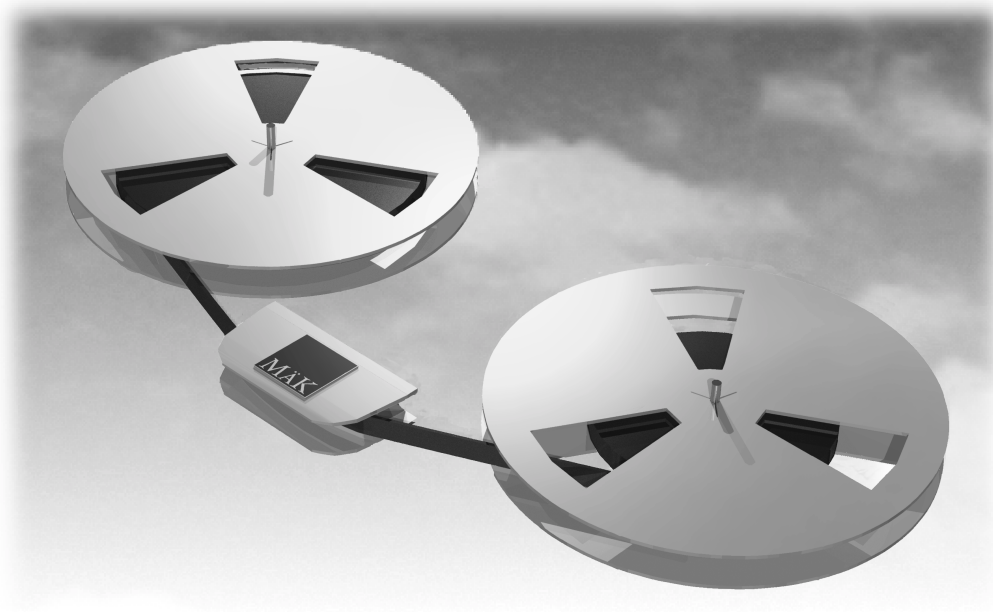




Logger

User's Guide



Logger

User's Guide

Copyright 1997, 1998, 2000 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 trademark of MÄK Technologies, Inc.
All other trademarks are owned by their respective companies.

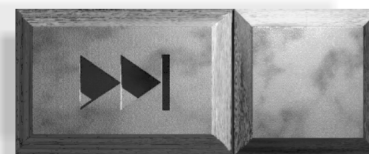
MÄK Technologies
185 Alewife Brook Parkway
Cambridge, MA 02138 USA

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

info@mak.com

www.mak.com

Revision LOG-3.5.1-1-000720



Contents

Preface

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

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 of the Logger	1-3
1.1.3. Sample Logger Files	1-4

Chapter 2 Installing and Configuring the Logger

2.1. System Requirements	2-2
2.1.1. HLA Support	2-2
2.2. Installing the Logger	2-3
2.2.1. Installing the Logger On A UNIX System	2-3
2.2.2. Installing the Logger On A PC	2-3
2.2.3. The Logger Directory Structure	2-5
2.3. Setting Up and Using the FLEXlm License Manager	2-6
2.3.1. The FLEXlm Client-Server Architecture	2-6
2.3.2. Installing and Configuring the License Manager	2-7
2.3.3. Adding Additional License Files	2-10
2.3.4. Running the License Server	2-11
2.3.5. Troubleshooting the FLEXlm License Manager	2-11

2.4. Installing An RTI	2-14
2.4.1. Installing the MÄK RTI	2-14
2.4.2. Installing the DMSO RTI	2-15
2.5. Configuring the Logger for HLA	2-17
2.5.1. Selecting A FOM Mapper	2-17
2.5.2. Specifying A FED File that is Appropriate for the FOM Mapper	2-17
2.5.3. Controlling the Classes and Interactions the Logger Subscribes To	2-18
2.6. Configuring the Logger for DIS	2-21
2.6.1. Specifying a Destination Address for Outgoing Data	2-21
2.6.2. Subscribing to Multicast Groups	2-21
2.6.3. Specifying a UDP Port	2-22

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. Effect of Reverse Playback on Dead Reckoning	3-7
3.4. The End-of-File Action	3-7

Chapter 4 Using The Logger

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	4-5
4.4. Navigating to A Specific Time in A Recording	4-6
4.4.1. Using the Set Time Feature	4-6
4.4.2. Using the Skip Time Feature	4-7
4.4.3. Annotating A Logger Recording	4-7
4.5. Setting Timescale and Direction	4-11
4.5.1. Using and Customizing the Timescale Slider	4-11
4.5.2. Setting a Timescale at Startup	4-11
4.5.3. Entering A Timescale Value Numerically	4-12
4.5.4. Message Processing for Non-Standard Timescales	4-12
4.6. Setting Configuration Options	4-13
4.7. Summary of Startup Options for the Logger	4-15

Chapter 5 Using The TLogger

5.1. Recording Exercises	5-2
5.1.1. TLogger Run-time Commands for Recording	5-2
5.1.2. Displaying Recording Status	5-2
5.1.3. Beginning a Recording Session in Record Mode	5-3
5.2. Playing Back Recordings	5-3
5.2.1. Basic TLogger Playback Commands	5-3
5.2.2. Displaying Playback Status	5-4
5.2.3. Beginning a Playback Session in Play Mode	5-4
5.3. Navigating Time	5-5
5.3.1. Controlling End-of-File Behavior	5-5
5.4. Controlling Entity Maintenance	5-6
5.5. Setting Playback Timescale and Direction	5-7
5.5.1. Message Processing for Non-Standard Timescales	5-7
5.6. Controlling DIS Heartbeat and Timeout	5-8
5.6.1. Setting the Heartbeat Rate	5-8
5.6.2. Setting the Timeout	5-8
5.7. Using a Script to Control Recording and Playback	5-9
5.7.1. Stopping A Script	5-9
5.8. Summary of TLogger Startup Options	5-10
5.9. Summary of TLogger Run-Time Commands	5-12

Chapter 6 Remote Control and the Logger File API

6.1. Remote Control PDUs	6-2
6.2. Using Logger Commands for Remote Control	6-3
6.2.1. Deleting Logger Files	6-4
6.3. Example of Remote Control for a DIS Exercise	6-5
6.4. Using the MÄK Logger File API	6-9
6.4.1. Reading A Logger File	6-9
6.4.2. Creating An LtLoggerPlaybackFile	6-9
6.4.3. Reading Data	6-11
6.4.4. Writing a Logger File	6-12
6.4.5. Creating an LtLoggerRecordFile	6-12
6.4.6. Writing Data	6-12
6.4.7. Interpreting LtPacketInfo	6-13
6.5. Editing Logger Files	6-15
6.6. Compiling and Linking with the Logger Library	6-17

Chapter 7 The Logger Plug-in API

7.1. Introduction	7-2
7.2. The Logger Architecture	7-2

7.3. Creating a Plug-in	7-4
7.3.1. Registering Subclasses with The Logger	7-4
7.3.2. Loading Plug-Ins	7-5
7.3.3. Creating An Instance of A Logger Element	7-6
7.4. Logger Core Elements	7-7
7.4.1. Sources	7-7
7.4.2. Actions	7-10
7.4.3. Filters	7-12
7.4.4. Senders	7-14
7.5. The Logger Plug-In Application Interface	7-16
7.5.1. Overriding the Logger Interface	7-22
7.6. Plug-in Example	7-23
7.6.1. Building and Compiling the Example Plug-In	7-23
7.6.2. Example Header File	7-23
7.6.3. Implementing the Plug-in	7-24
7.6.4. Registering the Example Plug-in	7-27
7.6.5. Using the Example Plug-in	7-28

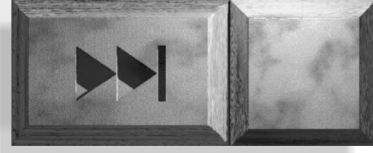
Chapter 8 Logger Tools

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

Appendix A Troubleshooting

A.1. Troubleshooting Logger Network Communications	A-2
A.1.1. Using the Correct Broadcast Address and UDP Port	A-2
A.1.2. Configuring the Network Mask on A PC	A-3

Glossary**Index**



Preface

The MÄK Logger records network activity of HLA or DIS exercises and plays it back for review. The Logger gives you a choice of a graphical, VCR-style interface or a command line interface.

This manual is for persons who will install the Logger and for persons who will record or play back simulations. It assumes that you are familiar with basic UNIX or DOS operations 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 Logger*, shows how to record and play back HLA/DIS network activity using the Logger's graphical user interface.

Chapter 5, *Using The TLogger*, shows how to use the TLogger, the command line version of the Logger.

Chapter 6, *Remote Control and the Logger File API*, explains how to control the Logger programmatically.

Chapter 7, *The Logger Plug-in API* describes the plug-in API for the Logger, which allows you to add new features.

Chapter 8, *Logger Tools*, describes the utility programs that combine, modify and inspect existing Logger recordings.

Appendix A, *Troubleshooting*, shows how to correct network problems that may prevent the Logger from communicating properly.

The *Glossary*, defines terms used in the manual.

Online Help

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

As with the Plan View Display, you can view remote entities, but you can also create your own local entities, aggregate them into hierarchical units, assign tasks, set state parameters, and create plans that have tasks, set statements, and conditional statements. 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 and the SGI computers and supports multi-channel 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.

How to Contact Us

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

For sales and upgrade information by e-mail: info@mak.com

For technical support by e-mail: support@mak.com

For license key files and information: www.mak.com/support/keyintro.htm

The MÄK web site: www.mak.com

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

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

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

Document Conventions

This document uses the following typographic conventions:

Monospaced Type	Indicates commands or values you enter.
<i>Monospaced Italic</i>	Indicates variables that you replace with appropriate values.
[]	Indicates optional arguments.
	Separates options in a command where only one option may be chosen at a time.
/	Indicates a directory. Since MÄK products run on both UNIX and 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.
i	Indicates supplemental or clarifying information.
!	Indicates additional information that you must observe to ensure the success of a procedure or other task.

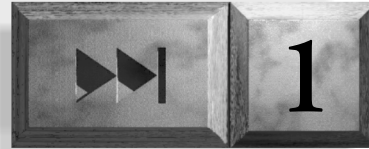
Directory names are preceded with dot and slash characters that show their position with respect to the Logger home directory. For example, the directory *logger/binRPR* appears in the text as *./binRPR*.

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 covers the following topics:

Logger Features	1-2
Logger Interfaces	1-3
Differences Between UNIX and Windows Versions of the Logger	1-3
Sample Logger Files	1-4

1.1. Logger Features

The Logger records network activity of HLA and DIS **exercises**¹ 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 with a VCR-style graphical user interface.
- ♦ The TLogger (or Text Logger), which performs the same functions as the Logger, but has an interactive, character-based, shell-style interface. The TLogger adds script support to the Logger's feature set.
- ♦ 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.
- ♦ 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:

- ♦ The **Logger**, a graphical user interface that uses VCR-style buttons to control the recording and playback of exercises.
- ♦ The **TLogger**, which uses a command line interface.
- ♦ An API which lets you control the Logger programmatically 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.

The only important functional differences between the Logger and TLogger are:

- ♦ You can use a script to control the TLogger.
- ♦ The Logger lets you change various options while the program is running. The TLogger requires that you set these options at startup through command-line switches.
- ♦ The Logger supports annotations.

1.1.2. Differences Between UNIX and Windows Versions of the Logger

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

Under UNIX, the entire Logger command line is case-sensitive, while under Windows, only the option letters in command-line options 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

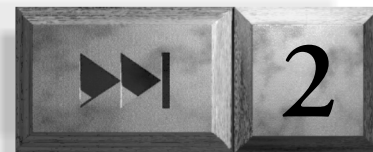
The Logger includes sample recordings of two exercises:

- ♦ An exercise from the December 2, 1993 I/ITSEC show, Orlando, Florida:
`iitsec93.dis204` — contains DIS 2.0.4 data.
- ♦ An exercise prepared for the MÄK/MultiGen/SGI demonstration at the 1995 I/ITSEC show:
`montyDemo.lgr` — contains DIS 2.0.4 data.
`montyDemoRPRxx.lgr` — contains HLA data.

This recording is of an exercise in which two A-10s attempt to destroy seven ground targets near an airfield.



The *montyDemo* recording contains View Control PDUs, which cause the 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.



Installing and Configuring the Logger

This chapter covers the following topics:

System Requirements	2-2
HLA Support	2-2
Installing the Logger	2-3
Installing the Logger On A UNIX System	2-3
Installing the Logger On A PC	2-3
The Logger Directory Structure.....	2-5
Setting Up and Using the FLEXlm License Manager	2-6
The FLEXlm Client-Server Architecture	2-6
Installing and Configuring the License Manager.....	2-7
Adding Additional License Files.....	2-10
Running the License Server	2-11
Troubleshooting the FLEXlm License Manager	2-11
Installing An RTI.....	2-14
Installing the MÄK RTI	2-14
Installing the DMSO RTI	2-15
Configuring the Logger for HLA	2-17
Selecting A FOM Mapper	2-17
Specifying A FED File that is Appropriate for the FOM Mapper.....	2-17
Controlling the Classes and Interactions the Logger Subscribes To	2-18
Configuring the Logger for DIS.....	2-21
Specifying a Destination Address for Outgoing Data.....	2-21
Subscribing to Multicast Groups	2-21
Specifying a UDP Port	2-22

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, your file can be played back successfully.

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 will 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 over the following sections.

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. If you are going to use the Logger with HLA, you need to know which version of the RTI you plan to use.

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 gateway.tar.gz
```
4. Untar the file:

```
tar -xvf gateway.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 need to know which version of the RTI you plan to use.

Installing the Logger from A CD

To install the Logger from a CD:

1. Insert the Logger CD into your CD-ROM drive. If the autorun feature is enabled, the MÄK PC Products Installation window opens (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.

2. In the MÄK PC Products Installation window, in the column with the RTI compatibility that you need, click Logger. (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	data
Documentation in PDF format.	doc
License manager files.	flexlm
Header files for Logger file API.	include
Library files for Logger file API.	lib
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</i>	Sample Logger files.
<i>doc</i>	Documentation in Adobe Portable Document Format (PDF).
<i>flexlm</i>	License management software.
<i>include</i> and <i>lib</i>	These directories contain header files and libraries for <i>libLgr-File</i> which provides an API to the MÄK Logger file format. You need MÄK VR-Link to use this 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 information about FLEXlm, view the FAQ page at www.globetrotter.com/flexlm.htm.) 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, *maklmgrd* runs on one computer, known as the license server. This machine services every machine (including possibly itself) on which you want to run a MÄK Technologies product. The machines on which the MÄK Technologies products are running are called clients.

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

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



The license key in the license file is tied to the host id of the server machine. Therefore, if you decide to change the server (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. This is not a trivial process, and is not recommended. 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.

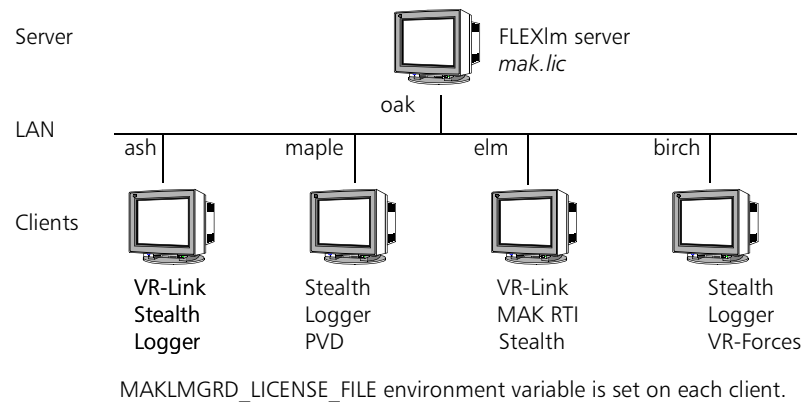


Figure 2-3. FLEXlm example architecture

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 2-10.

Installing the License Manager Software

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

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

Requesting A License File

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

To get a license file:

1. On the license server, in a command window, change to the *flexlm* directory.
2. 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.

3. Go to <http://www.mak.com/support/keyintro.htm> and complete the license key request form. MÄK will e-mail you a file named *mak.lic*.

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

UNIX

At a command prompt, type `hostname`.

Windows 95/98/NT

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

Putting the License File in the *flexlm* Directory

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



If you already have a *mak.lic* file in the *flexlm* directory, do not overwrite the file. See the instructions in [“Adding Additional License Files”](#), on page 2-10.

Setting the MAKLMGRD_LICENSE_FILE Environment Variable

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



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

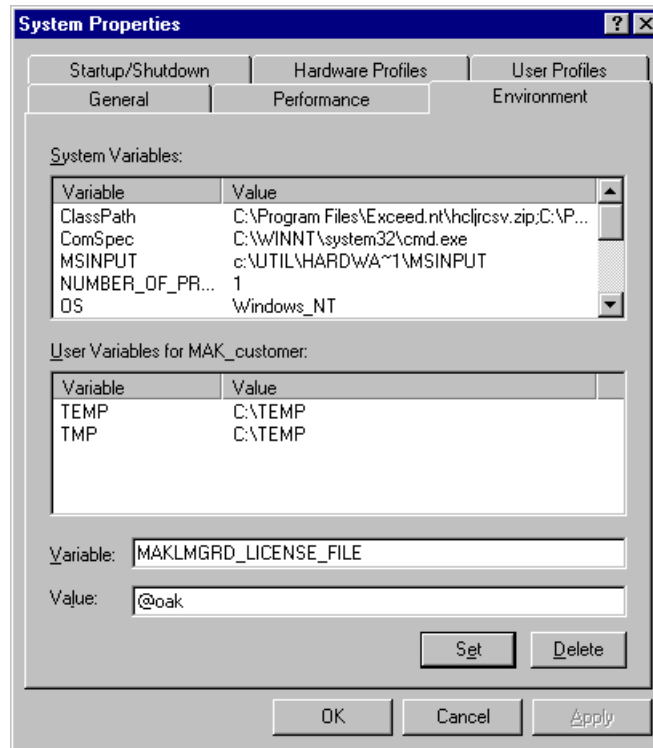


Figure 2-4. Setting an environment variable in Windows NT

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 2-11.

2.3.3. Adding Additional License Files

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

- To request an additional license file, follow the same procedure as for your first license file. See [“Requesting A License File”](#), on page 2-8.

To add a new license file to your license server:

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

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

2.3.4. Running the License Server

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

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



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

Shutting Down the License Server

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

2.3.5. Troubleshooting the FLEXlm License Manager

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

Unable to Get A License

If you are unable to get a license for a MÄK application and the FLEXlm log file (*LOG*) indicates that the port is in use, it is possible that you have more than one *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.

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.

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

2.4.1. Installing the MÄK RTI

The MÄK RTI is included 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 product's root directory, which points to the *makRti* directory, in which the MÄK RTI software is located.

PCs

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

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

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

Running Applications with the MÄK RTI

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

Documentation for the MÄK RTI

The documentation for the MÄK RTI consists of:

- ♦ *MÄK RTI Reference Manual*.
- ♦ *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 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 web site (<http://hla.dmsolm.com>). Click on Software Distribution Center, where you can register for an account, then later download the RTI software. Documentation for the DMSO RTI is provided on the MÄK product CD because the documentation applies to the MÄK RTI as well.

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

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

Configuring Your System to Use the DMSO RTI

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 resides in the RTI's *bin/platform_name* directory (UNIX) or in the RTI's *bin* directory (PCs). Run *rtiexec* only on one machine, regardless of how many HLA applications you are running.

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

2.5. 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. Replace *ExName* with the desired name:

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

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 v1RPRXX
Logger -f library:init_data
```

If the argument to the `-f` option is one of the `v1RPRxx` keywords, the Logger uses VR-Link's built-in RPR FOM mapper, configured for the RPR FOM version specified by the keyword, for example, `v1RPR08` 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 v1RPR08
```

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 *myFomMap.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 that is Appropriate 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 name, 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. Controlling the Classes and Interactions the Logger Subscribes To

You can specify the set of classes and interactions that the Logger subscribes to. By default the Logger subscribes to all classes in your FED file except *RTIPrivate* and any MOM object or interaction. To change this behavior, do the following:

1. Using the Logger executable, create a subscription control file that defines the default subscription list for a FED file. This file indicates exactly which classes and interactions the Logger is subscribing to for the indicated FOM. The `-F` option specifies the name of the file to write the subscription list to.

```
C:\> Logger -x VR-Link -F sublist.mtl
```

The *sublist.mtl* file should look as follows for the VR-Link FOM:

```
;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 should not be subscribed to.
3. Include the file in the *logger.mtl* configuration file by adding the following line.

```
(load "PathToSublistFile" "sublist.mtl")
```

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 will write to the file each class and interaction that it subscribes to.

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 through use of the `-S` command-line option. When subscribed to multicast groups, the Logger also continues to receive data through the machine's broadcast and unicast addresses.

- > To subscribe to more than one multicast address, use the `-S` option several times on the same command line:

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

Subscription pertains to record mode only.

Multicasting is fully supported under IRIX-5, IRIX-4, and Solaris-2. Under DEC OSF/1 v2.0, applications can receive, but not send, multicast.

i

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.

- > You can specify the UDP port start the Logger with the `-P` switch, as in these examples:

```
Logger -P 2727
```

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

3

This chapter describes features common to both Logger interfaces. For step-by-step instructions about how to operate the Logger, see [Chapter 4, “Using The Logger”](#) and [Chapter 5, “Using The TLogger”](#)

This chapter covers the following topics:

What The Logger Does During Recording.....	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
Effect of Reverse Playback on Dead Reckoning.....	3-7
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. When you start the Logger, you don't need to specify record or playback mode. You can do this in the interface. When you start the TLogger, you must specify the mode. For complete instructions for starting the Logger, see “Starting the Logger”, on page 4-2. For complete instructions for starting the TLogger, see “Recording Exercises”, on page 5-2.

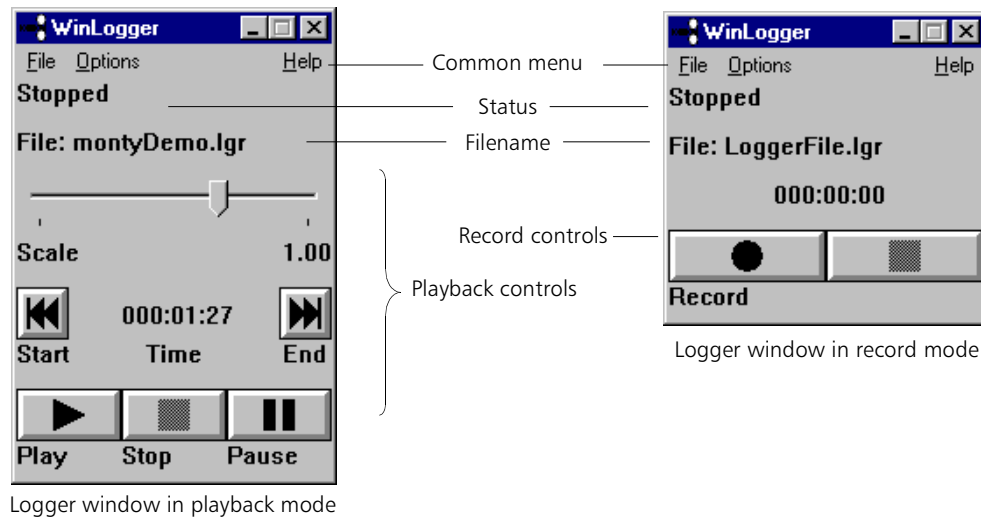


Figure 3-1. The Logger windows

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. You can interactively change the record state through the Logger window or with TLogger commands.

DIS only

In record mode, the DIS 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` switch, for example:

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

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

HLA only

Under HLA, 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.

3.1.2. What The Logger Does During Playback

The Logger may or may not begin playback immediately on startup, depending on which additional command-line switches you use.

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

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).

When stopped, the Logger halts at the current timepoint in the file and stops sending data.

In both the Logger and TLogger, you can interactively change the playback state.

During playback, you can control the playback speed and direction to produce fast motion, slow-motion, and reverse playback effects. For procedures, see [“Timescale and Direction During Playback”](#), on page 3-4 and [“Compensating for Timescale Effects During Playback”](#), on page 3-5 for details.

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 4-6 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` switch to specify a different port, for example:

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

HLA only

When playing back in HLA, the Logger acts as a federate that owns each of the objects represented 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.

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.

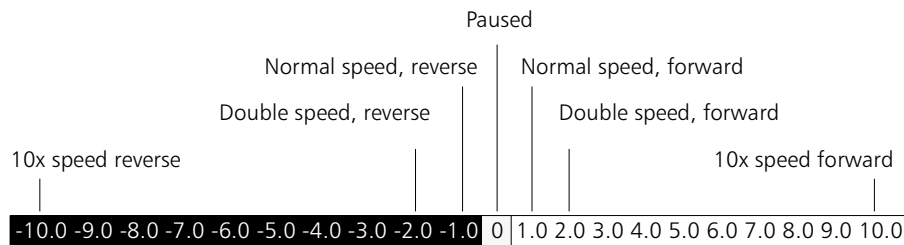


Figure 3-2. Sample timescale values

For procedures, see [“Setting Timescale and Direction”](#), on page 4-11 and [“Playing Back Recordings”](#), on page 5-3.

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 non-normal playback speeds. Under DIS, Interim PDU generation prevents 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 non-standard playback speed.

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 five seconds.

Most receiving applications typically 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 time the entity out. A typical timeout threshold is 12 seconds, or 2.4 times the heartbeat rate.

Using Interim PDUs to Maintain Heartbeat

During slow-speed playback, 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 DIS Logger calculates the contents of an interim PDU by applying a dead-reckoning algorithm to an entity's most recent PDU.

By default, the DIS 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 4-13 and [“Controlling DIS Heartbeat and Timeout”](#), on page 5-8.

Scaling the Timeout Period During Slow-Speed Playback

When an entity's PDUs stop occurring in the recording, the DIS 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 will generate 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 4-13 and [“Controlling DIS Heartbeat and Timeout”](#), on page 5-8.

3.3.2. Updating Entity Information Quickly After a Time Jump

During playback, the 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, but not for a Set Time command.

In the TLogger, you can turn entity maintenance on or off separately for the `time` and `skip` commands. See [“Controlling Entity Maintenance”](#), on page 5-6 for details.

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

3.3.3. Velocity and Acceleration Scaling

When playing at speeds above or below normal, the Logger can adjust the velocity and acceleration values in entity state messages accordingly. 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 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 `-a` command-line option at Logger startup.

DIS only

Scaling must be enabled in order for the DIS 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. Effect of Reverse Playback on 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 originally traveled when receiving a reverse playback.

3.4. The End-of-File Action

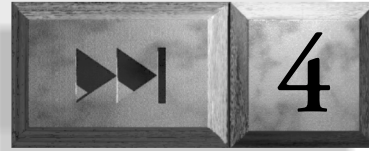
When the Logger reaches the end of a file, it can enter a stopped or paused state, or it can wrap to the other end of the file and continue playback.

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

```
Logger -esStop (default)
Logger -epPause
Logger -ewWrap
```

In the Logger window, you can also change the end-of-file action in the Configuration dialog box, available through the Options menu. See [“Setting Configuration Options”](#), on page 4-13.

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



Using The Logger

This chapter covers the following topics as they apply to the Logger's graphical user interface:

The Logger Windows.....	4-2
Starting the Logger	4-2
Recording Exercises.....	4-4
Playing Logger Recordings.....	4-5
Navigating to A Specific Time in A Recording.....	4-6
Using the Set Time Feature	4-6
Using the Skip Time Feature	4-7
Annotating A Logger Recording.....	4-7
Setting Timescale and Direction	4-11
Using and Customizing the Timescale Slider	4-11
Setting a Timescale at Startup.....	4-11
Entering A Timescale Value Numerically.....	4-12
Message Processing for Non-Standard Timescales.....	4-12
Setting Configuration Options.....	4-13
Summary of Startup Options for the Logger	4-15

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.

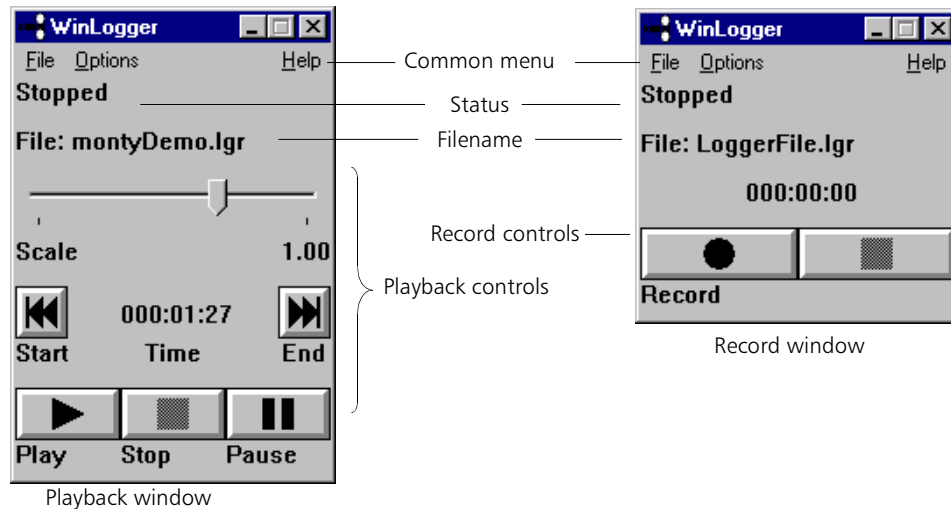


Figure 4-1. The Logger windows

4.1.1. Starting the Logger

You can use command-line options to set many of the Logger's initial settings. However, because the Logger window provides run-time access to most frequently-used options, you can usually start the Logger without options.

[Table 4-2](#), on page 15, lists startup options for the Logger.

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 console window, switch to the appropriate *\bin* directory, and enter:

```
Logger [options]
```

When the Logger window and its companion console window appear, you can close the DOS window if you want to.

i

Some options (such as the federation execution name in HLA, or the DIS port or destination address) cannot be set in the Logger windows. They must be specified on the command line at startup.

To invoke command-line options when starting the Logger, set up a shortcut on your Windows desktop and add the desired options to the startup command line in the icon's property sheet. See your Windows documentation for more information.

By default, the Logger opens up in Playback mode.

4.2. Recording Exercises

To begin recording:

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 a stopped state. To begin recording or return to a stopped state, click the appropriate button. The Status field displays either Recording or Stopped accordingly.

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

DIS only

In DIS, the Logger records all traffic that the host machine receives on the default UDP port (3000). If desired, you can use startup command-line options to specify a different UDP port or subscribe to additional multicast addresses. For more information about these options, see [“Configuring the Logger for DIS”](#), on page 2-21.

HLA only

In HLA, the Logger records all updates, interactions and object removals that it perceives through the RTI.

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, Stopped, Paused, or Playing is displayed to reflect the Logger's current operating state.

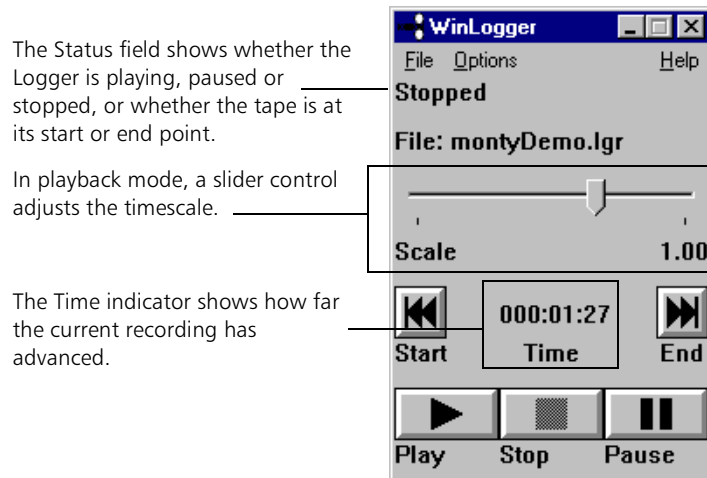


Figure 4-2. Logger playback controls

DIS only

In DIS, the Logger sends data to the host machine's default broadcast address over the default UDP port (3000). If desired, you can use command-line options to specify a different UDP port or destination IP address. See [“Configuring the Logger for DIS”](#), on page 2-21 for more information. The Logger transmits PDUs while in the playing or paused states. In the stopped state, no PDUs are sent.

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 option, the Logger performs entity maintenance to speed the transmission of up-to-date messages. When you use the *Set Time* option, the Logger does not perform entity maintenance. 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 (f).

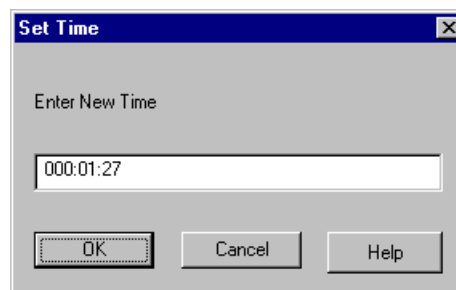


Figure 4-3. Set Time dialog box

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 4-4).

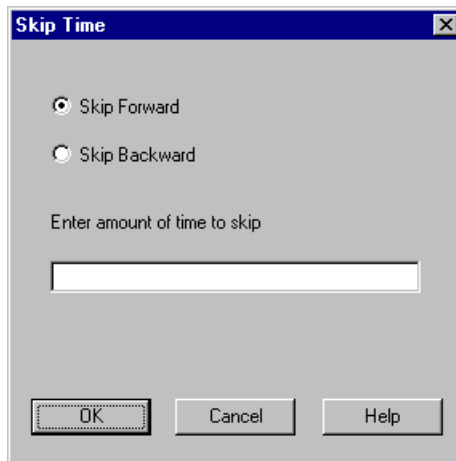


Figure 4-4. Skip Time dialog box

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.

4.4.3. 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.

You can annotate a Logger recording at as many points as you want.

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 4-5).

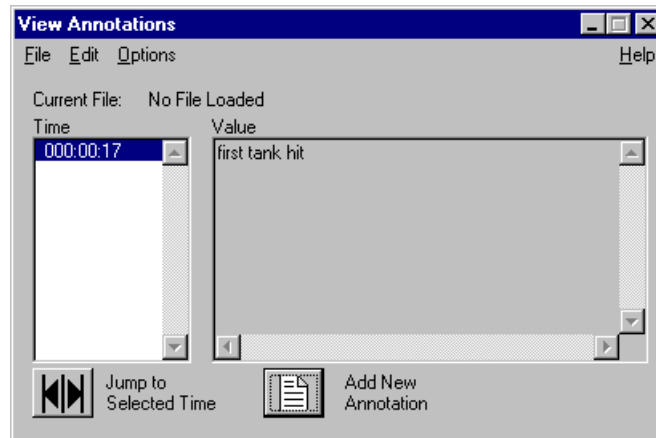


Figure 4-5. View Annotations window

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

The maximum length of an annotation is 4000 characters. There is no formal limit to the number of annotations you can add to a recording. From a practical standpoint, the maximum number of annotations depends on available memory.

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 4-8.
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.

Opening An Annotation File

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.

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.

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.

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.

Clearing the Annotation List

Clearing the View Annotations window 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, it just doesn't have any annotations in it.

4.5. Setting 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 main window. The slider can be customized to contain a specific set of timescale values.
- ♦ Specify an initial timescale at Logger startup through a command-line option.
- ♦ Enter the desired timescale into a dialog box.

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

4.5.1. Using and Customizing the Timescale Slider

The Timescale slider control in the Logger main 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.

- > To specify the values that appear on the timescale slider, start the Logger, with the `-t` command-line option followed by a comma-separated list of values, as shown below. Do not include any spaces in the value list:

```
Logger -t-5,-4,-3,-2.5,-2,-1.5,-1,0,1,1.5,2,2.5,5,7
```

The values appear in ascending order on the slider control, even if you enter them out of sequence.

4.5.2. Setting a Timescale at Startup

- > To specify a timescale at Logger startup, use the `-s` command-line option, as in this example:

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

4.5.3. 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.](#))

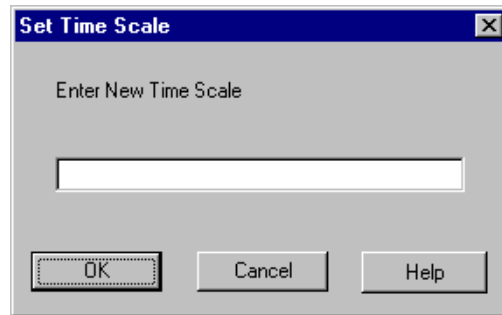


Figure 4-6. Set Time Scale dialog box

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.



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 displays the Timescale value in red and continues to process the messages as quickly as possible. None are discarded.

4.5.4. Message Processing for Non-Standard Timescales

When playing at speeds above or below normal, the Logger makes corrective adjustments to the velocity and acceleration values in outgoing entity state messages. For details, see section 3.3.3, "[Velocity and Acceleration Scaling](#)".

DIS only

In DIS, the Logger can generate interim PDUs during slow-speed playback to maintain heartbeat and prevent receiving applications from timing entities out too soon. For more information, see section 3.3.1, "[Generating Interim PDUs During Slow-Speed DIS Playback](#)", and section 4.6, "[Setting Configuration Options](#)".

4.6. Setting Configuration Options

To change configuration options:

1. Choose Options → Configuration. The Configuration dialog box opens (Figure 4-7.)



Figure 4-7. The Configuration dialog box

Table 4-1 describes the configuration options.

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. To set this option at startup, use the <code>-os</code> or <code>-op</code> option.
Record Action	Determines whether the Logger enters a stopped state (default) or begins recording immediately when a file is opened for recording. To set this option at startup, use the <code>-bs</code> or <code>-br</code> option.
End of File Action	When the end of a Logger tape is reached during playback, this option determines whether the Logger stops, pauses, or wraps around to the other end of the tape to continue playback (stop is the default). To set the end-of-file action at startup, use the <code>-es</code> , <code>-ep</code> , or <code>-ew</code> option. See section 3.4, “The End-of-File Action” for more information.
Heartbeat (DIS only)	During slow-speed playback in DIS, this option determines how frequently (in seconds) the Logger generates interim PDUs for each entity. For information, see section 3.3.1, “Generating Interim PDUs During Slow-Speed DIS Playback”.

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

Option	Description
Timeout (DIS only)	During slow-speed playback in DIS, this option determines how long the Logger continues to generate interim PDUs for an entity once that entity's PDUs have stopped occurring in the recording. For more information, see section 3.3.1, " Generating Interim PDUs During Slow-Speed DIS Playback ".



Configuration changes are valid only for a given Logger session. They are not retained for future sessions.

4.7. Summary of Startup Options for the Logger

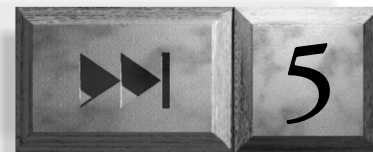
Table 4-2 lists command line options. They are grouped by mode and indicate if they are for HLA only or DIS only.

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

Option	Description
Record and Playback	
-h, -help, -?	Displays a brief summary of Logger command-line options.
-P <i>dis_port</i>	DIS only. Specifies which DIS port (UDP port) the Logger uses to send or receive data. The default port is 3000.
-X <i>ExName</i>	HLA only. Sets federation execution name. VR-Link is default.
-a "0:0:0"	DIS only. Specifies the site:host:entity for remote control of the Logger.
-v {0 1}	Specifies visibility of the Logger interface for remote control. 1 = Logger interface visible (default) 0 = Logger interface not visible
-L {0 1 2}	Specifies the response to Logger Control PDUs for remote control. 0 = Do not respond to Logger Control PDUs 1 = interpret Logger Control PDUs only if they are addressed to the entity ID specified by the -a argument (default) 2 = interpret all Logger Control PDUs
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. Set <i>state</i> to <i>r</i> for record, or <i>s</i> for stopped (default).
-r [<i>filename</i>]	Causes the Logger to open in record mode; specifies the destination file for recorded data. With the Logger, the -r switch is optional, and may be used without a filename. (You choose the filename from the Logger window.)
-R {0 1}	Specifies whether or not to record Logger Control PDUs during remote control. 0 = do not record Logger Control PDUs (default) 1 = record Logger Control PDUs

Table 4-2: Command-line options (Sheet 2 of 2)

Option	Description
<code>-S mcastAddr</code>	DIS only. Subscribes the Logger to the multicast address specified by <code>mcastAddr</code> . To subscribe to several multicast addresses, use this option more than once on the command line. See “Subscribing to Multicast Groups” , on page 2-21.
Playback Mode Only	
<code>-A addr</code>	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 2-21.
<code>-d</code>	Disables the generation of velocity and acceleration scaling data when playing at speeds above or below normal. See section 3.3.3, “Velocity and Acceleration Scaling” .
<code>-e action</code>	Specifies the Logger’s action upon reaching the end of the file. Set <code>action</code> to <code>p</code> to pause at the file’s end, <code>w</code> to wrap to the other end of the tape and continue, or <code>s</code> to stop (default).
<code>-O state</code>	When a file is opened for playback, specifies whether the Logger enters a stopped state, or begins playing immediately. Set <code>state</code> to <code>p</code> for playing, or <code>s</code> for stopped (default).
<code>-p [filename]</code>	Causes the Logger to open in playback mode; specifies the source file to be played. With the Logger, the <code>-p</code> switch is optional, and may be used without a filename. (You choose the filename from the Logger window.)
<code>-S timescale</code>	Sets the initial playback timescale factor. For more information, see section 3.2, “Timescale and Direction During Playback” .
<code>-T timepoint</code>	Sets the timepoint in the file where playback begins. Enter the <code>timepoint</code> in HH:MM:SS format. Default is 00:00:00
<code>-t list</code>	Specifies the list of values that appear on the Logger timescale slider control. Enter <code>list</code> as a series of comma-separated values without spaces. See section 4.5.1, “Using and Customizing the Timescale Slider” .



Using The TLogger

The TLogger offers the same features as the graphical user interface, plus script-based recording and playback capability. This chapter covers the following topics:

Recording Exercises.....	5-2
TLogger Run-time Commands for Recording.....	5-2
Displaying Recording Status.....	5-2
Beginning a Recording Session in Record Mode.....	5-3
Playing Back Recordings.....	5-3
Displaying Playback Status.....	5-4
Beginning a Playback Session in Play Mode.....	5-4
Navigating Time.....	5-5
Controlling End-of-File Behavior.....	5-5
Controlling Entity Maintenance.....	5-6
Setting Playback Timescale and Direction.....	5-7
Message Processing for Non-Standard Timescales.....	5-7
Controlling DIS Heartbeat and Timeout.....	5-8
Setting the Heartbeat Rate.....	5-8
Setting the Timeout.....	5-8
Using a Script to Control Recording and Playback.....	5-9
Stopping A Script.....	5-9
Summary of TLogger Startup Options.....	5-10
Summary of TLogger Run-Time Commands.....	5-12

5.1. Recording Exercises

To begin recording:

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

```
TLogger -r exercise_name.lgr
```

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

At startup, a message shows the NetSocket status.

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

DIS only

The TLogger records all DIS traffic that the host machine receives on UDP port 3000. You can use command-line options to specify a different UDP port or to subscribe to multicast groups. See [“Configuring the Logger for DIS”](#), on page 2-21 for more information.

HLA only

The TLogger records all updates, interactions, and object removals that it perceives through the RTI.

5.1.1. TLogger Run-time Commands for Recording

Recording with the TLogger is straightforward. At the TLogger prompt, enter:

go	To begin or resume recording.
stop	To stop recording without ending the session.
quit	To close the file and exit the TLogger.

5.1.2. Displaying Recording Status

At any time during the recording, 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
>
```

5.1.3. Beginning a Recording Session in Record Mode

By default, when you start the TLogger 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:

```
TLogger -br -r filename
```

5.2. Playing Back Recordings

To play a Logger file:

1. Open a console window.
2. Enter the TLogger command with the `-p filename` argument, for example:

```
TLogger -p ../logger_tapes/my-exercise.lgr
```

DIS only

The TLogger command opens the specified file, and sends the recorded DIS information to the host machine's default broadcast address over UDP port 3000. You can use command-line options to specify a different UDP port or destination IP address. See [“Configuring the Logger for DIS”](#), on page 2-21 for more information.

The DIS status message below shows the current port and source file.

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

5.2.1. Basic TLogger Playback Commands

In playback mode, the TLogger starts in a playing state. Thereafter, it is in either a playing, paused, or stopped state. At the TLogger prompt, enter:

- | | |
|--------------------|---|
| <code>go</code> | To begin or resume playback. When playing, the TLogger sends out PDUs in chronological sequence. |
| <code>pause</code> | To pause playback. When paused, the TLogger stops time and, for DIS only, repeatedly rebroadcasts each entity's most recent entity state PDU. |
| <code>stop</code> | To stop playback. When stopped, the TLogger stops time and does not transmit data. |
| <code>quit</code> | To close the file and exit the TLogger. |

5.2.2. Displaying Playback Status

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

5.2.3. Beginning a Playback Session in Play Mode

By default, when you start the TLogger 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:

```
TLogger -op -p filename
```

5.3. Navigating Time

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

- `start` moves to the beginning of the recording.
- `end` moves to the end of the recording.
- `time` sets the timepoint to a specified time. The command format is:
`time HH:MM:SS`
- `skip` 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:
`skip 00:02:30`
`skip -170`

If you execute a time jump while playback is paused or stopped, the TLogger retains that state. The TLogger also retains a playing state after a time jump unless the end of the file is reached and wrapping is disabled. For more information, see [“The End-of-File Action”](#), on page 3-7, and [“Controlling End-of-File Behavior”](#), on page 5-5.

When you use the `time` or `skip` command during slow-speed playback, the TLogger can perform entity maintenance to facilitate the transmission of up-to-date PDUs. For a description of entity maintenance, see [“Updating Entity Information Quickly After a Time Jump”](#), on page 3-6. For information about how to enable or disable entity maintenance, see [Figure 5.4](#).

5.3.1. Controlling End-of-File Behavior

The TLogger normally enters a stopped state when it reaches the end of a tape.

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

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

where `p` causes the Logger to enter a paused state at the end of a file, and `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:

```
TLogger -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
```

i

Although the TLogger 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 TLogger may be unable to keep up, especially during passages that contain an unusually high concentration of data. If the TLogger falls behind, it continues to process messages as quickly as possible. No messages are discarded.

5.5.1. Message Processing for Non-Standard Timescales

When playing at speeds above or below normal, the TLogger makes corrective adjustments to the velocity and acceleration values in outgoing state update messages. For details, see “[Velocity and Acceleration Scaling](#)”, on page 3-7.

DIS only

In DIS, the TLogger can generate interim PDUs during slow-speed playback to prevent receiving applications from timing entities out too soon. For more information, see “[Generating Interim PDUs During Slow-Speed DIS Playback](#)”, on page 3-5, and “[Controlling DIS Heartbeat and Timeout](#)”, on page 5-8.

5.6. Controlling DIS Heartbeat and Timeout

During slow-speed playback in DIS, the TLogger 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 TLogger generates interim PDUs for each entity. By default, the TLogger 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 TLogger continues to generate interim PDUs for an entity once the entity’s PDUs stop occurring in the recording. The TLogger’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 TLogger should wait before timing an entity out.

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

5.7. Using a Script to Control Recording and Playback

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

A script can contain any TLogger run-time commands listed in Table 5-2. Create the script in any text editor, then redirect it to the Logger as in these examples:

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

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

Controlling Processing with the Wait Command

The `wait` command controls how long the TLogger runs in its current state before processing the next command in the script. The sample script below provides an example.

At startup, jump 12 minutes into the tape.	_____	time 00:12:00
Maintain the current state (play) for 75 seconds.	_____	wait 75
Pause for 10 seconds.	_____	pause
	_____	wait 10
Jump to the 00:04:30 timepoint,	_____	time 00:04:30
set the timescale to 2.0, then	_____	scale 2.0
resume playing for 2½ minutes.	_____	go
	_____	wait 00:02:30
Jump to the 14 minute timepoint.	_____	time 00:14:00
Play the remainder of the tape to the end,	_____	wait
then exit.		

Figure 5-1. Sample TLogger script

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

If a script contains a `wait` command with no `time` argument, the TLogger continues operating in the current state until reaching the end of the tape, 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 TLogger Startup Options

Table 5-1 lists the parameters to the TLogger command.

Table 5-1: TLogger command-line options (Sheet 1 of 2)

Option	Description
Playback and Record	
-h, -help, -?	Displays a brief summary of TLogger command-line options.
-m <i>char</i>	Sets the TLogger's prompt character to <i>char</i> . Default is >.
-P <i>dis_port</i>	DIS only. Specifies which DIS port (UDP port) the TLogger uses to send or receive data. The default port is 3000.
-X <i>ExName</i>	HLA only. Sets the federation execution name. <i>VR-Link</i> is the default.
-a "0:0:0"	DIS only. Specifies the site:host:entity for remote control of the Logger.
-v {0 1}	Specifies visibility of the Logger interface for remote control. 1 = Logger interface visible (default) 0 = Logger interface not visible
-L {0 1 2}	Specifies the response to Logger Control PDUs for remote control. 0 = Do not respond to Logger Control PDUs 1 = interpret Logger Control PDUs only if they are addressed to the entity ID specified by the -a argument (default) 2 = interpret all Logger Control PDUs
Record Mode Only	
-b <i>state</i>	Specifies whether the TLogger opens in a stopped state or begins recording immediately. Set <i>state</i> to <i>r</i> for record, or <i>s</i> for stopped (TLogger default).
-r <i>filename</i>	Causes the TLogger to open in record mode; specifies the destination file for recorded data. When recording with the TLogger, the -r switch and filename are required.
-R {0 1}	Specifies whether or not to record Logger Control PDUs during remote control. 0 = do not record Logger Control PDUs (default) 1 = record Logger Control PDUs
-S <i>mcastAddr</i>	DIS only. Subscribes the TLogger to the multicast address specified by <i>mcastAddr</i> . To subscribe to several multicast addresses, use this option more than once. For information, see "Subscribing to Multicast Groups" , on page 2-21.

Table 5-1: TLogger command-line options (Sheet 2 of 2)

Option	Description
Playback Mode Only	
-A <i>addr</i>	DIS only. Sets the default IP address to which the TLogger sends PDUs. If this option is not used, the TLogger uses the computer's broadcast address. See "Subscribing to Multicast Groups" , on page 2-21.
-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 TLogger's action upon reaching the end of the file. Set <i>action</i> to <i>p</i> to pause at the file's end, <i>w</i> to wrap to the other end of the tape and continue, or <i>s</i> to stop (default).
-O <i>state</i>	Specifies whether the TLogger opens in a stopped state or begins playback immediately. Set <i>state</i> to <i>p</i> for playing, or <i>s</i> for stopped (TLogger default).
-p <i>filename</i>	Causes the TLogger to open in playback mode and specifies the source file to be played. When playing back with the TLogger, the <i>-p</i> switch and filename are required.
-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 is 00:00:00

5.9. Summary of TLogger Run-Time Commands

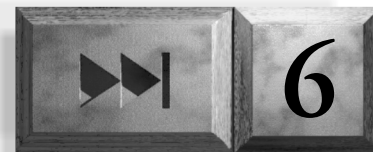
Table 5-2 lists the commands you can send to TLogger while it is running.

Table 5-2: TLogger 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 TLogger 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 TLogger run-time commands.
<code>wait time</code>	Used only in scripts. Specifies how long the TLogger 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.
Record Mode Only	
<code>start</code>	Sets the current timepoint to the beginning of the 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 <i>factor</i> 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 <i>timepoint</i> in the recording. Specify <i>timepoint</i> in HH:MM:SS format.
<code>skip offset</code>	Moves the current timepoint forward or backward by the specified <i>time 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.
<code>timemaint [n]</code>	Determines whether the TLogger generates up-to-date entity state messages after you jump to a new timepoint via the <code>time</code> command. Use a setting of 0 (default) to disable, 1 to enable, or no argument to display the current setting.

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

Command	Description
<code>skipmaint</code> [<i>n</i>]	Determines whether the TLogger generates up-to-date entity state messages after you jump to a new timepoint via the <code>skip</code> command. Use a setting of 0 to disable, 1 (default) to enable, or no argument to display the current setting.
<code>heartbeat</code> [<i>sec</i>]	DIS only. During slow-speed playback, specifies the number of seconds the TLogger allows before generating an interim PDU for a given entity. Default is 5. With no argument, <code>heartbeat</code> displays the current setting.
<code>timeout</code> [<i>sec</i>]	DIS only. During slow-speed playback, specifies how long the TLogger continues to transmit interim PDUs for an entity after the entity's PDUs have stopped occurring in the recording. Defaults to 12 recorded seconds.



Remote Control and the Logger File API

You can control the Logger from a remote application via DIS or HLA. Every command available on the Logger graphical user interface is available for remote control. This chapter discusses the following:

Remote Control PDUs	6-2
Using Logger Commands for Remote Control.....	6-3
Deleting Logger Files.....	6-4
Example of Remote Control for a DIS Exercise.....	6-5
Using the MÄK Logger File API	6-9
Reading A Logger File	6-9
Creating An LtLoggerPlaybackFile	6-9
Reading Data	6-11
Writing a Logger File.....	6-12
Creating an LtLoggerRecordFile.....	6-12
Writing Data	6-12
Interpreting LtPacketInfo	6-13
Editing Logger Files	6-15
Compiling and Linking with the Logger Library.....	6-17

6.1. Remote Control PDUs

Logger Control PDU classes work similarly to the way View Control PDUs work to control the MÄK Stealth. 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*.

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

The remote control command-line arguments control the following behavior:

- ♦ Run Logger without the graphic interface. Since the Logger can be operated remotely, the graphic interface is not a requirement, so you can run the Logger without the interface using the `-v` argument.
- ♦ Specify which remote control PDUs the Logger will accept. You can specify whether or not the Logger will respond to all PDUs, specific PDUs, or no PDUs, with the `-L` argument.
- ♦ Record remote control PDUs. You can keep a record of the control PDUs sent to the Logger. Use the `-R` argument.

DIS only

- ♦ Specify an entity ID for Logger. The Logger will be listening for instructions. It must be addressable so an entity ID is used. You specify the ID with the `-a` argument.

Table 6-1 describes each remote control argument to the Logger command and its possible values.

Table 6-1: Logger command arguments for remote control

Logger command arguments	MTL Syntax	Description
<code>-a "0:0:0"</code>	<code>entityIdStr</code>	Specify the site:host:entity for Logger
<code>-v {0 1}</code>	<code>LgrApp::visible</code>	1 = Logger interface visible (default) 0 = Logger interface not visible
<code>-R {0 1}</code>	<code>LgrAIF::recordLc</code>	0 = Do not record Logger Control PDUs (default) 1 = Record Logger Control PDUs
<code>-L {0 1 2}</code>	<code>LgrAIF::processLc</code>	0 = Do not respond to Logger Control PDUs 1 = Interpret Logger Control PDUs only if they are addressed to the entity ID specified by the <code>-a</code> argument (default) 2 = Interpret all Logger Control PDUs

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);
```

In HLA, add:

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

6.3. Example of Remote Control for a DIS Exercise

The following example demonstrates programmatic remote control of the Logger:

```

/*****
** Copyright (c) 1997 MaK Technologies, Inc.
** All rights reserved.
*****/
#include "stdio.h"
#include "exerciseConn.h"
#include "ProcControl.h"
#include "EntityTypes.h"

// Include each LoggerControl header
#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"

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"
"9  SetTime     float\n"
"10 SkipTime    float\n"
"11 TimeScale   float\n"
"12 PlayAction  {PLAY=0, PAUSE=1, 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"
"-1 Quit this application\n"
;

//
// Application
//
```

```
main()
{
    // Create a connection to the exercise or federation execution
    int port          = 3000;
    int exerciseId    = 1;
    int siteId        = 39;
    int applicationNum = 16;

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

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

    // Create Exercise Connection
    DtExerciseConn exConn(port, exerciseId, siteId, applicationNum);

    // Initialize simulation time
    DtTimeInit();

    // Init work variables
    char  cmdline[128], 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] = NULL;
        sscanf(cmdline, "%d%s",&cmd,optionalArg);

        // Construct and send desired PDU
        switch(cmd)
        {
            case DtLgrPbFile:
                // Set remote Logger playback-file
                exConn.sendStamped(DtLgrPbFilePdu(myId, target,
                                                    optionalArg));
                break;

            case DtLgrRecFile:
                // Set remote Logger record-file
                exConn.sendStamped(DtLgrRecFilePdu(myId, target,
                                                    optionalArg));
                break;

            case DtLgrQuit:
                // Terminate remote Logger
                exConn.sendStamped(DtLgrQuitPdu(myId,target));
                break;

            case DtLgrStart:
                // Rewind remote Logger (during playback)
                exConn.sendStamped(DtLgrStartPdu(myId,target));
                break;

            case DtLgrEnd:
```

```
// Fast-forward remote Logger to end (during playback)
exConn.sendStamped(DtLgrEndPdu(myId,target));
break;

case DtLgrPlay:
    // Start sending recorded traffic onto network
    // (during playback)
    exConn.sendStamped(DtLgrPlayPdu(myId,target));
    break;

case DtLgrPause:
    // Pause sending recorded traffic onto network
    // (during playback)
    exConn.sendStamped(DtLgrPausePdu(myId,target));
    break;

case DtLgrStop:
    // Stop sending recorded traffic onto network
    // (during playback)
    exConn.sendStamped(DtLgrStopPdu(myId,target));
    break;

case DtLgrRecord:
    // Start recorded traffic from network (during record mode)
    exConn.sendStamped(DtLgrRecordPdu(myId,target));
    break;

case DtLgrSetTime:
    // Jump to specified time during playback
    exConn.sendStamped(DtLgrSetTimePdu(myId,target,atof
        (optionalArg)));
    break;

case DtLgrSkipTime:
    // Move forward or backwards by specified amount of time
    // during playback
    exConn.sendStamped(DtLgrSkipTimePdu(myId,target,
        atof(optionalArg)));
    break;

case DtLgrTimeScale:
    // Control the speed of playback
    exConn.sendStamped(DtLgrTimeScalePdu
        (myId,target,atof(optionalArg)));
    break;

case DtLgrPlayAction:
    // Action to take when entering play mode

    exConn.sendStamped(DtLgrPlayActionPdu(myId,target,
        (DtLgrMode)atoi(optionalArg)));
    break;

case DtLgrRecordAction:
    // Action to take when entering record mode

    exConn.sendStamped(DtLgrRecordActionPdu(myId,target,
        (DtLgrMode)atoi(optionalArg)));
    break;
```

```
case DtLgrEofAction:
    // Action to take when play mode reaches end of tape

    exConn.sendStamped(DtLgrEofActionPdu(myId,target,
                                         (DtLgrMode)atoi(optionalArg)));

    break;

case DtLgrHeartbeat:
    // set heartbeat on remote Logger
    exConn.sendStamped(DtLgrHeartbeatPdu
                      (myId,target,atof(optionalArg)));

    break;

case DtLgrTimeout:
    // Set DIS timeout value for remote Logger
    exConn.sendStamped(DtLgrTimeoutPdu(myId,target,atof
                                       (optionalArg)));

    break;
    }
}
```



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

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

6.4. Using the MÄK Logger File API

In addition to the standalone tools that help you view or modify MÄK Logger files, we provide an API (application programmer's interface) to the MÄK 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.



The Logger also has a plug-in API, which allows you to add functionality to the Logger. The plug-in API is described in Chapter 7, *The Logger Plug-in API*.

Because the Logger file library uses MÄK VR-Link, you must have a licensed copy of 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:

- ♦ *lgrFile5MD.lib*
- ♦ *lgrFile5MT.lib*
- ♦ *lgrFileRPRMD.lib*.

See “Compiling VR-Link Applications” in the *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*.

6.4.1. 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.

6.4.2. Creating An LtLoggerPlaybackFile

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.

For both HLA and DIS, to read data from standard input, rather than from an actual file, pass the string "stdin" as the filename argument to the *LtLoggerPlaybackFile* constructor.

6.4.3. Reading Data

When you first create an *LtLoggerPlaybackFile*, you are at the beginning of the file. The functions described in Table 6-2 either read and return a packet, or change the location of a file pointer that determines where you are in the file:

Table 6-2: *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 <code>exConn</code> , and the packet cannot be converted to the <code>exConn</code> '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 <code>exConn</code> , and the packet cannot be converted to the <code>exConn</code> 's FOM.
<code>skipPacketForwards()</code> <code>skipPacketBackwards()</code>	Skip over the next packet in either a forwards or backwards direction. 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 <code>t</code> (in seconds) in the file.
<code>seekForward(DtTime t)</code> <code>seekBackward(DtTime t)</code>	Jump forward or backwards in the file by <code>t</code> seconds.

The functions described in Table 6-3 provide information about the file and about your current location in the file:

Table 6-3: *LtLoggerPlaybackFile* functions for reporting time and location

Function	Description
<code>atStart()</code> <code>atEnd()</code>	Returns 1 if you are at the beginning or end of the file, otherwise returns 0.
<code>fileTimeBackwards(DtTime *t)</code> <code>fileTimeForwards(DtTime *t)</code>	Fills <code>t</code> with the time of the next or previous packet. Returns 1 if successful, 0 if no there is packet in that direction.
<code>firstPacketTime(DtTime *t)</code> <code>lastPacketTime(DtTime *t)</code>	Fills <code>t</code> 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 6-13 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.

6.4.4. 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.

6.4.5. Creating an *LtLoggerRecordFile*

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());
```

For both HLA and DIS, to write data to standard output, rather than to a file, pass the string "stdout" as the filename argument to the *LtLoggerRecordFile* constructor.

6.4.6. 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 6-13 for information about filling out this object with the data you would like written.

6.4.7. 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 6-1).

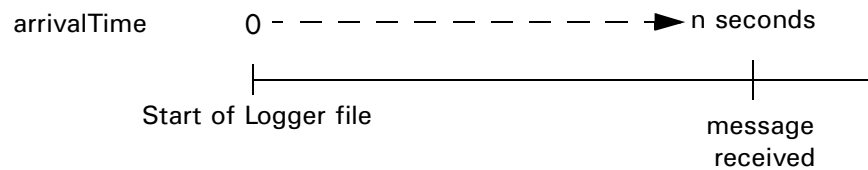


Figure 6-1. `arrivalTime`

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

Both the HLA version and the DIS version of the class include 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 what kind of packet you are dealing with:

- 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.

6.5. 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. Note that they use several classes from VR-Link:

Here is a DIS example of filtering out all Entity State PDUs 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 an 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;  
}
```

6.6. 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:

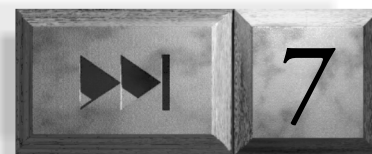
```
-L loggerDir/lib -llgrFilex
```

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 MÄK Logger File API”](#), on page 6-9.



The Logger Plug-in API

The Logger Plug-in API allows you to add features to the Logger or modify its behavior. The Plug-in API is distinct from the Logger File API, which is discussed in Chapter 6, *Remote Control and the Logger File API*.

Introduction	7-2
The Logger Architecture	7-2
Creating a Plug-in	7-4
Registering Subclasses with The Logger	7-4
Loading Plug-Ins	7-5
Creating An Instance of A Logger Element	7-6
Logger Core Elements	7-7
Sources	7-7
Actions	7-10
Filters	7-12
Senders	7-14
The Logger Plug-In Application Interface	7-16
Overriding the Logger Interface	7-22
Plug-in Example	7-23
Building and Compiling the Example Plug-In	7-23
Example Header File	7-23
Implementing the Plug-in	7-24
Registering the Example Plug-in	7-27
Using the Example Plug-in	7-28

7.1. Introduction

A plug-in is a dynamically linked module that is loaded by the Logger at runtime. These modules can be used to alter the basic functionality of the Logger or to extend the Logger functionality. The plug-in API allows you to create your own plug-ins.

7.2. The Logger Architecture

The Logger is an engine that accepts packets from external sources, performs simple processing and then sends the packets on to their appropriate destination. The internal architecture conforms to this structure.

Figure 7-1 illustrates the basic elements of the core engine. They are sources, actions, filters, and senders. Packets are produced by sources, passed to actions, filtered if appropriate, and exit the pipeline through the senders.

The engine makes no distinction between playback and record. It just has different sources and different outputs in each mode. Each of the points in this pipeline can be accessed through the plug-in interface.

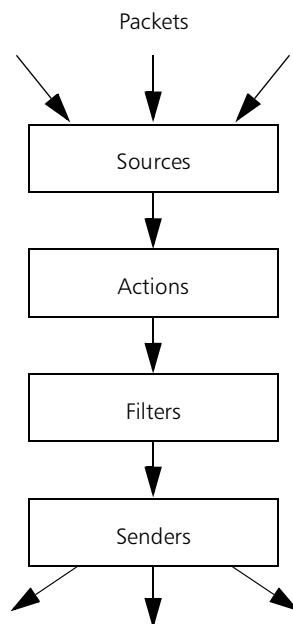


Figure 7-1. Logger engine

Examples of source and send objects are Logger files, HLA exercises, or DIS exercises. An action object might react to Logger Control PDUs. A filter object might remove all packets that involve a certain entity.

In normal operation the engine executes a loop through the following sequence:

1. The engine performs maintenance on the exercise connection. In HLA this means it ticks the RTI. Then it allows each of the elements processing time by calling a similar `tick()` method on each object.
2. The engine asks the sources for a list of all available packets. This list is passed to each element in turn: actions to filters to senders. Once all the elements have had a chance to deal with the packet, it is discarded.
3. The engine processes simulation time, and if possible, sleeps for a short frame.
4. The loop repeats until the engine is told to quit.

The class *DtLoggerInterface* provides the interface to the Logger engine. An instance of *DtLoggerInterface* is created by the engine and passed to each component object during its initialization period.

7.3. Creating a Plug-in

All plug-in libraries must export a “C” style function called `InitPlugin()`. An example of the method is as follows:

```
#include "initMethods.h"

extern "C"
{
    LGR_EXPORT void InitPlugin(void* data)
    {
        /* user register code belongs here */
    }
}
```

The logger application calls this method during initialization right after it finishes reading its configuration files. Any element you want to add to the application can be registered by executing one of the static methods found in *DtLoggerInterface*. Each of the methods is of the following form:

```
static void addNewActionCreator(void* data, const char* className,
                                createMethod method);
```

The **data** parameter is the same data that is passed into the `InitPlugin()` method. The **className** parameter is the name the element will be registered under. You need to write a `createMethod()` that returns an instance of the class being registered. This method looks like:

```
void* createMethod(DtString parameter);
```

Usually, `createMethod()` is declared a class static method of the class being created.

When the object is registered with the program, a message is printed that indicates the name of the object registered. A plug-in can associate an object with name, even those that are built into the program. If it registers to a built in name, the Logger program substitutes the plug-in class for the built-in class. In this way, you can replace the default functionality of the Logger itself.

7.3.1. Registering Subclasses with The Logger

The Logger API provides a base class that implements each of the elements of the Logger engine. A plug-in creates subclasses of the base class and registers them with the Logger. The API includes the following subclasses:

- ♦ `TCPSource` – reads packets from a TCP port
- ♦ `FileSource` – reads packets from a file.
- ♦ `HlaActiveSource` – obtains packets from an active subscription to an HLA exercise
- ♦ `HlaPassiveSource` – obtains packets from a passive subscription to an HLA exercise
- ♦ `HlaSimulate` – sends packets to an HLA exercise
- ♦ `DisSource` – reads packets from a DIS exercise
- ♦ `DisSimulate` – sends packets to a DIS exercise
- ♦ `SendToFile` – sends packets to a file

- ♦ ScreenDump – writes packets, in human readable form, out to the screen or to a file
- ♦ TcpPort – sends packets to a TCP connection.

Here is an example of registering a class to replace the *HlaActiveSource*:

```
LGR_EXPORT void InitPlugin(void* data)
{
    DtLoggerInterface::addNewSourceCreator(data, "HlaActiveSource",
        MySourceClass::create);
}
```

7.3.2. 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 defines the location the Logger should look to find the plug-in modules.
- ♦ **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 installed.

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"
  )
)
; list any objects that should automatically be created
; here. The paramters are Class, Type, Value
(setq enable
  (list
    (list "MyClass" "MyType" "A Value")
  )
)
```

See [“Using the Example Plug-in”](#), on page 7-28 for an example of using these parameters.

7.3.3. Creating An Instance of A Logger Element

An element class that has been registered with the Logger can be instantiated in two ways. The objects can be created in code with the `EnableObjects()` function in the plug-in DLL or by the **enable** parameter in the *logger.mtl* file.

The `EnableObjects()` function is similar to `InitPlugin()` however it is not mandatory. The function should 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 is an example of an `EnableObjects()` method:

```
extern "C"
{
    LGR_EXPORT DtList* EnableObjects( )
    {
        // always build this object
        static char enableTriple[][30] = { "filter",
                                           "MyFilter",
                                           "Some Data" };

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

The same functionality can be performed through the *logger.mtl* file by use of the **enable** parameter. This parameter is a list containing the same three string sets that are used in the `EnableObjects()` method. For an example of using the **enable** parameter, see [“Using the Example Plug-in”](#), on page 7-28.

The rest of this chapter contains definitions for the core elements, the *DtLoggerInterface* class, and the definition of the packets that are processed.

7.4. Logger Core Elements

There are four elements to the Logger core:

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

The Logger plug-in API provides base classes for each of these elements. A plug-in usually subclasses one or more of the base classes and registers them with the Logger. This section describes the base classes for the four API elements. The implementation for these classes is in the *libLogger* library.

7.4.1. Sources

All source elements must inherit from the *DtSourceObj* class. This class generates packets that are processed by other elements in the engine. [Figure 7-2](#) lists the attributes and methods in *DtSourceObj*.

DtSourceObj
protected DtString myName protected DtBoolean myStatus
<pre> public virtual void initSelf(DtLoggerInterface*) public virtual DtTime activate(DtTime) public virtual void deactivate() public virtual DtTime generatePackets(DtTime, DtList) public virtual LtPacketInfo* generatePacket(DtTime currentTime, DtTime* nextTime) public virtual void tick() public virtual const char* printData() const public virtual void reset() public const char* getName() const public virtual DtBoolean isOk() </pre>

Figure 7-2. DtSourceobj class

Attributes

myName

The **myName** attribute is a DtString 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()` method.

myStatus

The **myStatus** attribute is a `DtBoolean` 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()` method.

Methods

```
virtual void initSelf(DtLoggerInterface* theInterface)
```

The `initSelf()` method is the first method called after construction of the object. During this time the object is passed the *DtLoggerInterface* and is expected to perform initialization routines. If the object needs the *DtLoggerInterface* pointer later, it should save it here. This is a pure virtual method and must be overridden.

```
virtual DtTime activate(DtTime startTime)
```

When the core engine wants to begin receiving data from a source, it calls the `activate()` method. 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 core what time this object will be ready to send its first packet. If unknown, the object should return -1. The core begins calling processes to receive the data. This is a pure virtual method and must be overridden.

```
virtual void deactivate()
```

The `deactivate()` method indicates that the object should no longer produce data. If it currently has stored data, it may delete the data. The core engine will no longer be calling `process()` to receive data. This method is abstract and must be overridden.

```
virtual DtTime generatePackets(DtTime currentTime, DtList  
*packetList)
```

The `generatePackets()` method is used by the core 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 core engine. 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 method is to call `generatePacket()` until it returns a `NULL`.

```
virtual LtPacketInfo* generatePacket(DtTime currentTime,  
DtTime* nextTime)
```

The `generatePacket()` method creates one packet a time and is called by the `process()` method. The **currentTime** parameter is the simulation time that packets should be generated for. The **nextTime** parameter is a return value for when the next packet will be ready. This method is abstract and must be overridden.

`virtual void tick()`

The `tick()` method is used by the core engine to yield execution time to the object. This method is virtual, but not abstract. The default version of this method does nothing.

`virtual const char* printData() const`

The `printData()` method returns identifying information about the object. This method is not abstract and by default returns “A Source”.

`virtual void reset()`

When the Logger engine calls the `reset()` method, the object should return to its initial state. For example a file object should return to the beginning of the file. This method is abstract and must be overridden.

`const char* getName() const`

The `getName()` method returns the **myName** attribute.

`virtual DtBoolean isOk()`

If the `isOk()` method returns a value of false, it indicates that there is a problem with the object. The core engine removes the object and destroys it. The method is not abstract and by default returns the value of the **myStatus** attribute.

7.4.2. Actions

All action elements must inherit from the *DtActionObj* class. The purpose of this class is to take actions dependent upon the data that is processed by the class. [Figure 7-3](#) lists the attributes and methods for the *DtActionObj* class.

```
virtual void deactivate()
```

When the `deactivate()` method is called the object will no longer be receiving data in its `process()` method. This is a pure virtual method and must be overridden.

```
virtual void processData(const DtList *packetList)
```

The core sends the list of packets to the `processData()` method so the object can perform appropriate actions. This object cannot make any changes to the list. It can only parse the list and read the data. The default action for this method is to call `actOnPacket()` for every packet in the list.

```
virtual void actOnPacket(const LtPacketInfo* packet)
```

The `actOnPacket()` method is where the action performs any packet specific operation.

```
virtual void tick()
```

The `tick()` method is used by the core engine to yield execution time to the object. This method is virtual, but not abstract. The default version of this method does nothing.

```
virtual const char* printData() const
```

The `printData()` method returns identifying information about the object. This method is abstract and must be overridden.

```
virtual void reset()
```

When the core calls the `reset()` method the object should return to its initial state. For example a file object should return to the beginning of the file. This method is abstract and must be overridden.

```
const char* getName() const
```

The `getName()` method returns the **myName** attribute.

```
virtual DtBoolean isOk()
```

If the `isOk()` method returns a value of false, it indicates that there is a problem with the object. The core engine removes the object and destroys it. The method is not abstract and by default returns the value of the **myStatus** attribute.

7.4.3. Filters

All filter elements must inherit from the *DtFilterObj* class. The purpose of this class is to remove packets that do not belong in the packet list. [Figure 7-4](#) lists the attributes and methods in the *DtFilterObj* class.

DtFilterObj
<pre>protected DtString myName protected DtBoolean myStatus</pre>
<pre>public virtual void initSelf(DtLoggerInterface*) public virtual void activate() public virtual void deactivate() public virtual void processData(DtList) public virtual DtBoolean filterPacket(LtPacketInfo* packet) public virtual void tick() public virtual const char* printInfo() const public const char* getName() const public virtual DtBoolean isOk()</pre>

Figure 7-4. DtFilterObj class

Attributes

myName

The **myName** attribute is a *DtString* 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()` method.

myStatus

The **myStatus** attribute is a *DtBoolean* 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()` method.

Methods

```
virtual void initSelf(DtLoggerInterface* theAgent)
```

The `initSelf()` method is the first method called after construction of the object. During this time the object is passed the *DtLoggerInterface* and is expected to perform initialization routines. If the object needs the *DtLoggerInterface* pointer later it should save it here. This is a pure virtual method and must be overridden.

```
virtual void activate()
```

The `activate()` method is called to indicate that the object will begin to receive data in its `processData()` method. This is a pure virtual method and must be overridden.

```
virtual void deactivate()
```

When the `deactivate()` method is called the object will no longer be receiving data in its `processData()` method. This is a pure virtual method and must be overridden.

```
virtual void processData(DtList *packetList)
```

The core sends the list of packets to the `processData()` method so the object can apply a filter to the packets in the list. This object may then remove a packet from the list. If it removes a packet, it must delete the packet. The default action for this method is to call `filterPacket()` on each packet in the packet list.

```
virtual DtBoolean filterPacket(LtPacketInfo* packet)
```

The `filterPacket()` method is where the filter can make a decision about whether a packet should pass its filter or not. This method is allowed to change the packet. A return value of `DtTrue` indicates to the `processData()` method that the packet should be removed from the list.

```
virtual void tick()
```

The `tick()` method is used by the core engine to yield execution time to the object. This method is virtual, but not abstract. The default version of this method does nothing.

```
virtual const char* printInfo() const
```

The `printInfo()` method returns identifying information about the object. This method is abstract and must be overridden.

```
const char* getName() const
```

The `getName()` method returns the **myName** attribute.

```
virtual DtBoolean isOk()
```

If the `isOk()` method returns a value of false, it indicates that there is a problem with the object. The core engine removes the object and destroys it. The method is not abstract and by default returns the value of the **myStatus** attribute.

7.4.4. Senders

All send elements must inherit from the *DtSendObj* class. The purpose of this class is to direct the packets to their final destinations, for example, Logger files or exercises.

[Figure 7-5](#) lists the attributes and methods for the *DtSendObj* class.

DtSendObj
<pre>protected DtString myName protected DtBoolean myStatus</pre>
<pre>public virtual void initSelf(DtLoggerInterface*) public virtual void activate() public virtual void deactivate() public virtual void processData(DtList) public virtual void sendPacket(const LtPacketInfo*) public virtual void tick() public virtual const char* printData() const public const char* getName() const public virtual DtBoolean isOk()</pre>

Figure 7-5. DtSendObj class

Attributes

myName

The **myName** attribute is a *DtString* 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()` method.

myStatus

The **myStatus** attribute is a *DtBoolean* 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()` method.

Methods

```
virtual void initSelf(DtLoggerInterface* theAgent)
```

The `initSelf()` method is the first method called after construction of the object. During this time the object is passed the *DtLoggerInterface* and is expected to perform initialization routines. If the object needs the *DtLoggerInterface* pointer later it should save it here. This is a pure virtual method and must be overridden.

```
virtual void activate()
```

The `activate()` method is called to indicate that the object will begin to receive data in its `processData()` method. This is a pure virtual method and must be overridden.

`virtual void deactivate()`

When the `deactivate()` method is called the object will no longer be receiving data in its `processData()` method. This is a pure virtual method and must be overridden.

`virtual void processData(const DtList *packetList)`

The core sends the list of packets to the `processData()` method so the object can perform the final processing on the packet. This object cannot make any changes to the packet list. The default action for this method is to call `sendPacket()` for every packet.

`virtual void sendPacket(const LtPacketInfo* packet)`

The `sendPacket()` method is called by `processData()` to act on each individual packet. The method is abstract and must be overridden.

`virtual void tick()`

The `tick()` method is used by the core engine to yield execution time to the object. This method is virtual, but not abstract. The default version of this method does nothing.

`virtual const char* printInfo() const`

The `printInfo()` method returns identifying information about the object. This method is abstract and must be overridden.

`const char* getName() const`

The `getName()` method returns the **myName** attribute.

`virtual DtBoolean isOk()`

If the `isOk()` method returns a value of false, it indicates that there is a problem with the object. The core engine removes the object and destroys it. The method is not abstract and by default returns the value of the **myStatus** attribute.

7.5. The Logger Plug-In Application Interface

Each of the core elements is passed a pointer to the *DtLoggerInterface* class during the *initSelf()* method. This class acts as the interface between the core engine and the elements. It contains methods that control execution and return information regarding current status. This section lists the methods for the API. They are organized functionally, as follows:

- ♦ Simulation level
- ♦ Time related
- ♦ File related
- ♦ Logger control (pause, run, quit, and so on)
- ♦ Logger configuration
- ♦ API level.

Simulation Level Methods

`DtExerciseConn* exerciseConn()`

The *exerciseConn()* method returns a pointer to the current exercise connection.

`DtString fedName()`

The *fedName()* method returns the current federation name. This method will only return valid data if the Logger is in an HLA exercise. In a DIS exercise the return value is undefined.

`double heartbeat()`

The *heartbeat()* method returns the current value of the DIS heartbeat. The value is undefined in an HLA exercise.

`void setHeartbeat(double)`

The *setHeartbeat()* method is used to update the DIS heartbeat value.

`double setTimeout(double)`

The *setTimeout()* method is used to update the timeout value.

`double timeout()`

The *timeout()* method returns the current value of the DIS timeout. The value is undefined in an HLA exercise.

Time Related Methods

`DtTime getDisplayTime()`

The *getDisplayTime()* method returns the current simulation time.

`double getDisplayTimeScale()`

The *getDisplayTimeScale()* method returns the current timescale.

`void setDisplayTimeScale()`

The `setDisplayTimeScale()` method is used to update the timescale.

`void setTimeJumped(DtBoolean)`

The `setTimeJumped()` method is used to set the value of the time jump indicator.

`DtBoolean skipToEnd()`

The `skipToEnd()` method returns a flag to indicate that time has jumped to the end of the file.

`DtBoolean skipToStart()`

The `skipToStart()` method indicates that time has jumped to the beginning of the file.

`DtBoolean timeJumped()`

The `timeJumped()` method returns a flag indicating whether there was a discontinuity in simulation time.

Logger Control Methods

`DtBoolean getTimeToQuit()`

The `getTimeToQuit()` method returns `DtTrue` if the core engine is in the process of shutting down.

`void jumpTime(DtTime jumpTo)`

The `jumpTime()` method causes the Logger to jump to an absolute time value indicated by the parameter `jumpTo`.

`DtLgrMode mode()`

The `mode()` method returns the current Logger mode.

`void pause()`

The `pause()` method pauses the Logger. If the Logger is not currently running it does nothing.

`DtBoolean pb()`

The `pb()` method returns `DtTrue` if the core is currently in playback mode.

`void play()`

The `play()` method starts the Logger playing. If the Logger cannot enter play mode it does nothing.

`void quit()`

The `quit()` method causes the Logger to exit cleanly. After this method is called the `getTimeToQuit()` method returns true.

`void record()`

The `record()` method starts the Logger recording. If the Logger cannot begin recording the method does nothing.

`DtBoolean runFlag()`

The `runFlag()` method returns `DtFalse` if the core engine is not accepting new packets and `DtTrue` if it is.

`void setPb(DtBoolean)`

The `setPb()` method sets the playback mode indicator.

`virtual int DataScaling()`

The `dataScaling()` method indicates whether or not data scaling will take place.

`void skipTime(DtTime skipTo)`

The `skipTime()` method causes the Logger to jump a relative time indicated by the `skipTo` value.

`void stop()`

The `stop()` method stops the Logger from playing or recording.

`virtual void setDataScaling(int)`

The `setDataScaling()` specifies whether or not data scaling will take place. If the value is 0 then no data scaling will take place.

Logger Configuration Methods

`DtLgrMode getEofAction()`

The `getEofAction()` method returns the current value of the end-of-file action. This value determines what the Logger does when it reaches the end of a file.

`DtLgrMode getPlayAction()`

The `getPlayAction()` method returns the current value of the play action. This value determines what the Logger does when it opens a new file.

`DtLgrMode getRecordAction()`

The `getRecordAction()` method returns the current value of the record action. This value determines what the Logger does when it opens a file for recording.

`void setEofAction(DtLgrMode)`

The `setEofAction()` method sets the end-of-file action.

`void setPlayAction(DtLgrMode)`

The `setPlayAction()` method sets the play action.

`void setRecAction(DtLgrMode)`

The `setRecAction()` method sets the record action.

```
void setSkipToEnd(DtBoolean)
```

The setSkipToEnd() method sets the **skipToEnd** indicator.

```
void setSkipToStart(DtBoolean)
```

The setSkipToStart() method sets the **skipToStart** indicator.

File Related Methods

```
void atEOF()
```

The atEOF() method is used to hint to the Logger engine that the source has reached the end of the file. Its only action is to change the value returned by getFilePosition().

```
void atSOF()
```

The atSOF() method is used to hint to the Logger engine that the source has reached the start of the file. Its only action is to change the value returned by getFilePosition().

```
DtString defaultFileName()
```

The defaultFileName() method returns the name of the file that is being played back or recorded. Only returns the first file if more than one file is in use.

```
DtLgrFilePosition getFilePosition()
```

The getFilePosition() method returns the Logger's current position in the Logger file. The data is valid only if there is only one file in use. Otherwise the value is undefined.

```
void inMiddle()
```

The inMiddle() method is used to hint to the Logger engine that the source is in the middle of the file. Its only action is to change the value returned by getFilePosition().

```
void setDefaultFileName(const char* name)
```

The setDefaultFileName() method is used to set the default filename.

API Level Methods

```
void activatePluginFunction(char* name)
```

The activatePluginFunction() method is used to call a plug-in function.

```
int addObject(LtObjectType kind, const char* type, const char* param)
```

The addObject() method adds a new element to the core engine. The parameter **kind** indicates whether it is a source, action, filter, or send. The parameter **type** is the name the object was registered under. The parameter **param** is the parameter the object will be constructed with. This method allows you to have fine control over what objects you want to have running at any given time. The return value is a key that can be used later to delete the object.

```
void addPluginFunction(char* name, char* displayName, func-
tionMethod, void* data)
```

The `addPluginFunction()` method is used to allow an object to pass a function into a common area. This function will be displayed by the default GUI in the plug-ins menu.

```
LgrClock* baseClock()
```

The `baseClock()` method returns a pointer to the underlying clock that playback and record clocks are made from. It should be used if there is a reason to build another clock.

```
const int configChanged()
```

If the `configChanged()` method does not return 0, then the configuration of objects within the core process has changed. Objects (senders, sources, actions, and filters) have either been added or deleted.

```
int delObject(LtObjectType type, const int theKey)
```

The `delObject()` method removes an object from the core engine. The parameter **type** indicates what type of object should be removed. The parameter **key** is the indicator of which object should be removed. The key is either the return value from a previous `addObject()` call or can be found from the keys in the *mySourceNames* and *mySendNames* lists. The return value is the number of objects deleted.

```
void endOfFile()
```

The `endOfFile()` method causes the Logger elements that use files to jump to the end of the file.

```
DtMtlEnvironment* getMtlEnv()
```

The `getMtlEnv()` method returns a pointer to the MTL environment that was used to set up the Logger process.

```
const DtList* getPluginFunctionNames()
```

The `getPluginFunctionNames()` method returns a list of plug-in function names.

```
const DtList* getPluginFunctionDisplayNames()
```

The `getPluginFunctionDisplayNames()` method returns a list of the plug-in function display names.

```
int openFileForPlayback(char* filename)
```

The `openFileForPlayback()` method removes all current source and send objects. Then it creates the default source and send objects using **filename** for the default source name. Typically this creates a *FileSource* object using **filename** and either a *HlaSimulate* or *DisSimulate* object.

`int openFileForRecord(char* filename)`

The `openFileForRecord()` method removes all current source and send objects. Then it creates the default record send and source objects using `filename` as the `send` parameter. Typically this will create a *HlaActiveSource* or *DisSource* and a *SendToFile* object using `filename` as the parameter of the *SendToFile* object.

`LgrPbClock* runClock()`

The `runClock()` method returns a pointer to the run clock for the system.

`DtHashList* sendNames()`

The `sendNames()` method returns a hashlist of `DtStrings` that indicate the names of the currently active senders.

`void setConfigChanged()`

The `setConfigChanged()` method changes the `configChanged` variable to a non zero value depending on what state the core is in when the method is called.

`void setDefaultRecordParam(const char* param)`

The `setDefaultRecordParam()` method is used to set the default value for a record object. This value is sent to the default record source as an option.

`void setMirrorAction(DtBoolean action)`

When the mirror flag is set to true, all Logger actions are sent to the current exercise as Logger control PDU's.

`void setMtlEnvironment(DtMtlEnvironment*)`

The `setMtlEnvironment()` method is used to replace the MTL environment variable.

`DtHashList* sourceNames()`

The `sourceNames()` method returns a hashlist of `DtStrings` that indicate the names of the currently active sources.

`void startOfFile()`

The `startOfFile()` method causes the Logger elements that use files to jump to the beginning of the file.

`DtHashList* stateReps()`

The `stateReps()` method returns a pointer to a hashlist of `stateInfo` structures. The hash is based on `objectId`.

`DtTime stringToTime(char * timestring, int* errorFlag)`

The `stringToTime()` method can be used to convert a string to a time value.

`char* timeToString(double time)`

The `timeToString()` method converts the time to a string.

```
static void registerNewLoggerInterface(DtLoggerInterfaceCon-
structMethod);
```

This method allows a plugin to register a new *DtLoggerInterface*. This method can be called anytime before the creation of the *DtLoggerInterface*. This includes the `initPlugin()` function in plug-ins.

```
static void addNewActionCreator(void*, const char*, DtCreate-
Method);
```

```
static void addNewSourceCreator(void*, const char*, DtCreate-
Method);
```

```
static void addNewFilterCreator(void*, const char*, DtCreate-
Method);
```

```
static void addNewSendCreator(void*, const char*, DtCreate-
Method);
```

These methods can be called from the `initPlugin()` method to register the plug-in's objects with the interface. The initial `void*` is a private parameter that is passed into the `initPlugin()` method.

7.5.1. Overriding the Logger Interface

You can override the *DtLoggerInterface* class. To do this, create a class that inherits from *DtLoggerInterface*. You now have access to common methods such as `play()`. It is important for the overridden class to call the original *DtLoggerInterface* methods to preserve the original logger functionality. For Example:

```
void MyLoggerInterface::play()
{
    //perform my own action on play.
    doSomething();

    //call the original logger play method.
    DtLoggerInterface::play();
}

void MyLoggerInterface::stop()
{
    //perform my stop call
    stopSomething();

    //call the original stop method
    DtLoggerInterface::stop();
}
```

Once created, the new class can be registered from the `initPlugin()` function using the method `DtLoggerInterface::registerNewLoggerInterface()`.

7.6. Plug-in Example

This section presents an example of a plug-in. Although it does not use all of the plug-in interfaces the API provides, it demonstrates the essential components of a plug-in.

The example plug-in is a filter object that will remove entities of a specified type. The filter is configured to work during playback, but could easily be modified to work during record also.

7.6.1. Building and Compiling the Example Plug-In

The files for the example are in the `./plugins/examples` directory. The example is shipped with UNIX makefiles and MS-Visual Studio 6.0 workspace files.

7.6.2. Example Header File

The following code is the header file for the example plug-in:

```
#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 initial initialization
    virtual void initSelf( DtLoggerInterface* theLogger );

    // this method is called when the logger agent has selected
    // 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&);

DtLoggerInterface* myLIF;

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

#endif
```

7.6.3. Implementing the Plug-in

The plug-in 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*. This provides us the basic methods that we must implement.

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

The *DtLoggerInterface* pointer will be needed in later phases of executions, so during the *initSelf()* method this pointer is stored in the **myLoggerInterface** member variable.

The core engine will begin sending packet lists to the *processData()* method that is implemented in the *DtFilterObj* class. This method then proceeds to call the *filterPacket()* method that we have implemented. This filter has been created in a protocol independent manner. However the *filterPacket()* method 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.

The following code is the implementation of the example plug-in:

```
#include "exampleFilter.h"
#include "stateInfo.h"
#include "packetInfo.h"
#include "intHashKey.h"

// constructors
ExampleFilter::ExampleFilter( const char* theName ) :
    DtFilterObj("Example Filter")
{
    // parameter passed into the filter will serve as the entity
    // type that will be filtered.
    myEntityTypeToFilter = new DtEntityType(theName);
```

```

    }

    // destructors
    ExampleFilter::~ExampleFilter()
    {
        delete myEntityTypeToFilter;
    }

    // this method is called once after construction to perform
    // some initial initialization
    void ExampleFilter::initSelf( DtLoggerInterface* theLogger )
    {
        myLIF = theLogger;
    }

    // this method is called when the logger agent has selected
    // to send data to this object
    void ExampleFilter::activate()
    {
        // no action needed
    }

    // this method is called when the logger agent is no longer
    // going to send data to this object
    void ExampleFilter::deactivate()
    {
        // no action needed
    }

    // this method is where the filtering is performed. A return
    // value of DtTrue will cause the packet to be removed by
    // processData.
    DtBoolean ExampleFilter::filterPacket( LtPacketInfo* packet)
    {
        DtBoolean retval = DtFalse;

        if(myLIF->pb()) // only perform the filter during playback
        {
            #if DtHLA
                // In HLA the packets are of three types; Update, Removal, and
                // Interactions
                // the filter is only interested in updates and removals.
                switch(packet->packetKind())
                {
                    case LtPacketUpdate:
                    case LtPacketRemoval:
                    {
                        // In HLA a update packet does not need
                        // to encode all of the objects data. This means that we may
                        // receive and update for a packet we wish to filter, however the
                        // update does not have the information in it to allow us to
                        // apply the filter.
                        //
                        // To solve this problem the LoggerInterface class maintains a
                        // list of StateReps that is updated before any filter is applied.
                        // This information can be used to test if the packet should be
                        // filtered
            
```

```
DtHashList* theStateInfoList = myLIF->stateReps();

// the hashlist is a list of StateInfo Objects hashed on objectId
DtStateInfo* theStateInfo = (DtStateInfo*)theStateInfoList->lookup(
    DtIntHashKey( packet->objectId()));

// If the update is an entity then there will be a stateInfo
if(theStateInfo != NULL)
{
    // if the stateInfo says the packet is already filtered
    // then mark it for deletion but don't bother to test it.
    if(theStateInfo->filtered())
    {
        retval = DtTrue;
    }

    else if(theStateInfo->esr()->entityType().match(
        *myEntityTypeToFilter))
    {
        // Filter this entity
        retval = DtTrue;

        // mark its state as filtered
        theStateInfo->setFiltered(DtTrue);
    }
}
break;
default:
    // do nothing with the packet
    break;
}

#else
    // in DIS the packets represent PDU's
    if(packet->packetKind() == LtPacketPdu)
    {
        // DIS does not have the challenge that HLA has. All the data
        // regarding the object is in the PDU.
        if( ((DtNetPduHeader *) (packet->packet()))->DtKind ==
            DtEntityStatePduKind)
        {
            DtEntityStatePdu esPdu((DtBufferPlaceHolder*)packet->packet());
            if(esPdu.entityType().match(*myEntityTypeToFilter))
            {
                retval = DtTrue;
            }
        }
    }
#endif

return retval;
}

// this method returns a string that supplies identifying
// info about the object
const char* ExampleFilter::printInfo() const
{
```

```
DtString s = "Filter for ";
s += myName;
s += " Objects\n";

return s.string();
}

void* ExampleFilter::create(DtString param)
{
    void* retval = new ExampleFilter(param.string());
    return retval;
}

void ExampleFilter::reset()
{
    // this method is not used in this object
}
```

7.6.4. Registering the Example Plug-in

To register the code with the engine, we use the registration code. Notice the use of the `extern "C"` command. This protects the name of the function from the C++ compiler and allows its name to be found in the library during runtime.

The following code implements registration of the plug-in:

```
#include "exampleFilter.h"
#include "initMethods.h"

extern "C"
{
    // This function is mandatory for all plugins. All logger
    // elements should be registered here. Also the Logger interface
    // class may be overloaded in this method.
    LGR_EXPORT void InitPlugin(void* data)
    {
        // add the filter creator to the filter factory
        DtLoggerInterface::addNewFilterCreator(data, "ExampleFilter",
                                                ExampleFilter::create );
    }

    // This function is optional for plugins. This function is an
    // alternative to using the "enable" parameter in the logger.mtl
    // file.
    LGR_EXPORT DtList* EnableObjects( )
    {
        // always build this object
        static char enableTriple[][30] = { "filter",
                                            "ExampleFilter",
                                            "1:2:225:2:1:0:0" };

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

```
}
```

7.6.5. Using the Example Plug-in

To use the plug-in, move it to a location that can be found by the Logger executable. This location is defined by the `pluginDir` variable in the `logger.mtl` file. By default, all plug-ins in the directory are loaded. You can override this behavior by editing the `plugins` parameter in `logger.mtl`.

Add the DLL you created to the list of plug-ins in the `plugins` parameter, for example:

```
(setq plugins
  (list
    "exampleFilterRPR.dll"
  )
)
```

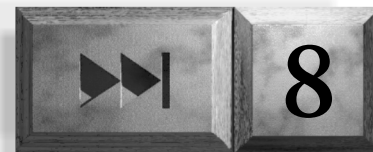
As an alternative to creating an instance of the filter in your code, you can tell the Logger to create an instance by modifying the `enable` parameter in the `logger.mtl` file.

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

If the `montyDemoRPR08.lgr` file is played back with the `logger.mtl` file modified in this way, the aircraft are filtered out. The `enable` parameter creates instances of the filter and could be used to install more than one example filter at a time. For example:

```
(setq enable
  (list
    (list "filter" "ExampleFilter" "1:2:225:2:1:0:0")
    (list "filter" "ExampleFilter" "1:1:222:8:10:0:0")
  )
)
```

This filters out the aircraft and radar entities from the tape.



Logger Tools

The Logger tools are utilities that let you combine, modify, and inspect recorded Logger tapes. This chapter describes:

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

Like the Logger itself, the Logger tools are provided in separate versions for HLA and DIS.

8.1. Viewing Logger Files with lgrDump

The lgrDump utility presents 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 tape. 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 8-1:

Table 8-1: PDU versions and DIS protocol versions

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

8.1.1. 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
```

8.1.2. How lgrDump Displays Unexpected Message Types

Like the Logger itself, this package provides separate lgrDump versions for HLA and DIS. Each lgrDump executable displays formatted data for messages of its intended protocol.

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.

8.1.3. 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.

8.2. Concatenating Files with lgrCat

The lgrCat utility concatenates together messages from a series of 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 8-1](#).

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 8-1. 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 on the command line 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

```

8.2.1. Inserting Time Between Concatenated Files

You can use the optional `-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 8-1](#). 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 8-2](#):

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 8-2. Effect of the lgrCat spacer argument

8.2.2. 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
```

8.3. 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 8-3](#) illustrates a merge of two Logger files:

If File1.lgr contains:	...and File2.lgr contains:
<pre>Msg #1 at time 1.0 Msg #2 at time 3.0 Msg #3 at time 5.0</pre>	<pre>Msg #1 at time 2.0 Msg #2 at time 4.0 Msg #3 at time 6.0</pre>
...then the merged file contains: <pre>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</pre>	

Figure 8-3. 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 on the command line 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
```

8.4. 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.

- > To remove a period of dead time from the beginning of a file, use the `first` keyword as the argument instead of a numeric `time` value. This causes lgrOffset to determine the timepoint of the first message, and to subtract that number of seconds from all messages 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
```

8.5. 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]
```

The *start* and *end* arguments determine the part of the file to be included in the output. lgrTrim discards any packets that fall outside of the specified range. If no *end* is specified, 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.

> To specify a numerical range of messages as the *start* and *end* points, use the *-n* option.

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:

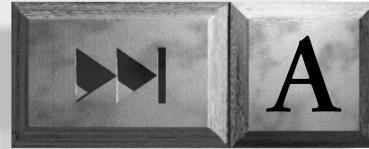
```
lgrTrim -n 10 20 < infile.lgr > outfile.lgr
```

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.....	A-2
 Using the Correct Broadcast Address and UDP Port	A-2
 Configuring the Network Mask on A PC	A-3

A.1. Troubleshooting Logger Network Communications

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 command line 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
```

A.1.2. Configuring the Network Mask on A PC

The Windows versions of the Logger and TLogger expect to run on a class C network using a netmask of 255.255.255.0. If your machine is configured to use a different netmask, create a DtNetmask environment variable to reflect your netmask.

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

If your netmask is	255.255.0.0
	
the hexadecimal representation is:	0xFFFF0000

The default value of DtNetmask is 0xFFFFFFFF00, which corresponds to the netmask 255.255.255.0.

To set the DtNetmask variable for Windows NT:

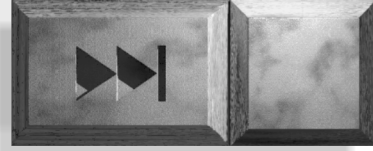
1. Open the Windows NT Control Panel.
2. Open the System applet.
3. Enter the DtNetmask variable name and appropriate hexadecimal value in the fields at the bottom of the System dialog box.
4. Click Set.

To set the DtNetmask variable for Windows 95/98:

1. Add the following line to your AUTOEXEC.BAT file. Replace the hexadecimal number with one that represents your netmask:

```
SET DtNetmask=0xFFFF0000
```

2. Reboot.



Glossary

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

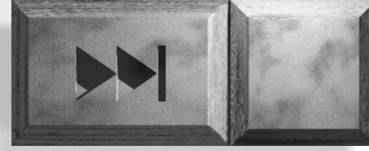
The period of time in which the Logger continues to display an entity after that entity's update messages have stopped appearing on the network.

timescale

A factor by which time is accelerated or slowed in

UDP port

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



Index

Symbols

-? option 4-15, 5-10

? command 5-12

A

-A option 2-21, 4-16, 5-11

acceleration scaling 3-7

action 7-2, 7-10

ActionObj class 7-10

activate() function 7-8, 7-10, 7-12, 7-14

activatePluginFunction() function 7-19

actOnPacket() function 7-11

addNewActionCreator() function 7-22

addNewFilterCreator() function 7-22

addNewSendCreator() function 7-22

addNewSourceCreator() function 7-22

addObject() function 7-19

addPluginFunction() function 7-20

agentAIF class 7-12, 7-14

AgentAIF pointer 7-24

allowRemoteFileDelete parameter 6-4

annotation 4-7

 creating 4-8

 deleting 4-9

 editing 4-9

 moving to 4-9

 opening annotation file 4-9

API

 Logger file 6-9

appending Logger tapes 8-5

architecture

 Logger 7-2

argument

 attrs 6-13

 classHandle 6-13

 name 6-14

 objectId 6-14

arrivalTime() function 6-13

atEnd() function 6-11

atEOF() function 7-19

atSOF() function 7-19

atStart() function 6-11

attribute

 myName 7-7, 7-9, 7-10, 7-11, 7-12, 7-13, 7-14, 7-15

 myStatus 7-8, 7-9, 7-10, 7-11, 7-12, 7-13, 7-14, 7-15

attrs argument 6-13

B

-b option 4-13, 4-15, 5-10

baseClock() function 7-20

broadcast address A-2

building

 plug-in 7-23

C

case sensitivity of command line 1-3

class

ActionObj 7-10

agentAIF 7-12, 7-14

DtFilterObj 7-24

DtLgrControlInteraction 6-2

DtLgrControlPdu 6-2

DtLgrDelFilePdu 6-4

- DtLoggerInterface*
 - overriding 7-22
- FilterObj* 7-12
- LtLoggerPlaybackFile* 6-9
- LtLoggerRecordFile* 6-9, 6-12
- LtPacketInfo* 6-13
- LtPacketInteraction* 6-13
- LtPacketRemoval* 6-14
- LtPacketUpdate* 6-14
 - registering 7-5
 - remote control 6-2
- SendObj* 7-14
- SourceObj* 7-7
- classHandle** argument 6-13
- command line options
 - general format 1-3
 - ? 4-15, 5-10
 - A 2-21, 4-16, 5-11
 - b 4-13, 4-15, 5-10
 - d 3-7, 4-16, 5-11
 - e 3-7, 4-13, 4-16, 5-5, 5-11
 - h 4-15, 5-10
 - help 4-15, 5-10
 - m 5-10
 - o 4-13, 4-16, 5-11
 - P 4-15, 5-10
 - p 4-16, 5-11
 - r 4-15, 5-10
 - S 2-21, 4-16, 5-10
 - s 4-11, 4-16, 5-11
 - startup under Windows 4-3
 - T 4-16, 5-11
 - t 4-11, 4-16
 - x 4-15, 5-10
- command, remote control 6-3
- compiling
 - Logger library 6-17
 - plug-in 7-23
- concatenating Logger tapes 8-5
- configChanged() function 7-20
- configuration file
 - logger.mtl 7-5, 7-28
- configuration options 4-13
- configuring
 - system for MÄK RTI 2-14
- conventions
 - manual xi
- createMethod() function 7-4
- creating
 - annotation 4-8

- Logger file 6-9, 6-12

D

- d option 3-7, 4-16, 5-11
- dataScaling() function 7-18
- deactivate() function 7-8, 7-11, 7-13, 7-15
- dead reckoning
 - acceleration scaling 3-7
 - effect of reverse playback on 3-7
 - velocity scaling 3-7
- dead time, removing from start of tape 8-8
- defaultFileName() function 7-19
- deleting
 - annotation 4-9
 - Logger file 6-4
- delObject() function 7-20
- destination address 2-21
- directory
 - ./bin* 2-5
 - ./doc* 2-5
 - ./logger_tapes* 2-5
- DIS port
 - Logger default A-2
 - specifying 4-15, 5-10, A-2
- DisSimulate module 7-4
- DisSource module 7-4
- DLL
 - plug-in 7-28
- DMSO RTI 2-15
- DtFilterObj* class 7-24
- DtLgrControlInteraction* class 6-2
- DtLgrControlPdu* class 6-2
- DtLgrDelFilePdu* class 6-4
- DtLoggerInterface* class
 - overriding 7-22
- DtNetmask* environment variable in
 - Windows A-3

E

- e option 3-7, 5-5, 3-7, 4-13, 4-16, 5-11
- Edit Annotation dialog box 4-9
- editing
 - annotation 4-9
 - Logger files 6-15
- editing a Logger tape 8-9
- EnableObjects() function 7-6
- End button 4-6
- end command 5-5, 5-12

end-of-file action 3-7, 4-13
 in TLogger 5-5
 endOfFile() function 7-20
 engine
 action loop 7-3
 Logger 7-2
 entity ID, remote control for 6-3
 entity maintenance 3-6, 4-6, 5-5, 5-6
 environment variable
 license manager 2-9
 MAKLMGRD_LICENSE_FILE 2-9
 RTI_CONFIG 2-15
 example, remote control 6-5
 exerciseConn() function 7-16
 exiting Logger during script 5-9
 extracting part of a Logger tape 8-9

F

FED file 2-15
 federation execution name 2-17
 specifying 4-15, 5-10
 fedName() function 7-16
 file
 FED 2-15
 logger.mtl 7-5, 7-28
 RID 2-15
 RTI.RID 2-15
 FileSource module 7-4
 fileTimeBackwards() function 6-11
 fileTimeForwards() function 6-11
 filter 7-2, 7-12, 7-24
FilterObj class 7-12
 filterPacket() function 7-13, 7-24
 firstPacketTime() function 6-11
 FLEXlm license manager 2-3, 2-6
 function
 activate() 7-8, 7-10, 7-12, 7-14
 activatePluginFunction() 7-19
 actOnPacket() 7-11
 addNewActionCreator() 7-22
 addNewFilterCreator() 7-22
 addNewSendCreator() 7-22
 addNewSourceCreator() 7-22
 addObject() 7-19
 addPluginFunction() 7-20
 arrivalTime() 6-13
 atEnd() 6-11
 atEOF() 7-19
 atSOF() 7-19

atStart() 6-11
 baseClock() 7-20
 configChanged() 7-20
 createMethod() 7-4
 dataScaling() 7-18
 deactivate() 7-8, 7-11, 7-13, 7-15
 defaultFileName() 7-19
 delObject() 7-20
 EnableObjects() 7-6
 endOfFile() 7-20
 exerciseConn() 7-16
 fedName() 7-16
 fileTimeBackwards() 6-11
 fileTimeForwards() 6-11
 filterPacket() 7-13, 7-24
 firstPacketTime() 6-11
 generatePacket() 7-8
 generatePackets() 7-8
 getDisplayTime() 7-16
 getDisplayTimeScale() 7-16
 getEofAction() 7-18
 getFilePosition() 7-19
 getMtlEnv() 7-20
 getName() 7-9, 7-10, 7-11, 7-12, 7-13, 7-14, 7-15
 getPlayAction() 7-18
 getPluginFunctionNames() 7-20
 getRecordAction() 7-18
 getTimeToQuit() 7-17
 heartbeat() 7-16
 InitPlugin() 7-4, 7-6
 initPlugin() 7-22
 initSelf() 7-8, 7-10, 7-12, 7-14, 7-24
 inMiddle() 7-19
 isOk() 7-8, 7-9, 7-10, 7-11, 7-12, 7-13, 7-14, 7-15
 jumpTime() 7-17
 jumpToEnd() 6-11
 jumpToStart() 6-11
 lastPacketTime() 6-11
 mode() 7-17
 openFileForPlayback() 7-20
 openFileForRecord() 7-21
 packet() 6-13
 packetKind() 6-13
 params() 6-14
 pause() 7-17
 pb() 7-17
 play() 7-17
 printData() 7-9, 7-11

printInfo() 7-13, 7-15
 process() 7-8, 7-10
 processData() 7-11, 7-12, 7-13, 7-14, 7-15
 quit() 7-17
 readPacket() 6-11, 6-13
 readPacketBackwards() 6-11
 record() 7-18
 reset() 7-9, 7-11
 runClock() 7-21
 runFlag() 7-18
 seekBackward() 6-11
 seekForward() 6-11
 seekTime() 6-11
 sendNames() 7-21
 sendPacket() 7-15
 setConfigChanged() 7-21
 setDataScaling() 7-18
 setDefaultFileName() 7-19
 setDefaultRecordParam() 7-21
 setDisplayTimeScale() 7-17
 setEofAction() 7-18
 setHeartbeat() 7-16
 setMirrorAction() 7-21
 setMtlEnvironment() 7-21
 setPb() 7-18
 setPlayAction() 7-18
 setRecAction() 7-18
 setSkipToEnd() 7-19
 setSkipToStart() 7-19
 setTimeJumped() 7-17
 setTimeout() 7-16
 size() 6-14
 skipPacketBackwards() 6-11
 skipPacketForwards() 6-11
 skipTime() 7-18
 skipToStart() 7-17
 sourceNames() 7-21
 startOfFile() 7-21
 stateReps() 7-21
 stop() 7-18
 stringToTime() 7-21
 tick() 7-9, 7-11, 7-13, 7-15
 timeJumped() 7-17
 timeout() 7-16
 timeToString() 7-21
 writePacket() 6-12, 6-13

G

generatePacket() function 7-8

generatePackets() function 7-8
 getDisplayTime() function 7-16
 getDisplayTimeScale() function 7-16
 getEofAction() function 7-18
 getFilePosition() function 7-19
 getMtlEnv() function 7-20
 getName() function 7-9, 7-10, 7-11, 7-12, 7-13, 7-14, 7-15
 getPlayAction() function 7-18
 getPluginFunctionNames() function 7-20
 getRecordAction() function 7-18
 getTimeToQuit() function 7-17
 go command 5-12

H

-h option 4-15, 5-10
 -help option 4-15, 5-10
 h command 5-12
 header file 7-23
 loggerFile.h 6-9
 heartbeat
 defined 3-5
 setting 4-13
 TLogger command 5-13
 heartbeat() function 7-16
 help command 5-12
 HlaActiveSource module 7-5
 HlaPassiveSource module 7-4
 HlaSimulate module 7-4

I

i command 5-12
 implementing
 plug-in 7-24
 info command 5-4, 5-12
 InitPlugin 7-6
 InitPlugin() function 7-4, 7-6
 initPlugin() function 7-22
 initSelf() function 7-8, 7-10, 7-12, 7-14, 7-24
 inMiddle() function 7-19
 installing
 DMSO RTI 2-15
 MÄK RTI 2-14
 interface, running without 6-3
 interim PDUs 3-5, 3-6, 4-12, 5-7
 isOk() function 7-8, 7-9, 7-10, 7-11, 7-12, 7-13, 7-14, 7-15

J

jumpTime() function 7-17
 jumpToEnd() function 6-11
 jumpToStart() function 6-11

L

lastPacketTime() function 6-11
lgrCat 8-5
 effect on PDU timestamp data 8-5
 piping output from another program 8-6
lgrDump 8-2
 compared with netdump 8-2
lgrDump
 display of unexpected PDU types 8-4
lgrMerge 8-7
lgrOffset 8-8
lgrTrim 8-9
 library
 compiling and linking Logger 6-17
 plug-in 7-4
 provided for Logger file API 6-9
 license manager 2-6
 license server
 running 2-11
 status 2-11
 stopping 2-11
 license, environment variable 2-9
 linking
 Logger library 6-17
lmdown program 2-11
lmhostid program 2-8
lmstat program 2-11
 Logger
 component files 2-5
 interfaces, compared 1-3
 remote control commands 6-3
 software components of 1-2
 tools
 described 8-1
 UNIX vs. Windows versions 1-3
 Logger Control PDUs 6-2
 Logger engine 7-2
 Logger file
 API 6-9
 creating 6-9, 6-12
 deleting 6-4
 editing 6-15
 reading 6-9

 reading data 6-11

 writing 6-12

 writing to 6-12

Logger library

 compiling and linking 6-17

Logger tapes

 combining, using *lgrMerge* 8-7

 concatenating with *lgrCat* 8-5

 editing 8-9

 extracting a portion of 8-9

 modifying time data with *lgrOffset* 8-8

 removing dead time from beginning 8-8

 samples 1-4

 viewing PDU contents of 8-2

 viewing with Stealth 1-4

loggerFile.h header file 6-9

logger.mtl configuration file 7-5

logger.mtl file 7-28

LtLoggerPlaybackFile class 6-9

LtLoggerRecordFile 6-9

LtLoggerRecordFile class 6-9, 6-12

LtPacketInfo class 6-13

LtPacketInteraction class 6-13

LtPacketRemoval class 6-14

LtPacketUpdate class 6-14

M

-m option 5-10

MÄK Plan View Display ix

MÄK RTI

 configuring 2-14

 installing 2-14

MÄK Technical Support x

MAKLMGRD_LICENSE_FILE environment
 variable 2-9

manual

 conventions xi

mode() function 7-17

module 7-4

 DisSimulate 7-4

 DisSource 7-4

 FileSource 7-4

 HlaActiveSource 7-5

 HlaPassiveSource 7-4

 HlaSimulate 7-4

 ScreenDump 7-5

 SendToFile 7-4

 TcpPort 7-5

 TCPSource 7-4

moving
to annotation point 4-9

multicast address 2-21

myName attribute 7-7, 7-9, 7-10, 7-11, 7-12, 7-13, 7-14, 7-15

myStatus attribute 7-8, 7-9, 7-10, 7-11, 7-12, 7-13, 7-14, 7-15

N

name argument 6-14

network mask (netmask) A-3

network, requirements for Logger A-2

O

-o option 4-13, 4-16, 5-11

objectId argument 6-14

openFileForPlayback() function 7-20

openFileForRecord() function 7-21

opening

annotation file 4-9

overriding

DtLoggerInterface class 7-22

P

-P option 4-15, 5-10

-p option 4-16, 5-11

packet 7-2

packet() function 6-13

packetKind() function 6-13

parameter

allowRemoteFileDelete 6-4

params() function 6-14

pause

command 5-12

compared to stop 5-3

pause() function 7-17

pb() function 7-17

PDU

how modified in Logger tape 8-3

remote control 6-2

viewing with *lgrDump* 8-2

Plan View Display ix

play action 4-13

in TLogger 5-4

play() function 7-17

playback 3-3, 4-5

described 3-3

direction 4-11, 5-7

in reverse 3-4

in TLogger 5-3

mode 3-3

speed 4-11, 5-7

allowable range of 4-12, 5-7

in TLogger 5-3

plug-in

building and compiling 7-23

DLL 7-28

header file 7-23

implementing 7-24

libraries 7-4

registering 7-27

printData() function 7-9, 7-11

printInfo() function 7-13, 7-15

process() function 7-8, 7-10

processData() function 7-11, 7-12, 7-13, 7-14, 7-15

products

Plan View Display ix

technical support x

upgrades x

Q

quit command 5-12

quit() function 7-17

R

-r option 4-15, 5-10

reading

data from Logger file 6-11

Logger file 6-9

readPacket() function 6-11, 6-13

readPacketBackwards() function 6-11

record action 4-13

in TLogger 5-3

record mode 3-2, 5-2

record() function 7-18

recording 3-2, 4-4

annotating 4-7

described 3-2

status, displaying in TLogger 5-2

with TLogger 5-2

red timescale display 4-12

registering

class 7-5

- plug-in 7-27
- remote control 6-2
 - commands for 6-3
 - example 6-5
- reset() function 7-9, 7-11
- reverse playback
 - and dead reckoning 3-7
- RID file 2-15
- RTI 2-3
 - definition of 2-14
 - DMSO, installing 2-15
 - MÄK, installing 2-14
- RTI_CONFIG environment variable 2-15
- rtiexec 2-16
- RTL.ridfile 2-15
- runClock() function 7-21
- runFlag() function 7-18
- runLm program 2-11
- running
 - license server 2-11
 - without interface 6-3

S

- S option 2-21, 4-16, 5-10
- s option 4-11, 4-16, 5-11
- sample tapes 1-4
- saving
 - annotation file
 - annotation
 - saving file 4-8
- scale command 5-7, 5-12
- scaling, time 3-7, 4-12
 - in TLogger 5-7
- ScreenDump module 7-5
- script
 - creating 5-9
 - executing 5-9
 - graphical interface 1-3
 - quitting Logger while running 5-9
 - wait command 5-9
- seekBackward() function 6-11
- seekForward() function 6-11
- seekTime() function 6-11
- send 7-2, 7-14
- sendNames() function 7-21
- SendObj class 7-14
- sendPacket() function 7-15
- SendToFile module 7-4
- Set Time 3-6

- entity maintenance 4-6
 - menu option 4-6
- Set Time Scale menu option 4-12
- setConfigChanged() function 7-21
- setDataScaling() function 7-18
- setDefaultFileName() function 7-19
- setDefaultRecordParam() function 7-21
- setDisplayTimeScale() function 7-17
- setEofAction() function 7-18
- setHeartbeat() function 7-16
- setMirrorAction() function 7-21
- setMtlEnvironment() function 7-21
- setPb() function 7-18
- setPlayAction() function 7-18
- setRecAction() function 7-18
- setSkipToEnd() function 7-19
- setSkipToStart() function 7-19
- setTimeJumped() function 7-17
- setTimeout() function 7-16
- shutting down
 - license server 2-11
- size() function 6-14
- skip command 5-5, 5-12
 - entity maintenance 3-6, 5-5, 5-6
- Skip Time 3-6
 - entity maintenance 4-6
 - menu option 4-6
- skipmaint command 3-6, 5-13
- skipPacketBackwards() function 6-11
- skipPacketForwards() function 6-11
- skipTime() function 7-18
- skipToStart() function 7-17
- slow-speed playback 3-4, 5-7
- source 7-2, 7-7
- sourceNames() function 7-21
- SourceObj class 7-7
- Start button 4-6
- start command 5-5, 5-12
- startOfFile() function 7-21
- startup
 - from DOS 4-3
 - from icon under Windows 4-3
 - playback mode 3-3
 - record mode 3-2
 - TLogger, record mode 5-2
 - UNIX 4-2
- stateReps() function 7-21
- status
 - license server 2-11
- status field 4-4, 4-5

- status message, in TLogger playback 5-4
- stop command
 - compared to pause 5-3
 - in playback mode 5-12
 - in record mode 5-2
- stop() function 7-18
- stopping
 - license server 2-11
- stringToTime() function 7-21
- support
 - technical x
- system requirements 2-2

T

- T option 4-16, 5-11
- t option 4-11, 4-16
- TcpPort module 7-5
- TCPSource module 7-4
- technical support x
- tick() function 7-9, 7-11, 7-13, 7-15
- time
 - between concatenated Logger tapes 8-6
 - navigating during playback 4-6
 - navigating during TLogger playback 5-5
 - shifting PDU times 8-8
- time command 5-5
 - and entity maintenance 5-6
 - entity maintenance 3-6
 - in TLogger 5-12
- time jump, updating entities after 3-6
- time navigation 4-6
 - in TLogger 5-5
- time scale
 - described 3-4
 - setting 4-11, 4-12, 5-7
 - setting in TLogger 5-7
 - slider control
 - controlling with keyboard 4-11
 - customizing 4-11
 - specifying at startup 4-11, 5-7
- timeJumped() function 7-17
- timemaint command 3-6, 5-12
- timeout
 - defined 3-5
 - setting 4-14
 - setting in TLogger 5-13
- timeout() function 7-16
- timepoint
 - setting 4-6
 - setting with TLogger skip command 5-5
 - setting with TLogger time command 5-5
- timescale 5-7
 - during recording 4-4
 - effects 3-5
 - red text, meaning of 4-12
- timeToString() function 7-21
- TLogger run-time commands
 - ? 5-12
 - end 5-5, 5-12
 - go 5-2, 5-3, 5-12
 - h 5-12
 - heartbeat 5-13
 - help 5-12
 - i 5-12
 - info 5-4, 5-12
 - pause 5-2, 5-3, 5-12
 - quit 5-12
 - scale 5-7, 5-12
 - skip 5-5, 5-6, 5-12
 - skipmaint 3-6, 5-6, 5-13
 - start 5-5, 5-12
 - stop 5-2, 5-3, 5-12
 - time 5-5, 5-6, 5-12
 - timemaint 3-6, 5-6, 5-12
 - timeout 5-13
 - wait 5-9, 5-12
- TLogger, remote control commands 6-3
- trimming beginning or end of Logger tapes 8-9

U

- UDP port, specifying 4-15, 5-10
- UNIX
 - Windows versions compared 1-3
- upgrades x
- User's Guide
 - organization of vii

V

- variable
 - environment 2-9
 - license environment 2-9
- velocity scaling 3-7
- View Annotations window 4-8
- view control PDUs 1-4
- VR-Link 6-9

W

wait command 5-9, 5-12

Windows and UNIX versions compared 1-3

wrapping 4-13

- at end of file 3-7, 5-5

- in reverse playback 3-7

writePacket() function 6-12, 6-13

writing

- Logger file 6-12

X-Y-Z

-x option 2-17, 4-15, 5-10



MÄK Technologies
185 Alewife Brook Parkway
Cambridge, MA 02138
617.876.8085