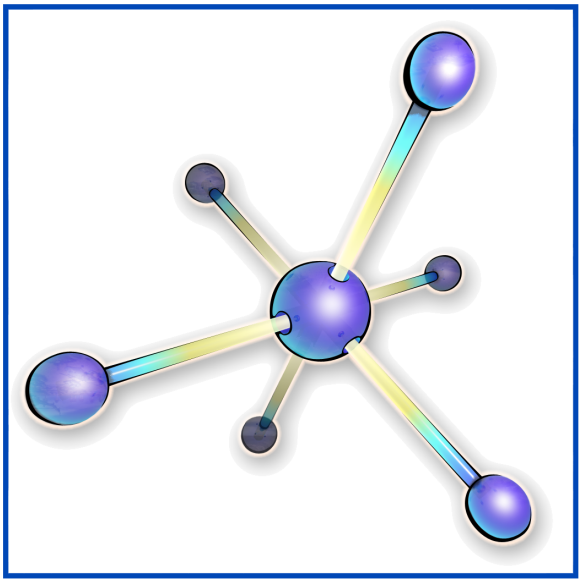
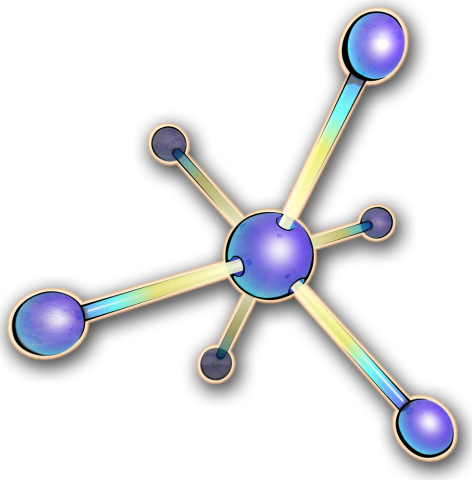


VR-Exchange

Users Guide

VT MÄK
A company of VT Systems



VR-Exchange

Users Guide

VT MÄK
A company of VT Systems

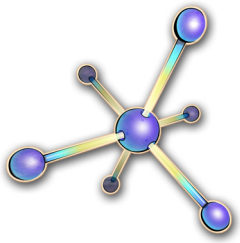
VT MÄK
68 Moulton St.
Cambridge, MA 02138 USA

Voice: 617-876-8085
Fax: 617-876-9208

info@mak.com

www.mak.com

Revision VRE-2.0-1-110210

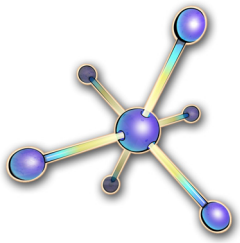


Contents

Preface

Chapter 1 Introduction

Chapter 5 The VR-Exchange API



Preface

This manual is written for persons who will install or use VR-Exchange. The manual assumes that you are familiar with your operating system and windowing environment.

Please see *VR-Exchange Release Notes* for updates to this manual and the latest information about your version of VR-Exchange.

How the Manual is Organized

This manual is organized as follows:

Chapter 1, *Introduction*, describes VR-Exchange and its architecture. It also contains an important note about the limitations of translation software.

Chapter 2, *Installing VR-Exchange*, describes installation, including setting up the license server and installing an RTI.

Chapter 3, *Using VR-Exchange*, explains how to run VR-Exchange and use it to manage connections and view the data being processed.

Chapter 4, *Working with Connections*, explains how to add, edit, and configure connections.

Chapter 5, *The VR-Exchange API*, explains how to create brokers using the VR-Exchange API.

Appendix A, *Broker Details* describes the configuration parameters for each broker, lists the supported translators, and lists the command-line options that you can use when you start brokers.

Appendix B, *A VR-Exchange/TENA Primer*, provides a brief explanation of how to download and install TENA software.

The *MÄK Products Glossary* defines terms common to real-time simulations and MÄK products.

MÄK Products

VR-Exchange is a member of the VT MÄK line of software products designed to streamline the process of developing and using networked simulated environments. The VT MÄK product line includes the following:

- VR-Link® Network Toolkit. VR-Link is an object-oriented library of C++ functions and definitions that implement the High Level Architecture (HLA) and the Distributed Interactive Simulation (DIS) protocol. VR-Link has built-in support for the RPR FOM and allows you to map to other FOMs. This library minimizes the time and effort required to build and maintain new HLA or DIS-compliant applications, and to integrate such compliance into existing applications.

  VR-Link includes a set of sample debugging applications and their source code. The source code serves as an example of how to use the VR-Link Toolkit to write applications. The executables provide valuable debugging services such as generating a predictable stream of HLA or DIS messages, and displaying the contents of messages transmitted on the network.

- MÄK RTI. An RTI (Run-Time Infrastructure) is required to run applications using the High Level Architecture (HLA). The MÄK RTI is optimized for high performance. It has an API, RTIspy®, that allows you to extend the RTI using plug-in modules. It also has a graphical user interface (the RTI Assistant) that helps users with configuration tasks and managing federates and federations.
- VR-Forces®. VR-Forces is a computer generated forces application and toolkit. It provides an application with a GUI, that gives you a 2D and 3D views of a simulated environment.

  You can create and view local entities, aggregate them into hierarchical units, assign tasks, set state parameters, and create plans that have tasks, set statements, and conditional statements. VR-Forces also functions as a plan view display for viewing remote entities taking part in an exercise. Using the toolkit, you can extend the VR-Forces application or create your own application for use with another user interface.

- VR-Vantage™. VR-Vantage is a line of products designed to meet your simulation visualization needs. It includes three end-user applications — VR-Vantage Stealth, VR-Vantage XR, and VR-Vantage IG — and the VR-Vantage Toolkit. VR-Vantage Stealth displays a realistic, 3D 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.

  VR-Vantage IG is a configurable desktop image generator (IG) for out the window (OTW) scenes and remote camera views. It has most of the features of the Stealth, but is optimized for its IG function.

VR-Vantage XR provides a common operational picture using the same 3D views as VR-Vantage Stealth, a 2D plan view, and an exaggerated reality (XR) view. Together these views provide both situational awareness and the big picture of the simulated world.

The VR-Vantage Toolkit is a 3D visual application development toolkit. Use it to customize or extend MÄK's VR-Vantage applications, or to integrate VR-Vantage capabilities into your custom applications. VR-Vantage is built on top of OpenSceneGraph (OSG). The toolkit includes the OSG version used to build VR-Vantage.

- MÄK Data Logger. The Data Logger, also called the 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 and can quickly jump to areas of interest. The Logger has a GUI and a text interface. The Logger API allows you to extend the Logger using plug-in modules or embed the Logger into your own application. The Logger editing features let you merge, trim, and offset Logger recordings.
- MÄK Plan View Display. The Plan View Display (PVD) provides a 2D, "big picture" of a virtual environment. It shows simulated entities, such as vehicles, soldiers, and tanks moving over a 2D-map display.
- VR-Exchange™. VR-Exchange allows simulations that use incompatible communications protocols to interoperate. For example, within the HLA world, using VR-Exchange, federations using the HLA RPR FOM 1.0 can interoperate with simulations using RPR FOM 2.0, or federations using different RTIs can interoperate. VR-Exchange supports HLA, TENA, and DIS translation.
- VR-TheWorld™ Server. VR-TheWorld Server is a simple, yet powerful, web-based streaming terrain server, developed in conjunction with Pelican Mapping. Delivered with a global base map, you can also easily populate it with your own custom source data through a web-based interface. The server can be deployed on private, classified networks to provide streaming terrain data to a variety of simulation and visualization applications behind your firewall.

How to Contact Us

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

Telephone

Call or fax us at:	Voice:	617-876-8085 (extension 3 for support)
	Fax:	617-876-9208

E-mail

Sales and upgrade information:	info@mak.com
Technical support:	support@mak.com
VR-Vantage support:	vrv-support@mak.com

Internet

MÄK web site home page:	www.mak.com
License key requests:	www.mak.com/support/licenses.php
Product version and platform information:	www.mak.com/support/productversions.php
For the free, unlicensed MÄK RTI:	www.mak.com/products/rti.php
Support FAQ:	www.mak.com/support/faq.php

Post

Send postal correspondence to:	VT MÄK 68 Moulton St. Cambridge, MA, USA 02138

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

Document Conventions

This manual uses the following typographic conventions:

`Monospaced`	Indicates commands or values you enter.
`Monospaced Bold`	Indicates a key on the keyboard.
`Monospaced Italic`	Indicates command variables that you replace with appropriate values.
`{ }`	Indicates required arguments.
`[ ]`	Indicates optional arguments.
|	Separates options in a command where only one option may be chosen at a time.
`(	)`	In command syntax, indicates equivalent alternatives for a command-line option, for example, `(-h | --help)`.
/	Indicates a directory. Since MÄK products run on both UNIX and Windows PC platforms, we use the / (slash) for generic discussions of pathnames. If you are running on a PC, substitute a \ (backslash) when you type pathnames.
Italic	Indicates a file name, pathname, or a class name.
sans Serif	Indicates a parameter or argument.
➤	Indicates a one-step procedure.
Menu → Option	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.

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

Hardware and Operating System Naming Conventions

Unless otherwise specified, references to UNIX include Linux, regardless of the type of computer it is running on. References to PCs refer to Intel processor-compatible computers running a version of Microsoft Windows.

Mouse Button Naming Conventions

An instruction to click the mouse button, refers to clicking the primary mouse button, usually the left button for right-handed mice and the right button for left-handed mice. The popup menu refers to the menu displayed when you click the secondary mouse button, usually the right button on right-handed mice and the left button on left-handed mice.

Third Party Licenses

MÄK software products may use code from third parties. This section contains the license documentation required by these third parties.

Boost License

VR-Link, and all MÄK software that uses VR-Link uses some code which is distributed under the Boost License. All header files that contain Boost code are properly attributed. The boost web site is: www.boost.org.

Boost Software License - Version 1.0 - August 17th, 2003

Permission is hereby granted, free of charge, to any person or organization obtaining a copy of the software and accompanying documentation covered by this license (the "Software") to use, reproduce, display, distribute, execute, and transmit the Software, and to prepare derivative works of the Software, and to permit third-parties to whom the Software is furnished to do so, all subject to the following:

The copyright notices in the Software and this entire statement, including the above license grant, this restriction and the following disclaimer, must be included in all copies of the Software, in whole or in part, and all derivative works of the Software, unless such copies or derivative works are solely in the form of machine-executable object code generated by a source language processor.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT SHALL THE COPYRIGHT HOLDERS OR ANYONE DISTRIBUTING THE SOFTWARE BE LIABLE FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

libXML and libICONV

VR-Link and all MÄK software that uses VR-Link, links in libXML and libICONV. On some platforms the compiled libraries and header files are distributed with MÄK Products. MÄK has made no modifications to these libraries. For more information about these libraries please see the following web sites:

- The LGPL license is available at: http://www.gnu.org/licenses/lgpl.html
- Information about IconV is at: http://www.gnu.org/software/libiconv/
- Information about LibXML is at: http://xmlsoft.org/

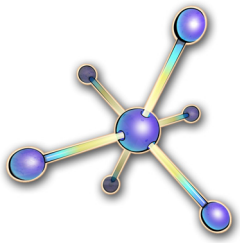
pThreads Library

VR-Exchange links with the pThreads win32 library. The library is distributed under the GNU Lesser General Public License (LGPL). MÄK has made no modification to this library. For information about the pThreads win32 library please see: http://sourceware.org/pthreads-win32/

EHS, PCRE, and PME

The MÄK RTI links with the EHS, PCRE, and PME libraries. These libraries are distributed under the GNU Lesser General Public License (LGPL). MÄK has made modification to these libraries only to allow them to compile on the supported platforms. For more information about these libraries please see the following web sites:

- EHS: http://xaxxon.slackworks.com/ehs/
- PCRE: http://www.pcre.org/
- PME: http://xaxxon.slackworks.com/pme/index.html



1 Introduction

This chapter describes VR-Exchange's features and its architecture.

1.1. The VR-Exchange Architecture

VR-Exchange allows simulations that use incompatible communications protocols to interoperate. For example, within the HLA world, using VR-Exchange, simulations using the HLA RPR FOM 1.0 can interoperate with simulations using RPR FOM 2.0, or simulations using different RTI implementations can interoperate.

VR-Exchange permits simulations to interoperate through the use of a shared memory space (Portal) and brokers (Figure 1-1). Each broker translates between its native protocol and the VR-Exchange common simulation representation. The translated data passes through the Portal and is translated by the other brokers to their protocol.

Given this architecture, it is not necessary to create multiple translation programs that map between specific sets of protocols. You only need to create a broker for a particular protocol, and simulations can interoperate with any other simulations for which a broker exists.

You can run multiple instances of a broker, each configured for a particular simulation. For example, Figure 1-1 shows three different federations. Each uses the broker, but each broker is configured to use different RTIs and different FOMs. Each unique broker configuration is called a connection.

Because the Portal and brokers use shared memory, the Portal and all brokers must run on the same computer.

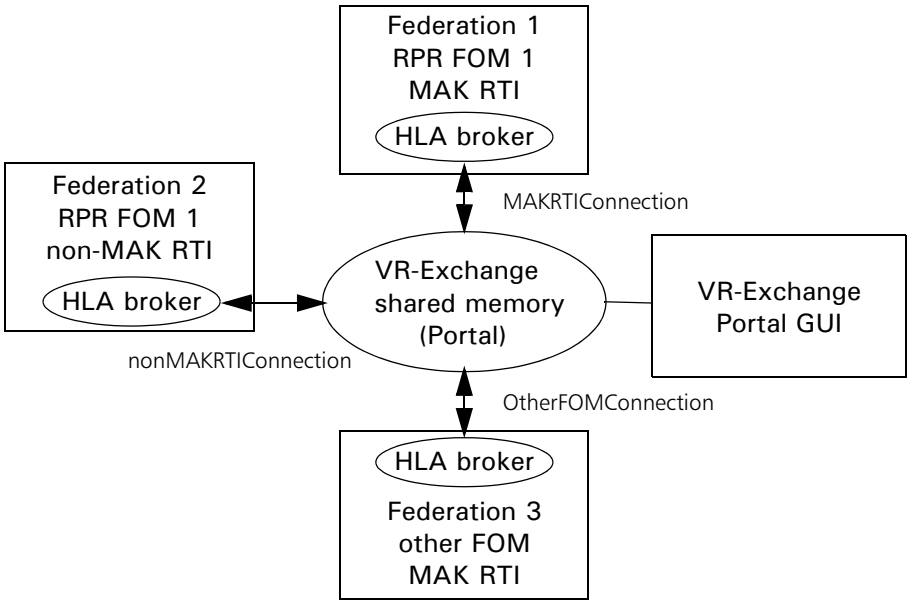


Figure 1-1. VR-Exchange architecture

1.1.1. Translators

Each broker has translators for the object or interaction classes that it supports. Figure 1-2 shows how a translator works. Imagine language-specific HLA FOMs (or TENA LROMs), in this case one for American English and one for British English. Their brokers have translators for the different concepts in the English language. The leftmost federation sends a “lorry” message. The vehicle translator in the broker translates “lorry” to a generic description “wheeled cargo vehicle” and puts the data in the VR-Exchange shared memory area. A second federation, also using the British English FOM, takes the vehicle data from the Portal and translates it back to “lorry”. Finally, a third federation, which uses an American English FOM, reads the vehicle message from the Portal and translates it to its American English equivalent, “truck.”

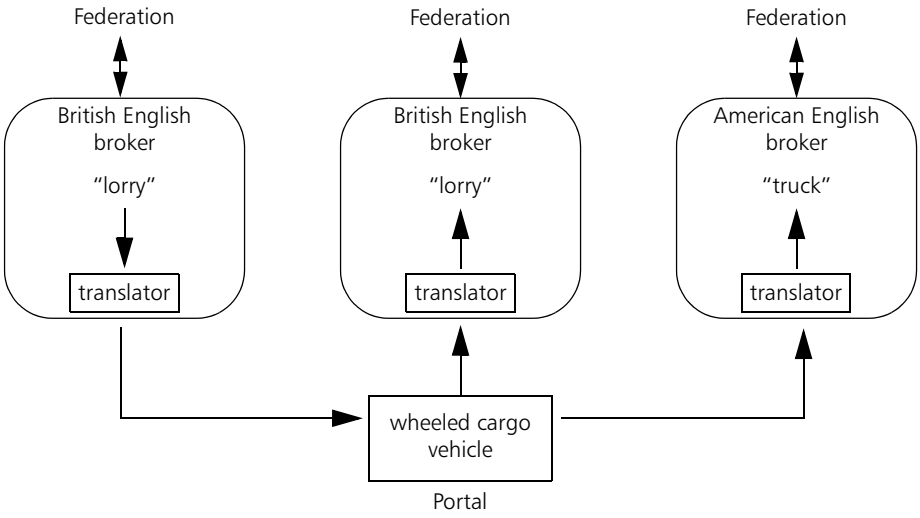


Figure 1-2. How translators work

You can disable translators by editing a connection. Since each connection has its own configuration, you can filter the objects and interactions that the federations are communicating through VR-Exchange.

1.1.2. Using VR-Exchange to Manage a Federation of Federations

In addition to using VR-Exchange to enable heterogeneous federations to interoperate, you can use it to help manage a federation of federations, regardless of whether they use the same FOMs and RTIs. Consider the configuration in Figure 1-3. Four federations are connected via brokers to a VR-Exchange Portal. These Portals are connected to brokers that form their own federation. If any of the individual federations fail, then each instance of VR-Exchange can detect this and drop it from the exercise, while the others can continue.

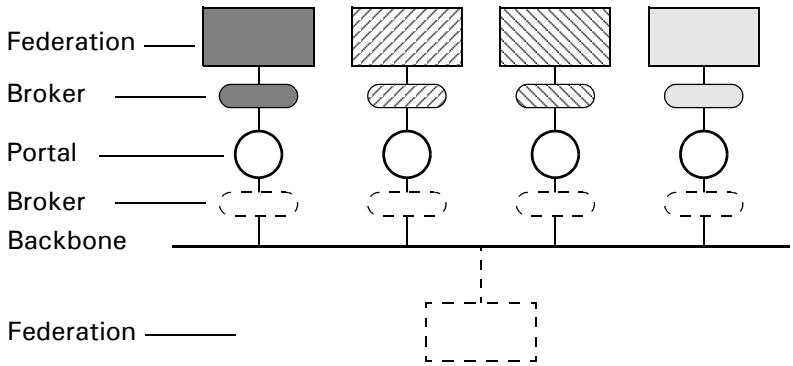


Figure 1-3. Using VR-Exchange over a WAN

1.1.3. The VR-Exchange Portal Application

VR-Exchange has a graphical user interface (Portal application) that provides a view into the shared memory space. It displays the connections in use and the objects and interactions being translated (Figure 1-4). The Portal application does not control the translation process. Translation takes place in the brokers, which are separate programs.

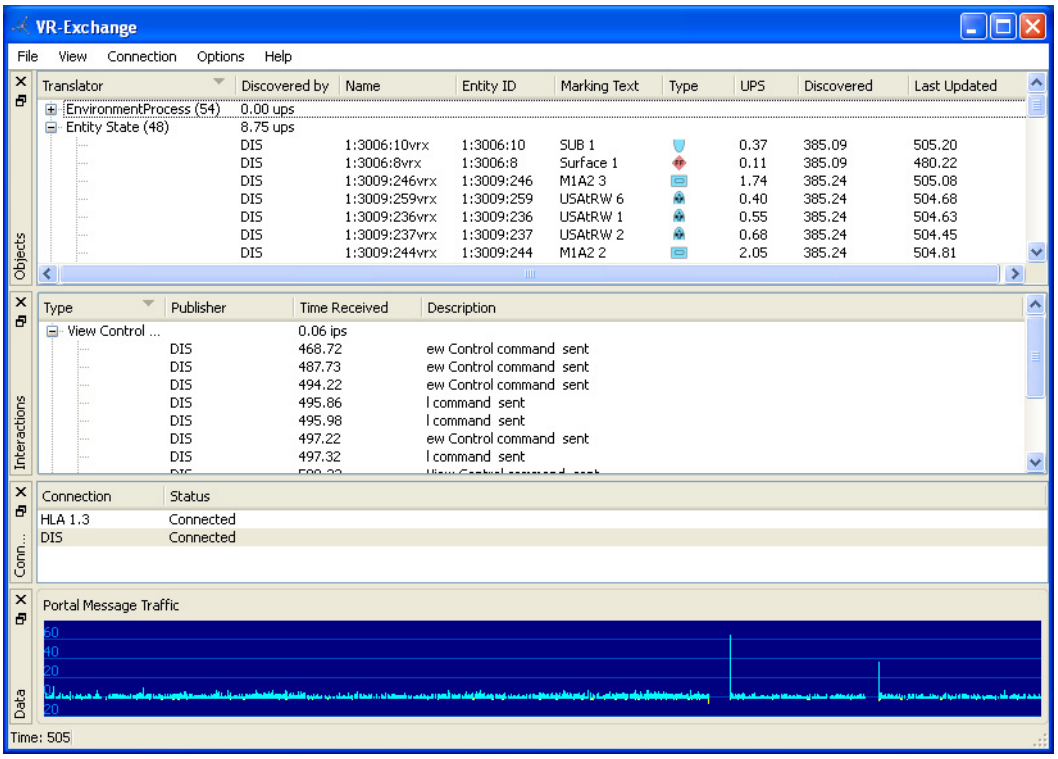


Figure 1-4. VR-Exchange Portal window

1.1.4. FOM-Agility and LROM-Agility

VR-Exchange brokers accommodate different FOMs and LROMs using VR-Link FOM Mappers and LROM Mappers. Mappers are dynamic link libraries that map the objects and concepts of a particular FOM or LROM to the VR-Link internal object model. Since the VR-Exchange brokers are VR-Link applications, they can use these mappers to switch between different FOMs and LROMs. For details about FOM and LROM mapping, please see *VR-Link Developers Guide*.

1.2. The VR-Exchange API

The VR-Exchange API has two primary components, the Broker API and the Portal API. The Broker API lets you create brokers for use with VR-Exchange or extend the brokers provided by MÄK. The Portal API allows brokers to send messages to and receive updates from, the Portal. For details, please see Chapter 5, *The VR-Exchange API*.

1.3. Brokers Supplied

VR-Exchange includes brokers for HLA, TENA, and DIS. It has HLA brokers for the HLA 1.3 specification, for the SISO Standard Dynamic Link Compatible C++ API for IEEE 1516 specification, and for HLA Evolved (IEEE 1516-2010) specification. Subject to licensing, you can run as many instances of these brokers as you want. The HLA brokers support the versions of the RPR FOM supported by VR-Link, or any FOM that has a VR-Link FOM Mapper.

The TENA broker allows VR-Exchange to participate in TENA executions. It uses LROM Mappers to map TENA objects to the VR-Exchange common simulation representation. VR-Exchange provides a common object LROM Mapper and a MÄK LROM Mapper. For details, please see "The TENA Broker," on page A-11.

The DIS broker receives DIS 2.0.4, IEEE 1278.1, and IEEE 1278.1a PDUs. It sends IEEE 1278.1 PDUs.

Table 1-1 lists the translation capabilities of each broker. For more information about translators, please see "HLA Translators," on page A-3, "TENA Translators," on page A-12, and "The DIS Broker," on page A-8.

Table 1-1: Broker translators

Translator	HLA	DIS	TENA
Acknowledge	X	X	X
Action Request	X	X	
Action Response	X	X	
Aggregate	X	X	X
Collision	X	X	X
Comment	X	X	X
Create Entity	X	X	
Data	X	X	X
Data Query	X	X	
Designator	X	X	X
Detonation	X	X	X
Emitter System	X	X	X
Entity State	X	X	X
Environment Process	X	X	X
Event Report	X	X	
Fire	X	X	X
Gridded Data	X	X	
IFF	X	X	X
Radio Receiver	X	X	X
Radio Signal	X	X	X
Radio Transmitter	X	X	X
Remove Entity	X	X	
Set Data	X	X	X
Start	X	X	X
Stop	X	X	X
View Control	X	X	

1.4. Translating Between DIS and the RPR FOM

DIS exercises sends updates at regular intervals, called heartbeats. HLA does not have this feature. VR-Exchange handles this difference as follows:

- VR-Exchange provides heartbeat updates for the appropriate PDUs when the time between HLA updates exceeds the heartbeat threshold.
- VR-Exchange deletes HLA objects when the time between DIS PDUs exceeds the timeout threshold (2.5*heartbeat). RPR FOM embedded system objects (radio transmitter/receiver and emitter system/beams) are deleted only if their host entity no longer exists.

VR-Exchange does not support the full set of PDUs in the 1278.1a specification. It supports the following PDUs (listed by protocol families):

- Entity Information and Entity Interaction PDUs
  - Entity State PDU
  - Collision PDU
- Warfare PDUs
  - Fire PDU
  - Detonation PDU
- Simulation Management PDUs
  - Start/Resume PDU
  - Stop/Freeze PDU
  - Acknowledge PDU
  - Action Request PDU
  - Action Response PDU
  - Data Query PDU
  - Set Data PDU
  - Data PDU
  - Event Report PDU
  - Comment PDU
  - Create Entity PDU
  - Remove Entity PDU
- Radio Communications PDUs
  - Transmitter PDU
  - Signal PDU
  - Receiver PDU

- Distributed Emission Regeneration PDUs
  - Electromagnetic Emission PDU
  - Designator PDU
  - IFF PDU
- Logistics PDUs
  - Service Request PDU
  - Resupply Offer PDU
  - Resupply Received PDU
  - Resupply Cancel PDU
  - Repair Complete PDU
  - Repair Response PDU
- Entity Management PDUs
  - Transfer Control Request PDU
- Synthetic Environment PDUs
  - Environmental Process PDU
  - Gridded Data PDU.
- Logger Control PDUs (for MÄK Logger)
- View Control PDUs (for MÄK Stealth and VR-Vantage applications).

The following sections describe the supported PDUs in greater detail.

1.4.1. Entity Information and Entity Interaction

The DIS Entity Information/Interaction PDUs are the Entity State PDU and the Collision PDU.

The Entity State PDU is translated into a RPR FOM object. The RPR FOM has broken down the Entity State PDU into an object class hierarchy rooted at the *BaseEntity* class with several levels of subclasses as shown in Table 1-2. The object class that is registered in the federation execution is dependent on the value of the entity type field. The default mapping between the entity type field and the RPR FOM object class is based on the entity type kind and domain values. For example, a DIS M1 tank would be instantiated as a *GroundVehicle* object in the RPR FOM, whereas a TOW missile would be a *MunitionEntity*.

The data fields of the Entity State PDU are translated to and from the corresponding attributes of the appropriate RPR FOM object class. The entity ID, entity type, and position attributes are required to send an Entity State PDU.

Table 1-2: BaseEntity Object Classes

Class1	Class2	Class3	Class4
BaseEntity	PhysicalEntity	PlatformEntity	Aircraft
			AmphibiousVehicle
			GroundVehicle
			MultiDomainPlatform
			Spacecraft
			SubmersibleVehicle
			SurfaceVessel
		Lifeform	Human
			NonHuman
		CulturalFeature	
		Expendables	
		Munition	
		Radio	
		Sensor	

The Collision PDU is translated into a RPR FOM *Collision* interaction. The *Collision* interaction is defined at a single level, with no inheritance.

1.4.2. Warfare

The DIS Warfare PDUs are the Fire PDU and the Detonation PDU, which are translated into RPR FOM *WeaponFire* and *MunitionDetonation* interactions, respectively. Each interaction class is defined at a single level, with no inheritance.

1.4.3. Simulation Management

The DIS Simulation Management (SIMAN) PDUs are translated into the corresponding RPR FOM interactions as shown in Table 1-3.

Table 1-3: SIMAN Representation in the RPR FOM

DIS PDU	RPR FOM Interaction Class
Create Entity	CreateEntity
Remove Entity	RemoveEntity
Acknowledge	Acknowledge
Set Data	SetData
Data Query	DataQuery
Data	Data
Action Request	ActionRequest
Action Response	ActionResponse
Event Report	EventReport
Comment	Comment
Start/Resume	StartResume
Stop/Freeze	StopFreeze

1.4.4. Radio Communications

The DIS Radio Communications PDUs are the Transmitter, Receiver, and Signal PDUs. The Receiver and Transmitter PDUs are translated into RPR FOM objects as *RadioReceiver* and *RadioTransmitter* object classes, respectively. The RPR FOM has broken down the Radio Transmitter and Receiver protocols into an object class hierarchy as shown in Table 1-4. The *EmbeddedSystem* root object class captures the common attributes that specify the objects as being attached to other objects. The specific object class that is used to register a radio object in the federation execution is dependent on the type of PDU received by VR-Exchange. The data fields of the Transmitter and Receiver protocol are translated to and from the corresponding attributes of the appropriate RPR FOM object class. The host entity ID or host object ID (mapping to an entity ID) is required to send a Radio Transmitter PDU.

Table 1-4: EmbeddedSystem Object Classes

Class1	Class2	Class3
EmbeddedSystem	Designator	
	EmitterSystem	
	RadioReceiver	
	RadioTransmitter	
	IFF	IffInterrogator
		aIffTransponder

The Signal protocol is translated into one of several RPR FOM *RadioSignal* interaction classes. A base RadioSignal interaction is defined from which the *ApplicationSpecificRadioSignal*, *DatabaseIndexRadioSignal*, *EncodedAudioRadioSignal*, and *RawBinaryRadioSignal* are derived. The HLA signal interaction that VR-Exchange generates is based on the Encoding Class in the DIS Signal PDU.

Table 1-5: RadioSignal Interaction Class

Class1	Class2
RadioSignal	ApplicationSpecificRadioSignal
	DatabaseIndexRadioSignal
	EncodedAudioRadioSignal
	RawBinaryRadioSignal

1.4.5. Distributed Emission Regeneration

The Distributed Emission Regeneration protocol family consists of the Electromagnetic Emission (EE), Designator, and IFF protocols. The EE PDU is transformed into several RPR FOM object classes. The Designator and IFF PDUs are represented in the RPR FOM by a derivation of the *EmbeddedSystem* class as Designator and IFF respectively (see Table 1-4). The IFF object class is further broken down into the *IffTransponder* and *IffInterrogator* object classes. These classes allow subscription based filtering of IFF data.

The EE PDU translation is different from any of the other PDU transformations in that it is a many-to-one and one-to-many transformation. A separate RPR FOM object is created for each emitter system and emitter beam in an EE PDU. Conversely, the emitter system and emitter beam objects reflected to VR-Exchange must be combined into a single EE PDU (for the State Update). The emitter system data is transformed into an *EmitterSystem* object class. An *EmitterSystem* object is created for each emitter system record in the EE PDU. The *EmitterSystem* is derived from the *EmbeddedSystem* class that associates each EmitterSystem object with its host entity object. Based on the beam function, the EE PDU emitter beam data is transformed into either a *RadarBeam* (the default) or a *JammerBeam*. The *EmitterBeam* class contains all of the emitter beam data parameters. Similar to the *EmbeddedSystem*, the *EmitterBeam* defines an *EmittingSystemID* that establishes the association between it and its EmitterSystem object instance.

VR-Exchange creates a Delta State EE PDU for each *EmitterSystem* or *EmitterBeam* reflection. VR-Exchange does not generate a delta EE PDU if the *EmitterSystem* has no active beams (unless that beam is being removed). VR-Exchange must also first receive the *EmitterBeam's EmitterSystem* data as well as the emitting entity data. The emitting system ID and beam ID are required for sending an EE PDU. VR-Exchange generates the State Update EE PDU at a heartbeat rate (regardless of intermediate delta updates). The State Update EE PDU contains the emission data for each emitter with active emitter beams.

1.4.6. Logistics

The DIS Logistics PDUs are translated into RPR FOM interaction classes as shown in Table 1-6.

Table 1-6: Logistics Representation in the RPR FOM

DIS PDU	RPR FOM Interaction Class
Service Request	ServiceRequest
Resupply Offer	ResupplyOffer
Resupply Received	ResupplyReceived
Resupply Cancel	ResupplyCancel
Repair Complete	RepairComplete
Repair Response	RepairResponse

The object ID attributes (mappable to DIS IDs) for the participating entities are required to send these PDUs.

1.4.7. Entity Management

The Transfer Control Request PDU is translated to the HLA *TransferControl* interaction class. The *TransferControl* class is available only if you are using RPR FOM version 2, draft 6 or later.

1.4.8. Synthetic Environment

VR-Exchange supports two PDUs in the Synthetic Environment family: Environmental Process PDU and Gridded Data PDU. The Environmental Process PDU is translated to the *EnvironmentProcess* class. The Gridded Data PDU is translated to the *GriddedData* class.

1.4.9. MÄK Logger Control and MÄK Stealth View Control PDUs

In addition to the standard DIS PDUs described in previous sections, VR-Exchange supports Logger Control PDUs for controlling the MÄK Logger and View Control PDUs for controlling the MÄK Stealth and VR-Vantage applications.

1.5. Hardware Requirements and Product Limitations

VR-Exchange was designed to work with up to 254 connections. However, the requirement that all brokers run on the same machine limits the actual number of connections that you can run. VR-Exchange has been tested with as many as 4 connections.

It is recommended that VR-Exchange be run with one processor for each connection. Though this is not required, and much testing has been done on single processor machines, a typical instance of VR-Exchange will have three or more processes competing for CPU time.

It is also recommended that each protocol supported exist on a dedicated network. This is not a requirement of VR-Exchange, but is recommended for high traffic exercises.

1.6. The Limitations of Translation Software

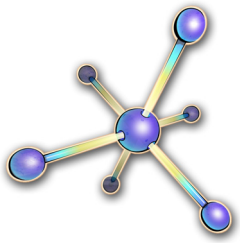
Translating between two protocols, or even two FOMs (in the case of HLA) is similar to translating between two spoken languages, in that you may lose something in translation. The goal of a good translator is to try to minimize that loss. When translating between spoken languages, you cannot simply perform word for word replacement. The source text needs to be examined holistically, and translated so that it conveys the same meaning to the target language. Sometimes idiomatic structures are not directly translatable and can only be approximated in the target language. And some concepts present in one language may not even exist in the other. As a result, some precision may be lost.

This same limitation occurs with all software translators. Messages often cannot be translated on a packet by packet basis; rather a stream of messages as a whole must be examined and translated. Some protocols map easily to each other. Others do not. Some protocols have no valid translation at all.

In the case of DIS to HLA translation, or even FOM-to-FOM translation, there is not always a one-to-one mapping between PDUs received from the DIS side and updates sent to the HLA side (or vice versa). This is due to fundamental differences in the rules and conventions of HLA versus DIS, and to differences in the ways different FOMs choose to represent similar concepts. For example, the DIS broker needs to send DIS heartbeat PDUs every 5 seconds, even though it may not receive HLA updates for an object for a much longer period of time. A single DIS PDU might contain data that must be translated into a series of attribute updates. For example, in DIS, a single Electromagnetic Emission PDU might contain information about a series of emitter systems and beams, but in HLA's RPR FOM, each emitter system and beam exists as a separate object, and must be updated through separate streams of messages.

In general, the strategy that VR-Exchange uses is to decode incoming data into state repositories that always contain the current state of each object. State repository information is passed to the other side of the bridge each frame, independently of when actual updates might have arrived. Then, the publishing rules of the other protocol and/or FOM are used to determine when to send out data describing the object. A consequence of this strategy is that no one piece of code is ever looking at both an incoming message and an outgoing message at the same time. While this strategy is necessary to support FOM agility and generic protocol-translation in VR-Exchange, a limitation is that “something” can get lost in the translation. For example, you should not expect that the timestamp on an outgoing HLA update will match the timestamp of a particular incoming DIS PDU.

Translators are very valuable solutions to real-world interoperabilty problems, and they often offer a much cheaper and easier solution than re-tooling applications to use a common protocol. But they can never quite provide the same level of close-coupling that you can get if all of your systems communicated natively. MÄK works hard to insure that our translators retain as much of the meaning and semantics of the data as is feasible, given the fundamental differences in the various protocols. But users must be aware that, as with any type of translator, there will always be limitations.



2

Installing VR-Exchange

This chapter explains how to install VR-Exchange and associated software.

2.1. Installing VR-Exchange

This section explains how to install VR-Exchange on a Windows or a UNIX operating system. You must also install the license manager files. (For details, please see "Installing and Setting Up the License Manager," on page 2-5.) For HLA, you must install an RTI. (For details, please see "Installing an RTI," on page 2-9.)

2.1.1. Installing VR-Exchange on Windows

Before you install VR-Exchange, please read *VR-Exchange Release Notes* to see if there are any special instructions for installation.

You must have administrator privileges to install MÄK products on Windows Vista.

Installing VR-Exchange from Optical Media

MÄK products are supplied on CDs or DVDs with a common installation interface.

To install VR-Exchange:

1. Insert the MÄK Products disc into your optical media drive. If the autorun feature on your computer is enabled, the MÄK PC Products Installation window opens. The PC Products Installation window is similar to Figure 2-1.

   If the MÄK PC Products Installation window does not open, open the Windows Explorer. In the root directory for your optical media drive, double-click *makInst.exe*. The MÄK PC Products Installation window opens.

2. In the MÄK PC Products Installation window, click VR-Exchange.
3. Follow the instructions of the installation program.

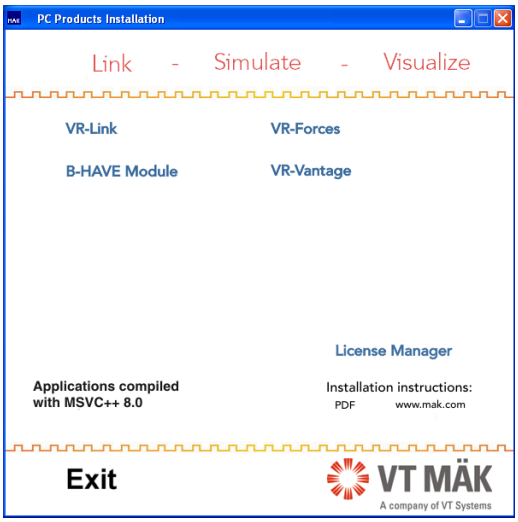


Figure 2-1. MÄK PC Products Installation window

Installing VR-Exchange from a Downloaded File

If you downloaded VR-Exchange, you received a compressed file. It may be a zip file or an executable file. (It is your responsibility to obtain an application that can unzip zip files.)

To install VR-Exchange from a zip file:

1. Copy the zip file to a temporary directory.
2. Unzip the zip file.
3. Run *setup.exe*.
4. Follow the instructions of the installation program.

➤ To install VR-Exchange from an executable file, run the executable.

2.1.2. Installing VR-Exchange on a Linux System

Before you install VR-Exchange, please read *VR-Exchange Release Notes* to see if there are any special instructions for installation.

Installing VR-Exchange from Optical Media

To install VR-Exchange:

1. Mount the optical drive.
2. Insert the MÄK Products disc into your optical media drive.
3. Run:

   ```
   ./install
   ```

4. Follow the on-screen instructions. You can install VR-Exchange into any directory for which you have write permissions.

Installing VR-Exchange from a Downloaded File

If you downloaded VR-Exchange, you received a compressed tar file.

To install VR-Exchange from a tar file:

1. Create the directory in which you want to install VR-Exchange.
2. Copy the tar file to the install directory.
3. Uncompress and untar the tar file:

   `gtar xzf vrexchange`*`xx-os`*`.tar.gz`

   where *`xx-os`* is the release, operating system, and other product release coding.

You must use gtar (from GNU) not the operating system's default tar.

2.2. Installing and Setting Up the License Manager

Before you can use a MÄK product, you must obtain a valid license file, install the License Manager, and configure the license server and client machines. The License Manager uses a client-server architecture, so you do not need to install the License Manager on every computer on which you install MÄK products. You only need to install it on the computer that you will use as the license server.

If you have already installed the License Manager for another MÄK product, you do not have to install it again. You just need to make sure you have licenses for your newly installed products.

The License Manager installer is included on MÄK installation media. It is separate from the product installers. The installers are also available for download at:

- Windows: ftp://ftp.mak.com/out/MAKLicenseManager-win-setup.exe
- Linux: ftp://ftp.mak.com/out/MAKLicenseManager-linux-setup.tar.gz

Complete installation and configuration instructions are included with the license manager installer. Instructions are also available at:
http://www.mak.com/products/install_license.php

Some customers use dongle licenses instead of running a license server. Instructions for using dongles are in the license manager documentation.

2.2.1. Specifying the License Server

The first time you run a MÄK application on a particular computer, the License Setup dialog box opens (Figure 2-2). It prompts you to enter the hostname of the license server and optionally, a port number.

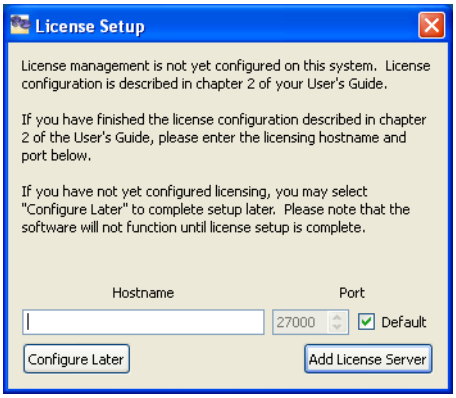


Figure 2-2. License Setup dialog box

If you do not know the hostname of the license server, click Configure Later. When you have the hostname, you can start the application again and complete the dialog box. You will not be able to run any MÄK applications until you set up license management.

If you know the hostname, type it in the Hostname box. Then click Add License Server. The application will start.

- If you are running MÄK products on the license server machine, it is also a client, so you must specify the license server on that machine too.
- If you change the license server, the saved configuration will no longer be valid and the License Setup dialog box will open the next time you start a MÄK application.
- You can clear the saved license configuration by deleting the cache file. On Windows it is *C:\Documents and Settings\user_name\Application Data\MAK\licenses<n>.xml*. On UNIX, it is *.mak/licenses<n>.xml*. (There may be more than one cache file, for example, *licenses1.xml* and *licenses2.xml*.)

The MAKLMGRD_LICENSE_FILE Environment Variable

As an alternative to using the License Setup procedure described in the previous section, you can configure the license server in an environment variable. The MAKLMGRD_LICENSE_FILE environment variable identifies the server machine. If you set this environment variable, it overrides the settings stored by the License Setup procedure.

To view a video that illustrates this procedure on Windows, please go to http://www.mak.com/html/setenvvar.htm

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 products run on.

Windows

To add the MAKLMGRD_LICENSE_FILE in Windows:

1. Open the System Properties dialog box.

   Windows XP:

   a. In Windows XP, on the Start menu, choose My Computer.

   b. In the My Computer window, under System Tasks, click View System Information. The System Properties dialog box opens.

   Windows Vista or Windows 7:

   a. In Windows Vista, open the Control Panel. On Vista, select System and Maintenance.

   b. Click System.

   c. Click Advanced System Settings. The System Properties dialog box opens.

2. Click the Advanced tab.
3. Click the Environment Variables button. The Environment Variables dialog box opens.
4. Click the New button. The New System Variable dialog box opens.
5. In the Variable Name field, enter MAKLMGRD_LICENSE_FILE.
6. In the Variable Value field, enter `@server_name`, where `server_name` is the name of the license server.
7. Click OK to back out of each dialog box and set the variable.

Linux

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

```
setenv MAKLMGRD_LICENSE_FILE @oak
```

If you are using the sh or bash shells, you set environment variables in your *.profile* file (or *.bashrc*). Set the variable similarly to the following example:

```
MAKLMGRD_LICENSE_FILE=@oak
export MAKLMGRD_LICENSE_FILE
```

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

You are ready to run the license server and use your new licenses or MÄK products.

2.3. Installing an RTI

An RTI 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.

For the most recent information about the RTI versions supported by MÄK products, please see the release notes for your MÄK application or the Product Versions page on the MÄK web site at: http://www.mak.com/product_version.php.

2.3.1. Installing the MÄK RTI

To install the MÄK RTI, follow the instructions in Chapter 2 of *MÄK RTI Reference Manual*.

Configuring Your System to Use the MÄK RTI

The RTI dynamic libraries must be located somewhere on your dynamic library search path. The path is specified in the PATH environment variable.The path is indicated by an environment variable as follows:

- Linux — LD_LIBRARY_PATH
- Windows — PATH.

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

- The federation configuration file (*.fed*, *.fdd*, or *.xml*) 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

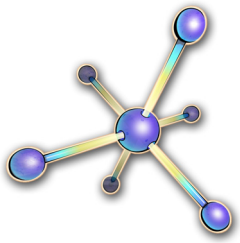
In many cases, you do not need an RTI server, such as the rtiexec, or a central application to use the MÄK RTI, however you can run the rtiexec if you want to. (It is required to use certain features of the MÄK RTI.) Be sure the FLEXlm license server is running, then start the applications, and they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

2.4. Installing TENA Software

If you plan to use the TENA broker, you must install the TENA Middleware and Object Model dynamic libraries. MÄK cannot distribute the majority of these files. They are under the control of the TENA SDA and you must get them from www.tena-sda.org. The software required to participate in a TENA execution is categorized as follows:

- TENA Middleware is the basic software required to run a TENA execution. It includes the Network Naming Service, Execution Manager, and some of the required dynamic libraries.
- The MÄK Object Model includes some of the dynamic libraries required to publish and subscribe to TENA messages and objects from the Common Object LROM and the MÄK LROM.
- The MÄK Object Model Basic Implementation is a set of dynamic libraries that is traditionally built by the end-user using code downloaded from the MÄK Object Model. These files are included in the installation of VR-Exchange.

For details about installing TENA software and the MÄK Object Model, please see Appendix B, *A VR-Exchange/TENA Primer*.



3

Using VR-Exchange

This chapter explains how to start the Portal and use it to manage connections.

3.1. Starting VR-Exchange

This section explains how to start VR-Exchange on the different operating systems it supports.

! All VR-Exchange components (the Portal and brokers) must run on the same computer.

3.1.1. Starting VR-Exchange on Linux

To start VR-Exchange on Linux:

1. In a command shell, change directory to *./bin*.
2. Enter:

   `vrx [command-line_options]`

The VR-Exchange Portal window opens (Figure 3-1). The VR-Exchange command-line options are described in "VR-Exchange Command-Line Options," on page 3-3.

3.1.2. Starting VR-Exchange on Windows

➤ To start VR-Exchange on Windows, on the Start menu, choose **Programs** → **MÄK Technologies** → **VR-Exchange x.x** → **VR-Exchange**.

The VR-Exchange Portal window opens (Figure 3-1).

If you want to start VR-Exchange with command-line options, start it from a command prompt or create a shortcut icon for it on your desktop. The VR-Exchange command-line options are described in "VR-Exchange Command-Line Options," on page 3-3.

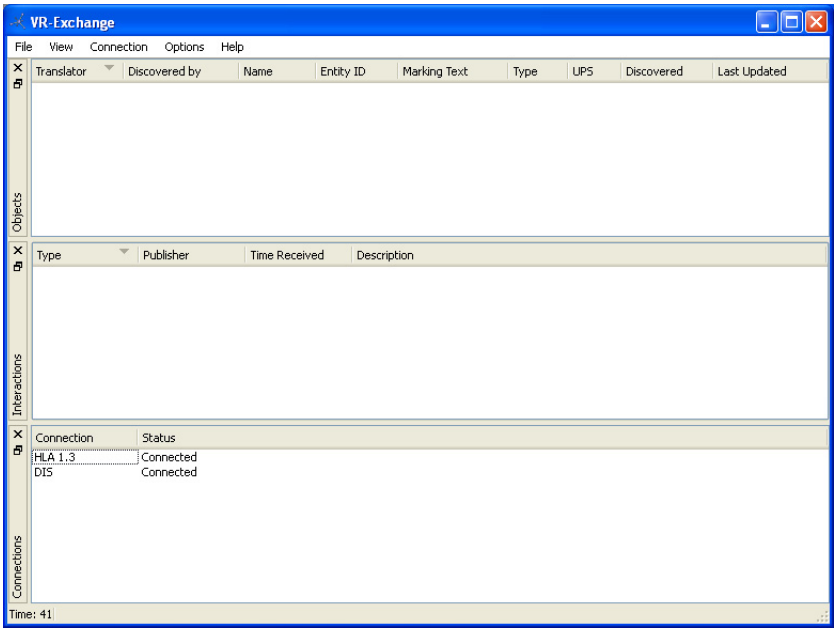


Figure 3-1. VR-Exchange Portal window

3.1.3. VR-Exchange Command-Line Options

The VR-Exchange command-line options override settings in *VR-Exchange.xml*. Table 3-1 describes the command-line options

Table 3-1: VR-Exchange Portal command-line options

Option	Description
`(--` | `--ignore_rest)`	Tells VR-Exchange to ignore the arguments following this flag.
`--controlQueueName` *queuename*	Specifies the name of the control shared memory queue.
`--dontStartBrokers`	Specifies that VR-Exchange not enable the connections listed in *VR-Exchange.xml*. If you use this option, you must start the connections manually.
`(-G` | `--noGui)`	Specifies console mode (no GUI).
`(-h` | `--help)`	Displays usage information and exits.
`--interactionQueue-Name` *queuename*	Specifies the name of the interaction shared memory queue.
`(-l` | `--configFile)` *filename*	Specifies a configuration file other than *VR-Exchange.xml*.
`(-L` | `--logFilename)` *filename*	Specifies a log file.

Table 3-1: VR-Exchange Portal command-line options

Option	Description
`--maxInteractionsTo-Show` *count*	Specifies the maximum number of interactions to show in the Portal application.
`--maxPortalControlMes-sages` *count*	Specifies the maximum number of control messages to process per tick.
`--maxPortalInterac-tionMessages` *count*	Specifies the maximum number of interaction messages to process per tick.
`--maxPortalObjectMes-sages` *count*	Specifies the maximum number of object messages to process per tick.
`(-n | --notifyLevel)` *notify_level*	Specifies the notification level for logging messages. Range: 0-4, where: ◆ 0 – Fatal error messages ◆ 1 – Warning messages ◆ 2 – Informational messages ◆ 3 – Verbose information ◆ 4 – Debugging data
`--objectQueueName` *queuename*	Specifies the name of the object shared memory queue.
`--queueBucketCount` *count*	Specifies the number of buckets inside the shared memory queue.
`--queueBucketPayload-Size` *size*	Specifies the size of buckets inside the shared memory queue.
`--tcpPort` *port_number*	Specifies the TCP port for connectivity tests.
`(-v | --version)`	Displays version information and exits.

3.1.4. The Portal Application

The Portal application houses several dockable windows. You can drag any of them off of the Portal and place them anyplace on your desktop. You can close any of the windows and open them to suit your viewing preferences.

The Portal has context-sensitive menus. The windows have a context-sensitive menu that matches the View menu. The individual connections in the Connections window have context-sensitive menus that match the Connection menu.

3.1.5. Running VR-Exchange without the Portal Application

You can run VR-Exchange without the Portal application. It automatically starts any connections that are configured to start when the Portal starts.

To run VR-Exchange without the Portal application:

1. Open *./bin/VR-Exchange.xml* in a text editor.
2. Set the value of the **noGui** attribute to true.

```
<!--Disable Portal GUI-->
<noGui value="true"/>
```

3. Save the file.

3.2. Configuring VR-Exchange

VR-Exchange has several parameters that you can set to optimize performance. Preferences are saved to *VR-Exchange.xml*.

To set VR-Exchange preferences:

1. Choose **Options** → **Preferences**. The VR-Exchange Preferences dialog box opens (Figure 3-2).

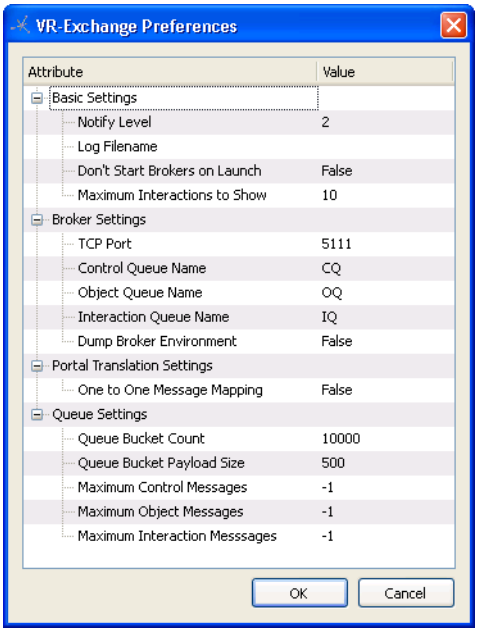


Figure 3-2. VR-Exchange Preferences dialog box

2. Expand the appropriate group of settings.
3. Click the value for the attribute that you want to change. The value becomes editable.

4. Edit the value. Some values can be chosen from a drop-down list. Others are text or numeric values that you edit directly.
5. Click OK. Changes do not take effect until you restart VR-Exchange.

Table 3-2 describes the attributes in the VR-Exchange Preferences dialog box.

Table 3-2: VR-Exchange preferences

Attribute	Description
Basic Settings	
Notify Level	Specifies the output level for the Portal's log file. Default: 2.
Log Filename	Specifies the log file for messages.
Maximum Interactions to Show	Specifies the maximum number of interactions to show in the Interactions window for each interaction type. When the Portal runs for a long time, the interaction list can become very long, which can degrade performance. If this number is set to -1 all interactions are shown. If it is set to any positive number, only the most recent maximum number of interactions is displayed. Default: 10.
Don't Start Brokers on Launch	Specifies that connections should not be enabled automatically when VR-Exchange starts.
Broker Communication Settings	
TCP Port	Specifies the TCP port to use to test broker lifetime. Each broker connects to this port. If the Portal crashes, the brokers will know and then shut down. If TCP Port is set to 0, this feature is disabled.
Object Queue Name Control Queue Name Interaction Queue Name	Specifies the names of the object, control, and interaction queues in the shared memory pool. If you run more than one instance of VR-Exchange, the queue names must be different for each Portal.
Dump Broker Environment	Print the brokers environment to a file at startup.
Portal Translation Settings	
One to One Message Mapping	Specifies whether or not to generate an outbound message for each inbound message. Changing this setting requires restarting all brokers.
Queue Settings	
Queue Bucket Count	Specifies the maximum number of messages in the shared memory queue.

Table 3-2: VR-Exchange preferences

Attribute	Description
Queue Bucket Payload Size	Specifies the maximum message size, in bytes.
Maximum Control Messages Maximum Interaction Messages Maximum Object Messages	Specifies the maximum number of Portal messages to process per call to drainInput(), where -1 means process all messages. Adjust these parameters to avoid process starvation if one of the queues is heavily used.

3.3. Enabling and Disabling Connections

The brokers are executable files that are independent of the Portal. You start them using connection configurations that specify a broker, a connection name, and configuration parameters. You can enable a connection automatically when the Portal is started or you can enable a connection manually. For details about adding and configuring connections, please see Chapter 4, *Working with Connections*.

! The TENA broker will not run if TENA is not installed. If TENA is not installed, you will be able to add a connection that specifies the TENA broker, but you will not be able to configure the connection or enable it. If you want to use the TENA broker, it is your responsibility to ensure that all required TENA files are installed. For more information, please see Appendix B, *A VR-Exchange/TENA Primer*.

3.3.1. Enabling Connections when the Portal Starts

To specify that a connection be enabled when the Portal starts:

1. Start VR-Exchange.
2. In the Connections window (Figure 3-1), select the connection that you want to configure.
3. If the connection is enabled, choose **Connection** → **Disable Connection**. The connection is disabled.
4. Choose **Connection** → **Edit Connection**. The Edit Connection dialog box opens (Figure 3-3).

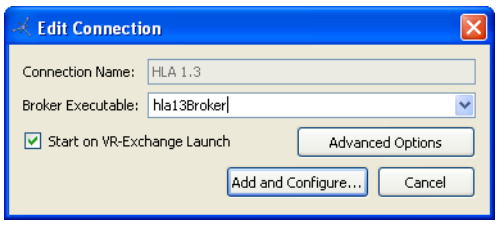


Figure 3-3. Edit Connection dialog box

5. Select the Start on VR-Exchange Launch check box.
6. Click Next.
7. Enable the connection, if applicable.

3.3.2. Enabling Connections During Runtime

To enable a connection after you start the Portal:

1. In the Connections window (Figure 3-1), select the connection that you want to enable.
2. Choose **Connection** → **Enable Connection**. The status changes to Connected.

3.3.3. Disabling Connections

Connections are disabled and brokers are shut down automatically when the Portal is shut down. Connections can also be disabled independently. If a broker crashes, or otherwise exits, and if the broker is connected using TCP, the Portal removes any connections that use it.

To disable a connection:

1. In the Connections window (Figure 3-1), select the connection you want to disconnect.
2. Choose **Connection** → **Disable Connection**.

3.4. Viewing Objects and Interactions

VR-Exchange displays lists of the objects and interactions that it discovers, which connection is publishing them, and statistics about them (Figure 3-4). As objects join and leave the exercise, VR-Exchange updates the display so that it only shows currently simulated objects. You can configure the number of interactions kept in the interaction list. (For details, please see "Configuring VR-Exchange," on page 3-5.)

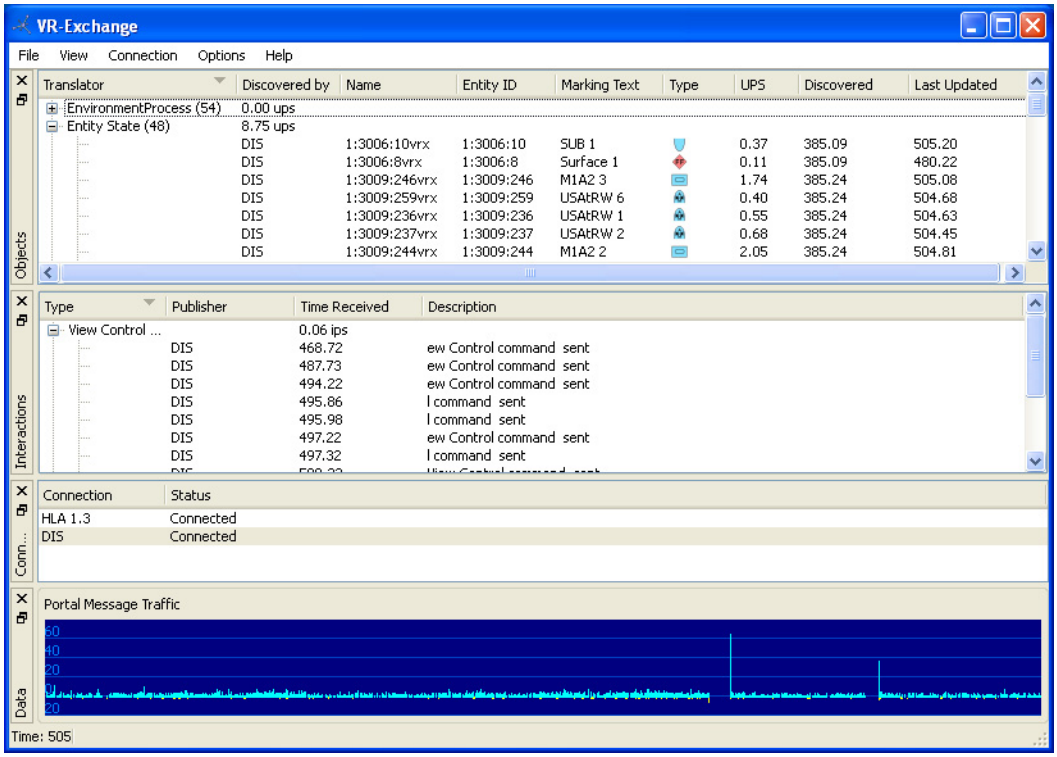


Figure 3-4. Object and interaction display

3.4.1. Customizing the Display of Object and Interaction Data

You can choose which columns of data to display and you can change the order in which columns are displayed.

To show or hide a column:

1. In the Objects window or Interactions window (Figure 3-4), right-click any column label. A list of columns is displayed.
2. To display a hidden column or hide a currently displayed column, click its entry in the list.

➤ To reorganize the order of data columns, drag the column title to the location where you want to display it.

3.4.2. Enabling and Disabling the Display of Object Data

You can enable and disable display of the following types of object data:

- Objects – show or hide all objects
- Updates – show or hide the Updated column.

➤ To enable or disable display of object data, choose **View** → **Objects** → *option*, where *option* is one of the following:

– Show Objects

– Show Updates.

3.4.3. Sorting the Object List by Column

➤ To sort the object list by column, click a column heading.

3.4.4. Enabling and Disabling the Display of Interactions

You can hide or display the interaction list.

➤ To enable or disable the display of interactions, choose **View** → **Interactions** → **Show Interactions**.

3.4.5. Clearing the Interactions List

You can clear the Interactions list. When you clear the interactions list, VR-Exchange continues to maintain a running count of the number of interactions. Figure 3-5 illustrates the interactions list after it is cleared. Notice the cumulative statistics next to Fire and Detonation.

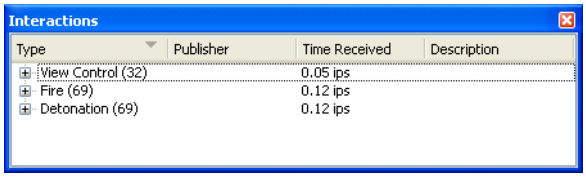


Figure 3-5. Cleared Interactions list

➤ To clear the Interactions list, choose **View** → **Interactions** → **Clear Interactions**.

Resetting Statistics

You can clear the cumulative statistics maintained by the Interactions window.

➤ To clear interaction statistics, choose **View** → **Interactions** → **Reset Statistics**.

3.5. Filtering Objects

You can filter out entire classes of objects (for details, please see "Specifying Which Translators to Use," on page 4-9). You can also filter individual objects. You might do this if you are interested in particular objects and want to reduce network traffic by filtering out the objects you are not interested in.

To filter out an object:

1. In the Objects window, select the objects that you want to filter.
2. Right-click the objects. A context-sensitive menu is displayed.
3. Choose **Filter Selected**. The word "filtered" is displayed in the Translators column for each filtered object (Figure 3-6).

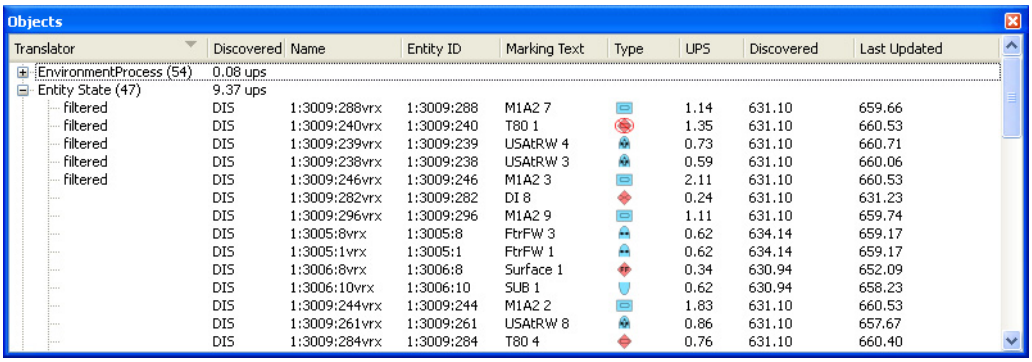


Figure 3-6. Filtered objects

To unfilter an object:

1. In the Objects window, select the objects that you want to unfilter.
2. Right-click the objects and on the popup menu, choose **Unfilter Selected**.

3.6. Displaying the Queue Status

You can display details about the shared memory queue.

➤ To display the queue status, choose **View** → **Queue Status**. The Queue Status dialog box opens (Figure 3-7).

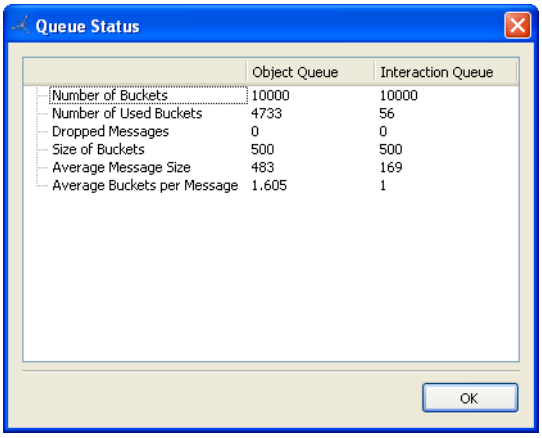


Figure 3-7. Queue Status dialog box

3.7. The Data View

You can display a histogram of the number of packets being processed by VR-Exchange (Figure 3-8). Blue lines (above the baseline) represent the number of object packets; yellow lines (below the baseline) represent the number of interaction packets; red lines represent VR-Exchange messages.

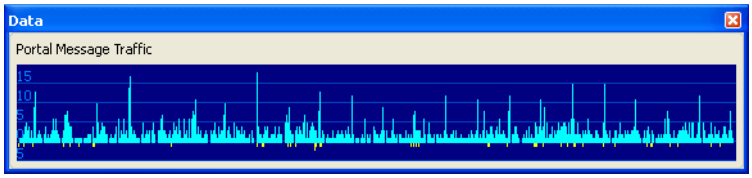


Figure 3-8. Data View

➤ To hide or display the Data View, choose **View** → **Show Data View**.

3.8. Closing VR-Exchange

➤ To close VR-Exchange, choose **File** → **Exit**.

3.9. Using vrxMessageDump

VR-Exchange includes a utility called *vrxMessageDump*. It displays all messages passing through shared memory in a console window (Figure 3-9). vrxMessageDump exits automatically when the Portal is shut down.

➤ To run *vrxMessageDump*, start VR-Exchange, then run the *vrxMessageDump* executable in the *./bin* directory.

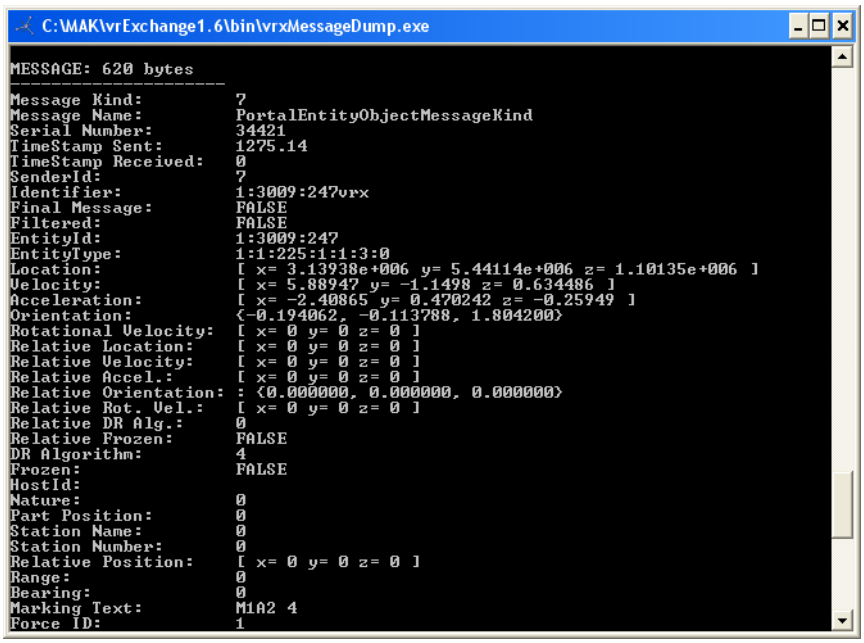
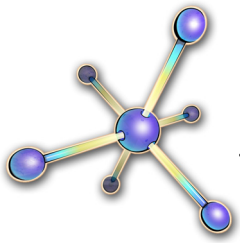


Figure 3-9. vrfMessageDump display



4

Working with Connections

This chapter explains how to add, edit, and configure connections.

4.1. Adding and Editing Connections

A connection runs a broker executable with a unique set of parameters. This allows you to, for example, run an instance of the HLA 1.3 broker with RTI A and another instance of the HLA 1.3 broker with RTI B.

A connection must have a name and it must specify the broker that it uses and a configuration file. It can include command-line options and can specify environment variables. For lists of command-line options, please see Appendix A, *Broker Details*.

VR-Exchange ships with connections for HLA 1.3, DIS, and TENA. The HLA and DIS connections are started automatically when you start VR-Exchange. The TENA connection is not displayed unless you have TENA installed.

! The TENA broker will not run if TENA is not installed. If TENA is not installed, you will be able to add a connection that specifies the TENA broker, but you will not be able to configure the connection or enable it. If you want to use the TENA broker, it is your responsibility to ensure that all required TENA files are installed. For more information, please see Appendix B, *A VR-Exchange/TENA Primer*.

4.1.1. Adding Connections

To add a connection:

1. Choose **Connection** → **Add New Connection**. The Add New Connection dialog box opens (Figure 4-1).

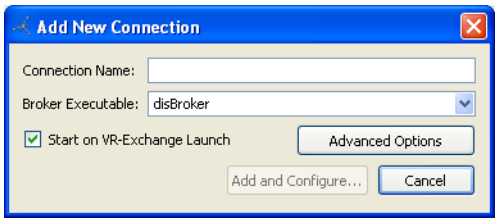


Figure 4-1. Add New Connection dialog box

2. In the Connection Name box, type a name for the connection.
3. In the Broker Executable box, select a broker from the drop-down list or type the name of a broker executable.
4. Optionally, specify advanced options, as follows:
   a. Click Advanced Options. The dialog box expands (Figure 4-2).

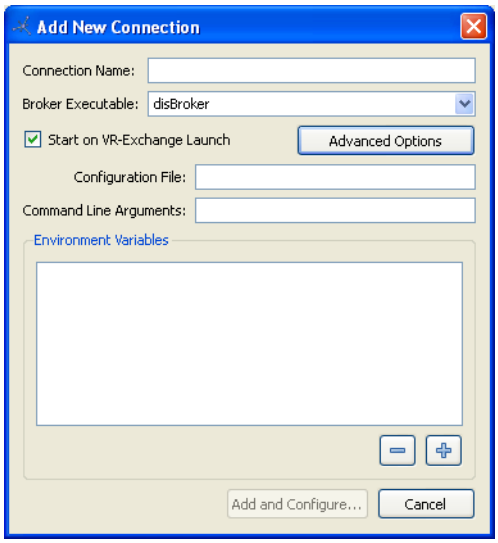


Figure 4-2. Add New Connection dialog box, Advanced Options

b. In the Configuration File box, type a name for the connection configuration file. (VR-Exchange will append the extension *.bcf*.) If you do not specify a configuration file name, VR-Exchange uses the connection name to name the configuration file.

c. Optionally, type command-line arguments in the Command Line Arguments box.

d. Optionally, specify environment variables for this connection, as described in "Specifying Environment Variables for a Connection," on page 4-4.

e. Click Advanced Options to collapse the dialog box.

5. If you want to enable the connection when VR-Exchange starts, select the Start on VR-Exchange Launch check box.

6. Click Add and Configure. The Connection Information dialog box for this connection opens.

7. Optionally, edit configuration parameters for the new connection. For details, please see "Configuring Connections," on page 4-5.

4.1.2. Specifying Environment Variables for a Connection

You can specify environment variables that only apply to a specific connection. You might want to do this to have different HLA connections use different RTIs.

To specify an environment variable for a connection:

1. In the Add New Connection dialog box or Edit Connection dialog box, click Advanced Options. The dialog box expands (Figure 4-2).
2. In the Environment Variables group box, click the Add button (+). The Environment Variable dialog box opens.
3. In the Variable Name box, type a name for the environment variable.
4. In the Variable Value box, type the value for the variable.
5. Click OK. The environment variable is added to the list.
6. Click Next.

4.1.3. Editing Connections

To edit a connection:

1. In the Connections window (Figure 3-1), select the connection that you want to edit.
2. If the connection is connected, choose **Connection** → **Disable Connection**.
3. Choose **Connection** → **Edit Connection**. The Edit Connection dialog box opens (Figure 4-3).

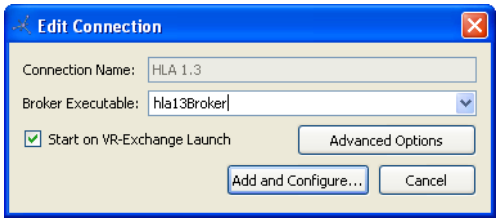


Figure 4-3. Edit Connection dialog box

4. Click Advanced Options to display all connection details.
5. Edit the connection as desired.
6. Click Next.

4.1.4. Deleting Connections

Deleting a connection deletes the association of a connection name with a broker and a configuration file. It does not affect the broker executable referenced by the configuration file and it does not affect the configuration file it references (if one exists).

To delete a connection:

1. In the Connections window (Figure 3-1), select the connection that you want to delete.
2. If the connection is connected, choose **Connection** → **Disable Connection**.
3. Choose **Connection** → **Remove Connection**.
4. Click OK. A confirmation dialog box opens.
5. Click OK.

4.2. Configuring Connections

Each connection has a set of attributes that you can set. If you use the default attribute values, VR-Exchange does not create a configuration file for the connection. If you change any of the attributes, VR-Exchange saves them to the configuration file you specified when you created the connection.

To configure a connection:

1. In the Connections window (Figure 3-1), select the connection that you want to configure.
2. Choose **Connection** → **Edit Connection Information**. The Connection Information dialog box opens (Figure 4-4). (When you add a new connection, the Connection Information dialog box opens automatically after you create the connection.)

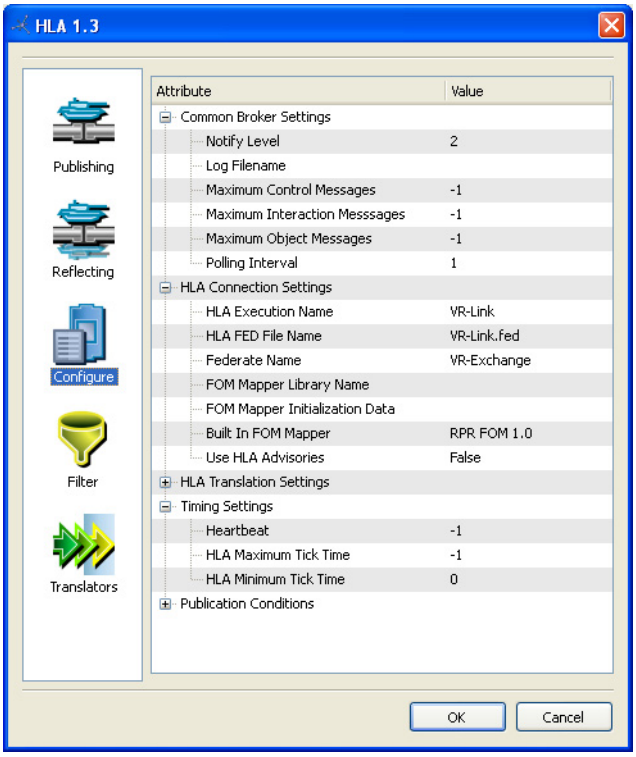


Figure 4-4. Connection Information dialog box

3. Select the Configure page. It lists all of the attributes for the connection, organized in logical groupings.
4. Expand the attribute tree to find the attributes you want to edit.
5. To edit an attribute, click its value. The field becomes editable.
   - If the field has a single text or numeric value, change the value as desired. Some values are available on drop-down lists. The attributes are listed and described in later sections in this chapter.
   - If the attribute value is a collection, edit it as described in "Specifying Collection Values," on page 4-7.
6. Click OK. If the connection was enabled, the changes do not take effect until you restart the connection. VR-Exchange prompts you to restart the connection.

4.2.1. Specifying Collection Values

Some attributes can take multiple values (collections).

To specify the values for a collection:

1. In the Configure page for a connection, expand the attribute list to display the collection attribute you want to edit.
2. Click the word Collection in the Value column (Figure 4-5). It changes to a button labeled Click to Edit.

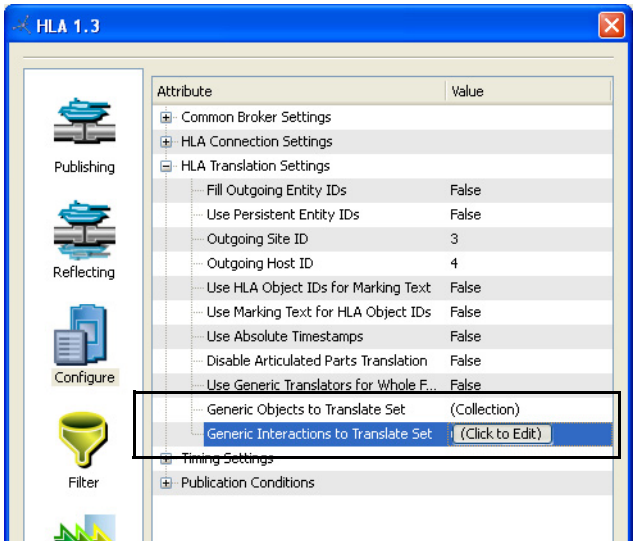


Figure 4-5. Collection value

3. Click the button. The Edit Collection dialog box opens (Figure 4-6).

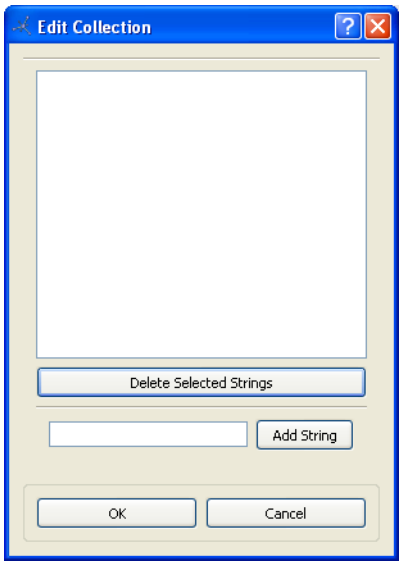


Figure 4-6. Edit Collection dialog box

4. In the text box, type the string you want to add.
5. Click Add String. The string is added to the list window.
6. Click OK.

Removing Values from a Collection

To remove values from a collection:

1. In the Configure page for a connection, click the word Collection in the Value column (Figure 4-5). It changes to the text Click to Edit.
2. Click the button. The Edit Collection dialog box opens (Figure 4-6).
3. In the list window, select the strings that you want to delete.
4. Click Delete Selected Strings.
5. Click OK.

4.2.2. Specifying Which Translators to Use

Each broker has a set of translators. You can specify which translators a connection should use during a session.

To specify which translators a connection should use:

1. In the Connections window (Figure 3-1), select the connection that you want to configure.
2. Choose **Connection** → **Edit Connection Information**. The Connection Information dialog box opens.
3. Select the Translators page (Figure 4-7).

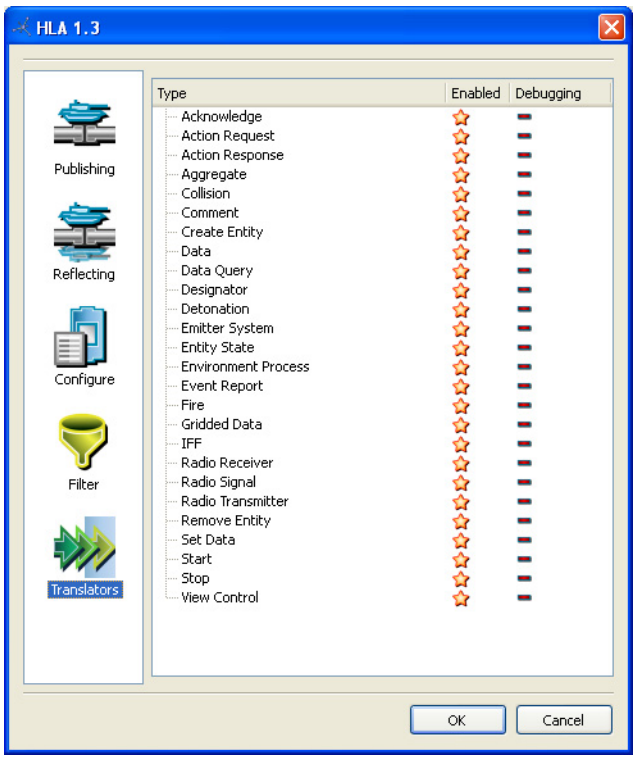


Figure 4-7. Translators page

4. In the Enabled column, select the check box for each translator that you want to use. Clear the check box for each translator that you do not want to use.
5. In the Debugging column, select the check box for a translator if you want VR-Exchange to send debugging messages for this translator. If debugging is enabled, the notify level must be at least 2.
6. Click OK.

4.2.3. Filtering Objects by ID

You can filter out objects by their ID. The filtering mechanism supports use of wildcards.

To filter an object by its ID:

1. Select the broker connection on which you want to configure a filter.
2. Choose **Connection** → **Edit Connection Information**. The Connection Information dialog box opens.
3. Select the Filter page (Figure 4-8).

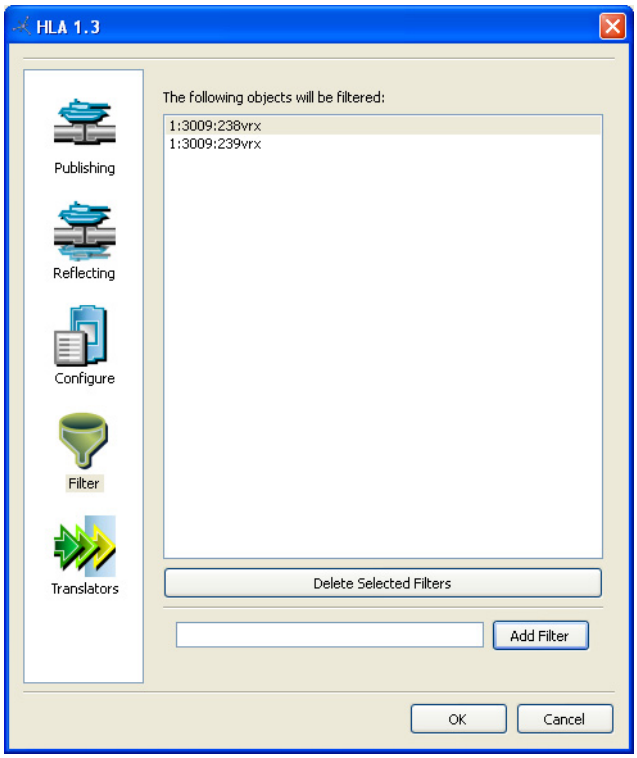


Figure 4-8. Filter page

4. In the text box, type the ID for the object you want to filter. You can use wildcards such as *, ?, and [abcd] to define a generalized ID.
5. Click Add Filter. The new filter is added to the list.

Deleting an Object Filter

To delete an object filter:

1. Select the broker connection from which you want to remove a filter.
2. Choose **Connection** → **Edit Connection Information**. The Connection Information dialog box opens.
3. Select the Filter page (Figure 4-8).
4. Select the filter that you want to delete.
5. Click Delete Selected Filters.

4.2.4. Configuring Publication Conditions

By default, VR-Exchange does not translate certain HLA objects until certain conditions are met. For example, an entity is not translated until its entity ID is a non-zero value. This is done because various mappings exist between DIS and HLA identifiers. However, there are situations in which IDs and other attributes are not specified. Turning off publication conditions helps allows you to translate these objects.

You can set discovery criteria for the following objects:

- Aggregate
- Designator
- Emitter System
- Entity State
- Environment Process
- Gridded Data
- IFF
- Radio Receiver
- Radio Transmitter.

To configure publication conditions:

1. Choose **Connection** → **Edit Connection Information**. The Connection Information dialog box opens.
2. Select the Configure page (Figure 4-9).

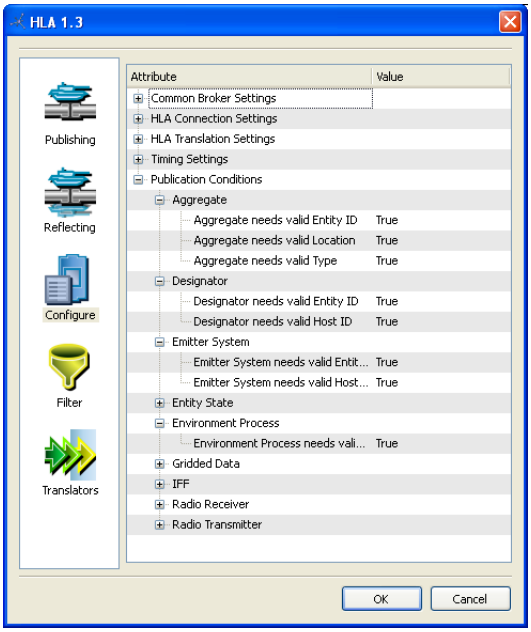


Figure 4-9. Publication conditions

3. Expand the Publication Conditions section.
4. Expand the hierarchy for the message type whose publication conditions you want to change.
5. Click the value to make it editable.
6. Select the value that you want.
7. Click OK.

4.2.5. Common Connection Attributes

Table 4-1 describes the connection attributes that are the same for all of the brokers.

Table 4-1: Common connection attributes (Sheet 1 of 2)

Attribute	Description
Common Broker Settings	
Notify Level	Specifies the logging output level for the broker's log file. Range 0-4, where: 0 = Fatal error messages 1 = Warning messages 2 = Informational messages 3 = Verbose information 4 = Debugging data Default: 2.
Log Filename	Specifies the name of the file to log broker output to. Default: "" (no log file).
Maximum Control Messages	Specifies the maximum number of Portal control messages to process per call to drainInput(). If set to -1, then all messages are processed. Default: -1.
Maximum Interaction Messages	Specifies the maximum number of Portal interaction messages to process per call to drainInput(). If set to -1, then all messages are processed. Default: -1.
Maximum Object Messages	Specifies the maximum number of Portal object messages to process per call to drainInput(). If set to -1, then all messages are processed. Default: -1.
Polling Interval	Specifies the interval, in seconds, between each call to drainInput() on the exercise connection. Default: 1.

Table 4-1: Common connection attributes (Sheet 2 of 2)

Attribute	Description
Publication Conditions	
	If a condition is set to True, an HLA object is not published until the condition is true. If a condition is set to false, the object will be published without satisfying this condition. The list of conditions is as follows: ◆ Aggregate: – Aggregate needs valid Entity ID – Aggregate needs valid Location – Aggregate needs valid Type ◆ Designator: – Designator needs valid Host ID – Designator needs valid Entity ID ◆ Emitter System: – Emitter System needs valid Entity ID – Emitter System needs valid Host ID ◆ Entity State: – Entity needs valid Entity ID – Entity needs valid Location – Entity needs valid Type ◆ Environment Process: – Environment Process needs valid Process ID ◆ Gridded Data: – Gridded Data needs valid Grid ID ◆ IFF: – IFF needs valid Entity ID – IFF needs valid Host ID ◆ Radio Receiver: – Radio Receiver needs valid Entity ID – Radio Receiver needs valid Host ID ◆ Radio Transmitter: – Radio Transmitter needs valid Entity ID – Radio Transmitter needs valid Host ID

4.2.6. HLA Connection Attributes

Table 4-2 describes the HLA-specific attributes that you can configure for connections that use the HLA broker.

Table 4-2: HLA connection attributes (Sheet 1 of 3)

Attribute	Description
HLA Connection Settings	
HLA Execution Name	Specifies the name of the HLA federation execution that this broker will join. Default: VR-Link.
HLA FED File Name	Specifies the name of the FED file being used for the federation execution. It must be the complete name including the three character extension (*.fed*). Default: VR-Link.fed.
Federate Name	Specifies the HLA Federate name. Default: VR-Exchange.
FOM Mapper Library Name	Specifies the name of a FOM mapper library, if you are using one. If you want to use one of the FOM Mappers supplied with VR-Exchange, leave this attribute blank and specify a FOM Mapper for the Built In FOM Mapper Attribute. Default: "".
FOM Mapper Initialization Data	Specifies a string that is passed to the FOM mapper library specified in FOM Mapper Library Name. This data is optional. Default: "".
Built In FOM Mapper	If the federation is using the RPR FOM, specifies the version, and for RPR FOM 2, draft 17, the revision. RPR FOM revision 2 for version 2.0017 fixes an encoding and decoding size issue with object ID lists to match RPR FOM 2 draft 17. The encoding and decoding issues affected the Aggregates, JammerBeams, and RadarBeams objects and the RemoveObjectRequest, AttributeChangeRequest, and ActionRequestToObject interactions. The built-in FOM Mappers are: ◆ RPR FOM 1.0 ◆ RPR FOM 2, draft 14 ◆ RPR FOM 2, draft 17, Revision 1 ◆ RPR FOM 2, draft 17, Revision 2. Default: RPR FOM 1.0.
Use HLA Advisories	Specifies whether the broker should use RTI advisory messages when determining what attributes to update. False = Send attribute updates regardless of whether or not someone has subscribed to the attribute with the RTI. True = Send attribute updates only if someone has subscribed to the attribute. Default: False.

Table 4-2: HLA connection attributes (Sheet 2 of 3)

Attribute	Description
HLA Translation Settings	
Fill Outgoing Entity IDs	Enables or disables filling out the entity ID (site:host:id) for all outgoing HLA entities. Use this parameter to spoof gateways that require DIS entity IDs for RPR FOM objects. Default: False.
Use Persistent Entity IDs	Specifies that entity IDs persist after disconnects. If this feature is enabled, when you replay an exercise, entities with common names will use the same entity IDs that they did in previous runs. To enable this feature, set Use Persistent Entity IDs to True. Default: False. You must also enable Fill Outgoing Entity IDs. This attribute saves entity IDs to the file *hlaPersistentEntityNumMap.csv*. Default: False.
Outgoing Site ID	The site ID to use when Fill Outgoing Entity IDs is true. Default: 3.
Outgoing Host ID	The host ID to use when Fill Outgoing Entity IDs is set to 1. Default: 4.
Use HLA Object IDs For Marking Text	When enabled, the broker uses the HLA object ID from reflected HLA objects to set the marking text when it sends information to the Portal. This is useful for mapping non-RPR FOM-based FOMs that do not have marking text to RPR FOM-based FOMs that might rely on it. Default: False.
Use Marking Text for HLA Object IDs	When enabled, the broker uses the marking text for objects it discovers in the Portal to name the HLA objects it is sending to its federation. If this parameter is enabled, HLA objects will not be published until a valid marking text is supplied. Default: False.
Use Absolute Time-stamps	Enables or disables use of absolute time stamping. Default: False.
Disable Articulated Parts Translation	Specifies whether or not to translate articulated parts. This can be used when your exercise has circular dependencies or otherwise non-standard configurations of articulated parts that might confuse the brokers.
Use Generic Translators for Whole FOM	Enables or disables use of generic translation for the entire FOM. Default: False.
Generic Objects to Translate Set	A list of HLA object classes to translate using the generic object translator. The generic object translator does not work with a FOM Mapper and the classes represented in this list must be exactly the same in all receiving FOMs. In cases where there are more than two brokers, only the two for which they are exactly the same should have this parameter set. (For information about generic translation, please see "Generic Translations," on page A-4.) Default: an empty list.

Table 4-2: HLA connection attributes (Sheet 3 of 3)

Attribute	Description
Generic Interactions to Translate Set	A list of HLA interaction classes to translate using the generic interaction translator. (For information about generic translation, please see "Generic Translations," on page A-4.) Default: an empty list.
Timing Settings	
Heartbeat	The frequency with which to send an update if there is no change. Default: -1 (means do not heartbeat).
HLA Maximum Tick Time	Specifies the maximum amount of time, in seconds, to tick the RTI before processing incoming messages. -1 = Tick the RTI until all messages are read. >0 = Tick the RTI at least HLA Minimum Tick Time, but no more than this value. Default: -1.
HLA Minimum Tick Time	Specifies the minimum amount of time, in seconds, to tick the RTI before processing incoming messages. Default: 0.0.

4.2.7. TENA Connection Attributes

Table 4-3 lists TENA-specific attributes that you can configure for a TENA connection. Attributes that take a list of translator names use the names listed in "TENA Translators," on page A-12.

Table 4-3: TENA connection attributes (Sheet 1 of 3)

Attribute	Description
TENA Connection Settings	
Execution Name	Specifies the name of the TENA execution this broker will join. The Execution Name value must match the TENA *executionManager.exe*'s `-executionname` parameter. Default: "VR-Link".
ORB Default Init Ref	Specifies the TENA ORB Initialization Reference. It must match the TENA *executionManager.exe*'s `-ORBdefaultInitRef` parameter. Example: corbaloc:iiop:10.0.1.250:50509 Default: "".
ORB Default Listen Endpoints	Specifies the TENA ORB Listen Endpoint. It must match the TENA *executionManager.exe*'s `-ORBlistenEndpoints` parameter. Example: iiop://10.0.1.250 Default: "".
TENA Transport Type	Specifies the transport type – best effort or reliable.
LROM Mapper Name	LROM Mapper DLL name. Default: *makLromMapper.dll*.
Multicast Time to Live	Specifies a multicast time-to-live value. Default: 1 (use the system default).
Multicast Interface	Specifies the IP address of the network device to use for sending PDUs. Default: 0.0.0.0 (the IP address of the primary network device).
TENA Translation Settings	
Parse Unknown IDs	When set, the broker will try to parse string identifiers into DIS IDs. Default: True.
Generate Fire PDU for Indirect Fire	If the broker receives a TENA detonation for which it does not have a corresponding Fire Interaction, it generates a Fire Interaction.
Fill Outgoing Entity IDs	Enables or disables filling out of the entity ID (site:host:id) for all outgoing TENA entities. Default: False.
Use Persistent Entity IDs	Specifies that entity IDs persist after disconnects. If this feature is enabled, when you replay an exercise, entities with common names will use the same entity IDs that they did in previous runs. To enable this feature, set Use Persistent Entity IDs to True. You must also enable Fill Outgoing Entity IDs. This attribute saves entity IDs to the file *hlaPersistentEntityNumMap.csv*. Default: False.

Table 4-3: TENA connection attributes (Sheet 2 of 3)

Attribute	Description
Site ID	Specifies the TENA site ID. Default: 1.
Application Number	The TENA application number. It is used for creating entity IDs for PDUs originating from the TENA broker. It must be unique. Default: 3338.
Use TENA Object ID for Marking Text	Enables or disables use of the TENA object ID from reflected TENA objects to set the marking text when it sends information to the Portal. Default: False.
Use Marking Text for TENA Object ID	Enables or disables use of the marking text for objects it discovers in the Portal to name the TENA objects it is sending to its execution. If enabled, TENA objects are not published until a valid marking text is supplied. Default: False.
Guise is Entity Type	Translate the TENA Guise attribute as the Entity Type when moving back and forth from DIS or HLA. Default: True.
Force Updates	Forces all objects to update on tick even if they have not changed at all. Default: False.
TENA AMO Settings	
AMO Name	Specifies the TENA broker's AMO name. Default: "".
AMO Hostname	Specifies the TENA broker's AMO hostname. Default: "".
AMO Host IP Address	Specifies the TENA broker's AMO hostIPaddress. Default: "".
AMO Operator Name	Specifies the TENA broker's AMO operatorName. Default: "".
AMO Operator Telephone Number	Specifies the TENA broker's AMO operatorTelephoneNumber. Default: "".
Debug Settings	
Debug Global ID Mappings	Enables or disables debug output for global ID mapping. Global ID mapping maps TENA names to Portal entity IDs. Notify Level must be set to at least 2 for this to work. Default: False.
Debug Entity ID Mappings	Enables or disables mapping of Portal entity IDs to TENA entity IDs in the debug output. Default: False.
Dump GUI to Text	Enables or disables logging of GUI updates to the log file. When enabled, the Portal outputs a line each time an update comes from the translator. This information is already displayed in the GUI, but using Dump GUI to Text makes a better record for debugging. Notify Level must be set to at least 2 for this to work. Default: False.

Table 4-3: TENA connection attributes (Sheet 3 of 3)

Attribute	Description
Timing Settings	
TENA Tick Time	Specifies the TENA exercise connection's tick time, that is, the time to spend within drainInput(). ! drainInput() will not exit early if there is no more input from the exercise. For more detail concerning this parameter, refer to VR-Link's TENA DtExerciseConn::drainInput() documentation. Default: 0.1 seconds
Object Throttle Rates	Specifies a list of entity types and update values. Limits the maximum number of updates per second for specific TENA object entity types. Default: 10.

4.2.8. DIS Connection Attributes

Table 4-4 lists DIS-specific attributes that you can configure for connections that use the DIS broker.

Table 4-4: DIS connection attributes (Sheet 1 of 3)

Parameter	Description
DIS Connection Settings	
Port	Specifies the port to receive network PDUs from and send them to. Default: 3000.
Exercise ID	Specifies the DIS exercise ID. Default: 1.
Destination Address	Specifies the address of outgoing DIS PDUs. If set to 0.0.0.0, the default broadcast address is used. Default: 0.0.0.0.
DIS Multicast Device	Specifies the IP address of the network device to use for sending DIS PDUs. Default: 0.0.0.0 (the IP address of the primary network device).
DIS Multicast Subscription Set	Specifies a list of multicast addresses the DIS broker should subscribe to for DIS communication. Default: an empty list.
Socket Send Buffer Size	Sets the socket's send buffer size. Default: 0.
Socket Receive Buffer Size	Sets the DIS socket's buffer received size. If you think the broker is dropping incoming packets during periods of heavy network traffic, increasing this value might help. A typical default value is 50000. If you set the size to zero, the DIS broker uses the default size for the operating system. Default: 0.

Table 4-4: DIS connection attributes (Sheet 2 of 3)

Parameter	Description
DIS Translation Settings	
Parse Unknown IDs	Tells the DIS Broker to try to parse a DIS ID out of a Portal Object ID if there is not already a known entity available. For example, if the Portal ID (a string) is "1:2:3", and if the DIS Broker does not already know about string "1:2:3", it tries to create a DIS ID from the string, in this case 1:2:3. Default: True.
Remap DIS IDs	Enables or disables mapping of all DIS entity IDs to match the site and host IDs specified in the configuration file. This corrects errors caused by rogue DIS objects in a DIS-to-HLA-to-DIS exercise. Default: False.
Use Persistent Entity IDs	Specifies that entity IDs persist after disconnects. If this feature is enabled, when you replay an exercise, entities with common names use the same entity IDs that they did in previous runs. To enable this feature, set Use Persistent Entity IDs to True. You must also enable Fill Outgoing Entity IDs. This attribute saves entity IDs to the file *disPersistentEntityNumMap.csv*. Default: False.
Application Number	The DIS application number. It is used for creating entity IDs for PDUs originating from the DIS broker. It must be unique. Default: 27.
Site ID	Specifies the DIS site ID. Default: 1.
Use Absolute Timestamps	Specifies the use of absolute timestamps on PDUs or interactions. False = Do not use absolute timestamps. True = Use absolute timestamps. Default: False.
DIS Protocol Version to Send	The DIS PDU version to send, where: 4 = DIS 2.0.4 5 = IEEE 1278.1 6 = IEEE 1278.1a Default: 5.
DIS Minimum Protocol Version to Receive	The minimum DIS PDU version to receive. Default: 4.
DIS Maximum Protocol Version to Receive	The maximum DIS PDU version to receive. Default: 6.
Disable Articulated Parts Translation	Specifies whether or not to translate articulated parts. This can be used when your exercise has circular dependencies or otherwise non-standard configurations of articulated parts that might confuse the brokers.

Table 4-4: DIS connection attributes (Sheet 3 of 3)

Parameter	Description
Debug Settings	
Debug Global ID Mappings	Enables or disables mapping of Portal object IDs to DIS entity IDs in the debug output. notifyLevel must be set to at least 2 for this to work. Default: False.
Debug Entity ID Mappings	Enables or disables mapping of Portal entity IDs to DIS entity IDs in the debug output. Default: False.
Debug Event ID Mappings	Enables or disables mapping of Portal event IDs to DIS event IDs in the debug output. Default: False.
Dump GUI to Text	Enables or disables logging of GUI updates to the log file. When enabled, the Portal outputs a line each time an update comes from the translator. This information is already displayed in the GUI, but using Dump GUI to Text makes a better record for debugging. Notify Level must be set to at least 2 for this to work. Default: False.
Timing Settings	
Heartbeat Interval	Specifies the number of seconds between heartbeats of published entities. Default: 5.0 seconds.
Timeout Interval	Specifies the number of seconds before a reflected entity times out and the DIS broker removes it. Default: 10.0 seconds.
Drain Timeout	Specifies the timeout value, in seconds, for reading PDUs from the network, as follows: -1 = No timeout. Poll network until all PDUs are read. >0 = Poll the network, reading the number of PDUs specified by Number of PDUs Read Per Tick, until all PDUs are read or the timeout value is reached, whichever comes first. Default: -1.
Number of PDUs Read Per Tick	Specifies the number of PDUs to read before checking to see if the amount of time specified by Drain Timeout has been reached. Default: 100.
Multicast Time-to-Live	Specifies a multicast time-to-live value. Default: 0 (use the system default).

4.3. Viewing Connection Information

You can view lists of the objects that a connection is publishing and reflecting. You can open information windows for multiple connections at the same time.

- The Publishing page displays the objects that the connection has discovered from the Portal and published to its simulation network.
- The Reflecting page displays the objects that the connection has discovered on its simulation network and published to the Portal.

To view connection information:

1. In the Connections window (Figure 3-1), select the connection whose details you want to display.
2. Choose **Connection** → **Edit Connection Information**. The Connection Information dialog box opens.
3. Select the Publishing page or the Reflecting page (Figure 4-10).

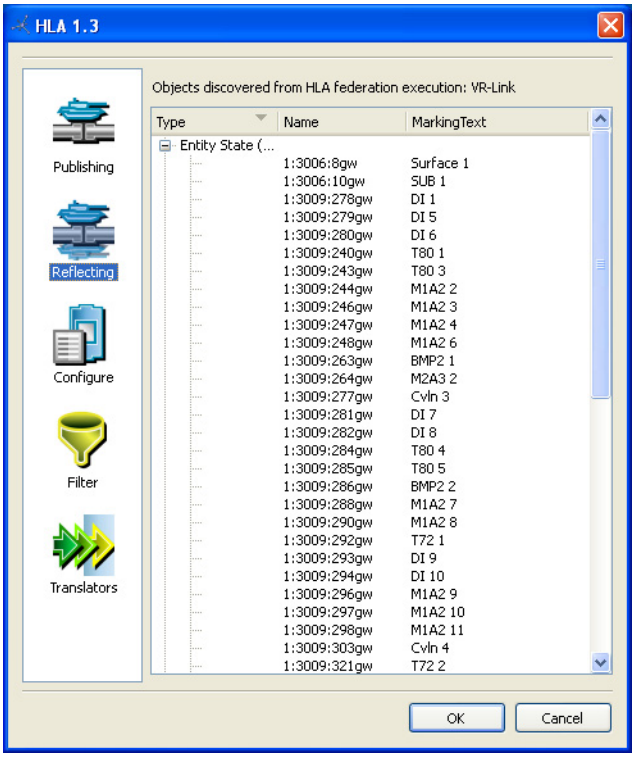
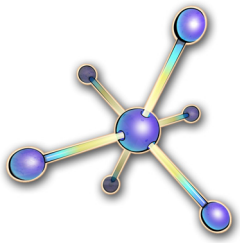


Figure 4-10. HLA connection Reflecting page



5

The VR-Exchange API

This chapter explains how to write brokers and have them interact with the Portal.

5.1. The VR-Exchange API

The VR-Exchange API has two primary components, the Broker API and the Portal API. The Broker API lets you create brokers for use with VR-Exchange or extend the brokers provided by MÄK. The Portal API allows brokers to send messages to, and receive updates from, the Portal. (Broker and Portal concepts are described in "The VR-Exchange Architecture," on page 1-2.)

All VR-Exchange API code is placed under the namespace MAKVrExchange. Each broker's API code is placed under its own namespace, as follows:

- HLA Broker API: MAKVrExchange::DtHlaBroker
- TENA Broker API: MAKVrExchange::DtTenaBroker
- DIS Broker API: MAKVrExchange::DtDisBroker.

5.1.1. The Portal Application

The portal is a separate application, or process, which is independent of the brokers. The Portal application executable is *vrx.exe*. It has the following functions:

- Create and manage the message queue
- Start and stop brokers (based on the connection configurations)
- Manage broker errors and lifetimes
- Displays basic shared memory queue information
- Display discovered and published objects and interactions
- Allow you to configure connections.

The display of message traffic through the GUI is a convenience for users. It is not required for operation of VR-Exchange. Therefore, if you add new broker types or messages, you do not have to rebuild the GUI.

5.1.2. Brokers

Brokers must be able to receive and send data in the exercises for which they are translating. Therefore, they behave much like HLA federates, DIS simulations, or TENA range resources. Brokers connect to an exercise the same way a VR-Link application would connect to TENA, DIS, or HLA (they do this using VR-Link libraries). Brokers publish objects and send interactions. While it is possible to write whole simulators that act as brokers, this is not recommended, because brokers must all run on the same machine to communicate and they require a VR-Exchange license to run. These conditions might limit their effectiveness as simulation applications.

Many of the terms and concepts in the Portal API, such as publishers, reflected objects, and callbacks are identical to or very similar to those in VR-Link. Please refer to VR-Link documentation for detailed discussions of these subjects.

5.2. The Portal Language

The Portal provides a framework for creating objects (which have state) and interactions (event messages) that pass through a distributed messaging system. VR-Exchange provides many object and interaction classes in the Portal language. The Portal language is defined in the *libPortal* library. Table 5-1 lists the objects and interactions supported by the Portal language.

Table 5-1: Object and interactions supported by the Portal language

Object/Interaction	*libPortal* Class
Acknowledge	*DtPortalAcknowledgeInteraction*
Action Request	*DtPortalActionRequestInteraction*
Action Response	*DtPortalActionResponseInteraction*
Aggregate	*DtPortalAggregate*
Application Specific	*DtPortalApplicationSpecificInteraction*
Collision	*DtPortalCollisionInteraction*
Comment	*DtPortalCommentInteraction*
Create Entity	*DtPortalCreateEntityInteraction*
Data	*DtPortalDataInteraction*
Data Query	*DtPortalDataQueryInteraction*
Designator	*DtPortalDesignatorObject*
Detonation	*DtPortalDetonationInteraction*
Emitter System	*DtPortalEmitterSystemObject*
Entity State	*DtPortalEntity*

Table 5-1: Object and interactions supported by the Portal language

Object/Interaction	***libPortal* Class**
Environment Process	*DtPortalEnvironmentProcessObject*
Event Report	*DtPortalEventReportInteraction*
Fire	*DtPortalFireInteraction*
Generic Interaction	*DtPortalGenericInteration*
Generic Object	*DtPortalGenericObject*
Gridded Data	*DtPortalGriddedDataObject*
IFF	*DtPortalIffObject*
Radio Receiver	*DtPortalRadioReceiverObject*
Radio Signal	*DtPortalRadioSignalInteraction*
Radio Transmitter	*DtPortalRadioTransmitterObject*
Remove Entity	*DtPortalRemoveEntityInteraction*
Set Data	*DtPortalSetDataInteraction*
Start	*DtPortalStartInteraction*
Stop	*DtPortalStopInteraction*
View Control	*DtPortalViewControlInteraction*

Objects and interactions have attributes that describe their state, such as location, type, velocity, and target. Each attribute is a unique data field transmitted in a protocol-independent way. That means attributes cannot always be directly copied into Portal messages. For a discussion of the limitations of translation software, please see "The Limitations of Translation Software," on page 1-16.

The Portal framework allows you to create new types of objects and interactions. Brokers can create, update, and delete objects, and send interactions to the Portal. They can also subscribe to the objects and interactions they want to receive from the Portal.

If you add a new object type you do not have to recompile all brokers, only those brokers that need to translate the new object type.

5.2.1. Translating Objects and Interactions to the Portal Language

The job of a broker is to translate object and interaction message attributes from the broker's simulation standard to the Portal language. The translation process involves the following:

- Managing object names
- Managing update times and rates
- Converting units
- Computing default values for unspecified attributes.

Each of these issues is highly protocol-dependent. The following sections provide a general description of how VR-Exchange's data exchange represents objects and attributes. For details, refer to the class documentation.

5.2.2. Identifying Objects

The Portal identifies each object by a string. The string must be unique, but this is not enforced by the Portal. Future versions of the Portal may choose to enforce this. For protocols that do not identify objects by a string, some form of conversion must be developed by the broker architect.

In the default HLA broker, objects are identified by their HLA object name.

5.2.3. Translating Attributes

The attribute representations used by the Portal language draw heavily on the data types used by the DIS protocol. For example, Entity Location is transmitted as three doubles: X, Y, and Z, using geocentric coordinates. This method of defining location was chosen because it is widely used by both DIS and the RPR FOM, as well as most RPR FOM variants. FOMs or protocols that use geodetic (latitude, longitude, and altitude) locations must convert to geocentric coordinates before updating location data in the Portal.

All enumeration attributes use values defined in SISO-REF-010-2005, the Enumerations document used for both DIS and the RPR-FOM.

The Portal uses a macro called DtDEC_ATTRIBUTE to declare attributes for objects and interactions. For example, an attribute called DamageState might be declared like this:

```
DtDEC_ATTRIBUTE(DtU32, DamageState, damageState);
```

This macro resets the access level to protected. It is recommended that classes group this macro usage in one place, then set the access level as needed.

This macro will create two public functions like this:

```
DtU32 damageState() const { return myDamageState;}
void setDamageState(const DtU32 &val) { myDamageState = val;}
```

and a protected variable called:

```
DtU32 myDamageState;
```

For more information about data types please refer to the class documentation.

5.3. The Portal API

Brokers exchange data with the Portal through the Portal API. Portal viewers, such as *vrxMessageDump* and the VR-Exchange Portal application use the Portal API to receive data.

5.3.1. Connecting to the Portal

Applications that want to receive messages from the Portal (brokers and Portal viewers) must create an instance of *DtPortalConnection* (in *pConnection.h*). This class opens a connection to the internal message queues and creates an instance of the message factory. To create a Portal connection do the following:

```
myPortalConnection = new DtPortalConnection(myBrokerName, myBrokerId,
  myControlQueueName, myInteractionQueueName, myObjectQueueName,
  myTcpPort);
```

The names of the Control, Interaction and Object queue are text strings that identify the shared memory queues to be used by the *DtPortalConnection* instance. If you want to run multiple copies of VR-Exchange on a single machine, each copy must use unique queue names. The queue names are set in the connection configuration files. The default values are:

- CQ, for the control queue
- IQ, for the interaction queue
- OQ, for the object queue.

Though brokers can connect to the Portal using this member function, it is much easier to use a *DtBroker* instance, which both creates the *DtPortalConnection* instance, and also registers callbacks to shut down the broker. For more information, please see "The Broker API," on page 5-15.

5.3.2. Receiving Interactions and Control Messages

A broker can receive three types of messages from the Portal: object updates, control messages, and interactions. Control messages (messages specific to the operation of VR-Exchange) and interactions are handled using callbacks. Object updates are handled using reflected objects (see "Reflecting Objects," on page 5-13).

Interaction messages derive from *DtPortalInteraction* (*pInteraction.h*). Control messages derive from *DtPortalControlMessage* (*pControlMessage.h*).

Object messages, control messages, and interaction messages have separate message queues. Control messages receive the highest priority.

To receive a message, a broker can subscribe to it via the message class. For example, a broker must subscribe to the *DtPortalBrokerShutdownRequestMessage*. This message is delivered to the broker when VR-Exchange exits. To subscribe to a message, do the following:

```
DtPortalBrokerShutdownRequestMessage::addCallback(myPortalConnection,
  shutdownRequest,0);
```

where shutdownRequest() is a function that shuts down the broker in an orderly way.

You can subscribe to interactions in much the same way:

```
DtPortalDetonationInteraction::addCallback(myPortalConnection,
  receiveDetonation, 0);
```

Please see the class documentation for details.

5.3.3. Sending Interactions and Control Messages

To send a control or interaction message to the Portal, create a message, fill it in, and then send it to the *DtPortalConnection*. For example, to send a detonation interaction, do the following:

```
DtPortalDetonationInteraction out;

// "in" is a structure containing detonation information
out.setAttackerId(in->attackerId());
out.setTargetId(in->targetId());
out.setMunitionId(in->munitionId());
out.setEventId(in->eventId().string());

myPortalConnection->send(out);
```

The syntax is the same for control messages.

5.3.4. Creating Interaction and Control Message Types

The procedure for creating interaction message types and control message types is the same, except that you derive them from different classes. Control messages are derived from *DtPortalControlMessage*; interaction messages are derived from *DtPortalInteraction*.

Each message type has a unique message kind. The message kind is an enumeration that is a positive integer value. All predefined message kinds are defined in *pMessageKinds.h*.

! If you create new message types for a user-developed broker and you use that broker in an exercise with other user-developed brokers, you are responsible for ensuring that there are no conflicts in message kinds.

As a service to our customers, if you create new message types and send a list of your message kind values to support@mak.com, we will add your message kinds to *pMessageKinds.h* so that future releases of VR-Exchange will show that those values are in use.

Because it is the type of the message that makes a message unique, and not the attributes of the message, there are two ways to make a new message type – a simple member function for messages that do not have attributes, and a more complex member function for messages that have attributes.

Creating a Message that Does Not Have Attributes

Control messages usually do not have attributes.

To create a message that has no attributes:

1. Choose a message kind. Select an integer that is not already defined in *pMessageKinds.h*.
2. Give the new message a name and define it as follows (assuming message kind 200 and a new message called PauseSimulation):

```
typedef DtPortalControlMessageTemplate<200> PauseSimulation;
```

The template class registers this message type with the factory and the *DtPortalConnection* instance will know how to use it. You can then use this type as you would any other control message type.

Creating a Message that has Attributes

Interaction messages usually have attributes. Some control messages have attributes. The modPortalApp example shows how to create a message that has attributes.

To create a message type that has attributes.

1. Choose a message kind. Select an integer that is not already defined in *pMessage-Kinds.h*.
2. If you are creating a control message, subclass *DtPortalControlMessage*. If you are creating an interaction message, subclass *DtPortalInteraction*.
3. Add data storage to your message for all attributes of the message. There is no required way to do this, however the built-in messages follow the VR-Link way of writing accessors and mutators. Accessors use the format:

   ```
   int attributeName() const
   ```

   Mutators are preceded with set, for example:

   ```
   void setAttributeName(int val);
   ```

4. For interactions, you must provide implementations for the following member functions:

   – Kind()
   – interactionKindName()
   – netSize()
   – writeSelfToBuffer()
   – readSelfFromBuffer()
   – printDataToStream().

   A full class header might look like the following:

```
class Foo: public DtPortalInteraction
{
public:
  ...
  // required to return your unique kind
  virtual DtPortalMessageKind kind() const { return 200;}
  // required to return a string unique to this type.
  // (for interactions only)
  virtual const char *interactionKindName() const {return "foo";}
  // required - returns the number of bytes required to pack
  // this message in the network.
  virtual unsigned int netSize() const;
  // required - called to write message to buffer of size netSize()
  virtual void writeSelfToBuffer(void *buffer) const;
  // required - called to decode network buffer
  virtual void readSelfFromBuffer(const void *buffer);
  // required - prints the internal state of the message to str
  virtual std::ostream &printDataToStream(std::ostream &str) const;

  // Attributes
  DtU32 height()const { return myHeight;}
```

```
  void setHeight(DtU32 val){ myHeight = val;};
  DtString string()const { return myString;}
  void setString(const DtString& val) { myString = val;}
protected:
  DtU32 myHeight;
  DtString myString;
};
```

5. To help implement the derived class, VR-Exchange includes encoder and decoder functions for basic types, as follows:

```
unsigned int Foo::netSize() const
{
  // get the size of the base message type
  unsigned int size = DtPortalInteraction::netSize();
  // append our attributes
  size += getSize(myHeight);
  size += getSize(myString);
  return size;
}

void Foo::writeSelfToBuffer(void *buffer) const
{
  // encode the base class
  DtPortalInteraction::writeSelfToBuffer(buffer);
  // advance the pointer
  char *ptr = (char *)buffer + DtPortalInteraction::netSize();

  // encode our attributes
  ptr = encode(myHeight, ptr);
  ptr = encode(myString, ptr);
}

void Foo::readSelfFromBuffer(const void *buffer)
{
  DtPortalInteraction::readSelfFromBuffer(buffer);
  const char *ptr = (const char *)buffer +
    DtPortalInteraction::netSize();
  ptr = decode(myHeight, ptr);
  ptr = decode(myString, ptr);
}

std::ostream &Foo::printDataToStream(std::ostream &str) const
{
  DtPortalInteraction::printDataToStream(str);
  DtPrintAttribute(str, "Height" , myHeight );
  DtPrintAttribute(str, "String" , myString );
  return str;
}
```

6. For interaction messages, add addCallback() and removeCallback() member functions. Although they are not required, because you can register callbacks on the *PortalConnection*, they provide added convenience. The line that begins connection->messageFactory() is particularly important, because it tells the Portal connection how to create a valid message.

```
typedef void (*CallbackType) (const Foo& interaction,
    void* callingObject);
```

```
void Foo::addCallback( DtPortalConnection* connection,
  CallbackType function, void* callingObject )
{
  // Create a factory for this class and tell the message factory
  // to use it. This can be done in the broker, but it is harder to
  // forget when it is done here.
  connection->messageFactory()->registerCreator( Foo() );

  connection->addMessageCallback( 200,
    (DtPortalConnection::MessageCbFcn) function, callingObject);
}

void Foo::removeCallback( DtPortalConnection* connection,
  CallbackType function, void* callingObject )
{
  connection->removeMessageCallback( 200,
    (DtPortalConnection::MessageCbFcn) function, callingObject );
}
```

5.3.5. Reflecting Objects

Object updates are sent as messages and can be received and transmitted the same way as control messages and interactions. However, because objects have state, they require a different API. The concept of receiving object updates is called object reflection.

To receive updates for a particular type of object from the Portal, you create an instance of a Portal Object Manager. For example, entities are a type of object, as are aggregates. If you want to receive entity updates, create a *DtPortalReflectedEntityManager* instance (*pReflectedObjectManagerTypes.h*). The code looks like this:

```
DtPortalReflectedEntityManager manager(myPortalConnection);
```

You can then register callbacks for when objects are created or deleted, as follows:

```
manager.addRemovalCallback(deletedCallback, 0);
manager.addAdditionCallback(addedCallback, 0);
```

When an entity object is created, your function addedCallback() will be called. The broker will have access to a *DtPortalReflectedEntity* at that time.

DtPortalReflectedEntity instances contain a pointer to a state repository, which keeps track of all the attributes of the object. They also contain a callback mechanism to notify the broker of object state changes. This way the objects do not need to be polled for changes each frame. To register for an update callback the broker does the following:

```
void addedCallback(const DtPortalReflectedEntity &reflectedObject)
{
  ...
  reflectedObj.addUpdateCallback(updatedCallback, 0);
}
```

5.3.6. Publishing Objects

Sending object updates is called publishing. Each object type has a corresponding publisher type. To create an entity publisher and fill it with basic data, the code would look like this:

```
publisher = new DtPortalEntityPublisher(myPortalConnection);
// Set the required, unique ID
publisher->sr()->setIdentifier("Object2-F18");
// Fill In basic information
publisher->sr()->setLocation(23.3,45.3,45.2);
publisher->sr()->setOrientation(1.3,3.2,10.3);
```

Each publisher must be ticked every frame:

```
publisher->tick();
```

During a tick, the publisher looks at its internal state and decides if data has changed. If it has changed, an update message is sent to the Portal. If the object has not changed, the tick() member function does nothing. The tick() member function also sends updates for late-joining brokers.

The object is destroyed when the publisher goes out of scope. All objects have a final message, which is sent when the publisher is destroyed.

5.3.7. Creating New Object Types

Creating a new object type is much like creating an interaction or control message.

To create an object type:

1. Select an integer that is not already defined in *pMessageKinds.h*. For details, please see "Creating Interaction and Control Message Types," on page 5-10.
2. Create a new class, that derives from *DtPortalObject*.
3. Add the attributes associated with the object. Do this the same way that you would add attributes for an interaction ("Creating a Message that has Attributes," on page 5-11).
4. Once this class is written, you must create a publisher, a reflected object, and a reflected object manager. These classes are created through templates, as follows:

```
// Given this class declaration
class PortalFruit : public DtPortalObject {};

// Create a Publisher
typedef DtPortalObjectPublisherTemplate<PortalFruit>
  PortalFruitPublisher;
// Create a Reflected Object
typedef DtPortalReflectedObjectTemplate<PortalFruit>
  PortalReflectedFruit;
// Create a Reflected Fruit Manager
```

```
typedef DtPortalReflectedObjectManager<PortalReflectedFruit>
  PortalReflectedFruitManager;
```

These typedefs let you create an object and use it like the built-in types. The modPortalApp example has a full implementation of the *DtPortalFruit* class.

5.4. The Broker API

The Broker API is in the *libBroker* library. It is built on top of the Portal API as a framework for building brokers. The various protocol-specific broker APIs provided by MÄK, such as the HLA broker API or the DIS broker API, are built on top of the broker API. VR-Exchange includes an example broker, SimpleBroker, that shows how to use the broker API.

The only things that a broker must do are create an instance of *DtPortalConnection* and process shutdown requests. However, to be useful, a broker should do the following:

- Connect to the exercise for which it is translating
- Register callbacks for the messages it wants to receive from the Portal (object updates, interactions, and control messages)
- Create new message types for objects it needs to translate that are not part of the Portal's predefined objects
- Process input from its exercise
- Translate input and send it to the Portal
- Tick the Portal
- Translate updates from the Portal and send them to its exercise.

5.4.1. Subclassing *DtBroker*

Most VR-Exchange brokers start by subclassing *DtBroker* (*broker.h*). This class manages connections to the Portal through a *DtPortalConnection*. The *DtBroker* class manages the lifetime of the application, and passes callbacks from the Portal requesting that a GUI be launched. Please see the SimpleBroker example to see how this is done.

5.4.2. The Broker Initializer

The broker initializer (*DtBrokerInitializer*, in *brokerInitializer.h*) initializes a *DtBroker* instance by reading a configuration file and command line options. This class provides a consistent configuration file format and consistent command line options among brokers.

5.4.3. Translators and MÄK-Supplied Brokers

All brokers supplied by MÄK are built around translators. A translator is a class that is responsible for translating some discrete protocol component, either an object or an interaction. Some translators translate several tightly related interactions or objects, for example both Emitter Systems and Emitter Beams are translated by the Emitter System Translators. Each broker supplied by MÄK has built-in translators. You can add new translators for new PDUs, interactions, objects, or messages.

Even though translators are fundamental to the MÄK broker architecture, there is no requirement to use translators in user-written brokers. Nor is there a requirement to group some objects together into a single translator. There is no base object called DtTranslator in *libBroker*. Translators are used in VR-Exchange brokers more as an architectural component than a specific object instance, even though the HLA, DIS, and TENA brokers all contain base *DtTranslator* object classes. Because of the fluid definition of a translator, and the different requirements of translating different protocols, there is no universal *DtTranslator* class that meets the needs of all brokers.

Enabling Translators

Every DIS, HLA, and TENA translator requires the enableTranslator() member function:

```
virtual void enableTranslator(bool enabled)
```

This function should connect your callback functions to and from the network and the Portal. It provides the ability to enable and disable a translator in real time. When a translator is created, it should start disabled.

Here is an example of using enableTranslator():

```
void DtEntityTranslator::enableTranslator(bool enabled)
{
  if(enabled)
  {
    myHlaReflectedList->addEntityAdditionCallback(
      DtEntityTranslator::hlaObjectAddedCb, this );
    myPortalReflectedManager->addAdditionCallback(
      DtEntityTranslator::portalObjectAddedCb, this );
  }
  else
  {
    myHlaReflectedList->removeEntityAdditionCallback(
      DtEntityTranslator::hlaObjectAddedCb, this );
    myPortalReflectedManager->removeAdditionCallback(
      DtEntityTranslator::portalObjectAddedCb, this );
  }
}
```

5.5. The HLA Broker API

The HLA Broker API is in the *libHlaBroker* library. It is built on the Broker API. The HLA Broker Library is the full implementation of the HLA broker.

DtHlaBroker (*hlaBroker.h*) is the central class in *libHlaBroker*, it is a subclass of *DtBroker* that connects to an HLA exercise using a VR-Link *DtExerciseConn*. The *DtHlaBroker* instance also opens a connection to Qt to process the broker's GUI. This instance is created by *DtHlaBrokerApp* (*hlaBrokerApp.h*).

The *DtHlaBroker* is responsible for creating object and interaction translators, configuring the HLA connection through an *DtHlaBrokerInitializer* (*hlaBrokerInitializer.h*) instance, and giving each translator processor time by calling tick().

The *DtHlaBroker* instance manages *DtTranslator* (*translator.h*). There are two types of *DtTranslator*: *DtInteractionTranslator* (*interactionTrans.h*) and *DtObjectTranslator* (*objectTrans.h*). The *DtHlaBroker* creates a translator for each message type. Each translator is responsible for translating its message type to an HLA class and back. This concept maps fairly well for DIS and the RPR FOM, but may not work for more complex FOMs. Therefore, the translator classes are not part of the *libBroker* API.

Interaction translators register for updates for their interaction type. They translate the updates and pass them to their local federation or the Portal, as appropriate.

Object translators register for object discovery, update, and removal messages, then translate them and publish them to the Portal or federation. A *DtBrokerObject* (*brokerObject.h*) is maintained for every object the translator knows about. The broker object contains pointers to the reflected object, the publisher, and other data about the object, such as whether it has been discovered or filtered.

You can create translators in the following ways:

- Create translators that override existing translators. You would do this if you want to replace the functionality of an existing translator.
- Create a translator to add a new type of message to the broker.

When you create a translator class, you must create a static function called translatorName() that returns a string. For example, the *DtEntityTranslator* class has this code:

```
static const char* DtEntityTranslator::translatorName()
  { return "Entity"; };
```

If you want to override this class, you add the same function to your class:

```
static const char *DtModEntityTranslator::translatorName()
  {return "Entity";};
```

If you want to add a new type of translator you provide a new name.

```
static const char *DtAppleTranslator::translatorName() {return "Apple;};
```

The translator name is displayed on the Translators page of a connection's information dialog box. The HLA translators supplied with VR-Exchange are listed in Table A-1.

You must enable translators, as described in "Enabling Translators," on page 5-16.

Brokers that want to replace a translator should register their translator with the factory after creating the *DtHlaBroker*. The broker should then tell the *DtHlaBroker* instance to load all its translators.

For example, code to replace the EntityState translator would look like this:

```
// Create an HLA broker application. This creates the DtHlaBroker
// instance, and registers all the default translators.
DtHlaBrokerApp brokerApp(hlaInit);

// Registers a new translator creator for entities. Its factory knows
// this class creates entities because the method translatorName()
// returns a string called "Entity"
brokerApp.hlaBroker()->objectTranslatorFactory()->addCreator
  <DtModEntityTranslator>();

// run() calls the creator functions to load all required translators. It
// uses the creators registered in the previous lines of code.
brokerApp.run();
```

5.6. The TENA Broker API

The TENA Broker API is in the *libTenaBroker* library. It is built on the Broker API. The TENA Broker Library is the full implementation of the TENA broker.

DtTenaBroker (*tenaBroker.h*) is the central class in *libTenaBroker*. It is a subclass of *DtBroker* that connects to a TENA execution using a VR-Link *DtExerciseConn*. The *DtTenaBroker* instance also opens a connection to Qt to process the broker's GUI. This instance is created by *DtTenaBrokerApp* (*tenaBrokerApp.h*).

The *DtTenaBroker* is responsible for creating object and interaction translators, configuring the TENA connection through a *DtTenaBrokerInitializer* (*tenaBrokerInitializer.h*) instance, and giving each translator processor time by calling tick().

The *DtTenaBroker* instance manages *DtTranslator*. There are two subclasses of *DtTranslator*: *DtInteractionTranslator* and *DtObjectTranslator*. The *DtTenaBroker* creates a translator for each message type. Each translator is responsible for translating its message type to a TENA class and back.

Interaction translators register for updates for their interaction type. They translate the updates and pass them to their local TENA execution or the Portal, as appropriate.

Object translators register for object discovery, update, and removal messages, then translate and publish them to the Portal or TENA execution. A *DtBrokerObject* is maintained for every object the translator knows about. The broker object contains pointers to the reflected object, the publisher, and other data about the object, such as whether it has been discovered or filtered.

You can create translators in the following ways:

- Create translators that override existing translators. You would do this if you want to replace the functionality of an existing translator.
- Create a translator to add a new type of message to the broker.

When you create a translator class, you must create a static function called translatorName() that returns a c-string. For example, the *DtAggregateTranslator* class's code is equivalent to the following:

```
static const char* DtAggregateTranslator::translatorName()
{
  Return "Aggregate";
}
```

If you want to override this class, you add the same function to your class:

```
static const char* MyNewAggregateTranslator::translatorName()
{
  Return "Aggregate";
}
```

If you want to add a new type of translator, you provide a new name:

```
static const char* MyAppleTranslator::translatorName()
{
  Return "Apple";
}
```

The translator name is displayed on the Translators page of a connection's information dialog box. All TENA broker translator names supplied with VR-Exchange are listed in Table A-3 and Table A-4.

You must enable translators, as described in "Enabling Translators," on page 5-16.

Brokers that want to replace a translator should register their translator with the factory after creating the *DtTenaBroker*. The broker should then tell the *DtTenaBroker* instance to load all its translators.

For example, code to replace the Aggregate translator would look like this:

```
// Create a TENA broker application. This creates the DtTenaBroker
// instance and registers all the default translators. It is initialized
// with a DtTenaBrokerInitializer object; in this case, tenaInit.
DtTenaBrokerApp  brokerApplication( tenaInit );

// Register a new translator creator for aggregates. Its factory knows
// this class creates aggregates because the method translatorName()
// returns a c-string called "Aggregate"
brokerApplication.tenaBroker()->objectTranslatorFactory()
  ->addCreator< MyNewAggregateTranslator >();

// run() calls the creator functions to load all required translators. It
// uses the creators registered in the previous lines of code.
brokerApplication->run();
```

5.7. The DIS Broker API

The DIS Broker API is in the *libDisBroker* library. It is built on the Broker API. The DIS Broker Library is the full implementation of the DIS broker.

DtDisBroker (*disBroker.h*) is the central class in *libDisBroker*. It is a subclass of *DtBroker* that connects to a DIS execution using a VR-Link *DtExerciseConn*. The *DtDisBroker* instance also opens a connection to Qt to process the broker's GUI. This instance is created by *DtDisBrokerApp* (*disBrokerApp.h*).

The *DtDisBroker* is responsible for creating object and interaction translators, configuring the DIS connection through a *DtDisBrokerInitializer* (*disBrokerInitializer.h*) instance, and giving each translator processor time by calling tick().

The *DtDisBroker* instance manages *DtTranslator*. There are two subclasses of *DtTranslator*: *DtInteractionTranslator* and *DtObjectTranslator*. The *DtDisBroker* creates a translator for each message type. Each translator is responsible for translating its message type to a DIS class and back.

Interaction translators register for updates for their interaction type. They translate the updates and pass them to their local DIS exercise or the Portal, as appropriate.

Object translators register for object discovery, update, and removal messages, then translate and publish them to the Portal or DIS exercise. A *DtBrokerObject* is maintained for every object the translator knows about. The broker object contains pointers to the reflected object, the publisher, and other data about the object, such as whether it has been discovered or filtered.

You can create translators in the following ways:

- Create translators that override existing translators. You would do this if you want to replace the functionality of an existing translator.
- Create a translator to add a new type of message to the broker.

When you create a translator class, you must create a static function called translatorName() that returns a c-string. For example, the *DtAggregateTranslator* class's code is equivalent to the following:

```
static const char* DtAggregateTranslator::translatorName()
{
  return "Aggregate";
}
```

If you want to override this class, you add the same function to your class:

```
static const char* MyNewAggregateTranslator::translatorName()
{
  return "Aggregate";
}
```

If you want to add a new type of translator, you provide a new name:

```
static const char* MyAppleTranslator::translatorName()
{
  return "Apple";
}
```

The translator name is displayed on the Translators page of a connection's information dialog box. The HLA translators supplied with VR-Exchange are listed in Table A-2.

You must enable translators, as described in "Enabling Translators," on page 5-16.

Brokers that want to replace a translator should register their translator with the factory after creating the *DtDisBroker*. The broker should then tell the *DtDisBroker* instance to load all its translators.

For example, code to replace the Aggregate translator would look like this:

```
// Create a DIS broker application. This creates the DtDisBroker
// instance and registers all the default translators. It is initialized
// with a DtDisBrokerInitializer object; in this case, disInit.
DtDisBrokerApp brokerApplication( disInit );

// Register a new translator creator for aggregates. Its factory knows
// this class creates aggregates because the method translatorName()
// returns a c-string called "Aggregate"
brokerApplication.disBroker()->objectTranslatorFactory()->addCreator
  < MyNewAggregateTranslator >();

// run() calls the creator functions to load all required translators. It
// uses the creators registered in the previous lines of code.
brokerApplication->run();
```

5.8. Extending the Portal Application

The VR-Exchange main executable *vrx.exe*, is referred to as the Portal application. The Portal application performs two basic functions: managing inter-broker message transfer, and managing the graphical display and logging translated messages. Although the graphical user interface facilitates management and configuration of connections, it is not required for VR-Exchange to operate.

The Portal application creates brokers and manages the internal message queues used for inter-broker communication. To accomplish this task the main Portal application class *DtPortalApp* (*portalApp.h*) creates the helper classes *DtQueueManager* (*queueManager.h*) and *DtAppBrokerManager* (*appBrokerManager.h*). The *DtQueueManager* instance creates and manages all internal message queues and maintains the status of all connected brokers. If a broker exits unexpectedly, the *DtQueueManager* instance sends a message to disconnect the broker. The *DtAppBrokerManager* also maintains a list of connections, displays the list in the GUI, and sends messages to individual brokers to shut down if their connections are disabled.

The Portal application acts as a read-only broker to display basic information transmitted between other brokers. When run with a GUI, the Portal application creates a *DtPortalAppGui* (*portalAppGui.h*) instance, which builds the Qt GUI interface and creates both object and interaction view components. View components are responsible for displaying some aspect of the internal Portal communications. Each view component instance connects via a *DtPortalConnection* and registers for updates much like a broker would. Instead of translating the updates it receives, the view component displays information about those updates on the screen.

Because message management occurs without knowledge of the message internals or message types, if you write new brokers, or develop new message types, you do not need to modify the Portal application. However, if you want new message types displayed in the GUI, or want to change the way the GUI looks, you must modify and recompile the Portal application.

A minimal example of a VR-Exchange Portal application is as follows:

```
DtPortalAppInitializer initializer("VR-Exchange.xml", __argc, __argv);
DtPortalApp pApp(initializer);
pApp.run();
```

The *DtPortalAppIntitializer* (*portalAppInitializer.h*) reads the specified configuration file and parses command-line arguments. Then it initializes the *DtPortalApp* instance, which creates a *DtPortalAppGui* instance. The example ModifiedPortalApp demonstrates this.

The *DtPortalAppGui*, when it exists, creates view component classes to display object and interaction updates. The classes are *DtInteractionViewComponent* (*pInteractionViewComponent.h*) and *DtObjectViewComponent* (*pObjectViewComponent.h*). Each of these classes can be subclassed and registered with the *DtPortalAppGui* class for later use.

For example, if you want to display a new interaction class, you would subclass the *DtInteractionViewComponent* and add a callback for that type of interaction class. The ModifiedPortalApp example does this with the *DtPortalBoom* interaction, as follows:

```
class DtModInteractionManager : public DtInteractionViewComponent
{
public:
  DtModInteractionManager(DtPortalConnection *pconn, Qwidget *parent);
  virtual ~DtModInteractionManager();
  static DtAppInteractionManager* create(DtPortalConnection *pconn,
    Qwidget *parent);
protected:
  static void boomCb(const DtPortalBoomInteraction &inter,
  void *userData);
};
```

The static member function boomCb() is called any time a new *DtPortalBoom* interaction is received. This callback is registered in the constructor as follows:

```
DtModInteractionManager::DtModInteractionManager(
  DtPortalConnection *pconn, Qwidget *parent):
  DtInteractionViewComponent(pconn, parent)
{
  DtPortalBoomInteraction::addCallback(pconn,
    DtModInteractionManager::boomCb, this);
}
```

This callback is registered like all callbacks for Portal objects. For more information, please see "The Portal API," on page 5-8.

The callback looks like this:

```
void DtModInteractionManager::boomCb(
const DtPortalBoomInteraction &inter, void *usrData)
{
  DtString description = "Something went boom!";
  DtModInteractionManager *p = (DtModInteractionManager*)usrData;
  p->processInteraction(inter, description);
}
```

The callback calls the virtual function processInteraction(), which takes an interaction to display and a string description. The string can explain that someone shot at someone else and provide a short detailed description of the event to be displayed on the GUI.

The new class creator member function is then registered in the *DtPortalAppGui* class as follows:

```
DtPortalAppGui::setInteractionManagerCreator(
  DtModInteractionManager::create);
```

DtPortalAppGui::setInteractionManagerCreator() is a static class that must be called before the *DtPortalAppGui* instance is created.

The same process happens for objects, as follows:

```
class DtModObjectManager : public DtObjectViewComponent
{
public:
  DtModObjectManager(DtPortalConnection *pconn, Qwidget *parent);
  virtual ~DtModObjectManager();
  static DtAppObjectManager* create(DtPortalConnection *pconn,
    QWidget *parent);
protected:
  static void discoverFruitCb( const DtPortalReflectedFruit &obj,
    void*usrData);
  static void updateFruitCb( const DtPortalReflectedFruit &obj,
    void*usrData);
  static void deleteFruitCb( const DtPortalReflectedFruit &obj,
    void *usrData);
protected:
  DtPortalReflectedFruitManager *myRefFruitMgr;
};
```

This class is similar to the *DtInteractionViewComponent* class, except that for every object there are three events to register for: discovery, update, and removal.

The callbacks look like the following:

```
void DtModObjectManager::discoverFruitCb(
const DtPortalReflectedFruit &obj, void *usrData)
{
  DtInfo << "DtModObjectManager discovered a new fruit. \n";
  DtModObjectManager *p = (DtModObjectManager*)usrData;
  obj.addUpdateCallback( DtModObjectManager::updateFruitCb, usrData);
  p->addObject(obj);
}
void DtModObjectManager::updateFruitCb(
const DtPortalReflectedFruit &obj, void *usrData)
{
  DtInfo << "DtModObjectManager updated a fruit. \n";
  DtModObjectManager *p = (DtModObjectManager*)usrData;
  p->updateObject(obj);
}
void DtModObjectManager::deleteFruitCb(
const DtPortalReflectedFruit &obj, void *usrData)
{
  DtInfo << "DtModObjectManager removed a fruit. \n";
  DtModObjectManager *p = (DtModObjectManager*)usrData;
  p->removeObject(obj);
```

```
}
```

In this case the base class provides three member functions that display the events: addObject(), updateObject(), and removeObject(). These three member functions add the object to the display, update the object, and remove the object from the display. Each of these member functions uses the message kind to keep track of statistics about the object and display the name of the object.

5.9. API Examples

VR-Exchange includes several examples. They are described in the class documentation under the Examples heading.

When you build the examples, the binary files are placed in the example directories. To use them, copy them to the *./bin* directory.

5.10. Building and Compiling Brokers

VR-Exchange includes solution and project files in the example directories of the Windows version and makefiles in the example directories for the Linux version. Please examine these files for a full picture of the build process.

5.10.1. VR-Exchange Libraries

VR-Exchange has the following libraries:

- *libPortal* – contains all classes and functions used to manage VR-Exchange, as well as the protocol-independent data messages that are exchanged between brokers. Brokers and pseudo-brokers, such as vrxMessageDump and the Portal require this library.
- *libBroker* contains classes that make development of brokers easier.
- *libHla13Broker*, *libHla1516Broker*, and *libHla1516eBroker* are the HLA broker implementations. These libraries contain code to translate the VR-Exchange protocol-independent messages to the HLA classes listed in Table A-1.
- *libTenaBroker* is the TENA broker implementation.
- *libDisBroker* is the DIS broker implementation.
- *libPortalApp* is the base library for the Portal application.

5.10.2. Third Party Libraries

VR-Exchange brokers require the following third party libraries:

- VR-Link – The non-HLA components use *vl*, *vlUtil*, *xml*, and *matrix*. The *libHla-Broker* library uses *vlHLA13* or *vlHLA1516*. The *libTenaBroker* library uses *vlTENA*. The *libDisBroker* library uses *vlDIS*.

Although VR-Link is included on the VR-Exchange CD, use of VR-Link is not covered by the VR-Exchange license; you must purchase a separate license to use it. Please see your MÄK sales representative for details.

- *pThreads* – VR-Exchange uses pThreads for shared memory queue locking. Because pThreads are not available on Windows, VR-Exchange uses the *PTHREADS-WIN32* library. The *PTHREADS-WIN32* library is distributed under the GNU Lesser General Public License. No modifications to the library have been made by MÄK. In future releases this dependency may be removed.

If you want to rebuild GUI components of the Portal or the brokers supplied with VR-Exchange, you need Qt. Qt is available as a free download under the LGPL version 2.1 license at www.qtsoftware.com. This version should be satisfactory for most VR-Exchange customers. If you need a Qt commercial license, you must purchase the license from Qt Software.

You do not need Qt to build new brokers or build the Portal without its GUI. You can purchase a Qt license from Trolltech: www.qtsoftware.com. Please see *VR-Exchange Release Notes* for the required version of Qt.

TENA users must install TENA middleware and any libraries required by their LROM. HLA users must install an RTI.

5.10.3. Building Brokers

There are no special pre-processor flags required to build a broker. VR-Exchange includes release and debug versions of Windows libraries. Debug versions use the naming convention *library_name*d.lib. Table 5-2 lists the libraries required to build brokers for Windows and Linux.

Table 5-2: Libraries required to build brokers

Windows Libraries	Linux LIbraries
lib{protocol}Broker.lib	*{protocol}broker*
libBroker.lib	*Broker*
libPortal.lib	*Portal*
vl.lib	*vl*
vlutil.lib	*vlutil*
vl{protocol}.lib	*vl{protocol}*
matrix.lib	*matrix*
mtl.lib	*mtl*
libxml2.lib	*xml*
pthreadVC2.lib	*pthread*
QtCore4.lib	*QtCore*
QtGui4.lib	*QtGui*
commonDialogs.lib	*commonDialogs*
commonWidgets.lib	*commonWidgets*

On Windows, the simpleBroker example requires the library *vl5*.

5.10.4. Rebuilding the HLA Broker

Table 5-3 lists the libraries you need to rebuild the HLA broker.

Table 5-3: HLA broker libraries

Windows Libraries	Linux LIbraries
vlPortal.lib	*portal*
libvl.lib	*vl*
vlutil.lib	*vlutil*
matrix.lib	*matrix*
mtl.lib	*mtl*
libxml2.lib	*xml*
pthreadVC2.lib	*pthread*
QtCore4.lib	*QtCore4*
QtGui4.lib	*QtGui4*
vlHLA13.lib, *vlHLA1516.lib*, or *vlHLA1516e.lib*)	*hla13Broker*, *hla1516Broker*, or *hla1516eBroker*
libHla1516Broker.lib (or *libHla13Broker.lib*)	*broker*
commonDialogs.lib	*commonDialogs*
commonWidgets.lib	*commonWidgets*

To rebuild the HLA broker on Windows, you must have the following RTI libraries (included with your RTI):

- *librti1516.lib* (for HLA 1516), *librti1516e.lib* (for HLA Evolved), or *libRTI-NG.lib* (for HLA 1.3)
- *libfedtime1516.lib* (for HLA 1516), *libfedtime1516e.lib* (for HLA Evolved), or *libFedTime.lib* (for HLA 1.3)

For the HLA broker several VR-Link compiler flags are required. Please consult the VR-Link documentation for instructions about how to build a VR-Link HLA federate.

5.10.5. LROM Mapper Library Dependencies

VR-Exchange includes release and debug versions of the following LROM Mappers: *coLromMapperRT* and *makLromMapperRT*. You can use VR-Link to build custom LROM Mappers.

The LROM mapper files have several dynamic library dependencies that are not supplied with either VR-Link or VR-Exchange. You can get these libraries the following ways from the TENA SDA web site:

- Download the TENA Middleware
- Download the appropriate Object Model.

5.10.6. Rebuilding the TENA Broker

If you build a TENA broker, you must link in the *vlTENART* library. You must set the VR-Link compiler flag DtTENA. This library has the dynamic library dependencies listed in this section. The libraries are available from the indicated sources.

Libraries required for each platform are essentially the same. The libraries supplied with TENA Middleware and the MÄK Object Model use the following naming convention:

Library_name`-`*os*`-[d-]`*version*`.`*extension*

where:

- *Library_name* is the platform-independent portion of the name
- *os* is the operating system: xp-vc71 for Windows VC++ 7.1; xp-vc80 for Windows VC++ 8.0; rhelws4-gcc34 for Red Hat Linux Enterprise Workstation 4 using gcc 3.4
- d specifies a debug version
- *version* is the version of the library
- *extension* is dll for Windows, or so for Linux.

The following libraries are supplied with the TENA Middleware:

- ACE-*os*-1.5.1d.*ext*
- libDCT_DIME-*os*-v5.2.2.*ext*
- libDCT_Utils-*os*-v5.2.2.*ext*
- TAO_PortableServer-*os*-1.5.1d.*ext*
- TAO-*os*-1.5.1d.*ext*
- libTENA_Middleware-*os*-v5.2.2.*ext*.

The following libraries are supplied with the MÄK Object Model:

- libTENA-PlatformType-v1-*os*-v5.2.2_0.*ext*
- libTENA-PlatformType-v1-csvReader-v2-*os*-v5.2.2_0.*ext*
- libTENA-Time-v1.1-*os*-v5.2.2_0.*ext*
- libTENA-Time-v1.1-fullConversion-v2-*os*-v5.2.2_0.*ext*
- libTENA-UniqueID-v2-*os*-v5.2.2_0.*ext*
- libTENA-UniqueID-v2-userDefined-v1-*os*-v5.2.2_0.*ext*.

The following MÄK libraries are provided with VR-Exchange. On Windows, debug libraries have a "d" before the extension, for example, *vlTENARTd.lib*.

Table 5-4:

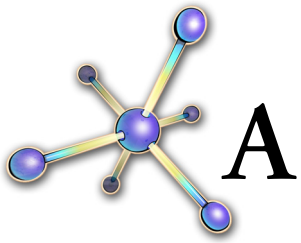
Windows	Linux
vlTENA.lib	*vlTENA*
vlutil.lib	*vlutil*
matrix.lib	*matrix*
vl.lib	*vl*
mtl.lib	*mtl*
libTenaBroker.lib	*TenaBroker*
commonWidgets.lib	*commonWidgets*
commonDialogs.lib	*commonDialogs*
QtCore4.lib	*QtCore4*
QtGui4.lib	*QtGui4*
libBroker.lib	*Broker*
libxml2.lib	*libxml2*
pthreadVC2.lib	*pthreadVC2*

5.10.7. Rebuilding the DIS Broker

To build the DIS broker, you must specify several VR-Link compiler flags. Please consult the VR-Link documentation for instructions about how to build a VR-Link application for DIS.

Table 5-5: Libraries required to build the DIS broker

Windows Libraries	Linux LIbraries
vl.lib	*vl*
vlutil.lib	*vlutil*
matrix.lib	*matrix*
mtl.lib	*mtl*
libxml2.lib	*xml2*
pthreadVC2.lib	*portal*
QtCore4.lib	*QtCore4*
QtGui4.lib	*QtGui4b*
vlDIS.lib	*vlDIS*
libDisBroker.lib	*disBroker*
commonDialogs.lib	*commonDialogs.lib*
commonWidgets.lib	*commonWidgets.lib*
	broker



Broker Details

This appendix lists the translators provided by each broker and lists their command-line options. You can add command-line options to a connection in the Add Broker dialog box and the Edit Broker dialog box.

A.1. The HLA Brokers

VR-Exchange includes HLA brokers for the HLA 1.3 specification, for the SISO Standard DLC C++ API for IEEE 1516 specification, and for the HLA Evolved specification. They have built-in support for the RPR FOM using the VR-Link FOM Mappers and FED files provided. They can support other FOMs through the use of user-written FOM Mappers or generic translation.

The brokers work identically and support the same configurations, so for simplicity, in the remainder of this manual we refer to them generically as the HLA broker.

A.1.1. HLA Translators

The HLA broker is composed of translators (described in "Translators," on page 1-3). Each translator is responsible for a specific FOM object class, or interaction class. Table A-1 lists the translators in the HLA broker. The HLA broker can use a VR-Link FOM Mapper to modify the translations of any of the translators. For information about FOM mapping, please see *VR-Link Developers Guide*.

Table A-1: HLA broker translators

Translator	RPR FOM Class
Acknowledge	InteractionRoot.Acknowledge
Action Request	InteractionRoot.ActionRequest
Action Response	InteractionRoot.ActionResponse
Aggregate	ObjectRoot.BaseEntity.AggregateEntity
Collision	InteractionRoot.Collision
Comment	InteractionRoot.Comment
Create Entity	InteractionRoot.CreateEntity
Data	InteractionRoot.Data
Data Query	InteractionRoot.DataQuery
Designator	ObjectRoot.Designator
Detonation	InteractionRoot.MunitionDetonation
Emitter System	ObjectRoot.EmitterBeam ObjectRoot.EmbeddedSystem.EmitterSystem
Entity State	ObjectRoot.BaseEntity.PhysicalEntity.Platform
Environment Process	ObjectRoot.EnvironmentProcess
Event Report	ObjectRoot.EventReport
Fire	InteractionRoot.WeaponFire
Gridded Data	ObjectRoot.GriddedData
IFF	ObjectRoot.EmbeddedSystem.IFF
Radio Receiver	ObjectRoot.EmbeddedSystem.RadioReceiver
Radio Signal	InteractionRoot.RadioSignal
Radio Transmitter	ObjectRoot.EmbeddedSystem.RadioTransmitter
Remove Entity	InteractionRoot.RemoveEntity
Set Data	InteractionRoot.SetData
Start	InteractionRoot.StartResume
Stop	InteractionRoot.StopFreeze
View Control	InteractionRoot.ViewControl

A.1.2. Generic Translations

The HLA broker has a special translator called a generic translator. Each broker can configure multiple generic translators, each responsible for one specific object or interaction class. You can also specify that all translations be done generically, using none of the translators supplied with VR-Exchange. Generic translation allows you to filter parts of a FOM. It also allows you to create RTI-to-RTI bridges if the federations being bridged use the same FOM.

The generic translator does not actually do any translation. It copies each attribute or parameter byte-for-byte from and to the Portal, and publishes it to the network.

Using the same configuration as Figure 1-2, Figure A-1 illustrates how a generic translator works. This time, suppose that these FOMs do not have a translator for vehicle names. The leftmost federation sends a "lorry" message. The broker simply sends the string "lorry" to the VR-Exchange shared memory area. The second federation reads the string "lorry" from the Portal. This string is meaningful to the FOM, which is identical to that of the first federation, so it sends the "lorry" message. The third federation does not have a generic translator, so it ignores the string.

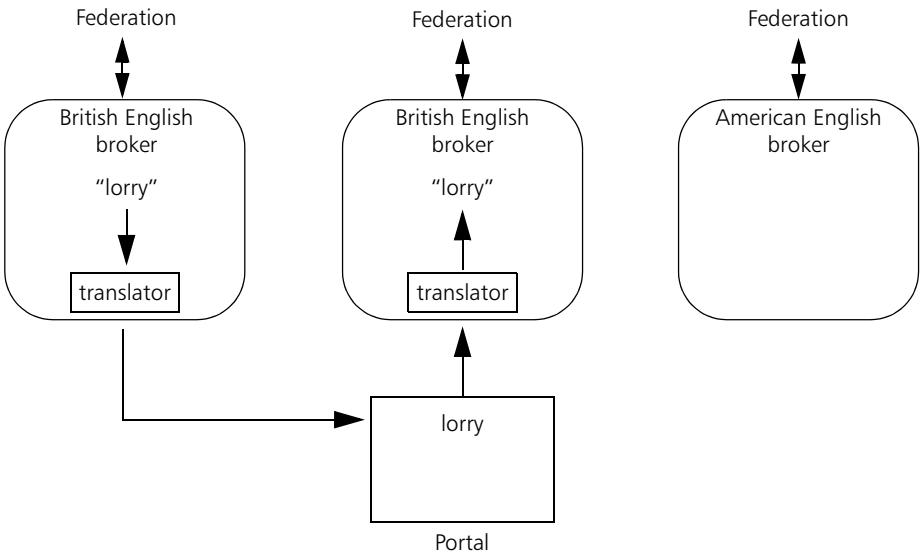


Figure A-1. How generic translators work

! To use generic translation, the sending and receiving federations must configure the same generic translators.

Generic translation only works with the HLA brokers, and only with one API at a time. This means:

- Generic messages do not work between the different HLA specifications
- The object or interaction class must be exactly the same in all connected FOMs
- You cannot use a FOM Mapper with generic translators.

Translating Individual Classes Generically

Translating individual classes generically is useful when you only want to translate parts of a FOM or you want to supplement the built-in translators. For example, if your exercise includes an interaction for which VR-Exchange does not have a translator, you can add a generic translator for that interaction. Bear in mind the restrictions on generic translations listed in the previous section.

To translate individual classes generically, you add the class names of the objects you want to translate to the appropriate collection in a connection's configuration. For example, you might add ObjectRoot.EnvironmentProcess to the Generic Objects to Translate Set attribute.

Translating a FOM Generically

Translating all FOM classes generically is useful if you have two or more federations that use the exact same FOM, but perhaps different RTI's. When you enable generic translation for all FOM classes, the HLA broker creates a generic translator for all objects and interactions in the FOM (except MOM data, and anything that starts with RTIPrivate). This mode prevents any built-in translators from starting.

To enable generic translation for the entire FOM, set the Use Generic Translators for Whole FOM attribute to True.

A.1.3. HLA Broker Command-Line Options

The HLA broker command-line options override the settings in a connections configuration file. The HLA broker has the following command-line syntax:

```
{hla13Broker | hla1516Broker | hla1516eBroker} [
  --
  --brokerName name
  --controlQueueName name
  -f library
  --fomMapperInitData data
  -F file
  -G
  -h
  --interactionQueueName name
  -l filename
  -L logfile_name
  -n level
  -N federate_name
  --objectQueueName name
  --rprFomRevision revision
  --rprFomVersion version
  --showInfoDialog
  -t port
  -v
  -x execution]
```

where:

`(--` | `--ignore_rest)` tells the broker to ignore the arguments following this flag.

`--brokerName` *`name`* specifies the name of the broker. The broker name must be unique among all brokers used. Default: Broker.

`--controlQueueName` *`name`* specifies the name of the shared memory control queue.

`(-f` | `--fomMapperLibName)` *`library`* specifies the FOM mapper library, if used.

`(-F` | `--fedFileName)` *`file`* specifies the FED file name. Default: *VR-Link.fed*.

`--fomMapperInitData` *`data`* specifies FOM Mapper initialization data, if it is required.

`(-G` | `--noGui)` specifies that the VR-Exchange Portal GUI should not be run.

`(-h` | `--help)` displays usage information and exits.

`--interactionQueueName` *`name`* specifies the name of the shared memory interaction queue.

`(-l` | `--configFile)` *`filename`* specifies the name of the configuration file for the specified broker. Default: *broker.xml*.

`(-L | --brokerLogFileName)` *`filename`* specifies the name of the log file for the broker.

`(-n | --notifyLevel)` *`level`* specifies the output level for the broker's log file. Value values are 0 – 4. As the level increases, so does the output generated. Default: 2. The meanings of the various levels are described in "VR-Exchange Command-Line Options," on page 3-3.

`(-N | --federateName)` *`federate_name`* specifies the name of the HLA broker federate.

`--objectQueueName` *`name`* specifies the name of the shared memory object queue.

`--rprFomRevision` *`revision`* specifies the revision of the RPR FOM version, for example, RPR FOM 2, revision 17.

`--rprFomVersion` *`version`* specifies the RPR FOM version. Default: 1.0.

`--showInfoDialog` specifies that the connection information dialog box be displayed at startup.

`(-t | --tcpPort)` *`port`* specifies the TCP port to use to maintain connectivity with the Portal. If the Portal dies, then the broker will know by testing this port. If this argument set to 0, the broker does not try to connect. Default: 5111.

`(-v | --version)` displays version information and exits.

`(-x | --execName)` *`execution`* specifies the HLA federation execution. Default: "VR-Link".

A.2. The DIS Broker

The DIS broker provides comprehensive translation of DIS PDUs. It receives DIS 4, 5, and 6. It sends DIS 5.

Table A-2 lists the translators supported by the DIS broker.

Table A-2: DIS broker translators

Translator	PDU
Acknowledge	Acknowledge PDU
Action Request	Action Request PDU
Action Response	Action Response PDU
Aggregate	Aggregate PDU
Collision	Collision PDU
Comment	Comment PDU
Create Entity	Create Entity PDU
Data	Data PDU
Data Query	Data Query PDU
Designator	Designator PDU
Detonation	Detonation PDU
Emitter Systems	Electromagnetic Emission PDU
Entity State	Entity State PDU
Environment Process	Environmental Process PDU
Event Report	Event Report PDU
Fire	Weapon Fire PDU
Gridded Data	Gridded Data PDU
IFF	IFF/ATC/NAVAIDS PDU
Radio Receiver	Radio Receiver PDU
Radio Signal	Signal PDU
Radio Transmitter	Radio Transmitter PDU
Remove Entity	Remove Entity PDU
Set Data	Set Data PDU
Start	Start Resume PDU
Stop	Stop Freeze PDU
View Control	Stealth Control PDU

A.2.1. DIS Broker Command-Line Options

The DIS broker command-line options override the settings in connection's configuration file. The DIS broker has the following command-line syntax:

```
disBroker [
   --
   -a application
   -A address
   --brokerName name
   --controlQueueName name
   --disMcastDevice device
   --disSendPort sendPort
   -G
   -h
   --interactionQueueName name
   -l filename
   -L logfile_name
   -n level
   --objectQueueName name
   -P port
   -s site
   --showInfoDialog
   -t port
   -v
   -x exercise]
```

where:

`(--` | `--ignore_rest)` tells the broker to ignore the arguments following this flag.

`(-a` | `--applicationNumber)` *`application`* specifies the unique DIS application number. It is used for creating entity IDs for PDUs originating from the DIS broker. Default: 27.

`(-A` | `--destinationAddress)` *`address`* is the address of outgoing DIS PDUs. If set to 0.0.0.0 the default broadcast address is used. Default: 0.0.0.0.

`--brokerName` *`name`* specifies the name of the broker. The broker name must be unique among all brokers used. Default: Broker.

`--controlQueueName` *`name`* specifies the name of the shared memory control queue.

`--disMcastDevice` *`device`* is the address of the device to use for multicast traffic. If set to 0.0.0.0, the default network address is used. Default: 0.0.0.0.

`--disSendPort` *`sendPort`* specifies the port the broker sends PDUs on. This port is only to be used by the broker. It knows that if it receives a message with this port, that it is its own message, so it does not translate it. Federates listen to the port specified by `--disPort`. Make sure this port does not conflict with others on your network. Default: 2999.

`(-G | --noGui)` specifies that the VR-Exchange Portal GUI should not be run.

`(-h | --help)` displays usage information and exits.

`--interactionQueueName` *`name`* specifies the name of the shared memory interaction queue.

`(-l | --configFile)` *`filename`* specifies the name of the configuration file for the specified broker. Default: *broker.xml*.

`(-L | --brokerLogFileName)` *`filename`* specifies the name of the log file for the broker.

`(-n | --notifyLevel)` *`level`* specifies the output level for the broker's log file. Value values are 0 – 4. As the level increases, so does the output generated. Default: 2. The meanings of the various levels are described in "VR-Exchange Command-Line Options," on page 3-3.

`--objectQueueName` *`name`* specifies the name of the shared memory object queue.

`(-P | --disPort)` *`port`* specifies the port to send and receive network PDUs on. Default: 3000.

`(-s | --siteId)` *`site`* specifies the site ID for the current exercise. This needs to be the same for all participants in the DIS exercise. Default: 1.

`--showInfoDialog` specifies that the connection information dialog box be displayed at startup.

`(-t | --tcpPort)` *`port`* specifies the TCP port to use to maintain connectivity with the Portal. If the Portal dies, then the broker will know by testing this port. If this argument is set to 0, the broker does not try to connect. Default: 5111.

`(-v | --version)` displays version information and exits.

`(-x | --exerciseId)` *`exercise`* specifies the exercise ID for the current exercise. This needs to be the same for all participants in the DIS exercise. Default: 1.

A.3. The TENA Broker

TENA (Test and Training Enabling Architecture) is an interoperability architecture used to pass data in the form of object updates and messages (like HLA objects and interactions) from one application to another. It was developed by the Foundation Initiative 2010 (FI2010) program, which is managed by the Director of the Central Test and Evaluation Investment Program (CTEIP). CTEIP is part of the Office of the Undersecretary of Defense for Test and Evaluation (DOTE-OSD). TENA was developed to serve as the interoperability architecture for the Live Range community. The direction of TENA is managed by the TENA Architecture Management Team (AMT) – a group that meets about once a quarter.

TENA's data definition format is called a Logical Range Object Model (LROM) (comparable to an HLA FOM). An LROM is a set of object/message definitions that may be used in an exercise. These object models are available via the TENA Software Development Activity (SDA) web site (https://tena-sda.org), and there are some standard TENA object models that were developed by the TENA community. TENA also uses a process called an Execution Manager that each application must connect to in order to be a part of the exercise.

The TENA broker supports the common object LROM and the MÄK LROM using an LROM Mapper. It can support other LROMs through the use of user-written LROM Mappers.

An LROM Mapper is conceptually the same thing as a FOM Mapper. For details, please see *VR-Link Developer's Guide*.

The TENA broker:

- Throttles the rate at which an object's data is published to the TENA execution by entity type
- Lets you specify the library (LROM Mapper) used to map the objects in an LROM to the VR-Exchange internal format
- Displays the throttle rate and actual published rate in the TENA broker's GUI.

A.3.1. TENA Translators

The *coLromMapper* LROM Mapper supports the translators listed in Table A-3. Each translator is responsible for the listed TENA LROM class. For a general explanation of translators, please see "Translators," on page 1-3.

The *makLromMapper* LROM Mapper supports the translators listed in Table A-3 and those listed in Table A-4. The MÄK LROM is based on the HLA RPR FOM and DIS standards.

Table A-3: Common Object LROM Mapper translators

Translator	TENA LROM Class
Entity State	TENA::Platform TENA::PlatformDetails
Fire	TENA::Engagement::WeaponFire
Detonation	TENA::Engagement::Detonation

Table A-4: MÄK LROM Mapper translators

Translator	TENA LROM Class
Acknowledge	MAK::Acknowledge
Aggregate	MAK::Aggregate
Collision	MAK::Collision
Comment	MAK::Comment
Data	MAK::Data
Designator	MAK::Designator
Emitter System	MAK::ElectromagneticEmission
Environment Process	MAK::EnvironmentProcess
IFF	JNTC::Airtransponder
Radio Receiver	MAK::RadioReceiver
Radio Signal	MAK::Signal
Radio Transmitter	MAK::RadioTransmitter
Set Data	MAK::SetData
Start	MAK::StartResume
Stop	MAK::StopFreeze

A.3.2. TENA Broker Command-Line Options

The TENA broker has the following command-line syntax:

```
tenaBroker [
   --
   --brokerName name
   --controlQueueName name
   -e endpoints
   -G
   -h
   -i
   --interactionQueueName name
   -l file
   -L logfile_name
   --lromMapper mapper
   -n level
   --objectQueueName name
   -O
   --showInfoDialog
   -t port
   -v
   -x execution]
```

where:

`(--` | `--ignore_rest)` tells the broker to ignore the arguments following this flag.

`--brokerName` *`name`* specifies the name of the broker. The broker name must be unique among all brokers used. Default: Broker.

The broker's name is used as its session name.

`--controlQueueName` *`name`* specifies the name of the shared memory control queue.

`(-e | --OrbListenEndpoints)` *`endpoints`* specifies the TENA ORB Listen Endpoint. It must match the TENA *executionManager.exe*'s `-ORBlistenEndpoints` parameter.

`(-G | --noGui)` specifies that the VR-Exchange Portal GUI should not be run.

`(-h | --help)` displays usage information and exits.

`(-i | --OrbInitRef)` *`reference`* specifies the TENA ORB Initialization Reference. It must match the TENA *executionManager.exe*'s `-ORBdefaultInitRef` parameter.

`--interactionQueueName` *`name`* specifies the name of the shared memory interaction queue.

`(-l | --configFile)` ***`filename`*** specifies the name of the configuration file for the specified broker. Default: *broker.xml*.

`--lromMapper` ***`mapper`*** specifies the LROM Mapper file name.

`(-L | --brokerLogFileName)` ***`filename`*** specifies the name of the log file for the broker.

`(-n | --notifyLevel)` ***`level`*** specifies the output level for the broker's log file. Range: 0 – 4. Default: 2. The meanings of the various levels are described in "VR-Exchange Command-Line Options," on page 3-3.

`--objectQueueName` ***`name`*** specifies the name of the shared memory object queue.

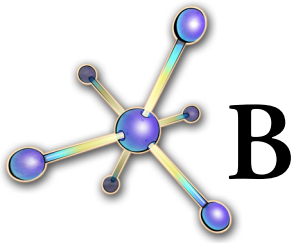
`(-O | --oneToOneMapping)` translate using one-to-one mapping.

`--showInfoDialog` specifies that the connection information dialog box be displayed at startup.

`(-t | --tcpPort)` ***`port`*** specifies the TCP port to use to maintain connectivity with the Portal. If the Portal dies, then the broker will know by testing this port. If this argument set to 0, the broker does not try to connect. Default: 5111.

`(-v | --version)` displays version information and exits.

`(-x | --execName)` ***`execution`*** specifies the TENA execution name. The value for ***`execution`*** must match the TENA *executionManager.exe*'s `-executionname` parameter. Default: "VR-Link".



A VR-Exchange/TENA Primer

TENA can be overwhelming for new users; however detailed documentation about TENA is beyond the scope of this manual. This appendix explains the most basic steps for getting up and running with TENA and the VR-Exchange TENA broker. For complete documentation about TENA, please refer to its home web site: www.tena-sda.org.

tena-sda.org is a restricted web site. Access must first be granted by the TENA SDA project office. On the home page, click Account Request.

B.1. Installing TENA Middleware

To use the TENA broker, you must download and install the TENA Middleware.

This section explains how to download, install, and verify the installation of the TENA Middleware.

- Some TENA files are compiler-specific. In the instructions in this appendix, we use the filenames for VC 7.1 versions. If you are downloading VC 8.0 versions, be sure to choose the correct version.
- Approximately 2.98 GB of hard drive space is required for building all the required TENA files.

B.1.1. Downloading the TENA Middleware

To download the TENA Middleware:

1. Log into the TENA web site, www.tena-sda.org.
2. On the Welcome page, click the TENA Middleware link.
3. On the TENA Middleware page, click the Middleware Downloads link. The TENA Repository page is displayed.
4. In the Middleware drop-down list, select TENA-v5.2.2.
5. In the table of downloads, click the download button for the version you want (*TENA-xp-vc71-v5.2.2.exe* or *TENA-xp-vc80-v5.2.2.exe*).

B.1.2. Installing the TENA Middleware

To install the TENA Middleware:

1. Close all open windows and running applications.
2. Run the TENA installer (for example, *TENA-xp-vc71-v5.2.2.exe*). This is a self-extracting zip file. In the WinZip Self-Extractor dialog box, make sure to select:

   ```
   When done unzipping open: TENA\5.2.2\scripts\post.install.bat
   ```

If you have problems install the TENA Middleware, please contact the TENA-SDA Help Desk.

B.2. Building the MÄK Object Model

You do not have to install or build the MÄK Object Model to run VR-Exchange. It is required if you want to build a customized TENA broker.

B.2.1. Downloading the MÄK Object Model

To download the MÄK Object Model:

1. Log into the TENA web site, www.tena-sda.org.
2. On the Welcome page, click the TENA Repository link.
3. At the top of the page, click Browse Repository. A list of object models is displayed.
4. Expand the MAK entry.
5. In the table of MAK object models, click MAK-LROM-v2.
6. In the Distributions list, select TENA-v5.2.2 (Figure B-1).

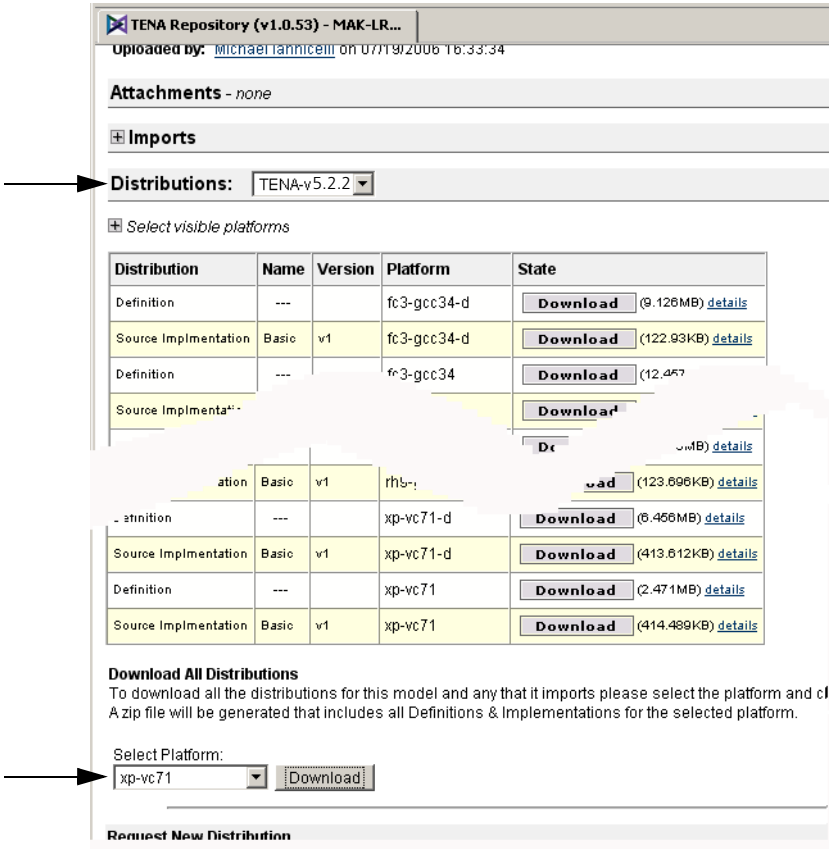


Figure B-1. MÄK Object Model download page

7. In the Select Platform list located below the Distributions table, select xp-vc71.

8. Click the Download button next to Select Platform. The distribution is assembled and displayed under the heading Object Model Imports.
9. Click Download Zip File. The file (*MAK-LROM-v2-distributions-xp-vc71-v5.2.2.zip* or *MAK-LROM-v2-distributions-xp-vc80-v5.2.2.zip*) is downloaded.

B.2.2. Installing the MÄK Object Model

To install the MÄK Object Model:

1. Download the MÄK Object Model files as described in "Downloading the MÄK Object Model," on page B-3.
2. Extract all files from the object model file (for example, *MAK-LROM-v2-distributions-xp-vc71-v5.2.2.zip*). The extracted files are either *.exe* files or zip files.
3. Run all of the *.exe* files. They are self-extracting zip files. Make sure the destination folder is the same as the TENA installation folder.

When you extract the files, do not select:

```
When done unzipping open: TENA\5.2.2\scripts\post.install.bat
```

B.2.3. Building the MÄK Object Model Basic Implementation

The dynamic libraries are included in the VR-Exchange installation. You do not need to build them unless you build a customized TENA broker.

To build the MÄK Object Model basic implementation:

1. Download the MÄK Object Model files as described in "Downloading the MÄK Object Model," on page B-3.
2. Extract all files from the object model zip file (for example, *MAK-LROM-v2-distributions-xp-vc71-v5.2.2.zip*). The extracted files are either *.exe* files or zip files.
3. Unzip each of the zip files that you extracted. Make sure the destination folder is the same as the TENA installation folder.
4. Open MS VC++ 7.1.
5. Open and build *TENA\5.2.2\src\MAK-LROM-v2\MAK-LROM-v2-2003.sln*.

MS VC++ 7.1 has experienced problems with large amounts of text in the Additional Include Directories project setting. If header files cannot be found, reduce the amount of text in this project setting. For example, change all instances of *$(TENA_HOME)\$(TENA_VERSION)\include* to *C:\TENA\5.2.2\include*.

6. Create the directory: *TENA\5.2.2\include\impls\MAK-LROM-v2-basic-v1*.
7. Copy *TENA\5.2.2\src\MAK-LROM-v2*.h* (2 files) to *TENA\5.2.2\include\impls\MAK-LROM-v2-basic-v1*.
8. Copy the files in *TENA\5.2.2\src\MAK-LROM-v2\MAK* to *TENA\5.2.2\include\impls\MAK-LROM-v2-basic-v1\MAK*.
9. Copy *TENA\5.2.2\src\MAK-LROM-v2*.lib* and **.exp* (2 files) to *TENA\lib*.
10. Copy *TENA\5.2.2\src\MAK-LROM-v2*.dll* (1 file) to *TENA\5.2.2\bin\xp-vc71*.

B.3. Running the TENA Broker

For the TENA broker to communicate with other TENA executions, two TENA services must be running: the Network Naming Service and the Execution Manager. These services do not have to reside on the same machine as VR-Exchange. However, they must be able to access that machine via TCP/IP.

1. Verify that the following environment variables are set:
   - TENA_BOOST_LIB_VERSION=1_31
   - TENA_BOOST_VERSION=1.31.0a
   - TENA_HOME=C:\TENA (should point to your TENA installation path)
   - TENA_OMC_VERSION=0
   - TENA_OS=xp
   - TENA_PLATFORM=xp-vc71
   - TENA_VERSION=5.2.2
2. Make sure that the PATH includes:

   ```
   %TENA_HOME%\%TENA_VERSION%\bin\%TENA_PLATFORM%;
     %TENA_HOME%\%TENA_VERSION%\scripts;
   ```

3. Change directory to the TENA installation directory (for example, *TENA\5.2.2\bin\xp-vc71*).
4. Start the Network Naming Service by running:

   ```
   networkNamingService.exe -ORBlistenEndpoints
     iiop://10.0.1.250:50509
   ```

5. Start the Execution Manager by running:

   ```
   executionManager.exe -ORBdefaultInitRef
     corbaloc:iiop:10.0.1.250:50509
     -executionname VR-Link
   ```

6. Start VR-Exchange.
7. Configure the TENA connection. Make sure that the connection attributes match the values you used to start the Network Naming Service and the Execution Manager, for example:
   - ORB Default Init Ref: corbaloc:iiop:10.0.1.250:50509
   - ORB Default Listen Endpoints: iiop://10.0.1.250
   - Execution Name: VR-Link.

   For details about configuring a TENA connection, please see "Configuring Connections," on page 4-5.

B.3.1. TENA Services

The information in this section is a very basic introduction to the TENA services.

Network Naming Service

The Network Naming Service (NNS) is a registry for TENA Execution Managers. The NNS must specify its ORBlistenEndpoints, which is where it should listen for requests, for example:

```
networkNamingService -ORBlistenEndpoints iiop://10.1.1.29:50509
```

The Exercise Manager

An Execution Manager registers its IP address and execution name with the NNS. Each TENA execution has one Execution Manager. In Figure B-2, two TENA executions are connected to the NNS. Execution VR-Link is running VR-Exchange.

```
executionManager -ORBdefaultInitRef
  corbaloc:iiop:10.1.1.29:50509 -executionname VR-Link
```

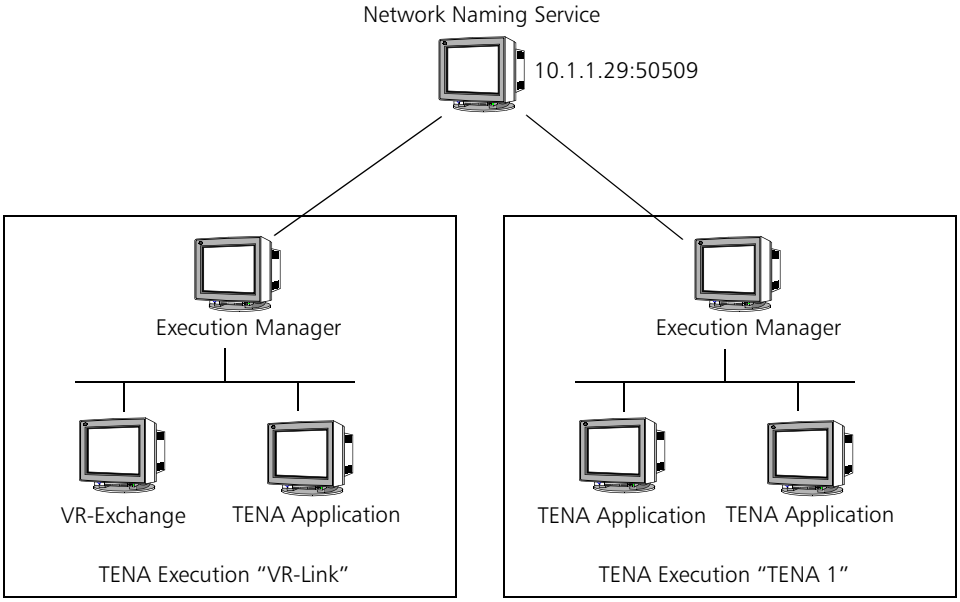
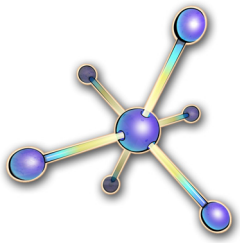


Figure B-2. TENA executions and Execution Manager



MÄK Products Glossary

This glossary defines terms used in MÄK product documentation.

absolute dead reckoning

A method of dead-reckoning in which an application 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.

AMO

See Application Management Object.

Application Management Object

An Application Management Object (AMO) is a standard TENA object that is supposed to be used by every TENA application. It gives other applications information about the local application. The VR-Link exercise connection, by default, publishes an AMO object, updates it at a specified rate (the rate is part of the AMO's state repository; the default is 30 seconds), and updates the number of objects published and proxies (TENA reflected objects) held.

API

Application Programming Interface.

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, an application uses Euler angles or a rotation matrix that rotates from a reference frame to the body coordinate frame.

channel

A channel renders the view of an observer in a 3D visualization application.

clipping

A feature that prevents the display of terrain and objects or parts of objects, that are closer than or farther than specific distances from the observer.

clipping planes

The range of distances from the observer (near clipping and far clipping) in which an application clips objects.

culling

The process of discarding database objects which are not within view, and therefore need not be rendered and displayed.

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 an application 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 former name of the executive secretariat for the Executive Council on Modeling and Simulation (EXCIMS), which provided a full-time focal point for information concerning Department of Defense modeling and simulation (M&S) activities. Renamed Modeling and Simulation Coordination Office. Currently the MSCO promulgates M&S policy, initiatives, and guidance to promote cooperation among DoD components to maximize efficiency and effectiveness.

DIS

Distributed Interactive Simulation.

DIS data packets

See Protocol Data Unit.

display engine

An object in an application that can render 3D data.

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 for use by the Department of Defense.

DMSO

See 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 maintenance

A process by which the MÄK Data Logger compensates for discontinuities following a time jump. The Logger sends out interim state messages for entities that were present before the time jump so that they do not time out before the next update message arrives.

entity-type

A list of seven components as specified by the DIS protocol and the HLA 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.

environmental process

According to the IEEE 1278.1a specification (DIS), environmental process PDUs communicate simple environment variables, small scale environmental updates, and embedded processes.

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.

execution

The TENA term for one or more interacting simulation applications.

Execution Manager

An Execution Manager governs a TENA execution for applications joining, resigning, or changing subscriptions to that execution.

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. (All MÄK products are VR-Link applications.)

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 applications also read 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

The federation execution represents the actual operation, over time, of a subset of the federates and the RTI initialization data taken from a particular federation. A federation execution is the logical equivalent of a DIS simulation exercise.

field of view

Controls the perspective of the observer.

A wide field of view creates an effect like that of a wide-angle camera lens. Objects appear smaller and farther away from the observer, since the observer coverage spans a wider area. Depth become exaggerated.

A narrow field of view creates an effect like that of a telephoto lens. Objects appear larger and closer to the observer, and the overall scene depth appears flattened. The distances between objects appears compressed.

filter range

A setting that prevents distant entities from being processed while allowing distant terrain to appear normally.

FOM

See Federation Object Model.

frame rate

The rate at which the application displays updated images.

ground clamping

A process by which the a 3D visualization application keeps an entity anchored to the surface of its terrain database, regardless of the altitude data contained in its entity state message.

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. In MÄK applications, geodetic coordinates consist of latitude and longitude in radians, and altitude in meters above the reference ellipsoid.

gridded data

Data that has been processed in a rectangular array of points, in X, Y or latitude/longitude, at which single data values define a two dimensional function. According to the IEEE 1278.1a specification, gridded data transmits information about large-scale or high-fidelity spatially and temporally varying ambient fields and about environmental processes and features.

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.

head-up display

A set of indicators and readouts superimposed onto a graphics display. Also called an overlay.

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 U. S. Department of Defense (DOD)-wide initiative to provide an architecture to support interoperability and reuse of simulations. The HLA supersedes DIS for the DOD.

HLA

See High-Level Architecture.

interaction

A message describing a simulation event. An interaction describes an event, it does not update an object's state.

Logger control PDUs

A set of DIS PDUs or HLA interactions that let you control the MÄK Logger remotely.

logical range

A suite of TENA Resources, sharing a common object model, that work together for a given range event. Similar in concept to the HLA Federation.

Local RTI Component

The libraries on a computer that implement an RTI. A federate communicates with the HLA network through the LRC.

Logical Range Object Model

The data definition file for TENA exercises. It contains the set of object and message definitions that may be used in the exercise.

LRC

See Local RTI Component.

LROM

See Logical Range Object Model.

MÄK Technologies Lisp

An adaptation of the Lisp language used in configuration files for MÄK products.

message

A general term used to refer to DIS PDUs and HLA interactions and state updates. In TENA, a message is similar to an HLA interaction.

Modeling and Simulation Coordination Office

The executive secretariat for the Executive Council on Modeling and Simulation (EXCIMS), which provided a full-time focal point for information concerning Department of Defense modeling and simulation (M&S) activities. The MSCO promulgates M&S policy, initiatives, and guidance to promote cooperation among DoD components to maximize efficiency and effectiveness. (Formerly DMSO)

MSCO

Modeling and Simulation Coordination Office. (Formerly DMSO.)

MTL

See MÄK Technologies Lisp.

Network Naming Service

Acts as a registry for TENA Execution Managers.

NNS

See Network Naming Service.

orientation clamping

Adjusts the pitch and roll of an entity so that it appears properly seated when the terrain is inclined. For example, if a tank is moving horizontally across the face of a hill, orientation clamping prevents the tank from appearing level, and therefore, partially embedded into the hillside. Used with ground clamping.

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. In HLA, 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.

observer

In MÄK 3D applications, the point of view into the scene. (Called the eyepoint in MÄK Stealth 6.x and earlier.)

PDU

See Protocol Data Unit.

Protocol Data Unit

A unit of data message (packet) that is passed on a network between DIS simulation applications.

proxy

In TENA, a reflected object (Stateful Distributed Object).

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.

range resource

A participant in a TENA execution (similar in concept to the HLA federate).

Realtime Platform Reference FOM

An HLA reference FOM based on the DIS protocol.

recording

A Data Logger file that stores a history of the interactions in a simulation for playback.

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 an application approximates the location of an object based on the local time of receipt of the last state update message.

remote display engine

A display engine running in an application that is separate from the master application, and which is controlled by the master application.

RID file

See RTI Initialization Data.

RTI Initialization Data

The initialization data required by the RTI for operation.

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

See Realtime Platform Reference FOM.

RTI

See 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.

SDO

See Stateful Distributed Object.

servant

A published object (SDO) in TENA.

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

In VR-Forces, simulation time is used in dead-reckoning of remote entities and thresholding of local entities. Typically, 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

See Simulation Interoperability Standards Organization.

smooth period

The period of time over which trajectory smoothing takes place.

smoothing

A method of ensuring that transitions from an entity's dead-reckoned position to its actual position are not 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.

Stateful Distributed Object

An object in a TENA exercise.

tape

An alternative term for a Logger recording. Not a physical magnetic tape.

TENA

See Test and Training Enabling Architecture.

TENA Middleware

"The software that links range resources together into a logical range. The "TENA Middleware" is the software component that is in between (that is, in the middle of) range resources during a logical range execution." – from www.tena-sda.org

terrain following

In a 3D visualization application, causes the observer (eyepoint) to maintain a constant distance above the terrain surface. The observer's height changes in tandem with the peaks and valleys as it passes over the geography.

Test and Training Enabling Architecture

Test and Training Enabling Architecture (TENA) is an interoperability architecture used to pass data in the form of object updates and messages (like HLA objects and interactions) from one application to another.

timeout

The period of time in which an application continues to display an entity after that entity's update messages have stopped appearing on the network. Time-outs are usually not used in HLA because there is no set frequency (no heartbeat) for transmitting messages.

timescale

A factor by which time is accelerated or slowed during playback of a Data Logger recording.

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.

trajectory smoothing

A method used by to smooth positional discontinuities that could occur when new state updates arrive.

UDP port

A network channel through which an application 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

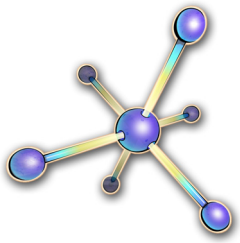
See Universal Transverse Mercator.

view control messages

A set of programmatic messages that let you control the VR-Vantage remotely.

view floor

In MÄK Stealth 6.x and earlier, a minimum height above the terrain in Absolute or Track mode, below which the observer is not allowed to go. The view floor is measured relative to the terrain surface.



Index

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X-Y-Z

Link - Simulate - Visualize

68 MOULTON STREET CAMBRIDGE, MA 02138 617.876.8085 www.mak.com