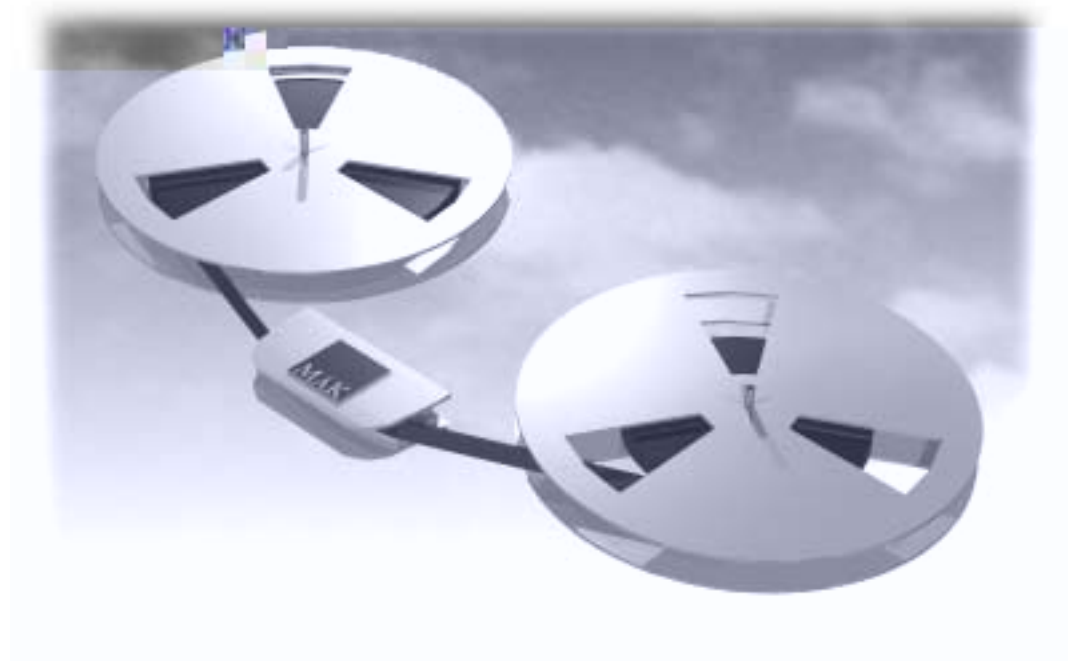




Data Logger

User's Guide



Data Logger

User's Guide

Copyright © 1997 - 2001 MÄK Technologies
All rights Reserved. Printed in the United States.

Under copyright laws, no part of this document may be copied or reproduced in
any form without prior written consent of MÄK Technologies, Inc.

VR-Link® is a registered trademark of MÄK Technologies, Inc.
All other trademarks are owned by their respective companies.

MÄK Technologies
185 Alewife Brook Parkway
Cambridge, MA 02138 USA

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

info@mak.com

www.mak.com

Revision LOG-3.7-1-010607



Contents

Preface

How the Manual is Organized	ix
Online Help	x
Other MÄK Products	xi
How to Contact Us	xii
Document Conventions	xiii
Hardware and Operating System Naming Conventions.....	xiii
Mouse Button Naming Conventions	xiv

Chapter 1 Introduction

1.1. Logger Features	1-2
1.1.1. Logger Interfaces	1-3
1.1.2. Differences Between UNIX and Windows Versions	1-3
1.1.3. Sample Logger Files	1-3
1.1.4. Sample Annotation Files	1-4
1.1.5. Sample Programs	1-4

Chapter 2 Installing and Configuring the Logger

2.1. System Requirements	2-3
2.1.1. HLA Support	2-3
2.2. Installing the Logger	2-4
2.2.1. Installing the Logger On A UNIX System	2-4
2.2.2. Installing the Logger On A PC	2-4
2.2.3. The Logger Directory Structure	2-6

2.3. Setting Up and Using the FLEXlm License Manager	2-8
2.3.1. The FLEXlm Client-Server Architecture	2-8
2.3.2. Installing and Configuring the License Manager	2-9
2.3.3. Installing the License Manager Software	2-9
2.3.4. Requesting A License File	2-10
2.3.5. Putting the License File in the flexlm Directory	2-11
2.3.6. Setting the MAKLMGRD_LICENSE_FILE Variable	2-11
2.3.7. Adding Additional License Files	2-13
2.3.8. Running the License Server	2-13
2.3.9. Troubleshooting the FLEXlm License Manager	2-14
2.4. Installing An RTI	2-17
2.4.1. Installing the MÄK Real-time RTI	2-17
2.4.2. Installing the DMSO RTI	2-18
2.5. Configuring the Logger for HLA	2-20
2.5.1. Selecting A FOM Mapper	2-20
2.5.2. Specifying A FED File for the FOM Mapper	2-21
2.5.3. Specifying Class and Interaction Subscriptions	2-22
2.6. Configuring the Logger for DIS	2-25
2.6.1. Specifying A Destination Address for Outgoing Data	2-25
2.6.2. Subscribing to Multicast Groups	2-25
2.6.3. Specifying A UDP Port	2-25

Chapter 3 Logger Concepts

3.1. What the Logger Does During Recording and Playback	3-2
3.1.1. What The Logger Does During Recording	3-2
3.1.2. What The Logger Does During Playback	3-3
3.2. Timescale and Direction During Playback	3-4
3.3. Compensating for Timescale Effects During Playback	3-5
3.3.1. Generating Interim PDUs During Slow-Speed DIS Playback ...	3-5
3.3.2. Updating Entity Information Quickly After A Time Jump	3-6
3.3.3. Velocity and Acceleration Scaling	3-7
3.3.4. How Reverse Playback Affects Dead-Reckoning	3-7
3.4. Specifying the End-of-File Action	3-7

Chapter 4 Using The Graphical User Interface

4.1. The Logger Windows	4-2
4.1.1. Starting the Logger	4-2
4.2. Recording Exercises	4-4
4.3. Playing Logger Recordings	4-4
4.4. Navigating to A Specific Time in A Recording	4-5
4.4.1. Using the Set Time Feature	4-5
4.4.2. Using the Skip Time Feature	4-6

4.5. Annotating A Logger Recording	4-7
4.5.1. Creating An Annotation	4-7
4.5.2. Saving An Annotation File	4-8
4.5.3. Opening An Annotation File	4-8
4.5.4. Importing An Annotation File	4-9
4.5.5. Moving to An Annotation Point	4-9
4.5.6. Editing An Annotation File	4-9
4.5.7. Deleting An Annotation	4-10
4.5.8. Clearing the Annotation List	4-10
4.6. Setting the Timescale and Direction	4-10
4.6.1. Using the Timescale Slider	4-11
4.6.2. Setting A Timescale at Startup	4-11
4.6.3. Entering A Timescale Value Numerically	4-11
4.7. Setting Configuration Options	4-12
4.8. Summary of Startup Options for the Logger	4-14

Chapter 5 Using The Command-Line Interface

5.1. Starting the Logger in Command-Line Mode	5-2
5.2. Recording Exercises	5-2
5.2.1. Beginning A Recording Session in Record Mode	5-2
5.2.2. Logger Run-time Commands for Recording	5-3
5.2.3. Displaying the Status of A Recording	5-3
5.3. Playing Back Recordings	5-4
5.3.1. Basic Logger Playback Commands	5-4
5.3.2. Displaying Playback Status	5-5
5.3.3. Beginning A Playback Session in Play Mode	5-5
5.4. Navigating Time	5-5
5.4.1. Controlling End-of-File Behavior	5-6
5.5. Setting Playback Timescale and Direction	5-6
5.6. Controlling the DIS Heartbeat and Timeout	5-7
5.6.1. Setting the Heartbeat Rate	5-7
5.6.2. Setting the Timeout	5-7
5.7. Using A Script to Control Recording and Playback	5-8
5.7.1. Stopping A Script	5-8
5.8. Summary of Logger Run-Time Commands	5-9

Chapter 6 Remote Control

6.1. Remote Control Messages	6-2
6.2. Using Logger Commands for Remote Control	6-3
6.2.1. Deleting Logger Files	6-3
6.3. Example of Remote Control	6-5

Chapter 7 Using The Logger API

7.1. Introduction	7-2
7.2. The DtLogger class	7-3
7.2.1. Including the Logger in an Application	7-3
7.2.2. Controlling the Logger Time Clock	7-6
7.2.3. Controlling Logger Files	7-7
7.2.4. Controlling the Logger's Configuration and Status	7-8
7.2.5. Controlling Logger Annotations	7-9
7.2.6. Setting Protocol-Specific Parameters	7-10
7.2.7. Managing Pipeline Objects	7-11
7.3. The Logger Architecture	7-12
7.4. Logger Pipeline Elements	7-13
7.4.1. Pipeline Element Member Functions	7-14
7.4.2. Default Pipeline Elements	7-17
7.4.3. Example of Overriding A Pipeline Element	7-17
7.5. Creating Plug-ins	7-22
7.5.1. Initializing A Plug-in	7-22
7.5.2. Modifying the Pipeline Using A Plug-in	7-23
7.5.3. Overriding <i>DtLogger</i> from A Plug-in	7-25
7.5.4. Loading Plug-Ins	7-26
7.6. Compiling and Linking with the Logger API	7-26

Chapter 8 Using the Logger File API

8.1. Using the Logger File API	8-2
8.1.1. Reading A Logger File	8-2
8.1.2. Creating An LtLoggerPlaybackFile	8-3
8.1.3. Reading Data	8-4
8.1.4. Writing A Logger File	8-5
8.1.5. Creating An <i>LtLoggerRecordFile</i>	8-5
8.1.6. Writing Data	8-6
8.1.7. Interpreting LtPacketInfo	8-6
8.2. Editing Logger Files	8-7
8.3. Compiling and Linking with the Logger Library	8-9

Chapter 9 Using Logger Tools

9.1. Viewing Logger Files with lgrDump	9-2
9.1.1. lgrDump for HLA	9-3
9.1.2. How lgrDump Displays Unexpected Message Types	9-4
9.1.3. Displaying Raw Data	9-4
9.2. Concatenating Files with lgrCat	9-5
9.2.1. Inserting Time Between Concatenated Files	9-6
9.2.2. Processing Input from Other Programs	9-6
9.3. Merging Files with lgrMerge	9-7
9.4. Using lgrOffset to Shift Messages in Time	9-8

9.5. Using lgrTrim to Extract Part of A File	9-9
--	-----

Appendix A Troubleshooting

A.1. Troubleshooting Logger Network Communications (DIS Only)	A-2
A.1.1. Using the Correct Broadcast Address and UDP Port	A-2

Appendix B Logger MTL Parameters

B.1. MÄK Technologies Lisp (MTL) Files	B-2
B.2. Logger MTL Parameters	B-2

Glossary

Index

Quick Reference Card



Preface

The MÄK Data Logger (Logger) records network activity of HLA or DIS exercises and plays it back for review. The Logger supports a graphical user interface, a command-line interface, and several APIs.

This manual is for persons who will install the Logger, for persons who will record or play back simulations, and for persons who will develop plug-ins or applications using the Logger APIs. It assumes that you are familiar with your operating system and that you know how to work in your computer's windowing environment.

Please consult the *Release Notes* for your version of the Logger for release-specific information and updates that occurred after this manual went to press.

How the Manual is Organized

The manual is organized as follows:

Chapter 1, [Introduction](#), describes the Logger's features.

Chapter 2, [Installing and Configuring the Logger](#), explains how to install the Logger and related software such as the FLEXlm license manager and the RTI. It also explains how to configure the Logger for use with HLA or DIS.

Chapter 3, [Logger Concepts](#), provides an overview of what the Logger does.

Chapter 4, [Using The Graphical User Interface](#), shows how to record and play back HLA/DIS network activity using the Logger's graphical user interface.

Chapter 5, [Using The Command-Line Interface](#), explains how to run the Logger from the command-line.

Chapter 6, [Remote Control](#), explains how to control the Logger using view control messages sent from other applications.

Chapter 7, [Using The Logger API](#) describes the Logger API, which allows you to add Logger functionality to your applications, and write plug-ins, which allow you to add new features to the Logger application.

Chapter 8, [Using the Logger File API](#) describe the Logger File API, which allows you to read and write Logger recordings from another application.

Chapter 9, [Using Logger Tools](#), describes the utility programs that combine, modify, and inspect Logger recordings.

Appendix A, [Troubleshooting](#), shows how to correct network problems that may prevent the Logger from communicating properly.

Appendix B, [Logger MTL Parameters](#) lists and describes MTL parameters that you can use with the Logger.

The [Glossary](#), defines terms used in the manual.

There is a [Quick Reference Card](#) at the end of the manual that you can remove and keep near your computer for convenience. It lists startup options, and the commands that you can use with the command-line interface.

Online Help

The Logger application has online help. The online help is written in HTML. It requires a web browser that supports javascript and Java, which must be enabled.

Other MÄK Products

The Logger is a member of the MÄK Technologies line of software products designed to streamline the process of developing and using networked simulated environments.

In addition to the Logger, the MÄK Technologies product line includes:

- ♦ The 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 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/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.

VR-Link is the basis of the other MÄK products.

- ♦ The MÄK Real-time RTI. An RTI (Run-Time Infrastructure) is required to run applications using the High Level Architecture (HLA). The MÄK Real-time RTI is the first commercial run-time infrastructure for HLA simulations. It is optimized for real-time simulations.
- ♦ MÄK VR-Forces. VR-Forces is a computer generated forces application and toolkit. It provides an application with a graphical user interface, that gives you a two dimensional view of a terrain database.

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.

- ♦ The MÄK Stealth. The Stealth displays a realistic, three-dimensional view of your virtual world. You can view this world from the inside of a simulated moving vehicle, or place the eyepoint at another moving or stationary location. The Stealth lets you switch rapidly among several predefined viewpoints while the simulation is underway. The Stealth runs on PCs, SGI IRIX, and Linux computers and supports multichannel views.
- ♦ The MÄK Plan View Display. The Plan View Display (PVD) provides a two-dimensional, “big picture” of a virtual environment. It shows simulated entities, such as vehicles, soldiers, and tanks moving over a 2D-map display.

- ♦ The MÄK Gateway. The Gateway is a translator application that allows DIS exercises to operate as members of HLA federation executions that use the RPR FOM, thereby preserving the value of simulation code written to support DIS. The MÄK Gateway is a standalone application and does not require you to write additional code for its use.

How to Contact Us

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

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

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

Send postal correspondence to:	MÄK Technologies 185 Alewife Brook Parkway Cambridge, MA, USA 02138
--------------------------------	---

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

Document Conventions

This document uses the following typographic conventions:

Monospaced Type	Indicates commands or values you enter.
<i>Monospaced Italic</i>	Indicates 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.
/	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.
Bold type	Indicates a new term.
<i>Italic</i>	Indicates a file name, pathname, or a class name.
sansSerif	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 Logger home directory. For example, the directory *logger/data* appears in the text as *./data*.

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, typically 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.



Introduction

This chapter describes the basic features of the Logger.

Logger Features	1-2
Logger Interfaces	1-3
Differences Between UNIX and Windows Versions.....	1-3
Sample Logger Files	1-3
Sample Annotation Files.....	1-4
Sample Programs	1-4

1.1. Logger Features

The Logger records network activity of HLA and DIS exercises¹ and plays them back for review.

DIS only

In record mode, the Logger monitors all DIS traffic on a specified **UDP** port and stores it in a file. In playback mode, the Logger transmits **DIS data packets** stored in a Logger **recording**.

HLA only

The HLA version of the Logger performs recording and playback while acting as a separate federate. In record mode, it subscribes to all **objects**, attributes, and **interactions** in the RPR FOM, and records updates, interactions, and object removals that it perceives through the RTI. It records values for all published attributes of an object by requesting attribute updates as it discovers each new object.

In playback mode, the HLA Logger acts as a **federate** that owns each of the objects for which updates appear in the playback file. The Logger sends interactions and updates only for classes that other federates are subscribed to. The Logger serves as a proxy simulator, answering requests for attribute updates and making objectRemove RTI calls when it encounters removals in the recorded file. In addition, just as in DIS, velocities, accelerations, and angular velocities are scaled in outgoing messages when the Logger's time scale is not equal to zero.

The Logger uses the same object IDs as those used by the federates when the **federation execution** was recorded.

The Logger does not store handles, so you do not need to use identical FED files, but it is recommended.

The Logger is available for UNIX platforms and for different versions of MS-Windows.

The Logger software package includes the following:

- ♦ The Logger, a data-recorder/player that you can control through a VCR-style graphical user interface, a command-line interface, or programmatically.
- ♦ A set of Logger Tools that let you combine, modify and inspect recorded Logger files. You can merge the activity from several exercises into a single Logger file, extract a portion of an exercise to a new file, and perform other file operations.
- ♦ Libraries and header files that support APIs that allow to write plug-ins for the Logger and allow you to embed Logger functionality in other applications.
- ♦ A set of sample Logger recordings (also called Logger tapes).

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

1.1.1. Logger Interfaces

The Logger provides the following interfaces:

- ♦ A graphical user interface (GUI) that uses VCR-style buttons to control the recording and playback of exercises.
- ♦ A command-line interface.
- ♦ APIs that lets you control the Logger through plug-ins, external applications, or over a network.

These interfaces have nearly identical capabilities. They let you:

- ♦ Start, stop, or pause when playing a recording.
- ♦ Navigate to a specific time-point when playing a recording.
- ♦ Control playback speed and direction.
- ♦ Select the network addresses used for the sending and receiving of data.

You can use scripts in the command-line interface. In the GUI, you can annotate recordings. You can use all the interfaces at the same time.

1.1.2. Differences Between UNIX and Windows Versions

There are a few minor differences between the UNIX and Windows versions of the Logger.

Under UNIX, Logger commands are case-sensitive, while under Windows, only the option letters in commands are case-sensitive.

For consistency across platforms, the hyphen is always used to precede command-line arguments, rather than the slash character typically used with DOS command-line arguments.

1.1.3. Sample Logger Files

The Logger includes several sample recordings:

- ♦ *montyDemo.lgr* — contains DIS 2.0.4 data
- ♦ *montyDemoRPRxx.lgr* — contains HLA data
- ♦ *aggDemDIS.lgr* —demonstrates aggregate entities.

The *montyDemo* recording is an exercise in which two A-10s attempt to destroy seven ground targets near an airfield. It uses a portion of the Monterey database.



The *montyDemo* recording contains View Control messages, which cause the MÄK Stealth to follow the action automatically, switching among many predetermined viewpoints. For information about manually controlling the eyepoint in this demo, see your Stealth documentation.

1.1.4. Sample Annotation Files

The file *montyDemoRPR.ant* contains annotations that works with any of the *montyDemoRPRxx.lgr* recordings supplied with MÄK products.

1.1.5. Sample Programs

The Logger includes the following example programs:

- ♦ A plug-in that filters out air entities
- ♦ A simple example of using the Logger API
- ♦ An example of using the Logger File API.



Installing and Configuring the Logger

This chapter explains how to install the Logger and related software.

System Requirements	2-3
HLA Support	2-3
Installing the Logger	2-4
Installing the Logger On A UNIX System	2-4
Installing the Logger On A PC	2-4
The Logger Directory Structure.....	2-6
Setting Up and Using the FLEXlm License Manager	2-8
The FLEXlm Client-Server Architecture	2-8
Installing and Configuring the License Manager.....	2-9
Installing the License Manager Software	2-9
Requesting A License File	2-10
Putting the License File in the flexlm Directory	2-11
Setting the MAKLMGRD_LICENSE_FILE Variable	2-11
Adding Additional License Files.....	2-13
Running the License Server	2-13
Troubleshooting the FLEXlm License Manager	2-14
Installing An RTI.....	2-17
Installing the MÄK Real-time RTI	2-17
Installing the DMSO RTI	2-18
Configuring the Logger for HLA	2-20
Selecting A FOM Mapper	2-20
Specifying A FED File for the FOM Mapper.....	2-21
Specifying Class and Interaction Subscriptions	2-22
Configuring the Logger for DIS.....	2-25
Specifying A Destination Address for Outgoing Data	2-25

Subscribing to Multicast Groups	2-25
Specifying A UDP Port	2-25

2.1. System Requirements

Please refer to the *Release Notes* for your version of the Logger for current system requirements.

2.1.1. HLA Support

This section describes compatibility and support issues for running the Logger in HLA. Please refer to the *Release Notes* and any supplementary documentation for your version of the Logger for current information about FOM support.

FOM Flexibility

You can change the FED file between recording and playback. As long as the FED file you are using on playback contains object and interaction classes, and attributes and parameters with the same names and format as those that were recorded, you can play back your file.

Classes, attributes, and parameters can be rearranged in the FED file, and new ones can be added. If any class names have been deleted from the FED file between recording and playback, recorded interactions and updates for those classes will not be included in the Logger's outgoing data stream. If any attributes or parameter names have been deleted from the FED file, those attributes and parameters do not appear in outgoing updates and interactions.

Backward Compatibility

Different versions of the RPR FOM represent many attributes and parameters differently. They are essentially different protocols. Therefore, you may not be able to play back a Logger file recorded with previous versions of the Logger in the same federation execution as applications based on the current RPR FOM.

Please refer to the *Release Notes* and any supplementary documentation for the latest release of the Logger for current information about backward compatibility.

2.2. Installing the Logger

A complete Logger installation involves installing the Logger itself, the FLEXlm license manager, and (for use with HLA) a Runtime Infrastructure (RTI). These tasks are described separately in the rest of this chapter.

2.2.1. Installing the Logger On A UNIX System

This section explains how to install the Logger on UNIX systems from CD-ROM or from an ftp download.

Before you install the Logger, read the *Release Notes* to see if there are any special instructions for installation.

Installing the Logger from CD-ROM

To install the Logger:

1. Mount the CD-ROM.
2. Run:
`./install`
3. Follow the on-screen instructions. You can install the Logger into any directory for which you have write permissions.

Installing the Logger from A Downloaded File

If you obtained the Logger via ftp, you received a compressed tar file. To install the Logger:

1. Create the directory in which you want to install the Logger.
2. Copy the tar file to the install directory.
3. Uncompress the tar file:
`gunzip product.tar.gz`
4. Untar the file:
`tar -xvf product.tar`

2.2.2. Installing the Logger On A PC

This section explains how to install the Logger from CD-ROM or from an ftp download on a PC.

Before you install the Logger, read the *Release Notes* to see if there are any special instructions for installation. If you are going to use the Logger with HLA, you may need to know which version of the RTI you plan to use.

Installing the Logger from A CD

The Logger may be supplied to you on an “All Products” CD with a common installation interface or a CD that does not support common installation. To install the Logger from a CD:

1. Insert the Logger CD into your CD-ROM drive.
 - ♦ If this is an All Products CD, and the autorun feature is enabled, the MÄK PC Products Installation window opens. The PC Products Installation window will be 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 CD drive, double-click *makInst.exe*. The MÄK PC Products Installation window opens.
 - ♦ If this is not an All Products CD, and autorun is enabled, an installation program may start immediately. Follow the instructions of the installation program.

If the installation does not start automatically, open the Windows Explorer and select the CD drive. In the root directory for the Logger, double-click *setup.exe*. Go to step 3.
2. In the MÄK PC Products Installation window, click Logger.

i

The PC Products Installation window may list products by the RTI that they support. If you plan to use the HLA version of the Logger, install the version that supports the RTI that you plan to use. You can install both versions if you have enough disk space. Both versions support DIS.

3. Follow the instructions of the installation program.



Figure 2-1. MÄK PC Products Installation window

Installing the Logger from A Downloaded File

The Logger is distributed via ftp as a zip file.

To install the Logger 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.

2.2.3. The Logger Directory Structure

The Logger installation process creates a directory structure like the one shown in [Figure 2-2](#).

Logger home directory.	loggerx.x.x
Executables for DIS.	bin5
Executables for HLA with RPR FOM.	binRPR
Sample Logger files (PCs only).	data
Documentation in PDF format.	doc
Example programs	examples
License manager files.	flexlm
Online help	help
Header files for Logger API.	include
Library files for Logger API.	lib
Sample Logger files (UNIX only).	logger_tapes
Plug-ins and example plug-ins.	plugins

Figure 2-2. Logger directory structure

[Table 2-1](#) describes the contents of the Logger directories.

Table 2-1: Logger directory structure

Directory	Contents
<i>bin5</i>	Executable for DIS versions 2.0.4, 2.1.4, IEEE 1278, and IEEE 1278.1a.
<i>binRPR</i>	Executable for use with the HLA.
<i>data (on PCs)</i> <i>logger_tapes (UNIX)</i>	Sample Logger files.
<i>doc</i>	Documentation in Adobe Portable Document Format (PDF).
<i>examples</i>	Examples of the Logger API and the Logger File API.
<i>flexlm</i>	License management software.

Table 2-1: Logger directory structure

Directory	Contents
<i>help</i>	Online help.
<i>include</i> and <i>lib</i>	These directories contain header files and libraries for the Logger API and Logger File API.
<i>plugins</i>	This directory and its subdirectories contains plug-ins and example code.

2.3. Setting Up and Using the FLEXlm License Manager

The Logger, like other MÄK Technologies products, uses the FLEXlm license manager. Before you can use the Logger, you must obtain a valid license file and configure the license server and client machines. (For general information about FLEXlm, see www.globetrotter.com.) Contact the MÄK Technologies sales department for information about special licensing agreements.

2.3.1. The FLEXlm Client-Server Architecture

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

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

Licenses “float”, so you can install product software on as many computers as you want, but at any given time, you can run only as many instances of a product as you have licensed.



The license key in the license file is tied to the host ID of the server machine. Therefore, if you decide to change the server (or replace the network card), you need to get a new license file. Then, you must update the `MAKLMGRD_LICENSE_FILE` environment variable on every client. Depending on how many clients you have, this may not be a trivial process. Keep this in mind when you choose the machine to use as a license server.

Figure 2-3 illustrates the client-server architecture for FLEXlm and MÄK products.

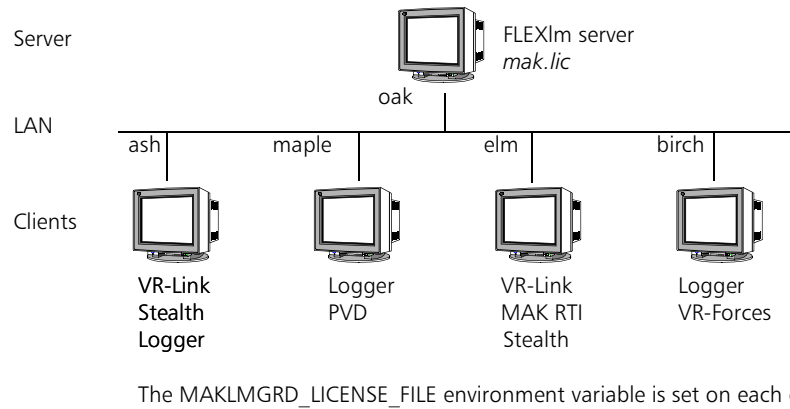


Figure 2-3. FLEXlm example architecture

2.3.2. Installing and Configuring the License Manager

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

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

If you have MÄK Technologies products installed and you purchase additional licenses or products, you must get a license file for the new products or licenses. For this procedure, see [“Adding Additional License Files”](#), on page 2-13.

2.3.3. Installing the License Manager Software

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

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

2.3.4. Requesting A License File

When you request a license file, you need to know the:

- ♦ Host ID of the license server
- ♦ Name of the license server
- ♦ MÄK invoice number of your purchase
- ♦ Basic contact information for you and your company
- ♦ Platforms for which you have purchased MÄK products
- ♦ Number of licenses for each platform and product.

When you have this information, go to <http://www.mak.com/licenses.htm> and complete the license key request form. MÄK will e-mail you a file named *mak.lic*.

Identifying the Host ID

To find out your host ID:

1. On the license server, in a command window, change to the *flexlm* directory.
2. On UNIX, execute *lmhostid*. This program generates a unique number that MÄK uses to generate your license activation codes. Write down the number.

On Windows platforms, execute *lmhostid.bat*. A command window opens and displays the host ID. Write down the number.



The host ID is keyed to your network card, so if you change your network card, you will need to request a new license file.

Identifying the License Server's Name

To find out the name of the server machine follow the procedure for your operating system:

UNIX

At a command prompt, type *hostname*.

Windows 95/98/NT

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

Windows 2000

1. Open the Properties dialog box for the My Computer icon.
2. Select the Network Identification tab. The machine name is listed in the Computer Name field.

2.3.5. Putting the License File in the *flexlm* Directory

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



If you already have a *mak.lic* file in the *flexlm* directory, do not overwrite the file. See the instructions in [“Adding Additional License Files”](#), on page 2-13.

2.3.6. Setting the MAKLMGRD_LICENSE_FILE Variable

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



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

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

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

Windows 95/98

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

```
set MAKLMGRD_LICENSE_FILE=@oak
```



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

Windows NT

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

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

@oak

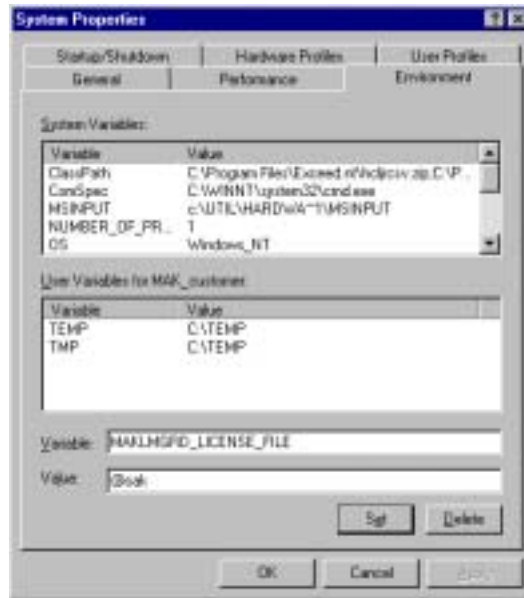


Figure 2-4. Setting an environment variable in Windows NT

Windows 2000

To add the MAKLMGRD_LICENSE_FILE in Windows 2000:

1. Right-click the My Computer icon on the Desktop.
2. On the pop-up menu, select Properties. The System Properties dialog box opens.
3. Click the Advanced tab.
4. Click the Environment Variables button. The Environment Variables dialog box opens.
5. Click the New button. The New System Variable dialog box opens.
6. In the Variable Name field, enter MAKLMGRD_LICENSE_FILE.

7. In the Variable Value field, enter `@server_name`, where `server_name` is the name of the license server.
8. Click OK to back out of each dialog box and set the variable.

UNIX

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

```
setenv MAKLMGRD_LICENSE_FILE @oak
```

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. See [“Running the License Server”](#), on page 2-13.

2.3.7. Adding Additional License Files

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

- > To request an additional license file, follow the same procedure as for your first license file. See [“Requesting A License File”](#), on page 2-10.

To add a new license file to your license server:

1. When you get the new license file, name it `mak2.lic`, or some other name ending in `.lic`, so that it has a different name from the license files currently in the `flexlm` directory on the license server.
2. Put the renamed file in the `flexlm` directory on the license server, with the other license files.
3. Restart the license server.

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

2.3.8. Running the License Server

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

- > To start the license server, on the server machine only, execute `runLm` from the `./flexlm` directory.
- > To obtain the current status of the server, run `lmstat`.



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

Shutting Down the License Server

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

2.3.9. Troubleshooting the FLEXlm License Manager

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

Unable to Get A License

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

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

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

On UNIX, enter the following commands:

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

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



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

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

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

```
kill -9 1385 878
```

where 1385 and 878 are process IDs.

On Windows NT, select a process in the Task Manager Processes window and click End Process.

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

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

Preventing Multiple License Manager Processes

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

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

License Manager Cannot Find MÄK License Management Executable

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

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

```
DAEMON maklmgrd .
```

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

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



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

The License Manager Reports an Unsupported Product

If you receive an error message indicating an unsupported product, you are trying to use a MÄK product for which you do not have a license. This may occur if you are running an HLA application with the MÄK Real-time RTI installed, but you do not have a license for the MÄK RTI. The license that you received for the products you purchased from MÄK may not include a license for the MÄK RTI. To get a license for the MÄK RTI, complete the license request form at http://www.mak.com/rti_license.htm.



You need a MÄK RTI license for each MÄK application that you run simultaneously. For example, if you are running the Logger, Stealth, and VR-Forces at the same time, you need three MÄK RTI licenses.

If you are trying to run your application with the DMSO RTI and you get the unsupported product message, it is possible that your application is finding the DLLs for the MÄK RTI before it finds the DMSO RTI. To fix this problem do the following:

- ♦ Make sure that the DMSO RTI path precedes the MÄK RTI path in your path environment variable.
- ♦ Check the directory in which the application is located to be sure that there are no DLLs from the MÄK RTI in that directory.

2.4. Installing An RTI

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



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

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

All MÄK products have been tested against the MÄK Real-time RTI (which comes with all MÄK products), and the DMSO RTI (which is available from the DMSO web site). For the most recent information about the RTI versions supported by the Logger, see the Product Versions page on the MÄK Technologies web site at: http://www.mak.com/product_version.htm.

2.4.1. Installing the MÄK Real-time RTI

The MÄK Real-time RTI is distributed with all MÄK products and you can install it when you install the Logger.

UNIX

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

PCs

Install the MÄK Real-time RTI from the PC Products Installation screen just as you installed the Logger.

Project files provided with the Logger (if any) point to the default installation directory for the RTI.

Configuring Your System to Use the MÄK Real-time RTI

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

- ♦ For IRIX6n32 use LD_LIBRARYN32_PATH
- ♦ On other UNIX platforms use LD_LIBRARY_PATH
- ♦ On PCs, use PATH.

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

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

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

Running Applications with the MÄK Real-time RTI

In most cases, you do not need an RTI server, such as the rtiexec, or a central application to use the MÄK Real-time RTI, however you can run the rtiexec if you want to. (It is required to use certain features of the MÄK Real-time 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, see “Setting Up and Using the FLEXlm License Manager”.) See the *MÄK RTI Reference Manual* for information about the rtiexec and changing the networking parameters such as multicast addresses and UDP ports.

Documentation for the MÄK Real-time RTI

The documentation for the MÄK Real-time RTI consists of:

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

2.4.2. Installing the DMSO RTI

You can get the DMSO RTI from DMSO's software distribution center (<http://sdc.dmsol.com>). (The URL for the DMSO Software Distribution Center has changed frequently. This reference was accurate at the time this manual was published.) Documentation for the DMSO RTI is provided on the MÄK product CD because the documentation applies to the MÄK Real-time RTI as well.

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

Configuring Your System to Use the DMSO RTI

Follow the configuration instructions provided by your version of the DMSO RTI.

Running Applications Using the DMSO RTI

To use the DMSO RTI, you need to run the RTI executive (rtiexec). This executable is in the *bin* directory or a subdirectory. Run the rtiexec only on one machine, regardless of how many HLA applications you are running.

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

2.5. Configuring the Logger for HLA

By default, the HLA Logger uses the federation execution name VR-Link.

> To specify a federation execution name, use the `-x` option at startup:

```
Logger -p my_exercise.lgr -x FedExName
```

where *FedExName* is the name of the federation execution.

2.5.1. Selecting A FOM Mapper

The Logger has built-in support for the RPR FOM. (See the *Release Notes* for your version of the Logger for a list of supported versions of the RPR FOM.) It also supports VR-Link's ability to support alternative FOMs through the FOM Mapper.

You can specify which FOM Mapping code to use with the `-f` startup option, in one of two ways:

```
Logger -f vLRPRxx
Logger -f library:init_data
```

If the argument to the `-f` option is one of the `vLRPRxx` keywords, the Logger uses VR-Link's built-in RPR FOM mapper, configured for the RPR FOM version specified by the keyword, for example, `vLRPR08` specifies RPR FOM 0.8.

If the argument to the `-f` option is not one of the keywords, it is interpreted as the name of a shared library (DSO or DLL) containing code for an alternate FOM Mapper that you would like the Logger to use, optionally followed by any initialization data the library requires. The library name and initialization data must be separated by a colon.

Examples

```
Logger -f vLRPR08
```

This command instructs the Logger to use the built-in RPR FOM Mapper, configured for RPR FOM version 0.8.

```
Logger -f myFomMapLib:ABC
```

This command instructs the Logger to use the FOM Mapper contained in a shared library called *myFomMapLib.so* (or *myFomMapLib.dll*). The string "ABC" is passed to the FOM Mapper creation function in the shared library as an initialization parameter.

2.5.2. Specifying A FED File for the FOM Mapper



The `-f` option only tells the application which FOM Mapping code to use. It does not ensure that you are using the right FED file for the FOM Mapper. Make sure that you are using a FED file that is consistent with the FOM Mapper you have chosen.

By default the application uses a FED file called *VR-Link.fed*, but you can specify an alternate FED file using the `-x` option. The `-x` option actually indicates the name of the federation execution, but we append *.fed*, and use that as the FED file name as well.

MÄK products come with a FED file for each version of the RPR FOM that they support. (See the *Release Notes* for your version of the Logger for a list of FED files supplied.)

The following example instructs the Logger to use the RPR FOM 0.8 FOM Mapping code (`-f vLRPR08`), and also instructs it to use the *VR-Link08.fed* file (`-x VR-Link08`).

```
Logger -f vLRPR08 -x VR-Link08
```

2.5.3. Specifying Class and Interaction Subscriptions

By default the Logger subscribes to all classes in your FED file except *RTIPrivate* and any MOM object or interaction. However, you can specify the set of classes and interactions that the Logger subscribes to by creating a subscription control file and loading it when you start the Logger.



On PCs, you can specify subscriptions through the GUI. See [“Specifying the Subscription List from the Logger GUI”](#), on page 2-24.

To create a subscription control file:

1. Start the Logger with the `-F` option and specify the name of a file to write the subscription list to, for example:

```
Logger -x VR-Link -F sublist.mtl
```

The Logger creates the file based on the information in the FOM that you are using. The *sublist.mtl* file looks as follows for the VR-Link FED file:

```
;subscription lists
(setq subscribedClasses
  (list
    "ObjectRoot.EnvironmentProcess"
    "ObjectRoot.GriddedData"
    "ObjectRoot.BaseEntity"
    "ObjectRoot.BaseEntity.EnvironmentalEntity"
    "ObjectRoot.BaseEntity.BaseEntityOther"
    "ObjectRoot.BaseEntity.AggregateEntity"
    "ObjectRoot.BaseEntity.PhysicalEntity"
    "ObjectRoot.BaseEntity.PhysicalEntity.Platform"
    "ObjectRoot.BaseEntity.PhysicalEntity.Platform.Aircraft"
    "ObjectRoot.BaseEntity.PhysicalEntity.Platform.AmphibiousVehicle"
    "ObjectRoot.BaseEntity.PhysicalEntity.Platform.GroundVehicle"
    "ObjectRoot.BaseEntity.PhysicalEntity.Platform.MultiDomainPlatform"
    "ObjectRoot.BaseEntity.PhysicalEntity.Platform.Spacecraft"
    "ObjectRoot.BaseEntity.PhysicalEntity.Platform.SubmersibleVessel"
    "ObjectRoot.BaseEntity.PhysicalEntity.Platform.SurfaceVessel"
    "ObjectRoot.BaseEntity.PhysicalEntity.Lifeform"
    "ObjectRoot.BaseEntity.PhysicalEntity.Lifeform.Human"
    "ObjectRoot.BaseEntity.PhysicalEntity.Lifeform.NonHuman"
    "ObjectRoot.BaseEntity.PhysicalEntity.CulturalFeature"
    "ObjectRoot.BaseEntity.PhysicalEntity.Expendables"
    "ObjectRoot.BaseEntity.PhysicalEntity.Munition"
    "ObjectRoot.BaseEntity.PhysicalEntity.Radio"
    "ObjectRoot.BaseEntity.PhysicalEntity.Sensor"
    "ObjectRoot.BaseEntity.PhysicalEntity.Supplies"
    "ObjectRoot.EmbeddedSystem"
    "ObjectRoot.EmbeddedSystem.Designator"
    "ObjectRoot.EmbeddedSystem.EmitterSystem"
    "ObjectRoot.EmbeddedSystem.RadioReceiver"
    "ObjectRoot.EmbeddedSystem.RadioTransmitter"
    "ObjectRoot.EmitterBeam"
    "ObjectRoot.EmitterBeam.RadarBeam"
    "ObjectRoot.EmitterBeam.JammerBeam"
  )
)
```

```
)

(setq subscribedInteractions
  (list
    "InteractionRoot.Acknowledge"
    "InteractionRoot.ActionRequest"
    "InteractionRoot.ActionResponse"
    "InteractionRoot.ActionRequestToObject"
    "InteractionRoot.ActionResponseFromObject"
    "InteractionRoot.AttributeChangeRequest"
    "InteractionRoot.AttributeChangeResult"
    "InteractionRoot.Collision"
    "InteractionRoot.Comment"
    "InteractionRoot.CreateObjectRequest"
    "InteractionRoot.CreateObjectResult"
    "InteractionRoot.CreateEntity"
    "InteractionRoot.Data"
    "InteractionRoot.DataQuery"
    "InteractionRoot.EventReport"
    "InteractionRoot.MunitionDetonation"
    "InteractionRoot.RadioSignal"
    "InteractionRoot.RadioSignal.ApplicationSpecificRadioSignal"
    "InteractionRoot.RadioSignal.DatabaseIndexRadioSignal"
    "InteractionRoot.RadioSignal.EncodedAudioRadioSignal"
    "InteractionRoot.RadioSignal.RawBinaryRadioSignal"
    "InteractionRoot.RemoveObjectRequest"
    "InteractionRoot.RemoveObjectResult"
    "InteractionRoot.RemoveEntity"
    "InteractionRoot.RepairComplete"
    "InteractionRoot.RepairResponse"
    "InteractionRoot.ResupplyCancel"
    "InteractionRoot.ResupplyOffer"
    "InteractionRoot.ResupplyReceived"
    "InteractionRoot.ServiceRequest"
    "InteractionRoot.SetData"
    "InteractionRoot.StartResume"
    "InteractionRoot.StopFreeze"
    "InteractionRoot.WeaponFire"
    "InteractionRoot.ViewControl"
  )
)
```

2. Edit the file by adding or removing the classes and interactions that you do not want the Logger to subscribe to.

3. Include the file in the *logger.mtl* configuration file by adding the following line.

```
(load "pathToSublistFile" "sublistfile")
```

where:

- *pathToSublistFile* is the relative or complete pathname of the directory in which the subscription list file is located.
- *sublistfile* is the name of the subscription list file.

4. To verify the subscription set is being used correctly, view the *logger.txt* output file. When the Logger is using a user-defined subscription set it writes to the file each class and interaction that it subscribes to.

Specifying the Subscription List from the Logger GUI

On PCs, you can control the subscription list from the graphical user interface.

To create a subscription control list:

1. In the Logger window, choose Options → Subscription Control. The Subscription Control dialog box opens. The Subscription Control window lists all object classes and interactions in the FOM. The objects and interactions that the Logger subscribes to are highlighted.
2. Edit the subscription list by clicking on the object classes or interactions that you want to subscribe to or remove from the list.
3. To save the edited subscription list, click Save. The Save As dialog box opens.
4. Enter a name for the subscription list and click Save.
5. Include the file in the *logger.mtl* configuration file by adding the following line.

```
(load "pathToSublistFile" "sublistfile")
```

where:

- ♦ *pathToSublistFile* is the relative or complete pathname of the directory in which the subscription list file is located.
- ♦ *sublistfile* is the name of the subscription list file.

2.6. Configuring the Logger for DIS

In DIS, the Logger can direct its output to a specific destination address during playback, and can listen (subscribe) to one or more multicast addresses during recording. You can also control which UDP port is used for playback or recording.

2.6.1. Specifying A Destination Address for Outgoing Data

Normally, the Logger sends outgoing data packets to the host machine's default broadcast address.

- > To specify a different destination address, use the `-A` command-line option, as in this example:

```
Logger -p my_exercise.lgr -A 123.234.123.234
```

The specified destination address may be a broadcast, multicast, or unicast address.

2.6.2. Subscribing to Multicast Groups

In record mode, the Logger can subscribe to multicast groups. When the Logger is subscribed to multicast groups, it also continues to receive data through the machine's broadcast and unicast addresses.

- > To subscribe to multicast groups, use the `-S` startup option. To subscribe to more than one multicast address, use the `-S` option several times in the same command:

```
Logger -r newfile.lgr -S 225.0.0.118 -S 225.0.0.202
```

Subscription pertains to record mode only.



Multicast subscription is a property of the host machine, not of individual applications. Therefore, if several applications are running on the same machine using the same port, and those applications subscribe to multicast groups, all the applications will receive packets for all multicast groups to which any application is subscribed. Accordingly, if the Logger is running on the same machine and using the same port as other applications that subscribe to multicast groups, the Logger may receive traffic intended for the other applications.

2.6.3. Specifying A UDP Port

By default, the Logger sends and receives data over UDP port 3000.

- > To specify the UDP port, start the Logger with the `-P` switch, for example:

```
Logger -P 2727
Logger -p mydemo.lgr -P 2727
```




Logger Concepts

This chapter describes Logger concepts that are common to all interfaces. For step-by-step instructions about how to operate the Logger, see Chapter 4, “[Using The Graphical User Interface](#)” and Chapter 5, “[Using The Command-Line Interface](#)”.

What the Logger Does During Recording and Playback	3-2
What The Logger Does During Recording	3-2
What The Logger Does During Playback	3-3
Timescale and Direction During Playback	3-4
Compensating for Timescale Effects During Playback.....	3-5
Generating Interim PDUs During Slow-Speed DIS Playback.....	3-5
Updating Entity Information Quickly After A Time Jump	3-6
Velocity and Acceleration Scaling	3-7
How Reverse Playback Affects Dead-Reckoning.....	3-7
Specifying the End-of-File Action	3-7

3.1. What the Logger Does During Recording and Playback

This section describes, in general, what happens when you record and play back exercises. [Figure 3-1](#) illustrates the Logger window in playback and record modes.

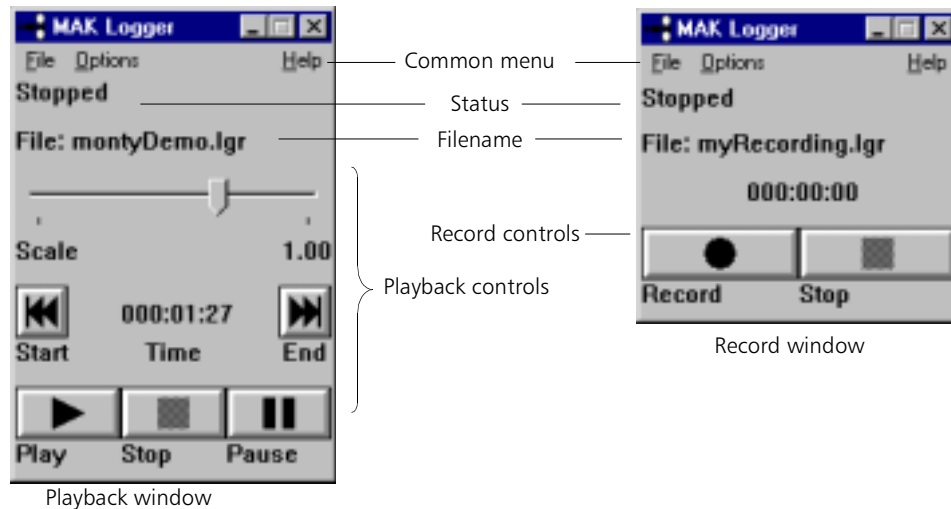


Figure 3-1. The Logger windows

3.1.1. What The Logger Does During Recording

In record mode, the Logger is in either a recording state or a stopped state. When stopped, the Logger does not record data and does not advance time.

DIS only

In record mode, the Logger records all DIS packets on UDP port 3000, without regard to exercise ID. You can specify a different UDP port at startup by using the `-P` argument, for example:

```
Logger -r newfile.lgr -P 2727
```

With each PDU it records, the Logger stores a timestamp indicating when the PDU occurred relative to the start of the recording.

HLA only

The Logger subscribes to objects, attributes, and interactions, and it records updates, interactions, and object removals that it perceives through the RTI. It requests updates as it discovers each new object, and records values for all published attributes of those objects. The Logger can subscribe actively or passively.

3.1.2. What The Logger Does During Playback

When the Logger is in playback mode, it is in either a playing state, a paused state, or a stopped state. When the Logger is playing, it reads messages from a recording and transmits them in sequence over the network. When the Logger is paused, it halts at the current timepoint in the file and stops sending data.

DIS only

When the Logger is paused, it repeatedly retransmits messages that represent the current state of each entity to maintain the DIS heartbeat (This retransmission is not needed in HLA).

During playback, you can control the playback speed and direction to produce fast motion, slow-motion, and reverse playback effects. For more information, see [“Timescale and Direction During Playback”](#), on page 3-4.

The Logger lets you navigate to a specified timepoint in a recording, or skip forward or backward by a specified time increment. For procedures, see [“Navigating to A Specific Time in A Recording”](#), on page 4-5 and [“Navigating Time”](#), on page 5-5.

DIS only

In DIS, when you start the Logger and specify a file for playback, the Logger sends the recorded PDUs to UDP port 3000. You can use the `-P` argument to specify a different port, for example:

```
Logger -p demofile.lgr -P 2727
```

For more information, see [“Specifying A UDP Port”](#), on page 2-25.

HLA only

In HLA, the Logger acts as a federate that owns each of the objects represented in the recording. The Logger sends interactions and updates only for classes that other federates are subscribed to. The Logger serves as a proxy simulator, answering requests for attribute updates and making objectRemove RTI calls when it encounters removals in the recorded file.

3.2. Timescale and Direction During Playback

The Logger lets you play a recording at any speed within the capabilities of your platform, forward or backward.

You determine the speed by specifying a timescale value, the factor by which time is multiplied during playback. The default timescale setting of 1.0 causes a Logger file to play forward at normal speed. A timescale of 0.5 results in forward, half-speed playback. A negative timescale causes reverse playback. [Figure 3-2](#) shows a set of typical timescale values and their effect on speed and direction.

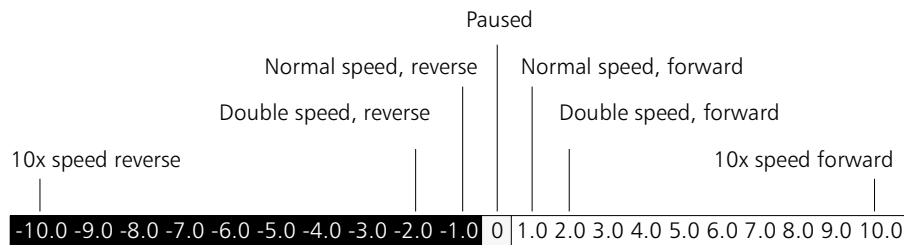


Figure 3-2. Sample timescale values

For procedures, see [“Setting the Timescale and Direction”](#), on page 4-10 and [“Playing Back Recordings”](#), on page 5-4.

You can specify an initial timescale at Logger startup with the `-s` option, for example:

```
Logger -p my-exercise.lgr -s 2.5
```

3.3. Compensating for Timescale Effects During Playback

The Logger has corrective features that adjust the content of outgoing messages for different playback speeds. Under DIS, the Logger generates interim PDUs that prevent receiving applications from inappropriately timing out entities during slow-speed playback. Under both DIS and HLA, velocity and acceleration scaling can adjust message content to prevent anomalies in vehicle movement that would otherwise occur at any playback speed other than real-time.

3.3.1. Generating Interim PDUs During Slow-Speed DIS Playback

In a DIS exercise, each participating entity sends PDUs to other applications at a regular time interval, called the heartbeat. A common heartbeat rate is 5 seconds.

Most receiving applications expect to receive PDUs from each entity at the agreed-upon heartbeat rate. If a receiving application does not receive a PDU from an entity for an unusually long time, it may conclude that the entity no longer exists, and then stop simulating (or time out) the entity. A typical timeout threshold is 12 seconds, or 2.4 times the heartbeat rate.

Using Interim PDUs to Maintain Heartbeat

During slow-speed playback in DIS, the long time lapse between an entity's recorded PDUs might cause receiving applications to time the entity out prematurely. To prevent this, the Logger generates interim PDUs as needed to maintain each entity's heartbeat.

The Logger calculates the contents of an interim PDU by applying a dead-reckoning algorithm to an entity's most recent PDU.

By default, the Logger ensures that at least one PDU is transmitted for each entity every five seconds. You can specify a different heartbeat rate. For procedures, see [“Setting Configuration Options”](#), on page 4-12 and [“Controlling the DIS Heartbeat and Timeout”](#), on page 5-7.

Scaling the Timeout Period During Slow-Speed Playback

When an entity's PDUs stop occurring in the recording, the Logger continues to generate interim PDUs for the entity for the specified timeout period.

During slow-speed playback, the timeout period is extended accordingly. For example, when playing a recording at half-speed, using the default timeout setting of 12 seconds, the Logger generates interim PDUs for an entity for 12 recorded seconds, or 24 real-time seconds before timing it out.

For procedures, see [“Setting Configuration Options”](#), on page 4-12 and [“Controlling the DIS Heartbeat and Timeout”](#), on page 5-7.

3.3.2. Updating Entity Information Quickly After A Time Jump

During playback, you can jump to a specified timepoint. After the time jump, the Logger resumes sending update messages for all entities, starting at the new timepoint. Some time may pass before a fresh update message is sent for each entity, so until updates arrive, receiving applications will reflect entity states that existed before the time jump.

DIS only

In DIS, these applications may time entities out.

To speed the update process, the Logger provides a feature called entity maintenance. After a time jump, the Logger can generate up-to-date interim messages to fill the period before the transmission of each entity's next recorded update. To create these interim updates, the Logger applies a dead-reckoning algorithm to each entity's most recent recorded message prior to the new time point.

In the Logger window, entity maintenance is applied when you issue a Skip Time command or a Set Time command.

Because additional processing is required by entity maintenance, you will experience a delay when jumping to a new timepoint. The length of the delay depends on the number of entities processed and the size of the time jump.

3.3.3. Velocity and Acceleration Scaling

When the Logger play at speeds above or below normal, it can adjust the velocity and acceleration values in entity state messages. This scaling allows the receiving applications to perform accurate dead-reckoning.

For example, if the Logger is playing at double speed, it can double the values in a message's velocity and acceleration fields. This adjustment causes a vehicle to move at double its original speed when viewed using an application such as the MÄK Stealth. Without this scaling, a moving vehicle would appear to be moving at its original slower speed, and would have to lurch forward with the arrival of each new state update message in order to catch up with the new location data contained in the update.

This scaling feature is active by default. To disable it, use the `-d` argument when you start the Logger.

DIS only

Scaling must be enabled in order for the Logger to send PDUs while in the paused state. If a file contains bundled PDUs, the Logger does not unbundle them on playback, and therefore cannot scale their speed and velocity.

3.3.4. How Reverse Playback Affects Dead-Reckoning

Applications receive state update messages in reverse order during reverse playback. Because dead-reckoning is based on acceleration, and acceleration is not a mathematically reversible operation, your applications may place a vehicle on a path different from the one it originally traveled.

3.4. Specifying the End-of-File Action

When the Logger reaches the end of a file, it can enter a stopped or paused state, or it can wrap to the other end of the file and continue playback.

> To specify an end-of-file action, start the Logger with the `-e` argument:

```
Logger -es (Stop (default))
Logger -ep (Pause)
Logger -ew (Wrap)
```

In the Logger window, you can change the end-of-file action in the Configuration dialog box. See [“Setting Configuration Options”](#), on page 4-12.

During reverse playback, if wrap is in effect and the beginning of the recording is reached, the recording wraps to the end and continues reverse playback.



Using The Graphical User Interface

This chapter how to use the Logger's graphical user interface (GUI):

The Logger Windows.....	4-2
Starting the Logger	4-2
Recording Exercises.....	4-4
Playing Logger Recordings.....	4-4
Navigating to A Specific Time in A Recording.....	4-5
Using the Set Time Feature	4-5
Using the Skip Time Feature	4-6
Annotating A Logger Recording.....	4-7
Creating An Annotation	4-7
Saving An Annotation File.....	4-8
Opening An Annotation File.....	4-8
Importing An Annotation File.....	4-9
Moving to An Annotation Point.....	4-9
Editing An Annotation File	4-9
Deleting An Annotation	4-10
Clearing the Annotation List.....	4-10
Setting the Timescale and Direction.....	4-10
Using the Timescale Slider.....	4-11
Setting A Timescale at Startup.....	4-11
Entering A Timescale Value Numerically.....	4-11
Setting Configuration Options.....	4-12
Summary of Startup Options for the Logger	4-14

4.1. The Logger Windows

The Logger has separate windows for playback and record modes. The Playback window has a set of navigation controls and a slider for scaling and reversing time. The Record window allows you to start and stop recording. [Figure 4-1](#) illustrates these windows. Both windows allow you to set configuration options.

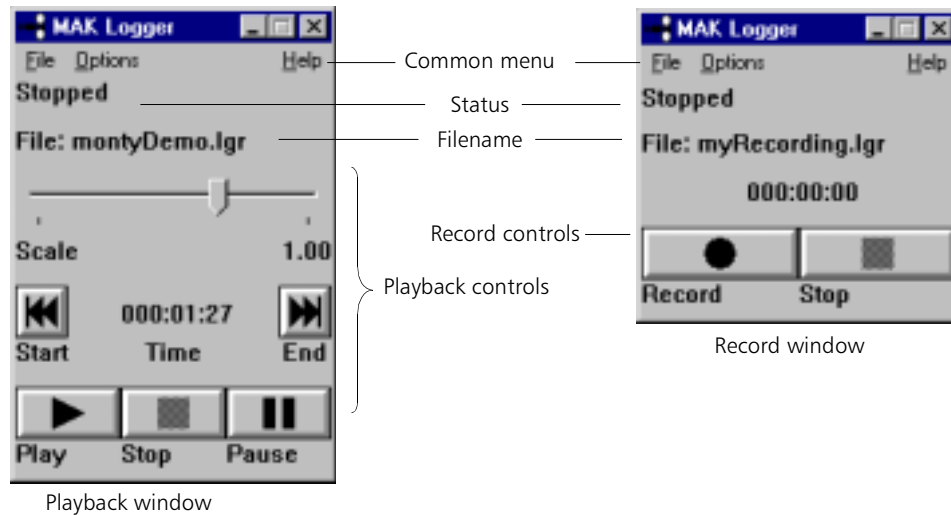


Figure 4-1. The Logger windows

4.1.1. Starting the Logger

You can use startup options to specify many of the Logger's initial settings. [Table 4-2](#), on page 14, lists startup options for the Logger.



Some options (such as the federation execution name in HLA, or the DIS port or destination address) cannot be set in the Logger windows. You must specify them in the startup command.

Starting the Logger in UNIX

To start the Logger:

1. Open a UNIX shell.
2. Change to the appropriate */bin* directory.
3. Enter:

`Logger [options]`

By default, the Logger opens up in playback mode.

Starting the Logger in Windows

Start Logger in any of the following ways:

- > On the Start Menu, choose Programs → MÄK Technologies → Logger *version*, where *version* is DIS5 or HLA.
- > In a DOS window, change to the appropriate *\bin* directory, and enter:
`Logger [options]`

When the Logger window opens, you can close the command window if you want to.

By default, the Logger opens up in playback mode.

4.2. Recording Exercises

To record an exercise:

1. In either the Playback window or Record window, choose File → Record New File. The Select File dialog box opens. The Select File dialog box is a standard open file dialog box for your windowing system.
2. If necessary, navigate to the directory where you want to create the file, then enter a new filename or select an existing file.



If you select an existing file, its contents will be overwritten by the new recording.

3. Click OK. If the Logger was in playback mode, it changes to record mode.

The Logger enters record mode in the state specified in the Configuration dialog box.

> To begin recording, click the Record button.

> To stop recording, click the Stop button.

The Status field displays either Recording or Stopped depending on the status.

As recording progresses, the Time field is updated to display elapsed time. In record mode, the timescale is fixed at 1.0 (normal speed).



If you are recording an exercise that is running faster than real-time, the Logger records the simulation at that rate and will play it back faster than real-time when the playback timescale is set to 1.0.

4.3. Playing Logger Recordings

To play a Logger recording:

1. In either the Playback window or Record window, choose File → Play Back Old File. The Select File dialog box opens. This is a standard open file dialog box for your windowing system.
2. If necessary, navigate to the desired directory, then select the file to be played.
3. Click OK. If the Logger was in record mode, it changes to playback mode.

By default, the Logger enters a stopped state when the file is loaded. To play, pause, or stop playback, click on the appropriate button.

If the recording is at its beginning or end, the Status field displays Start or End. Otherwise, it displays Stopped, Paused, or Playing to reflect the Logger's current operating state.

4.4. Navigating to A Specific Time in A Recording

The Logger provides the following ways to navigate to a specific time in the exercise:

- ♦ Use the Start and End buttons in the Logger's main window to jump to the beginning or end of the recording.
- ♦ Use the Set Time option to jump to a specific time.
- ♦ Use the Skip Time option to jump forward or backward in time by a specified offset.
- ♦ Create annotations.

The Skip Time and Set Time options are on the Options menu.

If you execute a time jump while playback is paused or stopped, the Logger remains in that state. The Logger also retains a play state after a time jump unless the end of the file is reached and wrapping is disabled. For more information, see section 3.4, “[The End-of-File Action](#)”, and section 4.6, “[Setting Configuration Options](#)”.

During slow-speed playback, when you use the Skip Time or Set Time option, the Logger performs entity maintenance to speed the transmission of up-to-date messages. For more information, see “[Updating Entity Information Quickly After A Time Jump](#)”, on page 3-6.

4.4.1. Using the Set Time Feature

The Set Time feature lets you move to a specific point in time measured from the start of the recording.

To move to a specific time:

1. In the Playback window, choose Options → Set Time. The Set Time dialog box opens ([Figure 4-2](#)).

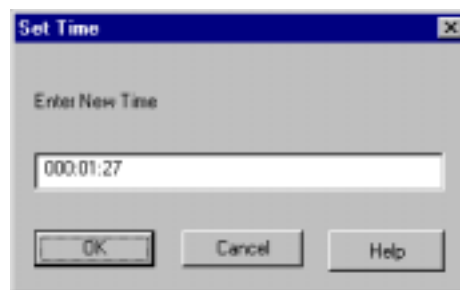


Figure 4-2. Set Time dialog box

2. In the Enter New Time box, enter a time in hours:minutes:seconds format. This time is measured from the start of the recording.
3. Click OK.

4.4.2. Using the Skip Time Feature

The Skip Time feature lets you move forward or backward from the current point in a recording by a specified amount of time.

To skip forward or backward in a recording:

1. In the Playback window, choose Options → Skip Time. The Skip Time dialog box opens ([Figure 4-3](#)).

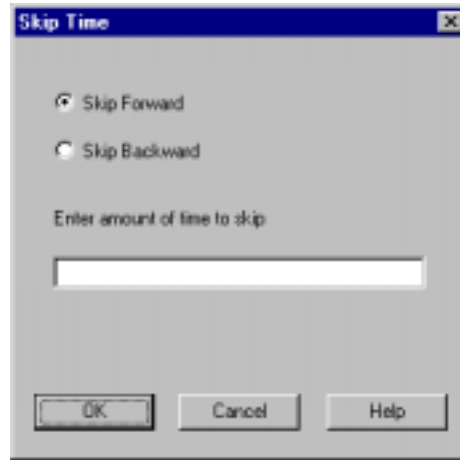


Figure 4-3. Skip Time dialog box

2. Select the Skip forward or Skip backward option.
3. In the Enter Amount of Time to Skip box, enter a time in hours:minutes:seconds format.
4. Click OK.

4.5. Annotating A Logger Recording

An annotation specifies a particular elapsed time in a Logger recording to which you can jump. It differs from the Set Time command in that you can save annotations to a file and reuse them. Optionally, you can associate text with an annotation.

The maximum number of annotations is 1000. You can override this limit with the Logger API.

4.5.1. Creating An Annotation

To annotate a Logger recording:

1. In the Record or Playback window, choose Options → View Annotations. The View Annotations window opens ([Figure 4-4](#)).

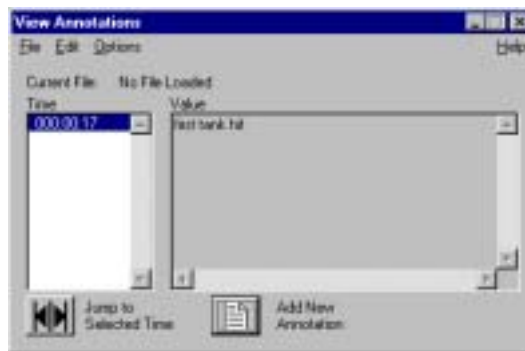


Figure 4-4. View Annotations window

2. Play a Logger recording or begin recording an exercise.
3. At any point while the Logger is playing, recording, or paused, choose Edit → Add (or click the Add New Annotation button.) The Insert Annotation dialog box opens ([Figure 4-5](#).) The dialog box displays the time at which the annotation will be inserted. You cannot edit the time.



Figure 4-5. Insert Annotation dialog box

4. Type the text you want to associate with this annotation.

5. Click OK.

You can leave the View Annotations window open and continue to add annotations as often as you want. Annotations can be as long as you want.

4.5.2. Saving An Annotation File

You can save annotations to a file. Annotation files have the extension *.ant*.

To save annotations to a file:

1. Create annotations as described in [“Creating An Annotation”](#), on page 4-7.
2. In the View Annotations window, choose File → Save As. The Save As dialog box opens.
3. Navigate to the directory in which you want to save the annotation file. The default directory is the directory from which you started the Logger.
4. Enter a name for the annotation file.
5. Click Save.

4.5.3. Opening An Annotation File



If you open an annotation file, any annotations currently listed in the View Annotations window are overwritten.

To open an annotation file:

1. In the Record or Playback window, choose Options → View Annotations.
2. In the View Annotations window, choose File → Open. An Open dialog box opens.
3. Select an annotation file.
4. Click Open. The annotations in the file are displayed.

4.5.4. Importing An Annotation File

If you import an annotations file, the annotations in the file are added to the annotations listed in the View Annotations window.

To import an annotation file:

1. In the Record or Playback window, choose Options → View Annotations.
2. In the View Annotations window, choose File → Import. An Open dialog box opens.
3. Select an annotation file.
4. Click Open. The annotations in the file are added to the list.

4.5.5. Moving to An Annotation Point

To move a recording to an annotated point:

1. In the Playback window, choose Options → View Annotations.
2. Open an annotation file or create new annotations for the recording you are working with.
3. Start the Logger recording that is associated with the displayed annotations.
4. At any point, select the annotation to which you want to move the recording.
5. Click the Jump to Selected Time button.

4.5.6. Editing An Annotation File

You can edit the text associated with an annotation. You cannot edit the time points, except to delete them.

To edit an annotation:

1. Select the annotation you want to edit.
2. Choose Edit → Modify. The Edit Annotation dialog box opens.
3. Edit the text in the text box.
4. Click OK.

4.5.7. Deleting An Annotation

You can delete an annotation from an annotation file or an unsaved list.

To delete an annotation:

1. In the View Annotations window, select the annotation you want to delete.
2. Choose Edit → Remove. The annotation is removed from the list.

If you removed an annotation from an annotation list that you had previously saved to a file, save the edited list with the File → Save As command.

4.5.8. Clearing the Annotation List

Clearing the View Annotations window removes all annotations from the display. If an annotation file is open, it closes the annotation file without closing the View Annotations window.

> To clear the View Annotations, choose File → Clear.



If you have opened a list of annotations from a file, clearing the View Annotations window is not the same as deleting all of the annotations in the list. Clearing the window closes the file without editing it. If you open an annotation file and remove each annotation with the Edit → Remove command, the file is still open, but now it does not have any annotations in it.

4.6. Setting the Timescale and Direction

The Logger lets you play a recording forward or backward at any speed, limited only by the capabilities of your hardware. You can set the timescale in the following ways:

- ♦ Adjust a slider control in the Logger playback window
- ♦ Specify an initial timescale at Logger startup
- ♦ Enter the desired timescale into a dialog box.

These options are described in the following sections. For conceptual information about timescale values, see section 3.2, [“Timescale and Direction During Playback”](#).

4.6.1. Using the Timescale Slider

The Timescale slider control in the Logger playback window is the most convenient way to change playback speed and direction. In its default configuration, the slider contains nineteen preset values ranging from -10 to +10.

> To change the timescale, drag the slider button.

If you need finer control, click once on the slider button to give it focus, then use the left and right arrows on your keyboard to step through the values.

4.6.2. Setting A Timescale at Startup

> To specify a timescale at Logger startup, use the -s argument, as in the following example:

```
Logger -p myexercise.lgr -s 2.5
```

4.6.3. Entering A Timescale Value Numerically

If the timescale you want is not available through the slider control, you can specify it directly.

1. In the Logger window, choose Options → Set Timescale. The Set Time Scale dialog box opens ([Figure 4-6.](#))

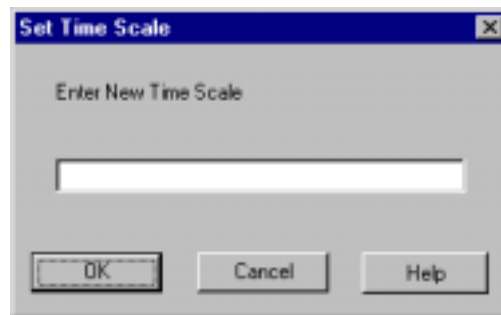


Figure 4-6. Set Time Scale dialog box

2. Enter the desired timescale.
3. Click OK.

You can enter the timescale value in scientific notation, for example, 1e+4 instead of 10,000.

i

Although the Logger lets you specify a very large timescale (beyond $1e+35$), the use of an unusually large timescale may produce unexpected results because of hardware limitations.

If a recording is played at a very high speed, the Logger may be unable to keep up, especially during passages that contain a high concentration of data. If the Logger falls significantly behind, it continues to process the messages as quickly as possible. It does not discard any messages.

4.7. Setting Configuration Options

To set configuration options,:

1. Choose Options → Configuration. The Configuration dialog box opens ([Figure 4-7](#).)
2. Set options as desired. [Table 4-1](#) describes the configuration options.



Figure 4-7. The Configuration dialog box

Table 4-1: Logger Configuration Options (Sheet 1 of 2)

Option	Description
Play Action	Determines whether the Logger enters a stopped state (default) or begins playing immediately when a file is opened for playback.
Record Action	Determines whether the Logger enters a stopped state (default) or begins recording immediately when a file is opened for recording.

Table 4-1: Logger Configuration Options (Sheet 2 of 2)

Option	Description
End of File Action	Determines the Logger's behavior when it reaches the end of a recording. It can stop, pause, or wrap around to the other end of the recording to continue playback. Default: stop. See "Specifying the End-of-File Action" , on page 3-7.
Heartbeat (DIS only)	Determines how frequently (in seconds) the Logger generates interim PDUs for each entity during slow-speed playback. For information, see "Generating Interim PDUs During Slow-Speed DIS Playback" , on page 3-5.
Timeout (DIS only)	Determines how long the Logger continues to generate interim PDUs for an entity once that entity's PDUs have stopped occurring in the recording. See "Generating Interim PDUs During Slow-Speed DIS Playback" , on page 3-5.

i

Configuration changes are valid only for a given Logger session. They are not retained for future sessions. If you want to create a set of startup options that you can reuse, write a script or batch file that specifies the configuration you want.

4.8. Summary of Startup Options for the Logger

Table 4-2 lists startup options. They are grouped by mode and indicate if they are for HLA only or DIS only.

Table 4-2: Command-line arguments (Sheet 1 of 3)

Option	Description
Record and Playback	
-a { <i>site:host:entity</i> <i>hla_object_name</i> }	DIS: Specifies the site:host:entity for remote control of the Logger. HLA: Specifies the object name (global object designator in VR-Link.)
-f { <i>VIRPRxx</i> <i>library:init_data</i> }	HLA: Specifies the FOM Mapper to use if you are not using the default FOM.
-F <i>sublist.mtl</i>	HLA: Specifies a subscription control file to load.
-G {w t wt}	Specifies interface to use, where: w = Graphical user interface (default) t = Command line interface wt = Both.
-h, -help, -?	Displays a brief summary of Logger command-line arguments.
-L {0 1 2}	Specifies the response to Logger Control messages for remote control. 0 = Do not respond to Logger Control messages 1 = interpret Logger Control messages only if they are addressed to the entity ID specified by the -a argument (default) 2 = interpret all Logger Control messages
-m	Causes the Logger to send a Logger control PDU each time you change the state of the Logger.
-P <i>dis_port</i>	DIS only. Specifies which DIS port (UDP port) the Logger uses to send or receive data. Default port: 3000.
-x <i>ExName</i>	HLA only. Sets federation execution name. Default: VR-Link
-v {0 1}	Specifies the Logger interface to use: 0 = Logger interface not visible 1 = Use GUI (default)

Table 4-2: Command-line arguments (Sheet 2 of 3)

Option	Description
Record Mode Only	
-b <i>state</i>	Specifies whether the Logger enters a stopped state or begins recording immediately when opening a file for recording. r = record S = stopped (default).
-P	HLA: Subscribe passively. Do not force federates to send all updates to the Logger. Record only those updates that other federates subscribe to.
-r [<i>filename</i>]	Causes the Logger to open in record mode. Specifies the destination file for recorded data.
-R {0 1}	Specifies whether or not to record Logger Control messages during remote control. 0 = do not record Logger Control messages (default) 1 = record Logger Control messages
-S <i>mcastAddr</i>	DIS only. Subscribes the Logger to the multicast address specified by <i>mcastAddr</i> . To subscribe to several multicast addresses, use this option more than once in the command. See “Subscribing to Multicast Groups” , on page 2-25.
Playback Mode Only	
-A <i>addr</i>	DIS only. Sets the default IP address to which the Logger sends PDUs. If this option is not used, the Logger uses the computer’s broadcast address. See “Specifying A Destination Address for Outgoing Data” , on page 2-25.
-d	Disables the generation of velocity and acceleration scaling data when playing at speeds above or below normal. See “Velocity and Acceleration Scaling” , on page 3-7.
-e <i>action</i>	Specifies the Logger’s action upon reaching the end of the file. p = pause at the file’s end w = wrap to the other end of the recording and continue S = stop (default).

Table 4-2: Command-line arguments (Sheet 3 of 3)

Option	Description
-o <i>state</i>	When a file is opened for playback, specifies whether the Logger enters a stopped state, or begins playing immediately. p = playing s = stopped (default).
-p [<i>filename</i>]	Causes the Logger to open in playback mode; specifies the source file to be played.
-s <i>timescale</i>	Sets the initial playback timescale factor. For more information, see “Timescale and Direction During Playback” , on page 3-4.
-T <i>timepoint</i>	Sets the timepoint in the file where playback begins. Enter the <i>timepoint</i> in HH:MM:SS format. Default: 00:00:00



Using The Command-Line Interface

The command-line interface has the same features as the graphical user interface, plus script-based recording and playback capability.

Starting the Logger in Command-Line Mode	5-2
Recording Exercises.....	5-2
Beginning A Recording Session in Record Mode	5-2
Logger Run-time Commands for Recording.....	5-3
Displaying the Status of A Recording	5-3
Playing Back Recordings	5-4
Basic Logger Playback Commands	5-4
Displaying Playback Status	5-5
Beginning A Playback Session in Play Mode.....	5-5
Navigating Time	5-5
Controlling End-of-File Behavior	5-6
Setting Playback Timescale and Direction.....	5-6
Controlling the DIS Heartbeat and Timeout	5-7
Setting the Heartbeat Rate.....	5-7
Setting the Timeout	5-7
Using A Script to Control Recording and Playback.....	5-8
Stopping A Script	5-8
Summary of Logger Run-Time Commands	5-9

5.1. Starting the Logger in Command-Line Mode

To start the Logger in command-line mode:

1. Open a command window.
2. Enter the following command:

```
Logger -G t
```

The Logger starts running and waits for instructions.

At startup, a message shows the NetSocket status.

```
NetSocket ok for port 3000 to 123.456.789.255
```

You can start the Logger in record or playback mode. See [“Recording Exercises”](#), on page 5-2 and [“Playing Back Recordings”](#), on page 5-4.

5.2. Recording Exercises

To record an exercise if the Logger is running:

1. In the command window, change to the *loggerx.x/bin5* or *binRPR* directory.
2. Enter the following command:

```
record logger_file.lgr
```

where *logger_file.lgr* is the name of the file to which you want to record the exercise.

To record an exercise if the Logger is not running:

1. In a command window, change to the *loggerx.x/bin5* or *binRPR* directory.
2. Enter the following command:

```
Logger -G t -r logger_file.lgr
```

5.2.1. Beginning A Recording Session in Record Mode

By default, when you start the Logger in record mode, it begins in a stopped state.

- > To begin recording immediately, use the *-b* option to set the record action as in this example:

```
Logger -br -r filename
```

5.2.2. Logger Run-time Commands for Recording

[Table 5-1](#) lists the run-time commands for the Logger.

Table 5-1: Logger run-time commands for recording exercises

Command	Description
go	Begins or resumes recording.
info	Displays a brief status report.
quit	Closes the file and exits the Logger.
record <i>filename</i>	Starts recording to the specified file. If the file already exists, it is overwritten.
stop	Stops recording without ending the session.

5.2.3. Displaying the Status of A Recording

At any time while you are recording an exercise, you can enter the `info` command (or the letter `i`) to display a brief status summary. The summary shows the destination file name, the time elapsed since the start of recording, and the current operating status.

```
> info
FILE: my-exercise.lgr
TIME: 000:01:43
SCALE: 1.000000
STATE: Recording
>
```

5.3. Playing Back Recordings

To play a Logger file if the Logger is already running:

1. In the command window, change to the *loggerx.x/bin5* or *binRPR* directory.
2. Enter the following command:

```
play recording_path/logger_file.lgr
```

where *recording_path/logger_file.lgr* is the full or relative path and file name of the recording you want to play.

To play a Logger file if the Logger is not running:

1. In a command window, change to the *loggerx.x/bin5* or *binRPR* directory.
2. Enter the following command:

```
Logger -v 2 -p ../logger_tapes/my-exercise.lgr
```

If you are running the DIS version of the Logger, a status message similar to the following is displayed:

```
NetSocket ok for port 3000 to 123.456.789.255
Logger file my-exercise.lgr is version 2 (writing)
```

5.3.1. Basic Logger Playback Commands

In playback mode, the Logger starts in a playing state. Thereafter, it is in either a playing, paused, or stopped state. [Table 5-2](#) describes the most common commands for playback.

Table 5-2: Logger run-time commands for playing recordings

Command	Description
go	Begins or resumes playback. When playing, the Logger sends out PDUs in chronological sequence.
info	Displays status information.
open	Opens a file for playback.
pause	Pauses playback. When paused, the Logger stops time and, for DIS only, repeatedly rebroadcasts each entity's most recent entity state PDU.
play <i>filename</i>	Opens the specified file for playback.
quit	Closes the file and exits the Logger.
stop	Stops playback. When stopped, the Logger stops time and does not transmit data.

5.3.2. Displaying Playback Status

- > To display a playback status summary, enter the `info` command (or the letter “i”) at the Logger prompt.

The summary shows the source file name, the current timepoint in the Logger file, the timescale (playback speed), and the current operating status.

```
> info
FILE: my-exercise.lgr
TIME: 000:00:09
SCALE: 1.000000
STATE: Playing
```

5.3.3. Beginning A Playback Session in Play Mode

By default, when you start the Logger in playback mode, it is in a stopped state.

- > To begin playing immediately, use the `-o` option to set the play action as in this example:

```
Logger -op -p filename
```

5.4. Navigating Time

During playback, four commands let you navigate to a specific point in a recording:

Table 5-3: Runtime commands during playback

Command	Description
<code>start</code>	Moves to the beginning of the recording.
<code>end</code>	Moves to the end of the recording.
<code>time</code>	Sets the timepoint to a specified time. The command format is: <code>time HH:MM:SS</code>
<code>skip</code>	Causes the Logger to jump forward or backward in time by a specified offset. Enter the offset in an HH:MM:SS format, or as a number of seconds. Use a negative value to skip backward in time. For example: <code>skip 00:02:30</code> <code>skip -170</code>

5.4.1. Controlling End-of-File Behavior

> To specify the end-of-file action, start the Logger with the `-e` option, as follows:

```
Logger -p filename -e [p|w]
```

where:

- `p` causes the Logger to enter a paused state at the end of a file.
- `w` causes the Logger to wrap around to the other end of the file and continue playback.

5.5. Setting Playback Timescale and Direction

You can play a recording at any practical speed, forward or backward.

> To set the playback speed and direction, enter the `scale` command:

```
scale timescale
```

where `timescale` is the factor by which time is to be multiplied. For information about choosing timescale values, see [“Timescale and Direction During Playback”](#), on page 3-4.

> To specify a timescale at startup, use the `-s` option, for example:

```
Logger -p my-exercise.lgr -s 2.5
```

You can enter the timescale value in scientific notation, as in the following example for 10,000:

```
scale 1e+4
```



Although the Logger lets you specify a very large timescale (beyond $1e+35$), the use of an unusually large timescale may produce unexpected results because of hardware limitations, and is not recommended.

If a recording is being played at a high speed, the Logger may be unable to keep up, especially during passages that contain an unusually high concentration of data. If the Logger falls behind, it continues to process messages as quickly as possible. No messages are discarded.

5.6. Controlling the DIS Heartbeat and Timeout

During slow-speed playback in DIS, the Logger generates interim PDUs as needed to maintain an appropriate heartbeat rate and timeout threshold for each entity in the recording. Heartbeat and timeout concepts are explained in section 3.3.1, [“Generating Interim PDUs During Slow-Speed DIS Playback”](#).

5.6.1. Setting the Heartbeat Rate

The heartbeat rate determines how frequently the Logger generates interim PDUs for each entity. By default, the Logger maintains a heartbeat rate of five seconds.

- > To set the heartbeat rate, use the `heartbeat` command:

```
heartbeat n
```

where *n* represents the maximum allowable time between PDUs.

- > To view the current `heartbeat` setting, enter the command name without a value.

5.6.2. Setting the Timeout

The timeout threshold determines how long the Logger continues to generate interim PDUs for an entity once the entity's PDUs stop occurring in the recording. The Logger's default timeout setting is twelve seconds.

- > To set the timeout value, use the `timeout` command:

```
timeout n
```

where *n* represents the number of recorded seconds the Logger should wait before timing an entity out.

- > To view the current `timeout` setting, enter the command without a value.

5.7. Using A Script to Control Recording and Playback

A script is a text file that can control the Logger in either playback or record mode. Scripts are especially useful for creating a prepared demo, or for automatically bypassing unwanted segments during playback.

A script can contain any Logger run-time commands listed in [Table 5-4](#). Create the script in any text editor, then redirect it to the Logger as in these examples:

```
Logger -p my_exercise.lgr < my_play_script.txt
Logger -r new_exercise.lgr < my_record_script.txt
```

At startup, the Logger enters a playing or recording state, then immediately begins processing commands in the script.

Controlling Processing with the Wait Command

The `wait` command controls how long the Logger runs in its current state before processing the next command in the script. The sample script in [Figure 5-1](#) provides an example.

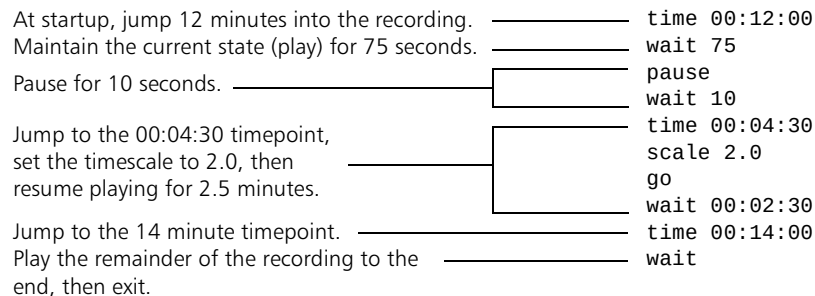


Figure 5-1. Sample Logger script

If `wait`, with a *time* argument, is the last command in a script, the Logger maintains the current state for the duration of the time argument, then exits.

If a script contains a `wait` command with no *time* argument, the Logger continues operating in the current state until reaching the end of the recording, then exits. Any subsequent commands in the script are ignored.

5.7.1. Stopping A Script

The Logger does not respond to keyboard input when running a script, so you cannot manually enter `quit` to stop execution.

> To stop a script, enter `Ctrl-C`.

5.8. Summary of Logger Run-Time Commands

Table 5-4 lists the commands you can send to the Logger while it is running in command-line mode.

Table 5-4: Logger Run-Time Commands (Sheet 1 of 2)

Command	Description
Record Mode and Play Mode	
<code>go</code>	Starts or resumes recording or playback from a previously stopped or paused state.
<code>stop</code>	Halts playback at the current timepoint, stops recording or transmission.
<code>quit</code>	Closes the Logger file and exits.
<code>info</code> or <code>i</code>	Displays a brief status report including the log file name, current timepoint, timescale, and current play state.
<code>h</code> , <code>help</code> , or <code>?</code>	Displays a summary of Logger run-time commands.
<code>wait time</code>	Used only in scripts. Specifies how long the Logger performs its current action before processing the next command in the script. Enter <code>time</code> in the HH:MM:SS format or as a number of seconds.
Play Mode Only	
<code>start</code>	Sets the current timepoint to the beginning of the recording.
<code>open filename</code>	Opens the specified Logger recording for playback.
<code>play</code>	Begins playing the Logger recording.
<code>end</code>	Sets the current timepoint to the end of the recording.
<code>pause</code>	Pauses playback at the current timepoint. In DIS, begins repeated resending of each entity's most recent entity state PDU with velocity and acceleration set to zero.
<code>scale [factor]</code>	Sets the timescale <code>factor</code> that determines playback speed and direction. Use 1.0 (default) for normal speed, 2.0 for double speed, 0.5 for half speed, and negative values for backward playback.
<code>time timepoint</code>	Navigates to a specific <code>timepoint</code> in the recording. Specify <code>timepoint</code> in HH:MM:SS format.
<code>skip offset</code>	Moves the current timepoint forward or backward by the specified <code>offset</code> . Specify the offset in HH:MM:SS format or as a number of seconds. A positive offset causes a jump forward in time; a negative offset causes a backward jump.

Table 5-4: Logger Run-Time Commands (Sheet 2 of 2)

Command	Description
heartbeat [sec]	(DIS only) During slow-speed playback, specifies the number of seconds the Logger allows before generating an interim PDU for a given entity. Default: 5. With no argument, heartbeat displays the current setting.
timeout [sec]	(DIS only) During slow-speed playback, specifies how long the Logger continues to transmit interim PDUs for an entity after the entity's PDUs have stopped occurring in the recording. Default: 12 recorded seconds.
Record Mode Only	
record <i>filename</i>	Begins recording to the specified filename.



Remote Control

You can control the Logger from a remote application using DIS or HLA update messages. Every command available in the Logger graphical user interface is available for remote control.

Remote Control Messages	6-2
Using Logger Commands for Remote Control.....	6-3
Deleting Logger Files.....	6-3
Example of Remote Control	6-5

6.1. Remote Control Messages

Logger Control PDU classes allow you to send messages to the Logger from an application based on VR-Link. There are 17 Logger Control PDUs - one for each feature that can be activated remotely. All are derived from *DtLgrControlPdu*, which is defined in *lcPdu.h*, and all are defined in header files that begin with “lc”. You can view the header files in *VR-Link Reference Manual*.



In DIS we send PDUs. In HLA, we send interactions. For convenience, in this manual, we refer to PDUs and interactions collectively as messages.

The remote control classes are:

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

HLA only

For HLA, *DtLgrControlInteraction*, defined in *hLcInter.h*, is a wrapper around *DtLgrControlPdu*.

6.2. Using Logger Commands for Remote Control

Several startup options and their equivalent MTL parameters facilitate remote control of the Logger. [Table 6-1](#) lists the relevant startup options and MTL parameter.

Table 6-1: Logger commands for remote control

Startup Option	MTL Parameter	Description
	<code>allowRemoteFileDelete</code>	Enables or disables file deletion in response to <i>DtLgrDelFilePdus</i> . 0 = Do not allow file deletions 1 = Allow file deletions.
<code>-v {0 1}</code>	<code>visible</code>	Specifies the Logger interface to use: 0 = Logger interface not visible 1 = Use GUI
<code>-R {0 1}</code>	<code>recordLc</code>	0 = Do not record Logger Control messages (default) 1 = Record Logger Control messages
<code>-L {0 1 2}</code>	<code>processLc</code>	0 = Do not respond to Logger Control messages 1 = Interpret Logger Control messages only if they are addressed to the entity ID specified by the <code>-a</code> argument (default) 2 = Interpret all Logger Control messages

6.2.1. Deleting Logger Files

The Logger Control PDU, *DtLgrDelFilePdu* instructs the Logger to delete a Logger file. When the Logger receives this PDU, it decides whether or not to delete the file specified in the PDU based on the following criteria:

- ♦ If the MTL variable `allowRemoteFileDelete` is non-zero
- ♦ If the `-D` flag is set.

The default is to not allow deletion.

The file to be deleted must be a Logger file of the correct protocol, that is, the DIS Logger can delete only DIS files and the HLA Logger can delete only HLA files.

If the Logger receives a *DtLgrDelFilePdu* that specifies the filename *foo.lgr*, it searches for *foo.lgr* in its working directory. If the PDU specifies a relative pathname, such as *../logger_tapes/foo.lgr*, the Logger searches for the file at that location relative to its working directory.

Diagnostic messages are sent to the *logger.txt* file.

To create a *DtLgrDelFilePdu*, use code similar to the following:

```
#include "lcDelFile.h"

DtLgrDelFilePdu deleteFile;
deleteFile.setFilename("logger_tape.lgr");
exerciseConn.sendStamped(deleteFile);
```

HLA only

In HLA, add:

```
DtLgrControlInteraction lcInter;
lcInter.setLcPdu(deleteFile);
exerciseConn.sendStamped(deleteFile);
```

6.3. Example of Remote Control

The following example demonstrates programmatic remote control of the Logger:

```
#include "stdio.h"
#include "exerciseConn.h"
#include "ProcControl.h"
#include "EntityTypes.h"

#include "lcEnd.h"
#include "lcEofAction.h"
#include "lcHeartbeat.h"
#include "lcInter.h"
#include "lcPause.h"
#include "lcPdu.h"
#include "lcPlay.h"
#include "lcPlayAction.h"
#include "lcPlayFile.h"
#include "lcQuit.h"
#include "lcRecAction.h"
#include "lcRecFile.h"
#include "lcRecord.h"
#include "lcSetTime.h"
#include "lcSkipTime.h"
#include "lcStart.h"
#include "lcStop.h"
#include "lcTimeScale.h"
#include "lcTimeout.h"
#include "lcDelFile.h"

char * helpstr =
    " PDUTYPE OPT_ARG\n"
    "0 PbFile filename\n"
    "1 RecFile filename\n"
    "2 Quit \n"
    "3 Rewind \n"
    "4 End \n"
    "5 Play \n"
    "6 Pause \n"
    "7 Stop \n"
    "8 Record \n"
    "11 TimeScale float\n"
    "12 PlayAction {PLAY=0, STOP=3}\n"
    "13 RecordAction {RECORD=2, STOP=3}\n"
    "14 EofAction {PAUSE=1, STOP=3, LOOP=5}\n"
    "15 HeartBeat float\n"
    "16 Timeout float\n"
    "17 Delete filename\n"
    "18 SetTime float\n"
    "19 SkipTime float\n"
    "-1 Quit this application\n"
;

sendPdu(DtExerciseConn& exConn, DtLgrControlPdu* pdu,
DtGlobalObjectDesignator& target, DtGlobalObjectDesignator& sender)
{
    if (pdu)
    {
        pdu->setReceiverId(target);
        pdu->setSenderId(sender);
    }
}
```

```
#if DtHLA
    DtLgrControlInteraction inter;
    inter.setLcPdu(*pdu);
    exConn.send(inter);
#else
    exConn.sendStamped(*pdu);
#endif
}

//
// Application
//

main()
{
    #if DtHLA
        DtExerciseConn exConn("VR-Link", "Remote", new DtRprFomMapper(1.0));
    #else
        DtExerciseConn exConn(3000, 1, 39, 16);
    #endif

    // Identify the receiving (remote) Logger
    DtGlobalObjectDesignator target("9:8:7");

    // Optionally identify yourself
    DtGlobalObjectDesignator myId("3:4:5");

    // Initialize simulation time
    DtTimeInit();

    char cmdline[128];
    char optionalArg[128];
    int cmd = 0;
    while (cmd >= 0)
    {
        // Print usage instructions and prompt for input
        printf("%s\n> ",helpstr);
        gets (cmdline);

        // Scan for input
        cmd = 0 ; optionalArg[0] = '\0';
        sscanf(cmdline, "%d%s",&cmd,optionalArg);

        // Construct and send desired PDU
        switch(cmd)
        {
            case DtLgrPbFile:
            {
                DtLgrPbFilePdu pdu;
                pdu.setFilename(optionalArg);
                sendPdu(exConn, &pdu, target, myId);
                break;
            }
            case DtLgrRecFile:
            {
                DtLgrRecFilePdu pdu;
                pdu.setFilename(optionalArg);
                sendPdu(exConn, &pdu, target, myId);
                break;
            }
        }
    }
}
```

```
}
case DtLgrRecord:
{
    // Start recorded traffic from network (during record mode)
    DtLgrRecordPdu pdu;
    sendPdu(exConn, &pdu, target, myId);
    break;
}
case DtLgrStart:
{
    // Rewind remote Logger (during playback)
    DtLgrStartPdu pdu;
    sendPdu(exConn, &pdu, target, myId);
    break;
}
case DtLgrPlay:
{
    // Start sending recorded traffic onto network
    // (during playback)
    DtLgrPlayPdu pdu;
    sendPdu(exConn, &pdu, target, myId);
    break;
}
case DtLgrStop:
{
    // Stop sending recorded traffic onto network
    // (during playback)
    DtLgrStopPdu pdu;
    sendPdu(exConn, &pdu, target, myId);
    break;
}
case DtLgrQuit:
{
    DtLgrQuitPdu pdu;
    sendPdu(exConn, &pdu, target, myId);
    break;
}
case DtLgrEnd:
{
    // Fast-forward remote Logger to end (during playback)
    DtLgrEndPdu pdu;
    sendPdu(exConn, &pdu, target, myId);
    break;
}
case DtLgrPause:
{
    // Pause sending recorded traffic onto network
    // (during playback)
    DtLgrPausePdu pdu;
    sendPdu(exConn, &pdu, target, myId);
    break;
}
case DtLgrSetTime:
{
    // Jump to specified time during playback
    DtLgrSetTimePdu pdu;
    pdu.setTime(atof(optionalArg));
    sendPdu(exConn, &pdu, target, myId);
    break;
}
```

```
case DtLgrSkipTime:
{
    // Move forward or backwards by specified amount of time
    // during playback
    DtLgrSkipTimePdu pdu;
    pdu.setTime(atof(optionalArg));
    sendPdu(exConn, &pdu, target, myId);
    break;
}
case DtLgrTimeScale:
{
    // Control the speed of playback
    DtLgrTimeScalePdu pdu;
    pdu.setTimeScale(atof(optionalArg));
    sendPdu(exConn, &pdu, target, myId);
    break;
}
case DtLgrPlayAction:
{
    // Action to take when entering play mode
    DtLgrPlayActionPdu pdu;
    pdu.setAction((DtLgrMode)atoi(optionalArg));
    sendPdu(exConn, &pdu, target, myId);
    break;
}
case DtLgrRecordAction:
{
    DtLgrRecordActionPdu pdu;
    pdu.setAction((DtLgrMode)atoi(optionalArg));
    sendPdu(exConn, &pdu, target, myId);
    break;
}
case DtLgrEofAction:
{
    DtLgrEofActionPdu pdu;
    pdu.setAction((DtLgrMode)atoi(optionalArg));
    sendPdu(exConn, &pdu, target, myId);
    break;
}
case DtLgrDelFile:
{
    DtLgrDelFilePdu pdu;
    pdu.setFilename(optionalArg);
    sendPdu(exConn, &pdu, target, myId);
    break;
}
#ifdef !DtHLA
case DtLgrHeartbeat:
{
    DtLgrHeartbeatPdu pdu;
    pdu.setHeartbeat(atof(optionalArg));
    sendPdu(exConn, &pdu, target, myId);
    break;
}
case DtLgrTimeout:
{
    DtLgrTimeoutPdu pdu;
    pdu.setTimeout(atof(optionalArg));
    sendPdu(exConn, &pdu, target, myId);
    break;
}
```

```
    }  
#endif  
    }  
}
```



A program can use wildcards to identify the Loggers it sends commands to.
For example:

```
DtObjectId target("255:255:7");
```



Using The Logger API

The Logger API allows you to add features to the Logger, modify its behavior, and embed Logger functionality into your own code.

Introduction	7-2
The DtLogger class	7-3
Including the Logger in an Application	7-3
Controlling the Logger Time Clock.....	7-6
Controlling Logger Files	7-7
Controlling the Logger's Configuration and Status	7-8
Controlling Logger Annotations.....	7-9
Setting Protocol-Specific Parameters	7-10
Managing Pipeline Objects	7-11
The Logger Architecture	7-12
Logger Pipeline Elements	7-13
Pipeline Element Member Functions.....	7-14
Default Pipeline Elements	7-17
Example of Overriding A Pipeline Element	7-17
Creating Plug-ins	7-22
Initializing A Plug-in	7-22
Modifying the Pipeline Using A Plug-in	7-23
Overriding DtLogger from A Plug-in	7-25
Loading Plug-Ins	7-26
Compiling and Linking with the Logger API	7-26

7.1. Introduction

The Logger application programming interface (API), allows you to extend and customize the functionality available in the Logger. For example, you can embed a Logger in your own executable, to play and record Logger files, or you can recompile the Logger executable with your own customizations. You can also use the Logger API to build dynamically linked modules, or plug-ins, that the Logger can load at runtime.



You must have a VR-Link developer's license to build applications or plug-ins that use the Logger API.

At the heart of the Logger API is the class *DtLogger*. With this class, you can programmatically access almost all core Logger functionality. This class manages the playing and recording of files, and provides access to the Logger's current configuration (such as whether data scaling is in use or whether the Logger should play immediately upon opening a file). Therefore, if you want to include a Logger in your application you simply need to instantiate a *DtLogger* object. In fact, this is how the Logger works. It creates a *DtLogger* object, and then adds mechanisms for interpreting command line arguments, accepting user input, and responding to Logger remote control messages.

If your goal in using the Logger API is to add standard Logger features to your application, the *DtLogger* class will be the only portion of the API that you need to understand. However, if you need to extend the Logger's functionality, add new file formats, or change the way standard commands are processed, you will need to understand the Logger's internal architecture, which is described in [“The Logger Architecture”](#), on page 7-12. A common reason for creating a plug-in is to extend the internal architecture.

In addition to the *DtLogger* class, the Logger API provides several classes that encapsulate the functionality of the various Logger user interfaces. There is a base class called *DtLgrInterface*, and subclasses that implement the Logger's graphical user interface, text interface, and remote control interface (for information about the remote control interface, see Chapter 6, [Remote Control](#)). These subclasses include *DtWinGui*, *DtMotifGui*, *DtTextGui*, and *DtLgrControlListener*. If you want to use any of these interfaces in your own applications, you can instantiate one or more of these classes. Alternatively, you can implement your own GUI and have it control the Logger's functionality by calling *DtLogger* member functions from your event handlers.

7.2. The DtLogger class

If you are using Logger functionality within your own application, you must instantiate a *DtLogger* yourself. If you are creating a Logger plug-in, then the standard Logger application creates the *DtLogger* object for you, and allows you to access it from your plug-in code.

All of *DtLogger*'s processing occurs in its `tick()` function. When you call `play()`, `record()`, `stop()`, or `pause()`, the Logger prepares itself for applying these modes. The action does not take effect until the next call to `tick()`. In the example in the next section, calling `DtLogger::play()` indicates that we want to enter play mode, but no packets get read from the file and sent until `DtLogger::tick()` is called.

7.2.1. Including the Logger in an Application

The following example illustrates how to include a Logger in your own code. It opens a Logger recording and plays it from its mid-point to the end. Once the end of the recording is reached, the example exits. The basic steps are:

1. Create a *DtLogger* instance.
2. Initialize the instance.
3. Invoke member functions such as `play()`
4. Call `DtLogger::tick()` periodically so that data can be processed and sent.
5. Optionally, create a graphical interface for displaying the Logger's status.

```

1  #include "logger.h"
2  #include "motifGui.h"
3  #include "ProcControl.h"
4
5  int main(int argc, char **argv)
6  {
7      // Create a Logger object.
8      #if DtHLA
9          DtLogger logger(
10             "VR-Link", "Logger API", new DtRprFomMapper(1.0));
11      #else
12          DtLogger logger(3000, 1, 15, 1);
13      #endif
14      // Let's set some the timescale to something other than zero.
15      logger.setTimeScale(1.0);
16
17      // Load a sample Logger recording for playback.
18      logger.loadFileForPlayback("example.lgr");
19
20      // Let's start in the middle of the recording.
21      logger.setTime(logger.fileLength()/2);
22
23      // Indicate that we want to enter play mode.
24      // No data will get read from the file or sent onto the
25      // network until DtLogger::tick() gets called.

```

```
26     logger.play();
27
28     // Let's put an interfaces on top of the Logger so it can repond to
29     // user input and display the Logger's current time and state.
30
31     // Instantiate a MAK Data Logger GUI.
32     DtMotifGui gui(&logger);
33
34     while (!gui.timeToQuit() && logger.filePosition() != DtLGR_END)
35     {
36         // Interfaces derived from DtLgrInterface should to be ticked
37         // each frame so that they can update their state to reflect the
38         // current internal state of the Logger.
39         gui.tick();
40
41         // You must call DtLogger::tick() each frame.
42         // This method does all Logger processing and returns
43         // the estimated arrival of the next packet. If the Logger
44         // does not know the arrival of the next packet, it returns
45         // -1.
46         logger.tick();
47
48         // Wait before processing more packets.
49         DtSleep(0.01);
50     }
51     return 0;
52 }
```

We create our *DtLogger* instance by passing to the constructor the arguments needed for creating an exercise connection [line 7]. The exercise connection will serve as our application's interface to the network regardless of whether we are using HLA or DIS. You can indicate that you want the Logger to use an existing exercise connection by passing a pointer to this connection to *DtLogger*'s constructor. In fact, if you are creating a *DtLogger* from within a VR-Link application, we recommend that you pass your application's normal exercise connection to the *DtLogger* in this fashion.

To prepare for playback, the timescale is set to 1.0 [line 15] and a sample Logger recording is loaded [line 18]. The timescale affects the speed at which the recording is read. In this example, we are keeping it simple and choosing to play the recording back at real time.

By default, the Logger will be in a stopped state after opening a file -- unless the default play option has been set using *DtLogger::setPlayAction()*. To indicate that we want to enter play mode, we call *DtLogger::play()* [line 26]. This call initializes the pipeline engine for playing back the file; however, no data will be actually read from the file and sent until *DtLogger::tick()* is called.

The next step initializes a Logger interface [line 32]. Interfaces take a pointer to the Logger and can be used to control the Logger and display Logger status information. The Logger library contains two graphical user interfaces -- one for UNIX platforms and one for Win32 platforms -- for controlling the Logger and displaying the Logger's status: *DtMotifGui* and *DtWinGui*. This example, which was written for UNIX, creates a Motif GUI.

The loop [line 34] executes until the end of the recording is reached, or the user presses "quit". In each iteration, we must call tick() for both the GUI and the Logger. Calling tick() for the GUI gives it time to update its appearance based on the current state of the Logger. Calling tick() for the Logger gives the Logger an opportunity to increment its internal clock, read the file, and send packets. This call returns an estimation of the time of the next packet arrival. In this example, we do not use this information, but this value can be used for calculating an amount of time to sleep until more packets from the file are ready for sending. For example, we could do something like the following:

```
1  DtTime estimatedNextPacketArrival = logger.tick();
2
3  // If we know the arrival of the next packet sleep either until the
4  // next packet is ready or 0.05 seconds has passed, whichever is
5  // sooner; otherwise, wait 0.01 seconds
6  if (estimatedNextPacketArrival > 0)
7  {
8      double theTime =
9          (estimatedNextPacketArrival - logger.idealTime());
10
11     if (logger.timeScale() != 0.0)
12     {
13         theTime = theTime / logger.timeScale();
14     }
15     else
16     {
17         theTime = 0.05;
18     }
19
20     DtSleep(theTime < 0.05? theTime : 0.05);
21 }
22 else
23 {
24     DtSleep(0.01);
25 }
```

This loop calculates an amount of time to sleep before processing more packets. If we know when more data is going to arrive, we try to sleep until that time. However, we set a limit on how long we are willing to wait, because we may want to do other processing in our application loop besides tick the Logger. The Logger's exercise connection checks the network for new information inside tick(), so it is also a good idea to call this regularly if you want to process information from the network. When the Logger is stopped, paused, recording, or at the beginning or end of a file, we will not know when the next packet is going to arrive, so we wait only 0.01 seconds.

7.2.2. Controlling the Logger Time Clock

The Logger keeps an internal clock that it uses for managing the time in a recording. This clock is separate from the simulation clock, which is used for sending network messages. The Logger's internal clock increments every time `DtLogger::tick()` is called, and it is this internal Logger time that is displayed in the Logger window's time indicator. In record mode, you cannot modify the state of this internal clock, but in playback mode, you can control the clock's speed and direction, and set and get the clock's current time. [Table 7-1](#) disrobes the DtLogger member functions for controlling the time clock.

Table 7-1: Time clock functions

Function	Description
<code>dataScaling()</code> <code>setDataScaling()</code>	Allows you to turn the scaling of entity information on and off. <code>dataScaling()</code> returns <code>DtTrue</code> if data scaling is in use, <code>DtFalse</code> , if it is not.
<code>idealTime()</code>	Returns the true current recording time. The <code>idealTime()</code> is the value that <code>time()</code> would always return if the Logger never fell behind while processing packets. Most of the time, the <code>idealTime()</code> and <code>time()</code> values match. If the they do not match, the Logger tries to process packets as quickly as possible until the two values match.
<code>timeScale()</code>	Gets the time scale.
<code>setTimescale()</code>	Sets the timescale. The timescale affects the speed at which the recording is read. ♦ A negative timescale causes the recording to be read backwards. ♦ A timescale between 1.0 and -1.0, excluding 0.0, plays the recording more slowly than real time. ♦ A timescale greater than 1.0 or less than -1.0 plays the recording faster than real time. ♦ A timescale equal to zero suspends the Logger clock, resulting in a paused state.
<code>time()</code>	Gets the Logger clock's time. The value returned by <code>time()</code> always corresponds to the current time in a loaded Logger recording.
<code>setTime()</code>	Sets the internal Logger clock's time. <code>setTime()</code> is only valid in playback mode, and only accepts values that fall between the beginning and end of a recording. Calling this function jumps playback to the specified time.

7.2.3. Controlling Logger Files

The Logger API contains many member functions for loading and managing Logger recordings. The *DtLogger* class has functions that provide high-level information about the current file (its name, length, and so on), and also includes a *file()* member function, which provides access to a *LtPlaybackFile* or *LtRecordFile* object. Chapter 8, [Using the Logger File API](#) describes how you can use these objects to iterate or search through the packets in a file, and modify or examine the contents of a file. [Table 7-2](#) describes *DtLogger* member functions that control Logger files.

Table 7-2: Logger file functions

Function	Description
<i>filePosition()</i>	Returns a <i>DtLgrFilePosition</i> (defined in VR-Link's <i>lcPdu.h</i> file) enumeration value describing the current file position. Valid values are <i>DtLGR_NOFILE</i> , <i>DtLGR_START</i> , <i>DtLGR_MIDDLE</i> , <i>DtLGR_END</i> .
<i>fileLength()</i>	Returns the length of a file. If no file is loaded, this returns 0. Valid in playback mode only.
<i>file()</i>	Returns a pointer to either a <i>LtPlaybackFile</i> or a <i>LtRecordFile</i> .
<i>gotoStart()</i> <i>gotoEnd()</i>	Moves to the beginning or end of a file. Valid in playback mode only.
<i>currentFile()</i>	Returns the name of the current file. In rare cases, it is possible for the Logger to have more than one file opened at a time. In this case, it returns the first loaded file.
<i>loadFileForRecording()</i>	Opens a file for recording and prepares the Logger pipeline for record mode.
<i>loadFileForPlayback()</i>	Opens a file for playback and prepares the Logger pipeline for playback mode.
<i>pause()</i>	Pauses playback of a recording.
<i>play()</i>	Causes the Logger to play a recording.
<i>record()</i>	Causes the Logger to start recording an exercise.
<i>setPlaybackFileCreator()</i> <i>setRecordFileCreator()</i>	Allows you to set a creator function for an object derived from either <i>LtPlaybackFile</i> or <i>LtRecordFile</i> . <i>DtFileSource</i> and <i>DtSendToFile</i> use these functions in <i>initSelf()</i> to create their file objects.
<i>stop()</i>	Causes the Logger to stop playing or recording.

7.2.4. Controlling the Logger's Configuration and Status

Table 7-3 describes *DtLogger* member functions for setting Logger default behaviors and for obtaining Logger state information.

Table 7-3: Configuration and status functions

Function	Description
<code>eofAction()</code> <code>setEofAction()</code>	Gets or sets the action the Logger takes when it reaches the end of a recording. Valid choices are: ♦ Stop at the end of the recording (which will delete all entities). ♦ Pause at the end of a recording (which will not delete any entities). ♦ Wrap to the beginning or ending of the recording, depending on the direction it is being played.
<code>inPlayback()</code>	Returns whether the Logger is currently set up for playing back a Logger recording or for recording a new file. This value changes whenever either <code>loadFileForPlayback()</code> or <code>loadFileForRecording()</code> is called.
<code>mode()</code>	Returns the current mode. This value is set internally when you open files for playback or for recording and when you call functions such as <code>play()</code> , <code>stop()</code> , <code>pause()</code> , and <code>record()</code> . This function returns a <code>DtLgrMode</code> enumeration (which is defined in VR-Link in <i>lcPdu.h</i>) value that can be either <code>DtLGR_PLAY</code> , <code>DtLGR_PAUSE</code> , <code>DtLGR_STOP</code> , or <code>DtLGR_EMPTY</code> .
<code>playAction()</code> <code>setPlayAction()</code>	Gets or sets what the Logger state when it opens a new file for playback. Valid choices are stop mode and play mode.
<code>recordAction()</code> <code>setRecordAction()</code>	Gets or sets what the Logger action when it opens a new file for recording. Valid choices are stop mode and record mode.
<code>recordingLogger- ControllInfo()</code> <code>setRecording- LoggerControllInfo()</code>	Gets or sets whether we want to record Logger control messages.
<code>setMirroring()</code>	Specifies whether the Logger will send Logger control messages with certain commands. For example, when mirroring is on and <code>play()</code> is called, a <code>DtLgrPlay</code> message is sent.
<code>setDefaultRecord- SourceParameter()</code>	Specifies a parameter to be sent to the default recording source. For example, if you want to override a default recording source (for example, <i>DtDisSource</i>) so that it uses the <code>DtString</code> parameter in its constructor, you set the value to pass here. The next time <code>loadFileForRecording()</code> is called, the value you have set is passed to your class's constructor. The default value is "None".

Table 7-3: Configuration and status functions

Function	Description
<code>setPacketLimit()</code>	Allows you to configure how many packets the Logger can generate during one pass through its internal processing pipeline. The default limit is 1000 packets and should only need to be modified in rare cases.

7.2.5. Controlling Logger Annotations

The *DtAnnotationManager* class maintains a list of annotations, which you can access through the member function `DtLogger::annotations()`. An individual Logger annotation is called a *DtNote* and consists solely of a time and message. When you add annotations to this list, the annotation manager creates a *DtNote* object for you, which you can access afterwards by calling `DtAnnotationManager::get()`. This class also contains member functions for loading and saving annotations from a file, and iterating through the list. For example, if you wanted to print out the contents of `montyDemo.ant`, you could do the following:

```
DtAnnotationManager* mgr = myLogger->annotations();
#ifdef DtHLA
mgr->loadFromFile("../config/montyDemo.ant", myLogger->exerciseConn());
#else
mgr->loadFromFile("../config/montyDemo.ant");
#endif
for (DtNote* n = mgr->first(); n; n = mgr->next())
{
    printf("%f: %s\n", n->time(), n->msg().string());
}
```

If you want to jump to the location of the first annotation in the list, you could do the following:

```
DtNote* note = mgr->first();
myLogger->setTime(note->time());
```

To remove an annotation from the list, call `DtAnnotationManager::remove()`. To remove all the elements in the list, call `DtAnnotationManager::empty()`.

7.2.6. Setting Protocol-Specific Parameters

The API lets you change protocol-specific parameters so that you can customize the Logger to your simulation and improve its performance. You have access to the Logger's exercise connection, so you can use network and protocol options that are available in VR-Link, in your Logger applications.

DIS only

For DIS, you can customize the heartbeat and timeout intervals.

HLA only

In HLA, you can set which FOM classes to which the Logger should subscribe. Also, you can optimize the manner in which interactions are recorded if you choose to use the Logger's federate ambassador.

[Table 7-4](#) describes DtLogger member functions for getting and setting protocol-specific parameters.

Table 7-4: Protocol-specific functions

Function	Description
<code>exerciseConn()</code>	Gets a pointer the Logger's exercise connection.
<code>heartbeat()</code>	Specifies the heartbeat interval. (DIS)
<code>setInterSubscription-List()</code>	Specifies a list of FOM interactions for the Logger to subscribe to. (HLA)
<code>setObjectSubscription-List()</code>	Specifies a list of FOM classes for the Logger to subscribe to. (HLA)
<code>setPassiveSubscription()</code>	Specifies whether the Logger should subscribe either actively or passively to FOM classes. (HLA)
<code>setUsingLogger-FederateAmbassador()</code>	Specifies whether the Logger's custom Logger ambassador should be used instead of VR-Link's. (This call only has an effect when the Logger is creating its own exercise connection). This Logger federate ambassador is part of an optimization for the interactions -- it bypasses some VR-Link processing by channeling interaction packets directly from the RTI to the Logger. (HLA)
<code>timeout()</code>	Specifies the timeout interval. (DIS)

7.2.7. Managing Pipeline Objects

[Table 7-5](#) describes DtLogger member functions for registering, adding, and removing objects from its internal packet processing pipeline. You can share data between elements in the pipeline and the main application. For a more detailed discussion of the pipeline and pipeline elements, see [“The Logger Architecture”](#), on page 7-12.

Table 7-5: Pipeline object functions

Function	Description
<code>addObject()</code>	Creates, initializes, and adds a registered element to the Logger pipeline’s source, filter, action, or send lists.
<code>fetchFromClipboard()</code>	Tries to get data from a pipeline element using a keyname.
<code>postToClipboard()</code>	Makes data available for pipeline elements and sets a keyname, which can be used to fetch the data later.
<code>registerObject()</code>	Registers a pipeline element creator with the Logger pipeline.
<code>removeFromClipboard()</code>	Removes the data. Whoever puts the data on the clipboard is responsible for deleting it later.
<code>removeObject()</code>	Removes a registered element from the Logger pipeline’s source, filter, action, or send lists. Then it deletes the element.

7.3. The Logger Architecture

This section describes how the Logger generates and processes packets. This information will be useful if you want to change the core functionality of the Logger (how it generates packets to or from a file, or how it reads packets from the network) or want to add to the core functionality (for example, filter certain packet types or play two files at once).

The *DtLogger*'s *DtLgrPipeline* object (defined in *lgrPipeline.h*) organizes the different stages of Logger packet processing. The Logger has four distinct processing steps:

- It generates packets
- Optionally, it inspects the generated packets
- Optionally, it filters the generated packets
- It sends the packets.

Figure 7-1 illustrates the basic elements of the Logger's core engine or packet processing pipeline. They are sources, actions, filters, and senders. Sources generate Logger packets from messages received from a file or the network. The packets are passed to actions, filtered if appropriate, and exit the pipeline through the senders.

The engine makes no distinction between playback and record modes. It just has different sources and different outputs in each mode. For example, when the Logger records a DIS exercise, it chooses a DIS connection as its source element and a file as its send element. The Logger can filter out packets so that they do not get sent to the recorded file or the network. You can add actions to monitor the packet stream and compile statistics.

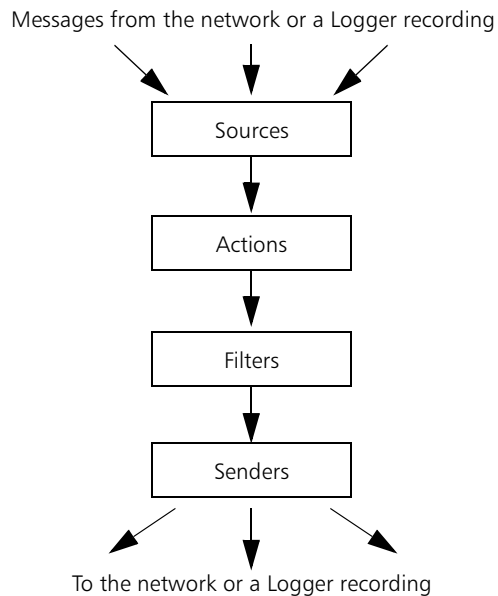


Figure 7-1. Logger engine

7.4. Logger Pipeline Elements

There are four basic types of pipeline elements:

- ♦ Sources
- ♦ Actions
- ♦ Filters
- ♦ Senders.

Each of the element types is derived from a pipeline element class called *DtLoggerStage*. All objects derived from *DtLoggerStage* contain abstract functions to initialize, activate, deactivate, and verify themselves. Source objects can generate Logger packets, which package the messages from the external source. Filters, actions, and sends can process a list of packets generated by a source object.

All *DtLoggerStage* classes have a *tick()* function for doing extra processing. All *DtLoggerStage* objects are given a *DtLgrState* object (defined in *lgrState.h*) to initialize themselves. *DtLoggerState* is a repository for Logger state data. This data includes the Logger's current play state, clock time, and exercise connection. This object also has the same clipboard mechanism as *DtLogger*, so that different pipeline elements can share data.

Objects are implemented by the following classes:

- ♦ Source objects are implemented by *DtSourceObj* classes (defined in *sourceObj.h*).
- ♦ Filter objects are implemented by *DtFilterObj* classes (defined in *filterObj.h*).
- ♦ Action objects are implemented by *DtActionObj* classes (defined in *actionObj.h*).
- ♦ Send objects are implemented by *DtSendObj* classes (defined in *sendObj.h*).

The implementation for all these classes is in the *liblgcore* library.

7.4.1. Pipeline Element Member Functions

Table 7-6 describes the member functions for the pipeline element classes. Most of the member functions are common to the four types of elements.

i

If you want to derive new element classes, you must override some of the functions, as indicated. If you are using default element classes supplied by MÄK (see “Default Pipeline Elements”, on page 7-17) you do not have to override these functions.

Table 7-6: Member functions for pipeline elements

Function	Description
Functions that apply to all elements:	
deactivate()	The deactivate() member function indicates that the object should no longer produce data. If it currently has stored data, it may delete the data. The pipeline will no longer be calling process() to receive data. This is a pure virtual function and you must override it if you want to derive a new type of element.
getName()	Returns the attribute myName. The myName attribute is a <i>DtString</i> that holds the name of the instance of the class. The name does not have to be unique; however, it is often useful for debugging purposes to give it a meaningful name. This name is typically passed in or created during object construction. It is returned by the getName() member function.
initSelf()	The initSelf() member function is the first function called after construction of an object. The object is passed the <i>DtLoggerState</i> and is expected to perform initialization routines. <i>DtLoggerState</i> is a pointer to a repository of Logger state data such as the Logger’s current play state, clock, and exercise connection. If the object needs the <i>DtLoggerState</i> pointer later, it should save it here. This is a pure virtual function and you must override it if you want to derive a new type of element.
isOk()	Returns the attribute myStatus. The myStatus attribute is a <i>DtBoolean</i> that is used to indicate that there is a problem with this object and the object should be removed. Typically it is returned by the isOk() member function. The status of a pipeline element is checked whenever the element has finished processing packet data. If it returns DtFalse, the pipeline removes the object and destroys it. The function is not abstract and by default returns the value of the myStatus attribute.
printData()	The printData() member function returns identifying information about the object. This function is not abstract and depending on the type of object, returns either “A Source”, “A Filter”, “An Action”, or “A Send” by default.

Table 7-6: Member functions for pipeline elements

Function	Description
reset()	When the Logger engine calls the reset() member function, the object should return to its initial state. For example a file object should return to the beginning of the file. This is a pure virtual function and you must override it if you want to derive a new type of element.
sendPacket()	The sendPacket() member function is called by processData() to act on each individual packet. This is a pure virtual function and you must override it if you want to derive a new type of element.
tick()	The tick() member function is used by the pipeline to yield execution time to the object. This function is virtual, but not abstract. The default version of this function does nothing.

Functions that apply only to action, filter, and send elements:

activate()	The activate() member function is called to indicate that the object will begin to receive data in its process() member function. This is a pure virtual function and you must override it if you want to derive a new type of element.
processData()	The pipeline sends the list of packets to the processData() member function so the object can perform appropriate actions. Action and Send objects should only parse the list and read the data. Filter objects may alter the list, but if a filter removes a packet, it must also delete it. By default: - Action objects call actOnPacket() for each packet in the list - Filter objects call filterPacket() on every packet in the list - Send objects call sendPacket() on every packet in the list.

Functions that apply only to source elements:

activate()	When the pipeline wants to begin receiving data from a source, it calls the activate() member function. The parameter startTime holds the time from which the source should begin sending data. If it has earlier data it should ignore it. The return value should indicate to the pipeline what time this object will be ready to send its first packet. If unknown, the object should return -1. The pipeline begins calling processes to receive the data. This is a pure virtual function and you must override it if you want to derive a new type of element.
generatePackets()	The generatePackets() member function is used by the pipeline to grab waiting data from the object. The parameter currentTime informs the object of the current simulated time. The object should create LtPacketInfo objects and add them to the list. The responsibility for deleting the LtPacketInfo objects belongs to the pipeline. The return value is the next time the object will have data to send. If unknown, the object should return -1. The default action of this function is to call generatePacket() until it returns a NULL.

Table 7-6: Member functions for pipeline elements

Function	Description
<code>generatePacket()</code>	The <code>generatePacket()</code> member function creates one packet a time and is called by the <code>process()</code> member function. The <code>currentTime</code> parameter is the simulation time that packets should be generated for. The <code>nextTime</code> parameter is a return value for when the next packet will be ready. This is a pure virtual function and you must override it if you want to derive a new type of element.
Functions that apply only to action elements:	
<code>actOnPacket()</code>	The <code>actOnPacket()</code> member function is where the action performs any packet-specific operation.
Functions that apply only to filter elements:	
<code>filterPacket()</code>	The <code>filterPacket()</code> member function is where the filter can make a decision about whether a packet should pass its filter or not. This function is allowed to change the packet. A return value of <code>DtTrue</code> indicates to the <code>processData()</code> member function that the packet should be removed from the list.

The Logger calls `initSelf()` immediately after an element is added to the pipeline. In the case of source and send objects, which are continually being added and removed from the pipeline as a result of calls to `loadFileForPlayback()` and `loadFileForRecording()`, this call will occur often. Filters and actions are reset whenever a new object is loaded for playback or recording. Whenever `play()` or `record()` are called, the Logger calls `activate()` on every element to indicate that all elements should prepare to either generate or process packets. Conversely, whenever `stop()` is called, the Logger calls `disable()` on every element to indicate that all elements should stop producing data.

Packets are processed as follows:

1. The `tick()` function is called on every element.
2. The pipeline calls `generatePackets()`, which calls `generatePacket()`, for each source.
3. The pipeline calls `processData()`, which in turn calls `actOnPacket()`, for each action.
4. The pipeline calls `processData()`, which in turn calls `filterPacket()`, on each filter.
5. The pipeline calls `processData()`, which calls `sendPacket()`, for each send.
6. The pipeline checks the status of each pipeline element using the function `isOk()`. If an error has occurred and `isOk()` returns `DtFalse`, the pipeline removes the offending object and destroys it.

7.4.2. Default Pipeline Elements

The Logger has several *DtLoggerStage* objects, which implement basic Logger functionality such as the playing and recording of files using DIS, HLA, or a TCP/IP connection.

- ♦ *DtFileSource* – reads packets from a file.
- ♦ *DtHlaActiveSource* – obtains packets from an active subscription to an HLA exercise.
- ♦ *DtHlaPassiveSource* – obtains packets from a passive subscription to an HLA exercise.
- ♦ *DtHlaSimulate* – sends packets to an HLA exercise.
- ♦ *DtDisSource* – reads packets from a DIS exercise.
- ♦ *DtDisSimulate* – sends packets to a DIS exercise.
- ♦ *DtSendToFile* – sends packets to a file.
- ♦ *DtScreenDump* – writes packets, in human readable form, out to the screen or to a file.

The following classes allow you to chain Logger pipelines together by sending or receiving packets through a special, protocol-independent TCP channel.

- ♦ *DtTCPSource* – reads packets from a TCP port.
- ♦ *DtTcpPort* – sends packets to a TCP connection.

You can override any of these elements by deriving your own classes from them and registering their creator with the Logger. For example, suppose we derived our own file source object called *MyFileSource*. We would replace *DtFileSource* with our own class:

```
myLogger->registerObject("FileSource", MyFileSource::create);
```

7.4.3. Example of Overriding A Pipeline Element

The following example illustrates how to create a filter by deriving a class called *ExampleFilter* from *DtFilterObj*. You could implement this class when you embed the Logger in your application, or as part of a plug-in. The full example creates a plug-in from this class. The full sample code plus makefiles and project files are in *./plugins/example/exampleFilter*.

The following code is the header file for the *ExampleFilter* class:

```
#ifndef ExampleFilter_H
#define ExampleFilter_H

#include "filterObj.h"

// Header file for the example filter object.

class DtEntityType;

class ExampleFilter : public DtFilterObj
{
```

```
public:
    // constructors
    ExampleFilter( const char* theName );

    // destructors
    virtual ~ExampleFilter();

    // This method is called once after construction to perform
    // some initialization.
    virtual void initSelf( DtLgrState* theLogger );

    // This method is called when the Logger agent wants
    // to send data to this object.
    virtual void activate();

    // This method is called when the Logger agent is no longer
    // going to send data to this object.
    virtual void deactivate();

    // This method is where the filtering is performed. A return
    // value of DtTrue will cause the packet to be removed by
    // processData.
    virtual DtBoolean filterPacket( LtPacketInfo* packet);

    // This method returns a string that supplies identifying
    // info about the object.
    virtual const char* printInfo() const;

    // This method is called to reset the filter
    virtual void reset();

    // factory method
    static void* create( DtString param );

private:

    // Protect against copy construction.
    ExampleFilter( const DtFilterObj&);

    DtLgrState* myLIF;

    // ID of the object that should be filtered.
    DtEntityType *myEntityTypeToFilter;
};

#endif
```

Example Source File

The example is designed to filter out an entity type from the playback of a Logger file. The appropriate part of the Logger architecture to perform this function is the filter. To implement a filter, we first inherit from *DtFilterObj* [line 7]. This provides us the basic member functions that we must implement.

The parameter that is passed into the constructor can be used to control what entity type the Logger should filter. To save this value we create a *DtEntityType* using the parameter and save it in the member variable, *myEntityTypeToFilter* [line 12].

The *DtLogger* pointer will be needed in later phases of executions, so during the *initSelf()* member function this pointer is stored in the *myLIF* member variable [line 25].

The pipeline will begin sending packet lists to the *processData()* member function that is implemented in the *DtFilterObj*. This function then proceeds to call the *filterPacket()* member function that we have implemented [line 45]. This filter has been created in a protocol-independent manner. However the *filterPacket()* member function differs greatly between DIS and HLA. The use of *#if DtHLA* indicates that the code is conditional depending on the HLA or DIS versions that are being compiled [line 53].

The following code is the implementation of the *ExampleFilter* class.

```
1  #include "exampleFilter.h"
2  #include "stateInfo.h"
3  #include "packetInfo.h"
4  #include "intHashKey.h"
5
6  // constructors
7  ExampleFilter::ExampleFilter( const char* theName ) :
8      DtFilterObj("Example Filter")
9  {
10     // Parameter passed into the filter will serve as the entity
11     // type that will be filtered.
12     myEntityTypeToFilter = new DtEntityType(theName);
13 }
14
15 // destructors
16 ExampleFilter::~ExampleFilter()
17 {
18     delete myEntityTypeToFilter;
19 }
20
21 // This method is called once after construction to perform
22 // some initialization.
23 void ExampleFilter::initSelf(DtLgrState* theState )
24 {
25     myLIF = theState;
26 }
27
28 // This method is called when the Logger agent wants
29 // to send data to this object.
30 void ExampleFilter::activate()
31 {
32     // no action needed
33 }
34
35 // This method is called when the Logger agent is no longer
36 // going to send data to this object.
37 void ExampleFilter::deactivate()
38 {
39     // no action needed
40 }
41
42 // This method is where the filtering is performed. A return
43 // value of DtTrue will cause the packet to be removed by
44 // processData.
45 DtBoolean ExampleFilter::filterPacket( LtPacketInfo* packet)
46 {
47     DtBoolean retval = DtFalse;
```

```
48
49     if(myLIF->mode() != DtLGR_RECORD)
50         // Only perform the filter during playback.
51         {
52
53     #if DtHLA
54         // In HLA the packets are of three types: Update, Removal, and
55         // Interactions.
56         // The filter is only interested in updates and removals.
57         switch(packet->packetKind())
58         {
59             case LtPacketUpdate:
60             case LtPacketRemoval:
61             {
62                 // In HLA an update packet does not need to encode
63                 // all of the object's data. This means that we may
64                 // receive and update for a packet we wish to filter,
65                 // however the update does not have the information in
66                 // it to allow us to apply the filter.
67                 //
68                 // To solve this problem the LoggerInterface class
69                 // maintains a list of StateReps that is updated before
70                 // any filter is applied. This information can be used
71                 // to test if the packet should be filtered.
72                 DtHashList* theStateInfoList = myLIF->stateReps();
73
74                 // The hashlist is a list of StateInfo Objects hashed on
75                 // objectId.
76                 DtStateInfo* theStateInfo =(DtStateInfo*)
77                     theStateInfoList->lookup(DtIntHashKey(
78                         packet->objectId()));
79
80                 // If the update is an entity then there will be a
81                 // stateInfo.
82                 if(theStateInfo != NULL)
83                 {
84                     // If the stateInfo says the packet is already
85                     // filtered then mark it for deletion, but don't
86                     // bother to test it.
87                     if(theStateInfo->filtered())
88                     {
89                         retval = DtTrue;
90                     }
91                     else if(theStateInfo->esr()->entityType() ==
92                         *myEntityTypeToFilter)
93                     {
94                         // Filter this entity.
95                         retval = DtTrue;
96
97                         // Mark its state as filtered.
98                         theStateInfo->setFiltered(DtTrue);
99                     }
100                 }
101             }
102             break;
103         default:
104             // Do nothing with the packet.
105             break;
106         }
```

```
106 #else
107     // In DIS the packets represent PDU's.
108     if(packet->packetKind() == LtPacketPdu)
109     {
110         // DIS does not have the challenge that HLA has. All the
111         // data regarding the object is in the PDU.
112         if( ((DtNetPduHeader *) (packet->packet()))->DtKind ==
            DtEntityStatePduKind)
113         {
114             DtEntityStatePdu esPdu((DtNetEntityStatePdu*)
                packet->packet());
115             if(esPdu.entityType().match(*myEntityTypeToFilter))
116             {
117                 retval = DtTrue;
118             }
119         }
120     }
121
122 #endif
123 }
124
125 return retval;
126 }
127
128 // This method returns a string that supplies identifying
129 // information about the object.
130 const char* ExampleFilter::printInfo() const
131 {
132     DtString s = "Filter for ";
133     s += myName;
134     s += " Objects\n";
135
136     return s.string();
137 }
138
139 void* ExampleFilter::create(DtString param)
140 {
141     void* retval = new ExampleFilter(param.string());
142     return retval;
143 }
144
145
146 void ExampleFilter::reset()
147 {
148     // This method is not used in this object.
149 }
```

7.5. Creating Plug-ins

A plug-in is a dynamically linked module that is loaded by the Logger at runtime. You can use plug-ins to modify the Logger executable. For example, you can change how the Logger processes packets, or you can override the Logger's *DtLogger* object with your own. When you use a plug-in, all the basic functionality of the Logger remains intact: command line arguments are read and interpreted, user interfaces are created, and Logger control messages are intercepted. After all the normal initialization work, the application loads your plug-in and your changes take effect.

The architecture section of this chapter ([“The Logger Architecture”](#), on page 7-12) explains the two steps involved in adding new elements to the pipeline. First, a creator function for the element must be registered with a call to `DtLogger::registerObject()`. Then, an instance of the element can be created, initialized, and added to the pipeline with a call to `DtLogger::addObject()`.

The first part of this section describes how to register and add new elements to the Logger's pipeline using a plug-in. The second part of this section describes how to override the application's default *DtLogger* instance with your own derived class.

7.5.1. Initializing A Plug-in

All plug-in libraries must export a “C” style function, which will be recognized by the Logger executable when it loads the plug-in. When you want to add new pipeline elements or override existing ones you must export a function called `InitPlugin()`. If you want to override *DtLogger*, you must export `CreateLogger()`. You can export either one or the other or both, but the functions must look like the following:

```
#include "initMethods.h"

extern "C"
{
    LGR_EXPORT DtLogger* CreateLogger(void* data)
    {
        // User Logger creation belongs here.
    }

    LGR_EXPORT void InitPlugin(void* data)
    {
        // User register code belongs here.
    }
}
```

The Logger application calls each of these functions during initialization. The data parameter is a pointer to the Logger application object, *DtLoggerApp*. If the Logger loads a module that exports `CreateLogger()`, it uses the function to create its *DtLogger* instance. If more than one module contains this function, the Logger only uses the first one. After a *DtLogger* instance is created, the Logger calls every module's `InitPlugin()` function. You can access start-up options and command line arguments through the *DtLoggerApp* object. Most importantly, the *DtLoggerApp* provides access to its *DtLogger* object.

7.5.2. Modifying the Pipeline Using A Plug-in

When the application calls `InitPlugin()`, the *DtLogger* instance has been created and can be used to register and add pipeline elements. For example, if we wanted to add the *ExampleFilter* object to the pipeline (see “[Example of Overriding A Pipeline Element](#)”, on page 7-17), the contents of `InitPlugin()` could look as follows:

```
LGR_EXPORT void InitPlugin(void* data)
{
    DtLogger* logger = ((DtLoggerApp*) data)->logger();
    logger->registerObject("ExampleFilter", ExampleFilter::create);

    // We can add some instances of our filter to the pipeline here, too.
    logger->addObject(DtFilter, "ExampleFilter", "1:2:225:2:1:0:0");
    // filter US planes

    logger->addObject(DtFilter, "ExampleFilter", "1:2:222:2:1:0:0");
    // filter Soviet planes
}
```

In this sample code, we register and add *ExampleFilter* to the pipeline. We could have simply registered the creator, then later, we could have added our filters using an `EnableObjects()` function, or the `enable` MTL parameter.

It is very easy to replace one of the default Logger pipeline elements using a plug-in's `InitPlugin()` function. For example, to override *DtFileSource* with your own class called *MyFileSource*, you simply need to do the following:

```
LGR_EXPORT void InitPlugin(void* data)
{
    ((DtLoggerApp*) data)->registerObject("DtFileSource",
    MyFileSource::create);
}
```

An element class that has been registered with the Logger can be instantiated and added to the Logger pipeline in two ways:

- ♦ The objects can be created in code with the `EnableObjects()` function in the plug-in DLL
- ♦ The objects can be specified by the `enable` parameter in the *logger.mtl* file.

The `EnableObjects()` function is similar to `InitPlugin()`; however, it is not mandatory. The function must be a “C” style function of the following form:

```
LGR_EXPORT DtList* EnableObjects()
```

The function returns a *DtList* containing arrays of three character strings, as follows:

- ♦ The first string describes the element type of the object to be created. The valid values for this string are “source”, “action”, “filter”, and “send”.
- ♦ The second string is the name of the object to be created. This value should match the name the object was registered with.
- ♦ The third string is the parameter that will be passed to the object's constructor.

The following code is an example of an `EnableObjects()` member function using our `ExampleFilter`:

```
extern "C"
{
    LGR_EXPORT DtList* EnableObjects( )
    {
        // Always build this object.
        static char enableTriple1[][30] = { "filter",
                                             "ExampleFilter",
                                             "1:2:225:2:1:0:0" };

        static char enableTriple2[][30] = { "filter",
                                             "ExampleFilter",
                                             "1:2:222:2:1:0:0" };

        static DtList aList;
        aList.add(enableTriple1);
        aList.add(enableTriple2);
        return &aList;
    }
}
```

The can enable object by adding the `enable` MTL parameter to *logger.mtl*. This parameter is a list that contains the same three string sets that are used in the `EnableObjects()` method:

```
(setq enable
  (list
    (list "MyClass" "MyType" "A Value")
  )
)
```

The MTL command that would create the object for the `ExamplePlugin` is as follows:

```
(setq enable
  (list
    (list "filter" "ExampleFilter" "1:2:225:2:1:0:0")
    (list "filter" "ExampleFilter" "1:1:222:2:1:0:0")
  )
)
```

The MTL lines are equivalent to the `EnableObjects()` function in the previous example.

7.5.3. Overriding *DtLogger* from A Plug-in

To create your own Logger class, create a class that inherits from *DtLogger*. You now have access to all the functions defined in *logger.h*. It is important for the overridden class to call the original *DtLogger* functions to preserve the original Logger functionality. For example:

```
void MyLogger::play()
{
    // Perform my own action on play.
    doSomething();

    // Call the original Logger play method.
    DtLogger::play();
}

void MyLogger::stop()
{
    // Perform my stop call.
    stopSomething();

    // Call the original stop method.
    DtLogger::stop();
}
```

To tell the application to use your new *DtLogger* class, define *CreateLogger()* in your plug-in and have it return a pointer to your new class.

```
extern "C"
{
    LGR_EXPORT DtLogger* CreateLogger(void* data)
    {
        DtLoggerApp* app = (DtLoggerApp*) data;
        return new MyLogger(app->port(), 1, 12, 3);
    }
}
```

The application pointer contains methods for obtaining configuration data such as the port number, federation name, and RPR FOM version. This allows you to use command line arguments to initialize your Logger object. Also, you can add your own command line arguments and interpret them here. In the previous example, we get the port number for the application and use it to instantiate our Logger.

7.5.4. Loading Plug-Ins

By default, the Logger loads all plug-ins that are in the *./plugins/bin* directory. However, you can override this behavior by specifying the plug-ins to load in the configuration file *logger.mtl*. This file defines three plug-in-related variables:

- ♦ **pluginDir** - The **pluginDir** variable specifies the directory in which the plug-in modules are located.
- ♦ **plugins** - The **plugins** variable is a list of the plug-ins that should be loaded by the Logger. By specifying which plug-ins should be loaded in the configuration file, different configurations can be maintained in different configuration files without having to specify multiple installation locations for the Logger code.
- ♦ **enable** - The **enable** variable specifies elements to create immediately. This is how plug-in objects such as filters, that have no inherent GUI support, can be added to the Logger pipeline.

An example of the plug-in section of *logger.mtl*:

```
; This lists the installed plugins.
; Each plug-in will be loaded before initialization.
(setq pluginDir "..\binRPR\" )
(setq plugins
  (list
    "myPlugin.dll"
  )
)
```

7.6. Compiling and Linking with the Logger API

UNIX:

To compile against the Logger API in UNIX, you must add the following include directive to your compile line:

```
-I loggerDir/include
```

where *loggerDir* is the path to the root of your Logger installation.

HLA only

For HLA, link against:

```
-LloggerDir/lib -llloggerRPR -llgrFileRPR -llgrcoreRPR
```

DIS only

For DIS, link against:

```
-LloggerDir/lib -lllogger5 -llgrFile5 -llgrcore5
```

Follow the directions for compiling and linking with VR-Link, in your VR-Link documentation. The Logger library link flag above must come before the VR-Link linking flags on your link line.

PC:

On PCs, you must set the paths to the Logger's *include* and *lib* directories and list the appropriate libraries in your project file. We distribute both debug and non-debug DLL versions of our *lgrcore*, *logger*, and *lgrapp* libraries.

HLA only

For HLA, link against *liblgrFileRPRMD.lib*, *liblgrcoreRPRRT.lib*, *libloggerRPRRT.lib*, and *liblgrappRPRRT.lib*.

DIS only

For DIS, link against *liblgrFile5MD.lib*, *liblgrcore5RT.lib*, *liblogger5RT.lib*, and *liblgrapp5RT.lib*.

Example makefiles and project files are in the *./example/loggerAPI* directory.



Using the Logger File API

In addition to the Logger API (see [Chapter 7, Using The Logger API](#)), the Logger supports a Logger File API. The Logger File API allows you to read and write Logger recordings from applications.

Using the Logger File API	8-2
Reading A Logger File	8-2
Creating An LtLoggerPlaybackFile	8-3
Reading Data	8-4
Writing A Logger File	8-5
Creating An LtLoggerRecordFile	8-5
Writing Data	8-6
Interpreting LtPacketInfo	8-6
Editing Logger Files	8-7
Compiling and Linking with the Logger Library	8-9

8.1. Using the **Logger File API**

In addition to the standalone tools that help you view or modify MÄK Data Logger files, we provide an API (application programmer's interface) to the Logger file format, and a library that implements the API. The API allows you to read and write MÄK Logger files from your own programs.



You must have a developer's license for VR-Link to use the Logger file library.

The Logger file API is in *include/loggerFile.h*. In UNIX, the libraries are in the *lib* directory:

- ♦ *liblgrFileRPR.a* - supports HLA
- ♦ *liblgrFile5.a* - supports DIS.

For PCs, the following libraries are provided:

- ♦ *liblgrFile5RT.lib*
- ♦ *liblgrFile5RT.dll*
- ♦ *liblgrFileRPRRT.lib*
- ♦ *liblgrFileRPRRT.dll*
- ♦ *liblgrFile5MD.lib*
- ♦ *liblgrFileRPRMD.lib*.

See “Compiling VR-Link Applications” in *VR-Link Developer's Guide* for information about the different libraries.

The Logger file API has two main classes: *LtLoggerRecordFile* and *LtLoggerPlaybackFile*. Use *LtLoggerRecordFile* when you want to write a Logger file, and *LtLoggerPlaybackFile* when you want to read a Logger file. Both of these classes are derived from *LtLoggerFile*.

8.1.1. Reading A Logger File

To read a Logger file, create an instance of *LtLoggerPlaybackFile*, and use its member functions to navigate through a file and read packets. When we use the term “packet”, we are referring to a packet of data in the Logger file: either a DIS PDU, or an HLA update, interaction, or object removal.

8.1.2. Creating An *LtLoggerPlaybackFile*

To read data from standard input, rather than from an actual file, pass the string “stdin” as the filename argument to the *LtLoggerPlaybackFile* constructor.

DIS only

In DIS, the *LtLoggerPlaybackFile* constructor takes only the name of the file you want to read:

```
LtLoggerPlaybackFile inFile("myFile.lgr");
```

HLA only

In HLA, a Logger file contains information about the FOM being used in the recorded exercise. The reason for this is that during playback, we may have to convert class, attribute, and parameter handles stored in the Logger file to their corresponding handles in a different federation execution. (There is no guarantee that the same classes, attributes and parameters are assigned the same handles in different federation executions, especially if the FED file has changed at all).

When you pass a valid *DtExerciseConn* (a VR-Link object that represents a connection to an HLA exercise) pointer as the second argument to the *LtLoggerPlaybackFile* constructor, *LtLoggerPlaybackFile*'s packet reading functions perform conversions to the handles assigned by the *DtExerciseConn*'s RTI connection. The following example shows the *LtLoggerPlaybackFile* constructor for HLA.

```
LtLoggerPlaybackFile("myFile.lgr", myExConn);
```

If the optional *DtExerciseConn* argument is omitted, the update and interaction information returned by its packet reading calls contain exactly the same class, attribute, and parameter handles that were recorded from the original exercise.

8.1.3. Reading Data

When you first create an *LtLoggerPlaybackFile*, you are at the beginning of the file. The functions described in [Table 8-1](#) either read and return a packet, or change the location of a file pointer that determines where you are in the file:

Table 8-1: *LtLoggerPlaybackFile* functions for reading or returning packets

Function	Description
<code>readPacket()</code>	Reads and returns the next packet, NULL if none. In HLA, packets might be skipped if the file was created with a non-NULL exercise connection, and the packet cannot be converted to the exercise connections's FOM.
<code>readPacketBackwards()</code>	Reads and returns the previous packet, NULL if none. In HLA, packets might be skipped if the file was created with a non-NULL exercise connection, and the packet cannot be converted to the exercise connection's FOM.
<code>skipPacketForwards()</code> <code>skipPacketBackwards()</code>	Skip over the next packet either forwards or backwards. Return 1 if successful, 0 if there are no packets in that direction.
<code>jumpToStart()</code> <code>jumpToEnd()</code>	Go to the beginning or end of the file.
<code>seekTime(DtTime t)</code>	Jump to time t (in seconds) in the file.
<code>seekForward(DtTime t)</code> <code>seekBackward(DtTime t)</code>	Jump forward or backwards in the file by t seconds.

The functions described in [Table 8-2](#) provide information about the file and about your current location in the file:

Table 8-2: *LtLoggerPlaybackFile* functions for reporting time and location

Function	Description
<code>atStart()</code> <code>atEnd()</code>	Returns 1 if at you are at the beginning or end of the file, otherwise returns 0.
<code>fileTimeBackwards(DtTime *t)</code> <code>fileTimeForwards(DtTime *t)</code>	Fills t with the time of the next or previous packet. Returns 1 if successful, 0 if there is no packet in that direction.
<code>firstPacketTime(DtTime *t)</code> <code>lastPacketTime(DtTime *t)</code>	Fills t with the time of the first or last packet in the file.

The functions `readPacket()` and `readPacketBackwards()` return their data in the form of a pointer to an instance of the *LtPacketInfo* class. See “[Interpreting LtPacketInfo](#)”, on page 8-6 for information about how to interpret this object.



The *LtPacketInfo* instance returned is valid only until the next call to one of *LtLoggerPlaybackFile*'s reading or jumping member functions. Do not save this pointer, or attempt to delete it.

8.1.4. Writing A Logger File

To write a Logger file, create an instance of *LtLoggerRecordFile*, and use its `writePacket()` member function to write each DIS PDU or HLA update, interaction, or removal.

8.1.5. Creating An *LtLoggerRecordFile*

To write data to standard output, rather than to a file, pass the string “stdout” as the filename argument to the *LtLoggerRecordFile* constructor.

DIS only

In DIS, the *LtLoggerRecordFile* constructor takes only a file name:

```
LtLoggerRecordFile outFile("myFile.lgr");
```

HLA only

In HLA, a Logger file contains information about the FOM being used in the recorded exercise. The reason for this is that during playback, we may have to convert class, attribute and parameter handles stored in the Logger file to their corresponding handles in a different federation execution. (There is no guarantee that the same classes, attributes, and parameters are assigned the same handles in different federation executions, especially if the FED file has changed at all).

When you create an *LtLoggerRecordFile* in HLA, you must provide it with information about the FOM being used, so that it can store it in the Logger file. You can do this in one of two ways: either by passing a *DtExerciseConn*, or by passing a FOM description object obtained from another Logger file that you have opened with *LtLoggerPlaybackFile*. That class's `fomDesc()` member function returns the FOM description in its file.

For example:

```
LtLoggerPlaybackFile inFile("inFile.lgr");  
LtLoggerRecordFile outFile("outFile.lgr", inFile.fomDesc());
```

8.1.6. Writing Data

LtLoggerRecordFile's `writePacket()` member function looks like this:

```
virtual int writePacket(LtPacketInfo *packetInfo);
```

Its argument is a pointer to an instance of the class *LtPacketInfo*. See “[Interpreting LtPacketInfo](#)”, on page 8-6 for information about filling out this object with the data you would like written.

8.1.7. Interpreting LtPacketInfo

This section describes aspects of the *LtPacketInfo* class.

The `arrivalTime()` method returns the time associated with this packet. It is not the timestamp of the HLA or DIS message. It is a time offset into the file, written by the Logger or other application writing the file. If the file was written by the MÄK Logger, `arrivalTime()` is the time, since the Logger started recording, that the packet was received ([Figure 8-1](#)).

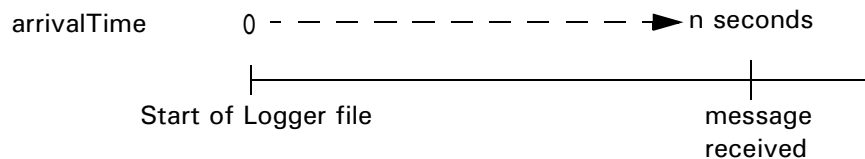


Figure 8-1. arrivalTime

The `readPacket()` and `writePacket()` methods represent data serialization methods.

The class includes the method `packetKind()` that returns an *LtPacketKind*. This value indicates the type of data the packet represents. Its values are as follows:

```
enum LtPacketKind
{
    LtPacketInteraction = 0,
    LtPacketUpdate,
    LtPacketRemoval,
    LtPacketAnnotation,
    LtPacketPdu,
    LtPacketFom
};
```

DIS only

DIS packets all use the value *LtPacketPdu*. The DIS PDU data can be retrieved using the `packet()` method.

The size of the packet in bytes is found with the `size()` method.

HLA only

The interpretation of the fields in *LtPacketInfo* depends on the packet kind:

- If the packet kind is *LtPacketInteraction*, then **classHandle** contains the interaction class handle associated with the interaction. The **attrs** field is a pointer to an *RTI::ParameterHandleValuePairSet* object that contains the actual interaction data. In files recorded by the Logger, this is the object passed to Logger by the RTI through the *receiveInteraction* service. The other fields are not valid.
- If the packet kind is *LtPacketUpdate*, then **classHandle** contains the object class handle of the object being described. The **objectId** field contains the object's object instance handle, as assigned by the RTI. The **name** field contains the name by which the object is known to the RTI. The **attrs** field contains a pointer to an *RTI::AttributeHandleValuePairSet* object that contains the actual update data. In files recorded by the Logger, this is the object passed to Logger by the RTI through the *reflectAttributeValues* service. Other fields are not valid.
- If the packet kind is *LtPacketRemoval*, then the packet represents the removal of an object from the exercise. In files written by the Logger, this type of packet is written in response to a *removeObjectInstance* RTI service invocation. Only the **objectId**, **name**, and **classHandle** fields are valid, and they have the same meanings as in *LtPacketUpdate* packets described above.
- If the packet is neither *LtPacketInteraction*, *LtPacketUpdate*, nor *LtPacketRemoval*, then the packet represents non-HLA execution data such as annotations. This data is encoded much the same way a DIS packet is encoded. The *packet()* method returns the data and the *size()* method returns the size of the data. Both the *attrs()* method and the *params()* method return NULL.

8.2. Editing Logger Files

One of the most common tasks users want to do with *LtLoggerFiles* is to read data from one Logger file, edit it, and write it to another Logger file. Below are two examples of this.



The examples use several classes from VR-Link:

Here is a DIS example of filtering out all Entity State messages from the entity with an ID of 3:4:5.

```
int main()
{
    // Open the input file.
    LtLoggerPlaybackFile infile("infile.lgr");

    // Open the output file.
    LtLoggerPlaybackFile outfile("outfile.lgr");

    // Create a PDU factory that can create instances of the right
    // kind of DtPdu subclass depending on the PDU kind in the packet.
    DtDisPduFactory factory;
```

```
    LtPacketInfo *info;
    while (info = infile.readPacket())
    {
        // Use the factory to create an instance of a DtPdu subclass.
        DtPdu* pdu = factory.createPdu((DtNetPacket*) info->packet());

        int writeIt = 1;
        // If it's an entity state PDU from 3:4:5, don't write it.
        if (pdu->kind() == DtEntityStatePduKind)
        {
            // Cast to DtEntityStatePdu - should be safe.
            DtEntityStatePdu* esPdu = (DtEntityStatePdu*) pdu;
            if (esPdu->entityType() == DtEntityType(3,4,5));
            {
                writeIt = 0;
            }
        }
        if (writeIt)
        {
            outfile.writePacket(info);
        }
    }
    delete info;
}
```

Here is an HLA example of filtering out all interactions:

```
int main()
{
    // Open the input file.
    LtLoggerPlaybackFile infile("inFile.lgr");

    // Open the output file, which should have the same FOM description
    // as the input file.
    LtLoggerPlaybackFile outfile("outFile.lgr", infile.fomDesc());

    LtPacketInfo *info;
    while (info = infile.readPacket())
    {
        if (info->packetKind() != LtPacketInteraction)
        {
            outfile.writePacket(info);
        }
    }
    delete info;
}
```

8.3. Compiling and Linking with the Logger Library

In order to use the Logger file API, you must add the following include directive to your compile line:

```
-I loggerDir/include
```

where *loggerDir* is the path to the root of your Logger installation.

You must link with *libLgrFile*, as follows:

```
-L loggerDir/lib -llgrFilex
```

The exact name of *lgrFilex* is keyed to DIS or HLA.

In addition, you should follow the directions for compiling and linking with VR-Link, in your VR-Link documentation. The Logger library link flag above must come before the VR-Link linking flags on your link line.



The preceding instructions are oriented towards UNIX. On a PC, make sure you are using the correct libraries in your project files, as listed in [“Using the Logger File API”](#), on page 8-2.



Using Logger Tools

The Logger tools are utilities that let you combine, modify, and inspect Logger recordings. The Logger tools are provided in separate versions for HLA and DIS.

Viewing Logger Files with lgrDump	9-2
lgrDump for HLA	9-3
How lgrDump Displays Unexpected Message Types.....	9-4
Displaying Raw Data.....	9-4
Concatenating Files with lgrCat	9-5
Inserting Time Between Concatenated Files	9-6
Processing Input from Other Programs.....	9-6
Merging Files with lgrMerge	9-7
Using lgrOffset to Shift Messages in Time.....	9-8
Using lgrTrim to Extract Part of A File.....	9-9

9.1. Viewing Logger Files with *lgrDump*

The *lgrDump* utility displays the contents of a Logger file in a readable form. The output format is very similar to that of the *netdump* and *hlaNetdump* utilities included with the VR-Link Toolkit.

Usage:

```
lgrDump [filename]
```

Because *lgrDump* directs its output to standard output, you can redirect the output to a destination file or program as in these examples:

```
lgrDump logfile.lgr > output.txt
lgrDump logfile.lgr | more
```

- > To pipe output from another program to *lgrDump*, omit the **filename** argument as in these examples:

```
lgrMerge File1.lgr File2.lgr | lgrDump
cat exercise_1.lgr | lgrDump
```



Owners of VR-Link can filter messages by specified field values by piping *lgrDump*'s output through the VR-Link *buffgrep* utility (UNIX only). See the VR-Link documentation for more information.

The following text shows a sample PDU as displayed by the DIS version of *lgrDump*.

```
*****
***** PDU #1077   Size=144           39.016 secs *****
*****
PduKind:           EntityStatePduKind   (1)
Version:           4
Exercise:          1
ProtocolFamily:    FamilyEntityInteraction   (1)
TimeStamp:         3297.213(seconds past hour)
TimeStampType:     Relative
Size:              144(bytes)
From:              1:15:1 (site:application:entity)
ForceID:           1
NumArtParams:      0
NumAttachedParts:  0
Type:              1:2:225:2:1:0:0 (kind:domain:country:cat:subcat:specific:extra)
Guise:             1:2:225:2:1:0:0 (kind:domain:country:cat:subcat:specific:extra)
Location:          { -2661471.288, -4358184.189, 3809002.286}
Velocity:          { 37.102, 87.402, 127.072}
Orientation:       { 1.169, -0.929, -2.305}
Acceleration:     { 0.000, 0.000, 0.000}
AngularVel:       { 0.000, 0.000, 0.000}
DrAlgorithm:       DrDrmRvw (4)
Appearance:        0x00000000
  PaintScheme      _PaintSchemeUniform
  MobilityKill      _MobilityKillNone
  FirePowerKill     _FirePowerKillNone
```

Damage	_DamageNone
Smoke	_SmokeNone
TrailingEffects	_TrailingEffectsNone
Hatch	_HatchNA
Lights	_LightsNone
Flaming	_FlamingNo
FinalPdu	_NotFinalPdu
AfterBurner	UnknownEnum
Marking:	A10
CharSet:	1
Capabilities:	0x00000000

Each PDU description begins with a heading containing a PDU number, packet size, and time elapsed since the start of the Logger recording. This information is added by the Logger during recording, and is not a part of the actual PDU.

After the heading, *lgrDump* displays all fields of the PDU.

lgrDump prints version information as described in Table 9-1:

Table 9-1: PDU versions and DIS protocol versions

PDU Version	DIS Protocol Version
4	2.0.4
5	IEEE 1278.1
6	2.1.4

9.1.1. lgrDump for HLA

The class, attribute and parameter handles printed by the HLA version of *lgrDump* may not be the actual handles that were originally recorded if you are not using the same FED file with *lgrDump* that you used when you recorded the file with the Logger. The handle numbers that are printed are those associated with the recorded names in the new FOM.

If you are using a different FED file with *lgrDump*, but would like to see the actual handles that were used at the time of recording, you can use *lgrDump*'s `-x` argument with an `execName` of "orig". This is only allowed when using *lgrDump* in raw mode. For example:

```
lgrDump -r -x orig myFile.lgr
```

In non-raw mode, we need to actually interpret the data, so we need to use the handles associated with the current FED file.

The HLA version of *lgrDump* decodes only the attribute data for entity objects. For attribute updates from other object classes, *lgrDump* prints the hex values of the attributes.

- > To decode all the data, use the Logger to play the data stream for the VR-Link *hlaNetdump* utility.

The following is an example of output from lgrDump for an HLA exercise:

```
*****
***** Packet 9: Update at          0.661 secs *****
*****
ObjectId: 25600002
ObjectName: (UNKNOWN)
ObjectClass: ObjectRoot.BaseEntity.AggregateEntity (8)
TimeStamp:      0.000000
TimeStampType:   Relative
3 attributes:
EntityIDs (13)
27 bytes:
 00 19 32 35
 36 3a 32 00
 32 35 36 3a
 33 00 32 35
 36 3a 34 00
 32 35 36 3a
 35 00 00
NumberOfEntities (16)
2 bytes:
 00 04
SubAggregateIDs (19)
3 bytes:
 00 01 00
```

9.1.2. How lgrDump Displays Unexpected Message Types

If *lgrDump* encounters unfamiliar message types, it displays formatted data for the messages.

DIS only

If the DIS version of *lgrDump* encounters a PDU of an incorrect DIS version (or a non-DIS PDU), it displays available header information followed by a hex dump of the rest of the PDU.

9.1.3. Displaying Raw Data

When you run *lgrDump* with the `-r` option (for raw), it behaves like the *nethex* utility that ships with VR-Link. For each attribute update or interaction received, it prints the name of the attribute, the size of the value of the attribute, and a hexadecimal dump of the value of the attribute.

9.2. Concatenating Files with *lgrCat*

The *lgrCat* utility concatenates messages from two or more Logger files. The resulting file contains all messages from the first file followed by all messages from the second file, and so on.

In the output file, messages from the first file retain their original times and message numbers. For each subsequent file, *lgrCat* increases the starting time and message numbers to create a continuous message sequence, as in [Figure 9-1](#).

If File1.lgr contains:	...and File2.lgr contains:
Msg #1 at time 5.0	Msg #1 at time 2.0
Msg #2 at time 10.0	Msg #2 at time 4.0
Msg #3 at time 20.0	Msg #3 at time 6.0

...then the concatenated file contains:

```
Msg #1 at time 5.0
Msg #2 at time 10.0
Msg #3 at time 20.0
Msg #4 at time 22.0
Msg #5 at time 24.0
Msg #6 at time 26.0
```

Figure 9-1. *lgrCat* example

lgrCat modifies only the message numbers and time values generated by the Logger (those that appear in the message heading when viewed with *lgrDump*).

DIS only

Under DIS, *lgrCat* does not change the TimeStamp values contained in original PDUs.

HLA only

When you use the HLA version of *lgrCat*, all of the input Logger files must have been recorded using the same FOM. Files recorded using FOMs that differ from that used in the first file specified in the command are skipped.

Usage:

```
lgrCat [-s spacing] file1 [file2 file3...]
```

lgrCat directs its output to standard output in a fashion similar to the UNIX *cat* or DOS *type* command. In most cases, you will want to redirect or pipe the output to a destination file or program, and specify two or more files to be concatenated, as in these examples:

```
lgrCat file1 file2 [file3 file4...] > output.file
lgrCat exercise1.lgr exercise2.lgr > ex_1-2.lgr
lgrCat ex1.lgr ex2.lgr ex3.lgr | lgrDump | more
```

9.2.1. Inserting Time Between Concatenated Files

You can use the `-s spacing` argument to insert a period of time between files. The *spacing* argument specifies the number of seconds to insert between the final message in each file and the starting point of the next file (timepoint 00:00:00). For example, the command,

```
lgrCat -s 4.0 File1.lgr File2.lgr > combined.lgr
```

creates a *combined.lgr* file whose first three messages match those in the output file in [Figure 9-1](#). Now, however, the last three messages have been delayed by four seconds and now occur at 26.0 28.0 and 30.0 seconds, as in [Figure 9-2](#):

If File1.lgr contains:

```
Msg #1 at time 5.0
Msg #2 at time 10.0
Msg #3 at time 20.0
```

...and File2.lgr contains:

```
Msg #1 at time 2.0
Msg #2 at time 4.0
Msg #3 at time 6.0
```

...then the concatenated file will contain:

```
Msg #1 at time 5.0
Msg #2 at time 10.0
Msg #3 at time 20.0
(four-second space)
Msg #4 at time 26.0
Msg #5 at time 28.0
Msg #6 at time 30.0
```

Figure 9-2. Effect of the *lgrCat* spacer argument

9.2.2. Processing Input from Other Programs

- > To pipe the output of another program to *lgrCat*, replace one of *lgrCat*'s input file arguments with a hyphen character to specify where the incoming data should appear in the concatenated output.

In the following example, *lgrCat* receives output piped from the *lgrOffset* program as its first input (indicated by the hyphen), and concatenates that data with *File2.lgr*. The combined data is directed to *Out.lgr*:

```
lgrOffset 20.0 < File1.lgr | lgrCat - File2.lgr > Out.lgr
```

9.3. Merging Files with *lgrMerge*

The *lgrMerge* utility combines one or more Logger files into a single file containing all messages from all files, sorted in chronological order and renumbered. [Figure 9-3](#) illustrates a merge of two Logger files:

If File1.lgr contains:	...and File2.lgr contains:
Msg #1 at time 1.0	Msg #1 at time 2.0
Msg #2 at time 3.0	Msg #2 at time 4.0
Msg #3 at time 5.0	Msg #3 at time 6.0
...then the merged file contains:	
Msg #1 at time 1.0	
Msg #2 at time 2.0	
Msg #3 at time 3.0	
Msg #4 at time 4.0	
Msg #5 at time 5.0	
Msg #6 at time 6.0	

Figure 9-3. How *lgrMerge* combines files

lgrMerge modifies only the message numbers and time values generated by the Logger (those that appear in the message heading when viewed with *lgrDump*).

DIS only

Under DIS, *lgrMerge* does not change the TimeStamp values contained in the original PDUs.

HLA only

When you use the HLA version of *lgrMerge*, all of the input Logger files must have been recorded using the same FOM. Files recorded using FOMs that differ from that used in the first file specified in the command are skipped.

Usage:

```
lgrMerge file1 file2 [file3 file4...]
```

Because *lgrMerge* directs its output to standard output, you typically redirect or pipe its output to a destination file or program as in these examples:

```
lgrMerge file1 file2 file3 > outputfile  
lgrMerge exercise1.lgr exercise2.lgr > ex_1-2.lgr  
lgrMerge ex1.lgr ex2.lgr ex3.lgr | lgrDump > PDUs.txt
```

9.4. Using *lgrOffset* to Shift Messages in Time

The *lgrOffset* utility can adjust the time of all messages in a file by a specified number of seconds. You can:

- ♦ Decrease or increase the timepoints of all messages in a file so they occur earlier or later.
- ♦ Remove an unknown length of dead time from the beginning of a file.

During recording, the Logger records the arrival time of each message relative to the start of recording. This is the only value that *lgrOffset* modifies; it is not a part of the original message. The original `TimeStamp` values in the messages are not affected.

Usage:

```
lgrOffset {time|first}
```

where:

- ♦ *time* is the desired time offset in seconds. *time* can be a positive or negative value.
- ♦ *first* removes a period of dead time from the beginning of a file. *lgrOffset* determines the timepoint of the first message, deletes the dead time from the start of the recording, and subtracts that number of seconds from the timepoint for each message in the file.

Because *lgrOffset* receives its input from standard input and directs its output to standard output, piping or redirection are used to specify the input source and output destination.

In either of the following two examples, *lgrOffset* increases the timepoints of all messages in *infile.lgr* by 15 seconds and saves the output in *outfile.lgr*:

```
lgrOffset 15.0 < infile.lgr > outfile.lgr  
lgrCat infile.lgr | lgrOffset 15.0 > outfile.lgr
```

In the next example, *lgrOffset* removes an unknown amount of dead time from the beginning of *infile.lgr* and saves the output in *outfile.lgr*:

```
lgrOffset first < infile.lgr > outfile.lgr
```

9.5. Using *lgrTrim* to Extract Part of A File

The *lgrTrim* utility reads and outputs a specified portion of a Logger file, in effect, trimming packets from the beginning and end of the file. You can specify the part of the file to be output, as either a range of messages or a span of time.

Usage:

```
lgrTrim [-n] start [end]
```

where:

- ♦ *start* specifies the time point at which to begin extracting messages from the recording.
- ♦ *end* specifies the time point at which to stop extracting messages from the recording.
- ♦ -n indicates that *start* and *end* specify a numerical range of messages to extract.

lgrTrim discards any packets that fall outside of the specified range. If you do not specify an end value, *lgrTrim* outputs all packets from the specified *start* point to the end of the file.

By default, *lgrTrim* assumes that the *start* and *end* arguments are time values, given in seconds.

Because *lgrTrim* receives its input from standard input and directs its output to standard output, piping or redirection are used to specify the input source and output destination.

Example 1

```
lgrTrim 5.0 10.0 < infile.lgr > outfile.lgr
```

lgrTrim extracts from *infile.lgr* all messages that occur between times 00:00:05 and 00:00:10 and saves them in *outfile.lgr*.

Example 2:

```
lgrTrim -n 10 20 < infile.lgr > outfile.lgr
```

lgrTrim extracts the 10th through 20th messages from *infile.lgr* and saves them in *outfile.lgr*.

Example 3:

```
lgrTrim -n 30 35 < exerc01.lgr | lgrDump
```

lgrTrim output piped to *lgrDump* for on-screen display of the 30th through 35th messages in *exerc01.lgr*.



Troubleshooting

This chapter describes how to identify and correct the most common network problems that prevent the Logger from sending and receiving data properly.

Troubleshooting Logger Network Communications (DIS Only).....	A-2
Using the Correct Broadcast Address and UDP Port	A-2

A.1. Troubleshooting Logger Network Communications (DIS Only)

In a typical DIS exercise, VR-Link applications (including the Logger) communicate with other applications using broadcast UDP/IP over a local area network. All applications in the exercise use the same IP broadcast address, and each application's host machine has a network interface device that is configured for that address. All machines must share a common netmask.

In addition, all participating applications must use the same UDP port.

If the Logger uses an IP address, netmask or port that is different from that used by your other applications, communication problems occur.

A.1.1. Using the Correct Broadcast Address and UDP Port

By default, the Logger communicates to the host machine's default broadcast address, using UDP port 3000. If the Logger appears to start normally, but fails to communicate with other DIS applications, the Logger may be using an incorrect broadcast address or UDP port. You can correct this problem by running the Logger with arguments that explicitly specify the correct address and port:

1. Determine the broadcast address used by your other DIS applications. This may be the network's default broadcast address, or may be specified through the application's startup command or configuration file at startup. For more information, see your System Administrator, or use the UNIX `ifconfig` command to display the broadcast address.
2. Determine the UDP port used by your other applications.
3. Start the Logger and notice the status message that appears at startup:

```
NetSocket ok for port 3000 to 123.456.789.255
```

4. If the port and address in the status message do not match those used by your other applications, exit the Logger and restart using the Address option (`-A`), the DIS port option (`-P`), or both, to specify the appropriate values. Enter:

```
Logger -A address -P portnum
```

where address and portnum are the broadcast address and UDP port used by your other applications, for example:

```
Logger -A 123.456.999.255 -P 3020
```



Logger MTL Parameters

This appendix lists the MTL commands that you can use in Logger configuration files.

MÄK Technologies Lisp (MTL) Files	B-2
Logger MTL Parameters	B-2

B.1. MÄK Technologies Lisp (MTL) Files

Configuration files for MÄK Technologies products are ASCII text files that use MÄK Technologies LISP (MTL) to encode configuration information. MTL files have the extension *.mtl*. You can edit them in any text editor. Be sure that your text editor uses the proper end-of-line characters for your platform.

For the Logger, most parameter entries take the format:

```
(setq parameter_name value)
```

For example:

```
(setq allowRemoteFileDelete 0)
```

To comment out a line, precede the text with two semi-colons (;). You can also add a comment at the end of a line by preceding it with two semi-colons.

In addition to the `setq` command, a commonly used command is the `load` command. This command instructs the Logger to load the file specified, for example:

```
(load ./mydata "mysublist.mtl")
```

You can put all MTL arguments in one file, or segregate them into files that have a special purpose, such as a file to list the subscribed classes and interactions. When you start the Logger, it automatically loads the *Logger.mtl* file in the directory in which the executable is located.

B.2. Logger MTL Parameters

Table B-1 lists MTL configuration parameters for the Logger.

Table B-1: Logger commands for remote control (Sheet 1 of 2)

MTL Syntax	Description
<code>allowRemoteFileDelete</code>	Enables or disables file deletion in response to <i>DtLgrDelFilePdus</i> . 0 = Do not allow file deletions 1 = Allow file deletions.
<code>enable</code>	Specifies a list of logger pipeline element plugins to enable.
<code>entityIdStr</code>	Specify the site:host:entity for Logger (DIS) or object name (HLA).
<code>fomMapperInitData</code>	Specifies initialization data to pass to a FOM Mapper library. Equivalent to the second portion of the <code>-f library: init_data</code> command.
<code>fomMapperLib</code>	Specifies a FOM Mapper library to use. Equivalent to the first portion of the <code>-f library: init_data</code> command.

Table B-1: Logger commands for remote control (Sheet 2 of 2)

MTL Syntax	Description
heartbeat	Specifies the DIS heartbeat.
ignoreAdvisories	Specifies that the Logger should send updates regardless of RTI advisory messages.
pluginDir	Specifies the directory in which plugins are located.
plugins	Specifies a list of plugins to load.
processLc	0 = Do not respond to Logger Control messages 1 = Interpret Logger Control messages only if they are addressed to the entity ID specified by the -a argument (default) 2 = Interpret all Logger Control messages
recordLc	0 = Do not record Logger Control messages (default) 1 = Record Logger Control messages
rprFomVersion	Specifies a version of the RPR FOM to use.
subscribedClasses	Specifies a list of classes for the logger to subscribe to. (HLA only)
subscribedInteractions	Specifies a list of interactions for the logger to subscribe to. (HLA only)
timeout	Specifies the DIS timeout.
visible	Specifies the Logger interface to use: 0 = Logger interface not visible 1 = Use GUI



Glossary

API

Application Programming Interface.

dead-reckoning

A method of approximating the location of an object based on its direction and velocity. The Logger uses dead-reckoning to generate interim state messages for [entity maintenance](#).

Defense Modeling and Simulation Office

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

DIS

[Distributed Interactive Simulation](#).

DIS data packets

See [Protocol Data Unit](#).

Distributed Interactive Simulation

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

The DIS protocol has been superseded by the [High-Level Architecture](#) protocol for use by the Department of Defense.

DMSO

[Defense Modeling and Simulation Office.](#)

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 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.

exercise

The DIS term for a one or more interacting simulation applications. Compare to [federation execution](#) in HLA.

exercise ID

A numeric identification for a DIS simulation exercise.

FED file

Federation Execution Data file. The Federation Execution Data is the subset of [FOM](#) data needed and read by the [RTI](#). VR-Link also reads the FED file.

federate

A connection to the [RTI](#). Typically a single simulation application can be thought of as a federate.

federation

A group of HLA federates capable of playing in the same [federation execution](#).

Federation Object Model

Defines the data content of a federation execution.

federation execution

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. It is the step where the executable code is run to conduct the exercise and produce the data for the measures of effectiveness for the federation execution. A federation execution is the logical equivalent of a DIS simulation exercise.

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

FOM

[Federation Object Model.](#)

frame rate

The rate at which the Logger displays updated images.

heartbeat

The frequency with which the Logger sends interim PDU messages. (DIS only)

High-Level Architecture

The High Level Architecture (HLA) for simulations is a DOD-wide initiative to provide an architecture to support interoperability and reuse of simulations. The HLA supersedes DIS for the Department of Defense.

HLA

[High-Level Architecture](#).

interaction

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

Logger control PDUs

A set of classes that you can use to control the Logger programmatically. Available in both DIS and HLA.

MÄK Technologies Lisp

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

mode

Refers to the Logger's current activity or readiness state, either recording, or playing. The Logger can do only one of these activities at a time.

MTL

[MÄK Technologies Lisp](#).

object

An element in a simulation. Also called an [entity](#).

PDU

[Protocol Data Unit](#).

Protocol Data Unit

A unit of data that is passed on a network between DIS simulation applications.

Realtime Platform Reference FOM

An [HLA](#) reference [FOM](#) based on the [DIS](#) protocol.

recording

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

reference FOM

A [FOM](#) designed to be used as a whole, or with modification, by a wide variety of [federation executions](#).

RTI Initialization Data (RID)

The data required by the RTI for operation. The required data come from two distinct sources, the [Federation Object Model](#) product, and the Federation Execution Details.

RPR FOM

[Realtime Platform Reference FOM](#).

RTI

[Run-Time Infrastructure](#).

Run-Time Infrastructure

A library and other supporting software that implements the HLA interface specification. All federates communicate with one another in an HLA environment through RTI functions.

state

Within each mode, refers to whether the Logger is stopped, paused, or active (recording or playing.)

tape

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

timeout

The period of time in which the Logger continues to display an entity after that entity's update messages have stopped appearing on the network.

timescale

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

UDP port

A network channel through which the Logger sends and receives data for DIS exercises.



Index

Symbols

-? option 4-14
? command 5-9

A

-A option 2-25, 4-15
acceleration scaling 3-7
action 7-12
actionObj.h header file 7-13
activate() function 7-15, 7-16
actOnPacket() function 7-16
adding
 objects 7-11
addObject() function 7-11, 7-22
AgentAIF pointer 7-19
allowRemoteFileDelete parameter 6-3
annotation 4-7
 creating 4-7
 deleting 4-10
 editing 4-9
 importing annotation file 4-9
 in API 7-9
 moving to 4-9
 opening annotation file 4-8
 saving file 4-8
API
 interface classes 7-2
 Logger file 8-2
appending Logger recording 9-5
application
 including Logger in 7-3
architecture 7-12
argument

attrs 8-7
classHandle 8-7
name 8-7
objectId 8-7
arrivalTime() function 8-6
atEnd() function 8-4
atStart() function 8-4
attribute
 myName 7-14
 myStatus 7-14
attrs argument 8-7

B

-b option 4-15
broadcast address A-2

C

case sensitivity of command 1-3
changing
 pipeline 7-23
class
 DtActionObj 7-13
 DtAnnotationManager 7-9
 DtFileSource 7-23
 DtFilterObj 7-13, 7-19
 DtLgrControlInteraction 6-2
 DtLgrControlListener 7-2
 DtLgrControlPdu 6-2
 DtLgrDelFilePdu 6-3
 DtLgrInterface 7-2
 DtLgrPipeline 7-12
 DtLgrState 7-13
 DtLogger 7-2, 7-3

- DtLoggerApp* 7-22
- DtLoggerStage* 7-13, 7-17
- DtLoggerState* 7-13
- DtMotifGui* 7-2
- DtNote* 7-9
- DtSendObj* 7-13
- DtSourceObj* 7-13
- DtTextGui* 7-2
- DtWinGui* 7-2
- LtLoggerPlaybackFile* 8-2
- LtLoggerRecordFile* 8-2, 8-5
- LtPacketInfo* 8-6
- LtPacketInteraction* 8-7
- LtPacketRemoval* 8-7
- LtPacketUpdate* 8-7
- remote control 6-2
- subscribed to 2-22
- classHandle** argument 8-7
- clipboard 7-11
- clock
 - Logger 7-6
- command options
 - general format 1-3
 - ? 4-14
 - A 2-25, 4-15
 - b 4-15
 - d 3-7, 4-15
 - e 3-7, 4-15, 5-6
 - h 4-14
 - help 4-14
 - o 4-16
 - P 4-14
 - p 4-16
 - r 4-15
 - S 2-25, 4-15
 - s 4-11, 4-16
 - startup under Windows 4-2
 - T 4-16
 - x 4-14
- command, remote control 6-3
- comment
 - MTL parameter B-2
- compiling
 - API code 7-26
 - Logger library 8-9
- component files 2-6
- concatenating Logger recordings 9-5
- configuration file
 - logger.mtl 7-26
 - logger.mtl 7-24

- configuration options 4-12
- configuring
 - system for MÄK Real-time RTI 2-18
 - with API 7-8
- controlling
 - classes and interactions subscribed to 2-22
- conventions
 - manual xiii
- CreateLogger() function 7-22
- creating
 - annotation 4-7
 - Logger file 8-3, 8-5
 - plug-in 7-22
- currentFile() function 7-7

D

- d option 3-7, 4-15
- dataScaling() function 7-6
- deactivate() function 7-14
- dead reckoning
 - acceleration scaling 3-7
 - effect of reverse playback on 3-7
 - velocity scaling 3-7
- dead time, removing from start of recording 9-8
- deleting
 - annotation 4-10
 - Logger file 6-3
- destination address 2-25
- directory
 - ./bin* 2-6
 - ./doc* 2-6
 - ./logger_tapes* 2-6
- DIS port
 - Logger default A-2
 - specifying 4-14, A-2
- disable() function 7-16
- DMSO RTI 2-18
- DtActionObj* class 7-13
- DtAnnotationManager* class 7-9
- DtDisSimulate* module 7-17
- DtDisSource* module 7-17
- DtFileSource* class 7-23
- DtFileSource* module 7-17
- DtFilterObj* class 7-13, 7-19
- DtHlaActiveSource* module 7-17
- DtHlaPassiveSource* module 7-17
- DtHlaSimulate* module 7-17
- DtLgrControlInteraction* class 6-2
- DtLgrControlListener* class 7-2

DtLgrControlPdu class 6-2
DtLgrDelFilePdu class 6-3
DtLgrInterface class 7-2
DtLgrPipeline class 7-12
DtLgrState class 7-13
DtLogger class 7-2, 7-3
DtLoggerApp class 7-22
DtLoggerStage class 7-13, 7-17
DtLoggerState class 7-13
DtMotifGui class 7-2
DtNote class 7-9
DtScreenDump module 7-17
DtSendObj class 7-13
DtSendToFile module 7-17
DtSourceObj class 7-13
DtTcpPort module 7-17
DtTCPSource module 7-17
DtTextGui class 7-2
DtWinGui class 7-2

E

-e option 3-7, 5-6, 3-7, 4-15
 Edit Annot ation dialog box 4-9
 editing

- annotation 4-9
- Logger files 8-7

 editing a Logger recording 9-9
 elements

- pipeline 7-13

 embedded Logger 7-3
 enable parameter 7-23, 7-26
 EnableObjects() function 7-23
 End button 4-5
 end command 5-5, 5-9
 end-of-file action 3-7, 4-12

- in Logger 5-6

 entity maintenance 3-6, 4-5
 environment variable

- license manager 2-11
- MAKLMGRD_LICENSE_FILE 2-11
- RTI_CONFIG 2-18

 eofAction() function 7-8
 example, remote control 6-5
 exerciseConn() function 7-10
 exiting Logger during script 5-8
 extracting part of a Logger recording 9-9

F

FED file 2-18, 2-21

- changing 2-3

 federation execution name 2-20

- specifying 4-14

 fetchFromClipboard() function 7-11
 file

- FED 2-18
- logger.mtl 7-26
- logger.mtl logger.mtl configuration file 7-24
- MTL B-2
- RID 2-18
- RTI.RID 2-18

 file() function 7-7
 fileLength() function 7-7
 filePosition() function 7-7
 fileTimeBackwards() function 8-4
 fileTimeForwards() function 8-4
 filter 7-12, 7-18
 filterObj.h header file 7-13
 filterPacket() function 7-16, 7-19
 firstPacketTime() function 8-4
 FLEXlm license manager 2-4, 2-8
 FOM

- flexibility 2-3

 FOM class 7-10
 FOM Mapper 2-20, 2-21
 function

- activate() 7-15, 7-16
- actOnPacket() 7-16
- addObject() 7-11, 7-22
- arrivalTime() 8-6
- atEnd() 8-4
- atStart() 8-4
- CreateLogger() 7-22
- currentFile() 7-7
- dataScaling() 7-6
- deactivate() 7-14
- disable() 7-16
- EnableObjects() 7-23
- eofAction() 7-8
- exerciseConn() 7-10
- fetchFromClipboard() 7-11
- file() 7-7
- fileLength() 7-7
- filePosition() 7-7
- fileTimeBackwards() 8-4
- fileTimeForwards() 8-4
- filterPacket() 7-16, 7-19

firstPacketTime() 8-4
generatePacket() 7-16
generatePackets() 7-15
get() 7-9
getName() 7-14
gotoEnd() 7-7
gotoStart() 7-7
heartbeat() 7-10
idealTime() 7-6
InitPlugin() 7-22, 7-23
initSelf() 7-14, 7-16, 7-19
inPlayback() 7-8
isOk() 7-14
jumpToEnd() 8-4
jumpToStart() 8-4
lastPacketTime() 8-4
loadFileForPlayback() 7-7, 7-16
loadFileForRecording() 7-7, 7-16
mode() 7-8
packet() 8-6
packetKind() 8-6
params() 8-7
pause() 7-3, 7-7
play() 7-3, 7-7, 7-16
playAction() 7-8
postToClipboard() 7-11
printData() 7-14
process() 7-15, 7-16
processData() 7-15
readPacket() 8-4, 8-6
readPacketBackwards() 8-4
record() 7-3, 7-7, 7-16
recordAction() 7-8
recordingLoggerControlInfo() 7-8
registerObject() 7-11, 7-22
removeFromClipboard() 7-11
removeObject() 7-11
reset() 7-15
seekBackward() 8-4
seekForward() 8-4
seekTime() 8-4
sendPacket() 7-15
setDataScaling() 7-6
setDefaultRecordSourceParameter() 7-8
setEofAction() 7-8
setInterSubscriptionList() 7-10
setMirroring() 7-8
setObjectSubscriptionList() 7-10
setPacketLimit() 7-9
setPassiveSubscription() 7-10

setPlayAction() 7-8
setPlaybackFileCreator() 7-7
setRecordAction() 7-8
setRecordFileCreator() 7-7
setRecordingLoggerControlInfo() 7-8
setTime() 7-6
setTimescale() 7-6
setUsingLoggerFederateAmbassador() 7-10
size() 8-7
skipPacketBackwards() 8-4
skipPacketForwards() 8-4
stop() 7-3, 7-7, 7-16
tick() 7-3, 7-15
time() 7-6
timeout() 7-10
timeScale() 7-6
writePacket() 8-5, 8-6

G

generatePacket() function 7-16
generatePackets() function 7-15
get() function 7-9
getName() function 7-14
go command 5-9
gotoEnd() function 7-7
gotoStart() function 7-7
graphical user interface
 API 7-2

H

-h option 4-14
-help option 4-14
h command 5-9
header file
 actionObj.h 7-13
 filterObj.h 7-13
 lgrPipeline.h 7-12
 lgrState.h 7-13
 loggerFile.h 8-2
 logger.h 7-25
 sourceObj.h 7-13
heartbeat 5-10, 7-10
 defined 3-5
 setting 4-12, 4-13
heartbeat() function 7-10
help command 5-9
HLA
 FOM flexibility 2-3

subscribing to classes and interactions 2-22

I

- i command 5-9
- idealTime() function 7-6
- implementing
 - plug-in 7-18
- importing
 - annotation file 4-9
- including Logger in application 7-3
- info command 5-5, 5-9
- initializing
 - plug-in 7-22
- InitPlugin() function 7-22, 7-23
- initSelf() function 7-14, 7-16, 7-19
- inPlayback() function 7-8
- installing
 - DMSO RTI 2-18
 - MÄK Real-time RTI 2-17
- interaction
 - subscribed to 2-22
- interface
 - API 7-2
- interim PDUs 3-5, 3-6
- internal clock 7-6
- isOk() function 7-14

J

- jumpToEnd() function 8-4
- jumpToStart() function 8-4

L

- lastPacketTime() function 8-4
- lgrCat* 9-5
 - effect on PDU timestamp data 9-5
 - pipng output from another program 9-6
- lgrDump* 9-2
 - compared with netdump 9-2
 - display of unexpected PDU types 9-4
- lgrMerge* 9-7
- lgrOffset* 9-8
- lgrPipeline.h* header file 7-12
- lgrState.h* header file 7-13
- lgrTrim* 9-9
- liblcore* library 7-13
- library

- compiling and linking Logger 8-9
- liblcore 7-13
- Logger API 7-26
 - provided for Logger file API 8-2
- license manager 2-8
- license server
 - running 2-13
 - status 2-14
 - stopping 2-14
- license, environment variable 2-11
- linking
 - API code 7-26
 - Logger library 8-9
- lmdown program 2-14
- lmhostid program 2-10
- lmstat program 2-14
- loadFileForPlayback() function 7-7, 7-16
- loadFileForRecording() function 7-7, 7-16
- loading
 - recording in API 7-7
- Logger API
 - annotations 7-9
 - architecture 7-12
 - compiling and linking 7-26
 - configuration functions 7-8
 - embedded Logger 7-3
 - loading recording 7-7
 - protocol-specific parameters 7-10
- Logger Control PDUs 6-2
- Logger file
 - API 8-2
 - creating 8-3, 8-5
 - deleting 6-3
 - editing 8-7
 - reading 8-2
 - reading data 8-4
 - writing 8-6
 - writing to 8-5
- Logger library
 - compiling and linking 8-9
- Logger recording
 - combining, using *lgrMerge* 9-7
 - concatenating with *lgrCat* 9-5
 - editing 9-9
 - extracting a portion of 9-9
 - modifying time data with *lgrOffset* 9-8
 - removing dead time from beginning 9-8
 - samples 1-3
 - viewing PDU contents of 9-2
 - viewing with Stealth 1-3

loggerFile.h header file 8-2
logger.h header file 7-25
logger.mtl configuration file 7-26
logger.mtl configuration file 7-24
LtLoggerPlaybackFile class 8-2
LtLoggerRecordFile class 8-2
LtLoggerRecordFile class 8-2, 8-5
LtPacketInfo class 8-6
LtPacketInteraction class 8-7
LtPacketRemoval class 8-7
LtPacketUpdate class 8-7

M

MÄK Real-time RTI
 configuring 2-18
 installing 2-17
MÄK Technologies LISP B-2
MAKLMGRD_LICENSE_FILE environment
 variable 2-11
manual
 conventions xiii
mode() function 7-8
module
 DtDisSimulate 7-17
 DtDisSource 7-17
 DtFileSource 7-17
 DtHlaActiveSource 7-17
 DtHlaPassiveSource 7-17
 DtHlaSimulate 7-17
 DtScreenDump 7-17
 DtSendToFile 7-17
 DtTcpPort 7-17
 DtTCPSource 7-17
Motif
 API 7-2
moving
 to annotation point 4-9
MTL file B-2
multicast address 2-25
myName attribute 7-14
myStatus attribute 7-14

N

name argument 8-7
network, requirements for Logger A-2

O

-o option 4-16
object
 managing from API 7-11
objectId argument 8-7
opening
 annotation file 4-8
overriding
 DtLogger class 7-25

P

-P option 4-14
-p option 4-16
packet
 processing 7-16
packet() function 8-6
packetKind() function 8-6
parameter
 allowRemoteFileDelete 6-3
 commenting out B-2
 enable 7-23, 7-26
 pluginDir 7-26
 plugins 7-26
 protocol-specific in API 7-10
params() function 8-7
pause
 command 5-9
 compared to stop 5-4
pause() function 7-3, 7-7
PC
 compiling and linking API 7-26
PDU
 how modified in Logger tape 9-3
 remote control 6-2
 viewing with *lgrDump* 9-2
pipeline
 changing with a plug-in 7-23
 elements
 7-13
 managing from API 7-11
 processing packets 7-16
play action 4-12, 5-5
play() function 7-3, 7-7, 7-16
playAction() function 7-8
playback 3-3, 4-4, 5-4
 described 3-3
 direction 4-10, 5-6
 in reverse 3-4

- mode 3-3
- speed 4-10, 5-6
 - allowable range of 4-12, 5-6
- plug-in
 - changing pipeline 7-23
 - creating 7-22
 - directory 2-7
 - implementing 7-18
 - initializing 7-22
- pluginDir parameter 7-26
- plugins directory 2-7
- plugins parameter 7-26
- postToClipboard() function 7-11
- printData() function 7-14
- process() function 7-15, 7-16
- processData() function 7-15
- processing
 - packets 7-11, 7-16
- products
 - technical support xii
 - upgrades xii
- protocol-specific parameter
 - Logger API 7-10

Q

- quit command 5-9

R

- r option 4-15
- reading
 - data from Logger file 8-4
 - Logger file 8-2
- readPacket() function 8-4, 8-6
- readPacketBackwards() function 8-4
- record action 4-12, 5-2
- record mode 3-2, 5-2
- record() function 7-3, 7-7, 7-16
- recordAction() function 7-8
- recording 3-2, 4-4, 5-2
 - annotating 4-7
 - described 3-2
 - managing in API 7-7
 - status, displaying 5-3
- recordingLoggerControlInfo() function 7-8
- registering
 - objects 7-11
- registerObject() function 7-11, 7-22
- remote control 6-2

- commands for 6-3
 - example 6-5
- remote control commands 6-3
- removeFromClipboard() function 7-11
- removeObject() function 7-11
- removing
 - objects 7-11
- reset() function 7-15
- reverse playback
 - and dead reckoning 3-7
- RID file 2-18
- RPR FOM 2-3, 2-20
- RTI 2-4
 - definition of 2-17
 - DMSO, installing 2-18
 - MÄK, installing 2-17
- RTI_CONFIG environment variable 2-18
- rtiexec 2-19
- RTI.rid file 2-18
- runLm program 2-13
- running
 - license server 2-13
- run-time commands
 - ? 5-9
 - end 5-5, 5-9
 - go 5-3, 5-4, 5-9
 - h 5-9
 - heartbeat 5-10
 - help 5-9
 - info 5-5, 5-9
 - pause 5-4, 5-9
 - quit 5-9
 - scale 5-6, 5-9
 - skip 5-5, 5-9
 - start 5-5, 5-9
 - stop 5-3, 5-4, 5-9
 - time 5-5, 5-9
 - timeout 5-10
 - wait 5-8, 5-9

S

- S option 2-25, 4-15
- s option 4-11, 4-16
- sample recording 1-3
- saving
 - annotation file 4-8
- scale command 5-6, 5-9
- scaling, time 3-7
- script

- creating 5-8
 - executing 5-8
 - quitting Logger while running 5-8
 - wait command 5-8
 - seekBackward() function 8-4
 - seekForward() function 8-4
 - seekTime() function 8-4
 - send 7-12
 - sendPacket() function 7-15
 - Set Time 3-6
 - entity maintenance 4-5
 - menu option 4-5
 - Set Time Scale menu option 4-11
 - setDataScaling() function 7-6
 - setDefaultRecordSourceParameter() function 7-8
 - setEofAction() function 7-8
 - setInterSubscriptionList() function 7-10
 - setMirroring() function 7-8
 - setObjectSubscriptionList() function 7-10
 - setPacketLimit() function 7-9
 - setPassiveSubscription() function 7-10
 - setPlayAction() function 7-8
 - setPlaybackFileCreator() function 7-7
 - setRecordAction() function 7-8
 - setRecordFileCreator() function 7-7
 - setRecordingLoggerControlInfo() function 7-8
 - setTime() function 7-6
 - setTimescale() function 7-6
 - setUsingLoggerFederateAmbassador() function 7-10
 - shutting down
 - license server 2-14
 - simulation clock 7-6
 - size() function 8-7
 - skip command 5-5, 5-9
 - entity maintenance 3-6
 - Skip Time 3-6
 - entity maintenance 4-5
 - menu option 4-5
 - skipPacketBackwards() function 8-4
 - skipPacketForwards() function 8-4
 - slow-speed playback 3-4, 5-6
 - software components 1-2
 - source 7-12
 - sourceObj.h header file 7-13
 - Start button 4-5
 - start command 5-5, 5-9
 - startup
 - from DOS 4-3
 - from icon under Windows 4-3
 - playback mode 3-3
 - record mode 3-2, 5-2
 - UNIX 4-2
 - status
 - API functions 7-8
 - license server 2-14
 - status field 4-4
 - status message, in playback 5-5
 - stop command
 - compared to pause 5-4
 - in playback mode 5-9
 - in record mode 5-3
 - stop() function 7-3, 7-7, 7-16
 - stopping
 - license server 2-14
 - subscribing
 - classes and interactions 2-22
 - support
 - technical xii
 - system requirements 2-3
- T**
- T option 4-16
 - technical support xii
 - tick() function 7-3, 7-15
 - time
 - between concatenated Logger tapes 9-6
 - navigating during playback 4-5, 5-5
 - shifting PDU times 9-8
 - time clock 7-6
 - time command 5-5, 5-9
 - entity maintenance 3-6
 - time jump, updating entities after 3-6
 - time navigation 4-5
 - 5-5
 - time scale
 - described 3-4
 - setting 4-10, 4-11, 5-6
 - slider control
 - controlling with keyboard 4-11
 - specifying at startup 4-11, 5-6
 - time() function 7-6
 - timeout 7-10
 - defined 3-5
 - setting 5-10
 - timeout() function 7-10
 - timepoint
 - setting 4-5
 - setting with skip command 5-5

- setting with time command 5-5
- timescale 5-6
 - during recording 4-4
 - effects 3-5
- timeScale() function 7-6
- tools 9-1
- trimming beginning or end of Logger recording 9-9

U

- UDP port, specifying 2-25, 4-14
- UNIX
 - compiling and linking API 7-26
 - Windows versions compared 1-3
- UNIX vs. Windows versions 1-3
- upgrades xii
- User's Guide
 - organization of ix

V

- variable
 - environment 2-11
 - license environment 2-11
- velocity scaling 3-7
- View Annotations window 4-7
- view control PDUs 1-3
- VR-Link 8-2

W

- wait command 5-8, 5-9
- Windows and UNIX versions compared 1-3
- wrapping 4-13
 - at end of file 3-7, 5-6
 - in reverse playback 3-7
- writePacket() function 8-5, 8-6
- writing
 - Logger file 8-5, 8-6

X-Y-Z

- x option 2-20, 4-14

MÄK Data Logger Startup Options Quick Reference (Sheet 1 of 2)

Option	Description
Record and Playback	
-a {"site:host:entity" "hla_object_name"}	DIS: Specifies the site:host:entity for remote control of the Logger. HLA: Specifies the object name (global object designator in VR-Link.)
-f {vIRPRxx library:init_data}	HLA: Specifies the FOM Mapper to use if you are not using the default FOM.
-F sublist.mtl	HLA: Specifies a subscription control file to load.
-G {w t wt}	Specifies the interface to use, where: ♦w = Graphical user interface (default) ♦t = Command line interface ♦wt = Both.
-h, -help, -?	Displays a brief summary of Logger command-line arguments.
-L {0 1 2}	Specifies the response to Logger Control messages for remote control. ♦0 = Do not respond to Logger Control messages ♦1 = Interpret Logger Control messages only if they are addressed to the entity ID specified by the -a argument (default) ♦2 = interpret all Logger Control messages
-m	Causes the Logger to send a Logger control PDU each time you change the state of the Logger.
-P dis_port	DIS only. Specifies which DIS port (UDP port) the Logger uses to send or receive data. Default port: 3000.
-x ExName	HLA only. Sets federation execution name. Default: VR-Link
-v {0 1}	Specifies the Logger interface to use: ♦0 = Logger interface not visible ♦1 = Use GUI (default)
Record Mode Only	
-b state	Specifies whether the Logger enters a stopped state or begins recording immediately when opening a file for recording. ♦r = record ♦s = stopped (default).
-P	HLA: Subscribe passively. Do not force federates to send all updates to the Logger. Record only those updates that other federates subscribe to.
-r [filename]	Causes the Logger to open in record mode. Specifies the destination file for recorded data.
-R {0 1}	Specifies whether or not to record Logger Control messages during remote control. ♦0 = do not record Logger Control messages (default) ♦1 = record Logger Control messages
-S mcastAddr	DIS only. Subscribes the Logger to the multicast address specified by mcastAddr. To subscribe to several multicast addresses, use this option more than once in the command.
Playback Mode Only	
-A addr	DIS only. Sets the default IP address to which the Logger sends PDUs. If this option is not used, the Logger uses the computer's broadcast address.
-d	Disables the generation of velocity and acceleration scaling data when playing at speeds above or below normal.

MÄK Data Logger Startup Options Quick Reference (Sheet 2 of 2)

Option	Description
-e {p w s}	Specifies the Logger's action upon reaching the end of the file. ♦p = pause at the file's end ♦w = wrap to the other end of the recording and continue ♦s = stop (default).
-o {p s}	When a file is opened for playback, specifies whether the Logger enters a stopped state, or begins playing immediately. ♦p = playing ♦s = stopped (default).
-p [<i>filename</i>]	Causes the Logger to open in playback mode; specifies the source file to be played.
-s <i>timescale</i>	Sets the initial playback timescale factor.
-T <i>timepoint</i>	Sets the timepoint in the file where playback begins. Enter the <i>timepoint</i> in HH:MM:SS format. Default: 00:00:00

MÄK Data Logger Run-Time Commands Quick Reference

Command	Description
Record Mode and Play Mode	
go	Starts or resumes recording or playback from a previously stopped or paused state.
stop	Halts playback at the current timepoint, stops recording or transmission.
quit	Closes the Logger file and exits.
info or i	Displays a brief status report including the log file name, current timepoint, timescale, and current play state.
h, help, or ?	Displays a summary of Logger run-time commands.
wait <i>time</i>	Used only in scripts. Specifies how long the Logger performs its current action before processing the next command in the script. Enter <i>time</i> in the HH:MM:SS format or as a number of seconds.
Play Mode Only	
start	Sets the current timepoint to the beginning of the recording.
open <i>filename</i>	Opens the specified Logger recording for playback.
play	Begins playing the Logger recording.
end	Sets the current timepoint to the end of the recording.
pause	Pauses playback at the current timepoint. In DIS, begins repeated resending of each entity's most recent entity state PDU with velocity and acceleration set to zero.
scale [<i>factor</i>]	Sets the timescale factor that determines playback speed and direction. Use 1.0 (default) for normal speed, 2.0 for double speed, 0.5 for half speed, and negative values for backward playback.
time <i>timepoint</i>	Navigates to a specific <i>timepoint</i> in the recording. Specify <i>timepoint</i> in HH:MM:SS format.
skip <i>offset</i>	Moves the current timepoint forward or backward by the specified time <i>offset</i> . Specify the offset in HH:MM:SS format or as a number of seconds. A positive offset causes a jump forward in time; a negative offset causes a backward jump.
heartbeat [<i>sec</i>]	(DIS only) During slow-speed playback, specifies the number of seconds the Logger allows before generating an interim PDU for a given entity. Default: 5. With no argument, heartbeat displays the current setting.
timeout [<i>sec</i>]	(DIS only) During slow-speed playback, specifies how long the Logger continues to transmit interim PDUs for an entity after the entity's PDUs have stopped occurring in the recording. Default: 12 recorded seconds.
Record Mode Only	
record <i>filename</i>	Begins recording to the specified filename.



MÄK Technologies
185 Alewife Brook Parkway
Cambridge, MA 02138
617.876.8085