn	hExerciseConn.h
DtFederateId	hostStructs.h
DtFedReader	fedReader.h
DtFedTarget	fedTarget.h
DtFilename	filename.h
DtFilterTest	Socket.h
DtFireDecoder	hFireDec.h
DtFireEncoder	hFireEnc.h
DtFireInteraction	hFireInter.h
DtFirePdu	firePdu.h
DtFom	fom.h
DtFomMapper	fomMapper.h
DtFundamentalParameterData	elecEmitPdu.h
DtFundamentalParameterData	emissionPdu.h
DtGenericObjectDecoder	genericDec.h
DtGenericObjectEncoder	genericEnc.h
DtGenericObjectPublisher	genericPub.h
DtGenericStateRepository	genericSR.h
DtGeodeticCoord	geodCoord.h
DtGlobalIdHashList	globIdHLst.h
DtGlobalObjectDesignator	globalObjDes.h
DtGlobalObjectDesignatorList	objDesList.h
DtGridAxisRecordFixed	gridAxiRecFix.h
DtGridAxisRecordVariable	gridAxiRecVar.h
DtGridDataRecord	gridDataRec.h
DtGridDataRecordType0	gridDataRecT0.h
DtGridDataRecordType1	gridDataRecT1.h
DtGridDataRecordType2	gridDataRecT2.h

Table A-2: Classes and their Corresponding Header Files (Page 8 of 19)

Class	Header File
DtGriddedDataPdu	gridDataPdu.h
DtGroundLogisticsVehicleRecord	gedLogVehicle.h
DtGroupVCPdu	vcPdu.h
DtHashedItem	DtOldHashlist, DtOldIntrusiveHashlist
DtHashItem	hashItem.h
DtHashlist	hashlist.h
DtHlaObject	hlaObj.h
DtHlaObjectFactory	hlaObjFact.h
DtHlaObjectWithStateRep	hlaObjWithSR.h
DtHlaStateDecoder	hStateDecoder.h
DtHlaStateEncoder	hStateEncoder.h
DtIffAtcNavaidsBeamData	iffAtcNavBeam.h
DtIffAtcNavaidsLayerHeader	iffAtcNavLayr.h
DtIffAtcNavaidsOperational	iffAtcNavOper.h
DtIffAtcNavaidsParameter	iffAtcNavParm.h
DtIffAtcNavaidsPdu	iffAtcNavPdu.h
DtIffAtcNavaidsSecondOperational	iffAtcNavOpr2.h
DtIffAtcNavaidsSystemID	iffAtcNavSyID.h
DtInteraction	dInteraction.h
DtInteraction	hInteraction.h
DtInteractionDecoder	interDecoder.h
DtInteractionDecoderFactory	intDecFact.h
DtInteractionEncoder	interEncoder.h
DtInteractionEncoderFactory	intEncFact.h
DtInteractionFactory	hInterFactory.h
DtInteractionManager	interMgr.h
DtInteractionWithData	hInterWData.h
DtInteractionWithEncDec	hIntWEncDec.h

Table A-2: Classes and their Corresponding Header Files (Page 9 of 19)

Class	Header File
DtInterClassDesc	intClassDesc.h
DtIntercomControlPdu	intercmCtlPdu.h
DtIntercomEntityDestRec	intercmEntDst.h
DtIntercomGroupAsgmtRec	intercmGrpAsg.h
DtIntercomGroupDestRec	intercmGrpDst.h
DtIntercomParmRec	intercmParRec.h
DtIntercomSignalPdu	intercmSigPdu.h
DtIntHashKey	intHashKey.h
DtIntrusiveList	vlList.h
DtIsGroupOfPdu	isGroupOfPdu.h
DtIsPartOfPdu	isPartOfPdu.h
DtJammerBeamDecoder	jammerBeamDec.h
DtJammerBeamEncoder	jammerBeamEnc.h
DtLaserPdu	laserPdu.h
DtLeAppearancePdu	leAppearPdu.h
DtLeArticulatedPartsPdu	leArtPartsPdu.h
DtLeDeadReckoningParameters	leDeadReck.h
DtLeDetonationPdu	leDetonPdu.h
DtLeEntityID	leEntityID.h
DtLeEntityLinearVelocity	leEntityVel.h
DtLeEntityLocationEntityCoordinates	leLocEntity.h
DtLeEntityLocationWorldCoordinates	leLocWorld.h
DtLeEntityOrientation	leEntityOri.h
DtLeFirePdu	leFirePdu.h
DtLeOrientationError	leOriError.h
DtLePositionError	lePosError.h
DtLeTspiPdu	leTspiPdu.h
DtLgrControlDecoder	hLcDec.h

Table A-2: Classes and their Corresponding Header Files (Page 10 of 19)

Class	Header File
DtLgrControlEncoder	hLcEnc.h
DtLgrControlInteraction	hLcInter.h
DtLgrControlPdu	lcPdu.h
DtLgrEndPdu	lcEnd.h
DtLgrEofActionPdu	lcEofAction.h
DtLgrHeartbeatPdu	lcHeartbeat.h
DtLgrPausePdu	lcPause.h
DtLgrPlayPdu	lcPlay.h
DtLgrPlayActionPdu	lcPlayAction.h
DtLgrPbFilePdu	lcPlayFile.h
DtLgrQuitPdu	lcQuit.h
DtLgrRecordActionPdu	lcRecAction.h
DtLgrRecFilePdu	lcRecFile.h
DtLgrRecordPdu	lcRecord.h
DtLgrSetTimePdu	lcSetTime.h
DtLgrSkipTimePdu	lcSkipTime.h
DtLgrStartPdu	lcStart.h
DtLgrStopPdu	lcStop.h
DtLgrTimeScalePdu	lcTimeScale.h
DtLgrTimeoutPdu	lcTimeout.h
DtLinearObjectStatePdu	linObjStPdu.h
DtLinearSegmentParameter	linSegParam.h
DtList	vlList.h
DtListedObj	vlList.h
DtListItem	vlList.h
DtLocalHlaObjectManager	locObjMgr.h
DtMessagePdu	messagePdu.h
DtMilitaryEntityDecoder	milEntDec.h

Table A-2: Classes and their Corresponding Header Files (Page 11 of 19)

Class	Header File
DtMilitaryEntityEncoder	milEntEnc.h
DtMilitaryPlatformEntityDecoder	milPlatDec.h
DtMilitaryPlatformEntityEncoder	milPlatEnc.h
DtMimicTrackVCPdu	vcPdu.h
DtMimicVCPdu	vcPdu.h
DtMinefieldDataPdu	mineDataPdu.h
DtMinefieldPerimeterRecord	minePerimeter.h
DtMinefieldQueryPdu	mineQueryPdu.h
DtMinefieldResponseNackPdu	mineRsNackPdu.h
DtMinefieldStatePdu	mineStatePdu.h
DtModelessVCPdu	vcPdu.h
DtModulationType	transmitPdu.h
DtMunitionEntityDecoder	munEntDec.h
DtMunitionEntityEncoder	munEntEnc.h
DtNet32Vector	NetStructs.h
DtNet64Vector	NetStructs.h
DtNetBoolean	netBoolean.h
DtNetBurstDescriptor	NetStructs.h
DtNetEntityIdentifier	NetStructs.h
DtNetEntityMarking	NetStructs.h
DtNetEntityType	NetStructs.h
DtNetEulerAngles	NetStructs.h
DtNetEventId	NetStructs.h
DtNetFederateId	NetStructs.h
DtNetFloat32	NetTypes.h
DtNetFloat64	NetTypes.h
DtNetGlobalObjectDesignator	NetStructs.h
DtNetInt16	NetTypes.h

Table A-2: Classes and their Corresponding Header Files (Page 12 of 19)

Class	Header File
DtNetInt32	NetTypes.h
DtNetRadioEntityType	NetStructs.h
DtNetSimulationAddress	NetStructs.h
DtNetSocket	NetSocket.h
DtNetTimeStamp	netTimeStamp.h
DtNetU16	NetTypes.h
DtNetU32	NetTypes.h
DtNullFederateAmbassador	nullFedAmb.h
DtObjClassDesc	objClassDesc.h
DtObjectId	hostStructs.h
DtObjectIdHashKey	objIdHashKey.h
DtObjectIdHashlist	objIdHashLst.h
DtObjectPublisher	hObjPub.h
DtObjectReflectAttributeValuesCbInfo	objRefCbInfo.h
DtObjectType	objectType.h
DtOldHashlist, DtOldIntrusiveHashlist In VR-Link 3.3, we significantly changed the semantics of our DtHashlist class. The new version is defined in hashlist.h. The older DtHashlist class has been renamed DtOldHashlist, and is still defined here in this file for backwards compatibility. While we suggest that you port your code to the new DtHashlist, which is cleaner and easier to use, if you do not want to make that change, you can just use DtOldHashlist wherever you used to use DtHashlist, and the rest of your old code should work.	
DtOrbitVCPdu	vcPdu.h
DtPdu	pdu.h
DtPduCallbackInfo	pduCbInfo.h
DtPduCallbackManager	pduCbMgr.h
DtPduDatumRecordSet	pduDatRecSet.h
DtPduFactory	pduFactory.h
DtPduWithData	pduWithData.h
DtPhysicalEntityDecoder	physEntDec.h

Table A-2: Classes and their Corresponding Header Files (Page 13 of 19)

Class	Header File
DtPhysicalEntityEncoder	physEntEnc.h
DtPointObjectStatePdu	pointObjStPdu.h
DtPropulsionSystem	supEmisEsProp.h
DtRadarBeamDecoder	radarBeamDec.h
DtRadarBeamEncoder	radarBeamEnc.h
DtRadioEntityType	hostStructs.h
DtRadioReceiverDecoder	radioRecvrDec.h
DtRadioReceiverEncoder	radioRecvrEnc.h
DtRadioReceiverPublisher	hRadioRcvrPub.h
DtRadioReceiverRepository	radioRecvrSR.h
DtRadioTransmitterDecoder	radioXmitDec.h
DtRadioTransmitterEncoder	radioXmitEnc.h
DtRadioTransmitterPublisher	hRadioXmitPub.h
DtRadioTransmitterRepository	radioXmitSR.h
DtRawBinaryRadioSignalDecoder	hRawBinRSDec.h
DtRawBinaryRadioSignalEncoder	hRawBinRSEnc.h
DtReceiveInteractionCbInfo	recIntrCbInfo.h
DtReceiverPdu	receivePdu.h
DtRecordPdu	recordPdu.h
DtRecordQueryPdu	recQueryPdu.h
DtRecordSet	recordSet.h
DtRecordSetPdu	recordSetPdu.h
DtReflectAttributeValuesCbInfo	refAttrCbInfo.h
DtReflectAttributeValuesCbList	refAttrCbList.h
DtReflectedAggregate	dReflectedAgg.h
DtReflectedAggregate	hReflectedAgg.h
DtReflectedAggregateList	dReflAggList.h
DtReflectedAggregateList	hReflAggList.h

Table A-2: Classes and their Corresponding Header Files (Page 14 of 19)

Class	Header File
DtReflectedEmitterBeamList	hRefBeamList.h
DtReflectedDesignatorList	hRefDesigList.h
DtReflectedEmitterSystemList	hRefEmSysList.h
DtReflectedEmitterBeam	hReflBeam.h
DtReflectedDesignator	hReflDesigntr.h
DtReflectedEmitterSystem	hReflEmitrSys.h dReflEmitrSys.h
DtReflectedEmitterSystemList	dRefEmSysList.h
DtReflectedEntity	dReflectedEnt.h hReflectedEnt.h
DtReflectedEntityList	hReflEntList.h dReflEntList.h
DtReflectedGenericObject	refGenericObj.h
DtReflectedGenericObjectList	refGenList.h
DtReflectedHlaObjectManager	refObjMgr.h
DtReflectedObject	hReflectedObj.h dReflectedObj.h
DtReflectedObjectCallbackInfo	refObjCbInfo.h
DtReflectedObjectList	hRefObjList.h dRefObjList.h
DtReflectedRadioReceiver	hReflRadRecvr.h
DtReflectedRadioTransmitter	hReflRadXmitr.h
DtReflectedRadioRecieverList	hRefRadRvList.h
DtReflectedRadioTransmitterList	hRefRadXmList.h
DtRegControlCbInfo	regCtlCbInfo.h
DtRegControlCbList	regCtlCbList.h
DtRemoveEntityDecoder	hRemovEntDec.h
DtRemoveEntityEncoder	hRemovEntEnc.h
DtRemoveEntityInteraction	hRemovEntInt.h

Table A-2: Classes and their Corresponding Header Files (Page 15 of 19)

Class	Header File
DtRemoveEntityPdu	removePdu.h
DtRemoveEntityRPdu	removeEntRPdu.h
DtRemoveObjectCbInfo	remObjCbInfo.h
DtRemoveObjectRequestDecoder	hRemObjReqDec.h
DtRemoveObjectRequestEncoder	hRemObjReqEnc.h
DtRemoveObjectRequestInter	hRemObjReqInt.h
DtRemoveObjectResultDecoder	hRemObjResDec.h
DtRemoveObjectResultEncoder	hRemObjResEnc.h
DtRemoveObjectResultInter	hRemObjResInt.h
DtRepairCompleteDecoder	hRepCompDec.h
DtRepairCompleteEncoder	hRepCompEnc.h
DtRepairCompleteInteraction	hRepCompInter.h
DtRepairCompletePdu	repairPdu.h
DtRepairResponseDecoder	hRepRespDec.h
DtRepairResponseEncoder	hRepRespEnc.h
DtRepairResponseInteraction	hRepRespInter.h
DtRepairResponsePdu	repairPdu.h
DtResupplyCancelDecoder	hResCancDec.h
DtResupplyCancelEncoder	hResCancEnc.h
DtResupplyCancelInteraction	hResCancInter.h
DtResupplyCancelPdu	resupplyPdu.h
DtResupplyOfferDecoder	hResOffDec.h
DtResupplyOfferEncoder	hResOffEnc.h
DtResupplyOfferInteraction	hResOffInter.h
DtResupplyOfferPdu	resupplyPdu.h
DtResupplyReceivedDecoder	hResRecDec.h
DtResupplyReceivedEncoder	hResRecEnc.h
DtResupplyRecInteraction	hResRecInter.h

Table A-2: Classes and their Corresponding Header Files (Page 16 of 19)

Class	Header File
DtResupplyReceivedPdu	resupplyPdu.h
DtRprFomMapper	rprFomMap.h
DtRprHlaObjectFactory	rprObjFact.h
DtRprInteractionDecoderFactory	rprIDecFact.h
DtRprInteractionEncoderFactory	rprIEncFact.h
DtRprInteractionFactory	rprInterFact.h
DtRprStateDecoderFactory	rprDecFact.h
DtRprStateEncoderFactory	rprEncFact.h
DtScript	script.h
DtSelectParam	vlutil.h
DtServiceRequestDecoder	hServReqDec.h
DtServiceRequestEncoder	hServReqEnc.h
DtServiceRequestInteraction	hServReqInter.h
DtServiceRequestPdu	servReqPdu.h
DtSetDataDecoder	hSetDataDec.h
DtSetDataEncoder	hSetDataEnc.h
DtSetDataInteraction	hSetDataInter.h
DtSetDataPdu	setDataPdu.h
DtSetDataRPdu	setDataRPdu.h
DtSetRecordPdu	setRecordPdu.h
DtSignalInteraction	hSignalInter.h
DtSignalPdu	signalPdu.h
DtSilentAggregateList	aggAggList.h
DtSilentEntityList	aggEntList.h
DtSimanPdu	simanPdu.h
DtDimpleRprFomMapper	simRprFomMap.h
DtSmoother	smoother.h
DtSmoothOrient	smoothUtil.h

Table A-2: Classes and their Corresponding Header Files (Page 17 of 19)

Class	Header File
DtSmoothOriWithDr	smoothOriWDR.h
DtSmoothPos	smoothUtil.h
DtSmoothPosWithDr	smoothPosWDR.h
DtSocket	Socket.h
DtSocketPairSocket	pairSock.h
DtSoldierDecoder	soldierDec.h
DtSoldierEncoder	soldierEnc.h
DtStartResumeDecoder	hStartDec.h
DtStartResumeEncoder	hStartEnc.h
DtStartResumeInteraction	hStartInter.h
DtStartResumePdu	startPdu.h
DtStartResumeRPdu	startResRPdu.h
DtStartStopInterCbInfo	stIntCbInfo.h
DtStartStopInterCbList	stIntCbList.h
DtStateEncoderFactory	encFactory.h
DtStateMsg	dStateMsg.h
DtStateMsg	hStateMsg.h
DtStateRepository	stateRep.h
DtStopFreezeDecoder	hStopDec.h
DtStopFreezeEncoder	hStopEnc.h
DtStopFreezeInteraction	hStopInter.h
DtStopFreezePdu	stopPdu.h
DtStopFreezeRPdu	stopFrzRPdu.h
DtString	str.h
DtStringHashKey	strHashKey.h
DtSuppEmisEntStatePdu	supEmisEsPdu.h
DtSystemHashKey	sysHashKey.h
DtTaitBryan	taitBryan.h

Table A-2: Classes and their Corresponding Header Files (Page 18 of 19)

Class	Header File
DtTcpForwarder	tcpForwarder.h
DtTcpSocket	tcpSocket.h
DtTcpSocketMgr	tcpSocketMgr.h
DtTetherTrackVCPdu	vcPdu.h
DtTetherVCPdu	vcPdu.h
DtThresholder	thresholder.h
DtTopoView	topoView.h
DtTrackJamBeamDecoder	trkJamBeamDec.h
DtTrackJamBeamEncoder	trkJamBeamEnc.h
DtTrackJamDesignator	trackJamDes.h
DtTrackJamDesignatorList	tkJmDesLst.h
DtTrackJamTarget	elecEmitPdu.h
DtTrackJamTarget	emissionPdu.h
DtTrackVCPdu	vcPdu.h
DtTransferControlRequestPdu	transCtlPdu.h
DtTransmitterPdu	transmitPdu.h
DtUnderwaterAcousticAddtnlPassiveAct	undwaAcPasAct.h
DtUnderwaterAcousticBeam	undwaAcBeam.h
DtUnderwaterAcousticEmitterSystem	undwaAcEmtSys.h
DtUnderwaterAcousticPdu	undwaAcPdu.h
DtUnderwaterAcousticShaftSpeed	undwaAcShaft.h
DtUnknownHlaObject	unknownHObj.h
DtUnknownInteraction	hUnknownInter.h
DtUnknownPdu	unknownPdu.h
DtUnknownVCPdu	vcPdu.h
DtUpdateControlCbInfo	updCtlCbInfo.h
DtUpdateControlCbList	updCtlCbList.h

Table A-2: Classes and their Corresponding Header Files (Page 19 of 19)

Class	Header File
DtUtmCoord	utmCoord.h
DtUtmView	utmView.h
DtVariableDatumRecord	varDatumRec.h
DtVector	vlVector.h
DtVectoringNozzleSystem	supEmisEsNozz.h
DtViewControlDecoder	hVcDec.h
DtViewControlEncoder	hVcEnc.h
DtViewControlInteraction	hVcInter.h
DtViewControlPdu	vcPdu.h
DtVrlFederateAmbassador	vrlFedAmb.h
DtVrlFedTime	vrlFedTime.h
DtVrlRtiAmbassador	vrlRtiAmb.h

A.3. Header-to-Class Cross Reference

Table A-3 contains an alphabetical list of VR-Link header files, along with the classes they contain. In the *VR-Link Reference Manual*, the The names of the header files are links to the header file code.

Table A-3: Header Files and the Classes they Contain (Page 1 of 18)

Header File	Class
absNotifyObs.h	DtAbsNotifyObserver
ackPdu.h	DtAcknowledgePdu
ackRPdu.h	DtAcknowledgeRPdu
actionPdu.h	DtActionRequestPdu DtActionResponsePdu
actReqRPdu.h	DtActionRequestRPdu
actRespRPdu.h	DtActionResponseRPdu
aggAggList.h	DtSilentAggregateList
aggEntDec.h	DtAggregateEntityDecoder
aggEntEnc.h	DtAggregatedEntityEncoder
aggEntList.h	DtSilentEntityList
aggregateSR.h	DtAggregateStateRepository
aggStatePdu.h	DtAggregateStatePdu
antPatRec.h	DtAntennaPatternRecord
approximator.h	DtApproximator
arealObjStPdu.h	DtArealObjectStatePdu
artPart.h	DtArtPart
artPartList.h	DtArtPartList
artPartThresh.h	DtArtPartThresholder
attPartList.h	DtAttachedPartList
baseEmbedDec.h	DtBaseEmbeddedSystemDecoder
baseEmbedEnc.h	DtBaseEmbeddedSystemEncoder
baseEmittrDec.h	DtBaseEmitterBeamDecoder
baseEmittrEnc.h	DtBaseEmitterBeamEncoder
baseEntDec.h	DtBaseEntityDecoder

Table A-3: Header Files and the Classes they Contain (Page 2 of 18)

Header File	Class
baseEntEnc.h	DtBaseEntityEncoder
baseEntitySR.h	DrBaseEntityStateRepository
baseExConn.h	DtBaseExerciseConn
baseHashKey.h	DtBaseHashKey
baseMsg.h	DtBaseMsg
baseReflAgg.h	DtBaseReflectedAggregate
baseReflEnt.h	DtBaseReflectedEntity
beamAntRec.h	DtBeamAntennaPatternRecord
cartView.h	DtCartesianView
ClientSocket.h	DtClientSocket DtCsFilterTest
collidePdu.h	DtCollisionPdu
collisElasPdu.h	DtCollisionElasticPdu
commentPdu.h	DtCommentPdu
commentRPdu.h	DtCommentRPdu
createEntRPdu.h	DtCreateEntityRPdu
createPdu.h	DtCreateEntityPdu
dAggDec.h	DtAggregateStateDecoder
dAggEnc.h	DtAggregateStateEncoder
dAggPub.h	DtAggregatePublisher
dataPdu.h	DtDataPdu
dataQueryRPdu.h	DtDataQueryRPdu
dataRPdu.h	DtDataRPdu
datumMap.h	DtDatumMap DtDatumMapEntry DtDmeGlobalFcns
datumRecSet.h	DtVarDatumSpec DtDatumRecordSet
dcm.h	DtDcm
deadReckoner.h	DtDeadReckoner

Table A-3: Header Files and the Classes they Contain (Page 3 of 18)

Header File	Class
decFactory.h	DtStateDecoderFactory
dEntityDec.h	DtEntityStateDecoder
dEntityEnc.h	DtEntityStateEncoder
dEntityPub.h	DtEntityPublisher
designatePdu.h	DtDesignatorPdu
designatorDec.h	DtDesignatorDecoder
designatorEnc.h	DtDesignatorEncoder
designatorSR.h	DtDesignatorRepository
detonatePdu.h	DtDetonationPdu
dExerciseConn.h	DtExerciseConn
dInteraction.h	DtInteraction
directRotRate.h	DtDirectedRotRate
dirRtiAmb.h	DtDirectRtiAmbassador
disPduFactory.h	DtDisPduFactory
dRefEmSysList.h	DtReflectedEmitterSystemList
dReflAggList.h	DtReflectedAggregateList
dReflectedAgg.h	DtReflectedAggregate
dReflectedEnt.h	DtReflectedEntity
dReflectedObj.h	DtReflectedObject
dReflEmitrSys.h	DtReflectedEmitterSystem
dReflEntList.h	DtReflectedEntityList
dRefObjList.h	DtReflectedObjectList
dscvObjCbInfo.h	DtDiscoverObjectCbInfo
dStateMsg.h	DtStateMsg
elecEmitPdu.h	DtElectromagneticEmissionPdu DtFundamentalParameterData DtTrackJamTarget
embedSystemSR.h	DtEmbeddedSystemRepository

Table A-3: Header Files and the Classes they Contain (Page 4 of 18)

Header File	Class
emissionPdu.h	DtEmissionPdu DtFundamentalParameterData DtTrackJamTarget
emitterSysDec.h	DtEmitterSystemDecoder
emitterSysEnc.h	DtEmitterSystemEncoder
emitterSysSR.h	DtEmitterSystemRepository
emittrBeamSR.h	DtEmitterBeamRepository
emptyFomMap.h	DtEmptyFomMapper
encFactory.h	DtStateEncoderFactory
entitySR.h	DtEntityStateRepository
envDec.h	DtEnvironmentalEntityDecoder
envEnc.h	DtEnvironmentalEntityEncoder
enviroTypeRec.h	DtEnvironmentTypeRecord
envProcPdu.h	DtEnvironmentalProcessPdu
esPdu.h	DtEntityStatePdu
esUpdatePdu.h	DtEntityStateUpdatePdu
eventId.h	DtEventId
eventReptPdu.h	DtEventReportPdu
eventReptRPdu.h	DtEventReptRPdu
fedReader.h	DtFedReader
fedTarget.h	DtFedTarget
filename.h	DtFilename
firePdu.h	DtFirePdu
fom.h	DtFom
fomMapper.h	DtFomMapper
gedBasFixWing.h	DtBasicFixedWingAircraftRecord
gedBasRotWing.h	DtBasicRotorWingAircraftRecord
gedBasSoldier.h	DtBasicGroundCombatSoldierRecord
gedBasVehicle.h	DtBasicGroundCombatVehicleRecord

Table A-3: Header Files and the Classes they Contain (Page 5 of 18)

Header File	Class
gedEnhFixWing.h	DtEnhancedFixedWingAircraftRecord
gedEnhRotWing.h	DtEnhancedRotorWingAircraftRecord
gedEnhSoldier.h	DtEnhancedGroundCombatSoldierRecord
gedEnhVehicle.h	DtEnhancedGroundCombatVehicleRecord
gedLogVehicle.h	DtGroundLogisticsVehicleRecord
genericDec.h	DtGenericObjectDecoder
genericEnc.h	DtGenericObjectEncoder
genericPub.h	DtGenericObjectPublisher
genericSR.h	DtGenericStateRepository
geodCoord.h	DtGeodeticCoord
globalObjDes.h	DtGlobalObjectDesignator
globIdHLst.h	DtGlobalIdHashList
gridAxiRecFix.h	DtGridAxisRecordFixed
gridAxiRecVar.h	DtGridAxisRecordVariable
gridDataPdu.h	DtGriddedDataPdu
gridDataRec.h	DtGridDataRecord
gridDataRecT0.h	DtGridDataRecordType0
gridDataRecT1.h	DtGridDataRecordType1
gridDataRecT2.h	DtGridDataRecordType2
hAckDec.h	DtAcknowledgeDecoder
hAckEnc.h	DtAcknowledgeEncoder
hAckInter.h	DtAcknowledgeInteraction
hActReqDec.h	DtActionRequestDecoder
hActReqEnc.h	DtActionRequestEncoder
hActReqInter.h	DtActionRequestInteraction
hActReqObjDec.h	DtActionRequestToObjDecoder
hActReqObjEnc.h	DtActionRequestToObjEncoder
hActReqObjInt.h	DtActionRequestToObjInter

Table A-3: Header Files and the Classes they Contain (Page 6 of 18)

Header File	Class
hActRespDec.h	DtActionResponseDecoder
hActRespEnc.h	DtActionResponseEncoder
hActRespInter.h	DtActionResponseInteraction
hActRspObjDec.h	DtActionResponseFromObjDecoder
hActRspObjEnc.h	DtActionResponseFromObjEncoder
hActRspObjInt.h	DtActionResponseFromObjInter
hAggPub.h	DtAggregatePublisher
hAppRadSigDec.h	DtAppSpecificRadioSignalDecoder
hAppRadSigEnc.h	DtAppSpecificRadioSignalEncoder
hashItem.h	DtHashItem
hashlist.h	DtHashlist
hAttChgReqDec.h	DtAttributeChangeReqDec
hAttChgReqEnc.h	DtAttributeChangeReqEnc
hAttChgReqInt.h	DtAttributeChangeReqInter
hAttChgResDec.h	DtAttributeChangeResDec
hAttChgResEnc.h	DtAttributeChangeResEnc
hAttChgResInt.h	DtAttributeChangeResInter
hCollideDec.h	DtCollisionDecoder
hCollideEnc.h	DtCollisionEncoder
hCollideInter.h	DtCollisionInteraction
hCommentDec.h	DtCommentDecoder
hCommentEnc.h	DtCommentEncoder
hCommentInter.h	DtCommentInteraction
hCreatEntDec.h	DtCreateEntityDecoder
hCreatEntEnc.h	DtCreateEntityEncoder
hCreatEntInt.h	DtCreateEntityInteraction
hCrtObjReqDec.h	DtCreateObjReqDecoder
hCrtObjReqEnc.h	DtCreateObjReqEncoder

Table A-3: Header Files and the Classes they Contain (Page 7 of 18)

Header File	Class
hCrtObjReqInt.h	DtCreateObjReqInter
hCrtObjResDec.h	DtCreateObjectResultDecoder
hCrtObjResEnc.h	DtCreateObjectResultEncoder
hCrtObjResInt.h	DtCreateObjectResultInter
hDataDec.h	DtDataDecoder
hDataEnc.h	DtDataEncoder
hDataInter.h	DtDataInteration
hDataQDec.h	DtDataQueryDecoder
hDataQEnc.h	DtDataQueryEncoder
hDataQInter.h	DtDataQueryInteraction
hDbiRadSigDec.h	DtDatabaseIndexRadioSignalDecoder
hDbiRadSigEnc.h	DtDatabaseIndexRadioSignalEncoder
hDesignatrPub.h	DtDesignatorPublisher
hDetDec.h	DtDetonationDecoder
hDetEnc.h	DtDetonationEncoder
hDetInter.h	DtDetonationInteraction
hEmitrBeamPub.h	DtEmitterBeamPublisher
hEmittrSysPub.h	DtEmitterSystemPublisher
hEncARSigDec.h	DtEncodedAudioRadioSignalDecoder
hEncARSigEnc.h	DtEncodedAudioRadioSignalEncoder
hEntityPub.h	DtEntityPublisher
hEventRepDec.h	DtEventReportDecoder
hEventRepEnc.h	DtEventReportEncoder
hEventRepInt.h	DtEventReportInteraction
hExerciseConn.h	DtExerciseConn
hFireDec.h	DtFireDecoder
hFireEnc.h	DtFireEncoder
hFireInter.h	DtFireInteraction

Table A-3: Header Files and the Classes they Contain (Page 8 of 18)

Header File	Class
hInteraction.h	DtInteraction
hInterFactory.h	DtInteractionFactory
hInterWData.h	DtInteractionWithData
hIntWEncDec.h	DtInteractionWithEncDec
hlaObj.h	DtHlaObject
hlaObjFact.h	DtHlaObjectFactory
hlaObjWithSR.h	DtHlaObjectWithStateRep
hLcDec.h	DtLgrControlDecoder
hLcEnc.h	DtLgrControlEncoder
hLcInter.h	DtLgrControlInteraction
hObjPub.h	DtObjectPublisher
hostStructs.h	DtBurstDescriptor DtEntityIdentifier DtEntityType DtFederateId DtObjectId DtRadioEntityType
hRadioRcvrPub.h	DtRadioReceiverPublisher
hRadioXmitPub.h	DtRadioTransmitterPublisher
hRawBinRSDec.h	DtRawBinaryRadioSignalDecoder
hRawBinRSEnc.h	DtRawBinaryRadioSignalEncoder
hRefBeamList.h	DtReflectedEmitterBeamList
hRefDesigList.h	DtReflectedDesignatorList
hRefEmSysList.h	DtReflectedEmitterSystemList
hReflAggList.h	DtReflectedAggregateList
hReflBeam.h	DtReflectedEmitterBeam
hReflDesigntr.h	DtReflectedDesignator
hReflectedAgg.h	DtReflectedAggregate
hReflectedObj.h	DtReflectedObject
hReflEmitrSys.h	DtReflectedEmitterSystem

Table A-3: Header Files and the Classes they Contain (Page 9 of 18)

Header File	Class
hReflectedEnt.h	DtReflectedEntity
hReflEntList.h	DtReflectedEntityList
hReflRadRecvr.h	DtReflectedRadioReceiver
hReflRadXmitr.h	DtReflectedRadioTransmitter
hRefObjList.h	DtReflectedObjectList
hRefRadRvList.h	DtReflectedRadioReceiverList
hRefRadXmList.h	DtReflectedRadioTransmitterList
hRemObjReqDec.h	DtRemoveObjectRequestDecoder
hRemObjReqEnc.h	DtRemoveObjectRequestEncoder
hRemObjReqInt.h	DtRemoveObjectRequestInter
hRemObjResDec.h	DtRemoveObjectResultDecoder
hRemObjResEnc.h	DtRemoveObjectResultEncoder
hRemObjResInt.h	DtRemoveObjectResultInter
hRemovEntDec.h	DtRemoveEntityDecoder
hRemovEntEnc.h	DtRemoveEntityEncoder
hRemovEntInt.h	DtRemoveEntityInteraction
hRepCompDec.h	DtRepairCompleteDecoder
hRepCompEnc.h	DtRepairCompleteEncoder
hRepCompInter.h	DtRepairCompleteInteraction
hRepRespDec.h	DtRepairResponseDecoder
hRepRespEnc.h	DtRepairResponseEncoder
hRepRespInter.h	DtRepairResponseInteraction
hResCancDec.h	DtResupplyCancelDecoder
hResCancEnc.h	DtResupplyCancelEncoder
hResCancInter.h	DtResupplyCancelInteraction
hResOffDec.h	DtResupplyOfferDecoder
hResOffEnc.h	DtResupplyOfferEncoder
hResOffInter.h	DtResupplyOfferInteraction

Table A-3: Header Files and the Classes they Contain (Page 10 of 18)

Header File	Class
hResRecDec.h	DtResupplyReceivedDecoder
hResRecEnc.h	DtResupplyReceivedEncoder
hResRecInter.h	DtResupplyRecInteraction
hServReqDec.h	DtServiceRequestDecoder
hServReqEnc.h	DtServiceRequestEncoder
hServReqInter.h	DtServiceRequestInteraction
hSetDataDec.h	DtSetDataDecoder
hSetDataEnc.h	DtSetDataEncoder
hSetDataInter.h	DtSetDataInteraction
hSignalInter.h	DtSignalInteraction
hStartDec.h	DtStartResumeDecoder
hStartEnc.h	DtStartResumeEncoder
hStartInter.h	DtStartResumeInteraction
hStateDecoder.h	DtHlaStateDecoder
hStateEncoder.h	DtHlaStateEncoder
hStateMsg.h	DtStateMsg
hStopDec.h	DtStopFreezeDecoder
hStopEnc.h	DtStopFreezeEncoder
hStopInter.h	DtStopFreezeInteraction
hUnknownInter.h	DtUnknownInteraction
hVcDec.h	DtViewControlDecoder
hVcEnc.h	DtViewControlEncoder
hVcInter.h	DtViewControlInteraction
iffAtcNavBeam.h	DtIffAtcNavaidsBeamData
iffAtcNavLayr.h	DtIffAtcNavaidsLayerHeader
iffAtcNavOper.h	DtIffAtcNavaidsOperational
iffAtcNavOpr2.h	DtIffAtcNavaidsSecondOperational
iffAtcNavParm.h	DtIffAtcNavaidsParameter

Table A-3: Header Files and the Classes they Contain (Page 11 of 18)

Header File	Class
iffAtcNavPdu.h	DtIffAtcNavaidsPdu
iffAtcNavSyID.h	DtIffAtcNavaidsSystemID
intClassDesc.h	DtInterClassDesc
intDecFact.h	DtInteractionDecoderFactory
intEncFact.h	DtInteractionEncoderFactory
intercmCtlPdu.h	DtIntercomControlPdu
intercmEntDst.h	DtIntercomEntityDestRec
intercmGrpAsg.h	DtIntercomGroupAsgmtRec
intercmGrpDst.h	DtIntercomGroupDestRec
intercmParRec.h	DtIntercomParmRec
intercmSigPdu.h	DtIntercomSignalPdu
interDecoder.h	DtInteractionDecoder
interEncoder.h	DtInteractionEncoder
interMgr.h	DtInteractionManager
intHashKey.h	DtIntHashKey
isGroupOfPdu.h	DtIsGroupOfPdu
isPartOfPdu.h	DtIsPartOfPdu
jammerBeamDec.h	DtJammerBeamDecoder
jammerBeamEnc.h	DtJammerBeamEncoder
laserPdu.h	DtLaserPdu
lcEnd.h	DtLgrEndPdu
lcEofAction.h	DtLgrEofActionPdu
lcHeartbeat.h	DtLgrHeartbeatPdu
lcPause.h	DtLgrPausePdu
lcPdu.h	DtLgrControlPdu
lcPlay.h	DtLgrPlayPdu
lcPlayAction.h	DtLgrPlayActionPdu
lcPlayFile.h	DtLgrPbFilePdu

Table A-3: Header Files and the Classes they Contain (Page 12 of 18)

Header File	Class
lcQuit.h	DtLgrQuitPdu
lcRecAction.h	DtLgrRecordActionPdu
lcRecFile.h	DtLgrRecFilePdu
lcRecord.h	DtLgrRecordPdu
lcSetTime.h	DtLgrSetTimePdu
lcSkipTime.h	DtLgrSkipTimePdu
lcStart.h	DtLgrStartPdu
lcStop.h	DtLgrStopPdu
lcTimeScale.h	DtLgrTimeScalePdu
lcTimeout.h	DtLgrTimeoutPdu
leAppearPdu.h	DtLeAppearancePdu
leArtPartsPdu.h	DtLeArticulatedPartsPdu
leDeadReck.h	DtLeDeadReckoningParameters
leDetonPdu.h	DtLeDetonationPdu
leEntityID.h	DtLeEntityID
leEntityOri.h	DtLeEntityOrientation
leEntityVel.h	DtLeEntityLinearVelocity
leFirePdu.h	DtLeFirePdu
leLocEntity.h	DtLeEntityLocationEntityCoordinates
leLocWorld.h	DtLeEntityLocationWorldCoordinates
leOriError.h	DtLeOrientationError
lePosError.h	DtLePositionError
leTspiPdu.h	DtLeTspiPdu
LibMatrix.h	DtCoordTransform DtCoordTranslation
linObjStPdu.h	DtLinearObjectStatePdu
linSegParam.h	DtLinearSegmentParameter
locObjMgr.h	DtLocalHlaObjectManager

Table A-3: Header Files and the Classes they Contain (Page 13 of 18)

Header File	Class
messagePdu.h	DtMessagePdu
milEntDec.h	DtMilitaryEntityDecoder
milEntEnc.h	DtMilitaryEntityEncoder
milPlatDec.h	DtMilitaryPlatformEntityDecoder
milPlatEnc.h	DtMilitaryPlatformEntityEncoder
mineDataPdu.h	DtMinefieldDataPdu
minePerimeter.h	DtMinefieldPerimeterRecord
mineQueryPdu.h	DtMinefieldQueryPdu
mineRsNackPdu.h	DtMinefieldResponseNackPdu
mineStatePdu.h	DtMinefieldStatePdu
munEntDec.h	DtMunitionEntityDecoder
munEntEnc.h	DtMunitionEntityEncoder
netBoolean.h	DtNetBoolean
NetSocket.h	DtNetSocket
NetStructs.h	DtNet32Vector DtNet64Vector DtNetBurstDescriptor DtNetEntityIdentifier DtNetEntityMarking DtNetEntityType DtNetEulerAngles DtNetEventId DtNetFederateId DtNetGlobalObjectDesignator DtNetRadioEntityType DtNetSimulationAddress
netTimeStamp.h	DtNetTimeStamp
NetTypes.h	DtNetFloat32 DtNetFloat64 DtNetInt16 DtNetInt32 DtNetU16 DtNetU32
nullFedAmb.h	DtNullFederateAmbassador
objClassDesc.h	DtObjClassDesc

Table A-3: Header Files and the Classes they Contain (Page 14 of 18)

Header File	Class
objDesList.h	DtGlobalObjectDesignatorList
objectType.h	DtObjectType
objIdHashKey.h	DtObjectIdHashKey
objIdHashLst.h	DtObjectIdHashlist
objRefCbInfo.h	DtObjectReflectAttributeValuesCbInfo
oldHashList.h	In VR-Link 3.3, we significantly changed the semantics of our DtHashlist class. The new version is defined in hashlist.h. The older DtHashlist class has been renamed DtOldHashlist, and is still defined here in this file for backwards compatibility. While we suggest that you port your code to the new DtHashlist, which is cleaner and easier to use, if you do not want to make that change, you can just use DtOldHashlist wherever you used to use DtHashlist, and the rest of your old code should work.
pairSock.h	DtSocketPairSocket
pdu.h	DtPdu
pduCbInfo.h	DtPduCallbackInfo
pduCbMgr.h	DtPduCallbackManager
pduDatRecSet.h	DtPduDatumRecordSet
pduFactory.h	DtPduFactory
pduWithData.h	DtPduWithData
physEntDec.h	DtPhysicalEntityDecoder
physEntEnc.h	DtPhysicalEntityEncoder
pointObjStPdu.h	DtPointObjectStatePdu
queryPdu.h	DtDataQueryPdu
radarBeamDec.h	DtRadarBeamDecoder
radarBeamEnc.h	DtRadarBeamEncoder
radioRecvrDec.h	DtRadioReceiverDecoder
radioRecvrEnc.h	DtRadioReceiverEncoder
radioRecvrSR.h	DtRadioReceiverRepository
radioXmitDec.h	DtRadioTransmitterDecoder

Table A-3: Header Files and the Classes they Contain (Page 15 of 18)

Header File	Class
radioXmitEnc.h	DtRadioTransmitterEncoder
radioXmitSR.h	DtRadioTransmitterRepository
receivePdu.h	DtReceiverPdu
recIntrCbInfo.h	DtReceiveInteractionCbInfo
recordPdu.h	DtRecordPdu
recordSet.h	DtRecordSet
recordSetPdu.h	DtRecordSetPdu
recQueryPdu.h	DtRecordQueryPdu
refAttrCbInfo.h	DtReflectAttributeValuesCbInfo
refAttrCbList.h	DtReflectAttributeValuesCbList
refGenericObj.h	DtReflectedGenericObject
refGenList.h	DtReflectedGenericObjectList
refObjCbInfo.h	DtReflectedObjectCallbackInfo
refObjMgr.h	DtReflectedHlaObjectManager
regCtlCbInfo.h	DtRegControlCbInfo
regCtlCbList.h	DtRegControlCbList
remObjCbInfo.h	DtRemoveObjectCbInfo
removeEntRPdu.h	DtRemoveEntityRPdu
removePdu.h	DtRemoveEntityPdu
repairPdu.h	DtRepairCompletePdu DtRepairResponsePdu
resupplyPdu.h	DtResupplyCancelPdu DtResupplyOfferPdu DtResupplyReceivedPdu
rprDecFact.h	DtRprStateDecoderFactory
rprEncFact.h	DtRprStateEncoderFactory
rprFomMap.h	DtRprFomMapper
rprIDecFact.h	DtRprInteractionDecoderFactory
rprIEncFact.h	DtRprInteractionEncoderFactory

Table A-3: Header Files and the Classes they Contain (Page 16 of 18)

Header File	Class
rprInterFact.h	DtRprInteractionFactory
rprObjFact.h	DtRprHlaObjectFactory
script.h	DtScript
servReqPdu.h	DtServiceRequestPdu
setDataPdu.h	DtSetDataPdu
setDataRPdu.h	DtSetDataRPdu
setRecordPdu.h	DtSetRecordPdu
signalPdu.h	DtSignalPdu
simanPdu.h	DtSimanPdu
simRprFomMap.h	DtDimpleRprFomMapper
smoother.h	DtSmoother
smoothOriWDR.h	DtSmoothOriWithDr
smoothPosWDR.h	DtSmoothPosWithDr
smoothUtil.h	DtSmoothOrient DtSmoothPos
smUtil.h	DtAckPredOpt
Socket.h	DtFilterTest DtSocket
soldierDec.h	DtSoldierDecoder
soldierEnc.h	DtSoldierEncoder
startPdu.h	DtStartResumePdu
startResRPdu.h	DtStartResumeRPdu
stateRep.h	DtStateRepository
stIntCbInfo.h	DtStartStopInterCbInfo
stIntCbList.h	DtStartStopInterCbList
stopFrzRPdu.h	DtStopFreezeRPdu
stopPdu.h	DtStopFreezePdu
str.h	DtString
strHashKey.h	DtStringHashKey

Table A-3: Header Files and the Classes they Contain (Page 17 of 18)

Header File	Class
supEmisEsNozz.h	DtVectoringNozzleSystem
supEmisEsPdu.h	DtSuppEmisEntStatePdu
supEmisEsProp.h	DtPropulsionSystem
sysHashKey.h	DtSystemHashKey
taitBryan.h	DtTaitBryan
tcpForwarder.h	DtTcpForwarder
tcpSocket.h	DtTcpSocket
tcpSocketMgr.h	DtTcpSocketMgr
thresholder.h	DtThresholder
tkJmDesLst.h	DtTrackJamDesignatorList
topoView.h	DtTopoView
trackJamDes.h	DtTrackJamDesignator
transCtlPdu.h	DtTransferControlRequestPdu
transmitPdu.h	DtModulationType DtTransmitterPdu
trkJamBeamDec.h	DtTrackJamBeamDecoder
trkJamBeamEnc.h	DtTrackJamBeamEncoder
tStampType.h	DtTimeStampType
undwaAcBeam.h	DtUnderwaterAcousticBeam
undwaAcEmtSys.h	DtUnderwaterAcousticEmitterSystem
undwaAcPasAct.h	DtUnderwaterAcousticAddtnlPassiveAct
undwaAcPdu.h	DtUnderwaterAcousticPdu
undwaAcShaft.h	DtUnderwaterAcousticShaftSpeed
unknownHObj.h	DtUnknownHlaObject
unknownPdu.h	DtUnknownPdu
updCtlCbInfo.h	DtUpdateControlCbInfo
updCtlCbList.h	DtUpdateControlCbList
utmCoord.h	DtUtmCoord

Table A-3: Header Files and the Classes they Contain (Page 18 of 18)

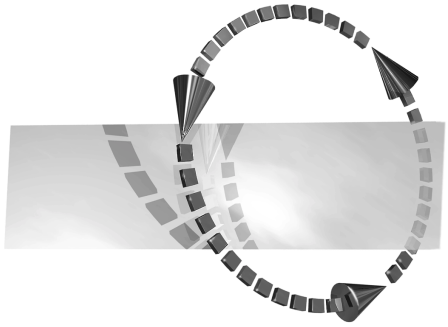
Header File	Class
utmView.h	DtUtmView
varDatumRec.h	DtVariableDatumRecord
vcPdu.h	DtAbsoluteVCPdu DtCompassVCPdu DtGroupVCPdu DtMimicTrackVCPdu DtMimicVCPdu DtModelessVCPdu DtOrbitVCPdu DtTetherTrackVCPdu DtTetherVCPdu DtTrackVCPdu DtUnknownVCPdu DtViewControlPdu
vlList.h	DtIntrusiveList DtList DtListItem DtListedObj
vlutil.h	DtSelectParam
vlVector.h	DtVector
vrlFedAmb.h	DtVrlFederateAmbassador
vrlFedTime.h	DtVrlFedTime
vrlRtiAmb.h	DtVrlRtiAmbassador

A.4. Other Header Files

Table A-4 lists header files that do not contain classes.

Table A-4: Other Header Files

Header FIle	Contents
datumIdType.h	Definitions and functions that map from DtDatumParam to type enumerations.
disEnums.h	Contains enumerations specified by the DIS protocol and RPR FOM.
EntityTypes.h	Contains enumerations of entity-type fields.
enum2string.h	Contains functions that map from DtPduKind enumerations to strings.
Euler.h	Contains functions that convert between Euler angles and transformation matrices.
MachineTypes.h	Defines primitive data types in a machine-independent manner.
mathTypes.h	Contains math macros, includes other math-related headers.
printUtil.h	Defines printing functions.
ProcControl.h	Defines functions for sleeping and polling stdin.
topoCoord.h	Defines functions to generate matrices that convert between geodetic and topographic orientation.
vlprint.h	Contains functions related to the printing of notification messages.
vlTime.h	Contains functions related to time.



Glossary

For additional terms and definitions, see the HLA glossary at:
http://hla.dmso.mil/hla/general/hlagloss.html

absolute dead reckoning

A method of dead-reckoning in which the VR-Link uses the time stamps contained within state updates if the sender is using absolute time stamping. Contrast with relative dead reckoning.

aggregate

The combination of individual entities, aggregates, or both into a single object. For example, an organizational group such as a platoon.

articulated part

A part of an entity that is capable of movement relative to the entity, such as a turret or gun.

attached part

An object that is attached to another object, such as a rocket attached to a jet.

body coordinate frame

A coordinate framework centered on an entity. To represent the orientation of an entity, VR-Link uses Euler angles or a rotation matrix that rotates from a reference frame to the body frame.

Cartesian coordinates

A system for indicating location by means of three planes intersecting at right angles to one another at the origin.

dead-reckoning

A process by which the VR-Link calculates the expected location of an entity during periods between state updates, based on velocity, acceleration, and rotational velocity.

Defense Modeling and Simulation Office

The executive secretariat for the Executive Council on Modeling and Simulation (EXCIMS), which provides a full-time focal point for information concerning Department of Defense modeling and simulation (M&S) activities. Currently the DMSO promulgates M&S policy, initiatives, and guidance to promote cooperation among DoD components to maximize efficiency and effectiveness.

designator

A simulated device capable of lasing, or designating, a target to be destroyed by guided munitions.

DIS

Distributed Interactive Simulation.

Distributed Interactive Simulation

The DIS protocol is a set of standards that govern how participating applications share information about the virtual world. The protocol specifies a set of packets, called protocol data units (PDUs), that communicate this information. Each PDU identifies the sender and contains other information depending on the PDU type. The DIS protocol also specifies when and how frequently PDUs are sent.

The DIS protocol has been superseded by the High-Level Architecture protocol for use by the Department of Defense.

DMSO

Defense Modeling and Simulation Office.

emitter

A simulated device on an entity that emits electromagnetic radiation, for example, radar.

entity

An element in a simulation, such as a vehicle or a person, that is represented in the simulation through the issuance of state messages.

entity-type

A list of seven components as specified by the DIS protocol and RPR FOM. If none of the seven components contains a wildcard (-1) value, then the entity type refers to a specific, narrowly-defined entity. If some components contain a wildcard, then the entity type refers to a class of entities. A larger number of wildcards indicates a broader, more general class.

The components of an entity-type are:

- Entity kind
- Domain
- Country
- Category
- Subcategory
- Specific
- Extra

Euler angles

A set of three angles used to describe the orientation of an entity as a set of three successive rotations about three different orthogonal axes (x, y, and z). These angles specify successive rotations needed to transform from a reference coordinate system to the entity's body coordinate system.

event

An interaction between objects or between an object and the terrain, such as firing of a munition, or a collision of entities.

exercise

The DIS term for a one or more interacting simulation applications. Compare to federation execution in HLA.

exercise connection

The object (DtExerciseConn) through which a VR-Link application connects to the network. State messages are sent through the exercise connection and information about remote entities is received through the exercise connection.

exercise ID

A numeric identification for a DIS simulation exercise.

FED file

Federation Execution Data file. The Federation Execution Data is the subset of FOM data needed and read by the RTI. VR-Link also reads the FED file.

federate

A connection to the RTI. Typically a single simulation application can be thought of as a federate.

federation

A group of HLA federates capable of playing in the same federation execution.

Federation Object Model

Defines the data content of a federation execution.

federation execution

A federation execution is the HLA equivalent of a DIS simulation exercise.

FOM

Federation Object Model.

geocentric coordinates

A coordinate system calculated with respect to the earth's center. The origin of the geocentric coordinate system is the center of the earth. The positive X-axis passes through the prime meridian at the equator; the positive Y-axis passes through 90 degrees east longitude at the equator; and the positive Z-axis passes though the north pole.

geodetic coordinates

A coordinate system in which position is determined relative to a reference ellipsoid, such as the surface of the earth at sea level. VR-Link geodetic coordinates consist of latitude and longitude in radians, and altitude in meters above the reference ellipsoid.

guise

An alternative entity type used to display an object depending on the force ID. For example, a tank could look like an M1A1 to friendly forces and a T72 to hostile forces.

heartbeat

In DIS, the frequency with which current PDUs are sent to the network regardless of whether or not the entity's state has changed.

High-Level Architecture

The High Level Architecture (HLA) for simulations is a DOD-wide initiative to provide an architecture to support interoperability and reuse of simulations. The HLA supersedes DIS for the Department of Defense.

HLA

High-Level Architecture.

interaction

A message describing a simulation event. An HLA interaction describes an event, rather than an object's state.

logger control messages

A set of DIS PDUs or HLA interactions that let you control the MÄK Logger remotely.

MÄK Technologies Lisp

An adaptation of the Lisp language used in configuration files for MÄK Technologies products.

message

A general term used to refer to DIS PDUs and HLA interactions and state updates.

MTL

MÄK Technologies Lisp.

object

An element in a simulation that has persistence, as opposed to an interaction, which is a transient element.

object handle

An integer that an application uses to identify a particular object in RTI service calls. An object handle is meaningful only to a particular federate. The same object can be known to different federates by different object handles.

object name

A character string that can be used to identify an object. The object name is known to the RTI, and the RTI provides functions to find out an object's name, given its handle, and vice versa. Object names can be chosen by applications that register the objects with the RTI, however if you do not want to choose names for objects, the RTI will assign names for you.

PDU

Protocol Data Unit.

Protocol Data Unit

A message (packet) that is passed on a network between DIS simulation applications.

radio transmitter

A simulated device on an entity, capable of transmitting radio communications.

radio receiver

A simulated device on an entity, capable of receiving radio communications.

Realtime Platform Reference FOM

An HLA reference FOM based on the DIS protocol.

reflected entity list

A list of entities simulated by remote applications (federates in HLA) and about which the local simulation has received information over the network.

reference FOM

A FOM designed to be used as a whole, or with modification, by a wide variety of similar federation executions.

relative dead reckoning

A method of dead reckoning in which the VR-Link approximates the location of an object based on the local time of receipt of the last state update message.

RTI Initialization Data (RID)

Data required by an RTI during initialization, independent of the FOM being used. RID data is usually dependent on a specific implementation of the RTI.

RPR FOM

Realtime Platform Reference FOM.

RTI

Run-Time Infrastructure.

Run-Time Infrastructure

A library and other supporting software that implements the HLA interface specification. All federates communicate with one another in an HLA environment through RTI functions.

Simulation Interoperability Standards Organization

A group that seeks to promote modeling and simulation interoperability and reuse for the benefit of diverse M&S communities, including developers, procurers, and users, world-wide.

simulation time

VR-Link simulation time is used in dead-reckoning of remote entities and thresholding of local entities. Typically, VR-Link simulation time is set once during each iteration of the application's main simulation loop so that all entities are dead-reckoned based on the same value of current time.

SISO

Simulation Interoperability Standards Organization

smoothing

A method of ensuring that transitions from an entity's dead-reckoned position to its actual position aren't so abrupt as to be visually disconcerting.

state

The current status of an object, including location, direction of movement, extent of damage, and so on.

topographic coordinates

A right-handed Cartesian coordinate system whose X-Y plane is tangent to the earth's surface at the origin, with the positive X axis pointing north, the positive Y axis pointing east, and the positive Z axis pointing down. There are an infinite number of topographic coordinate systems – one for each point on the earth's surface.

topographic coordinate frame

A coordinate frame in the context of the terrain. When the eyepoint moves with respect to the terrain, we describe it as movement in a topographic coordinate frame.

UDP port

A network channel through which the VR-Link sends and receives data for DIS exercises.

Universal Transverse Mercator

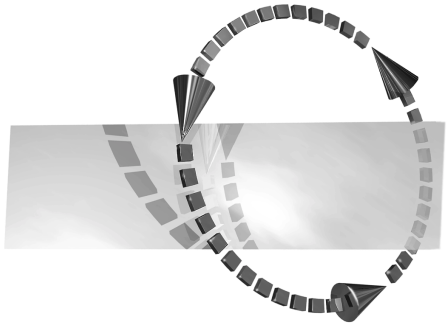
In general, a non-cartesian coordinate system in which the X, Y, and Z correspond to easting (nearly east), northing (nearly north), and height above an ellipsoid that approximates the surface of the earth.

UTM coordinates

Universal Transverse Mercator coordinates.

view control messages

A set of DIS PDUs or HLA interactions that let you control the MÄK Stealth remotely.



Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X-Y-Z

MÄK Technologies
185 Alewife Brook Parkway
Cambridge, MA 02138
617.876.8085

VRL-3.4-1-990513