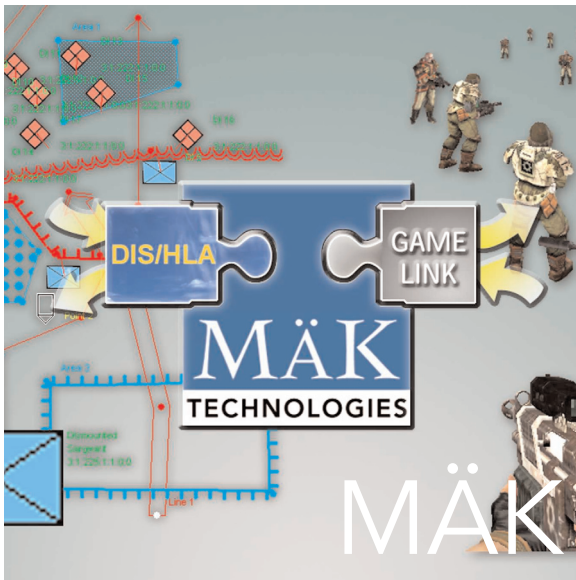
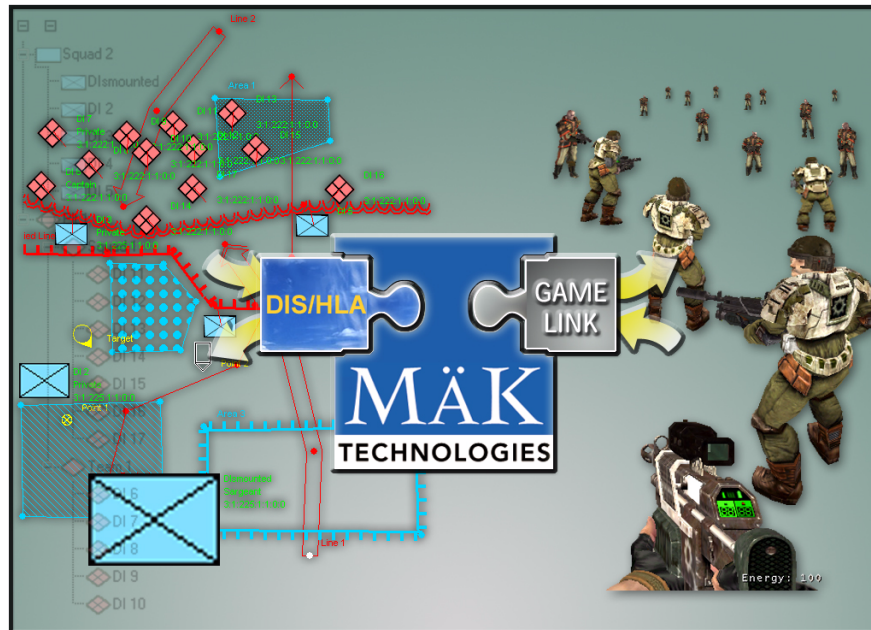




DIS/HLA Game-Link

Developer's Guide



MÄK DIS/HLA Game-Link



Developer's Guide

Copyright © 2005, 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.

MÄK Technologies®, VR-Link®, RTIsPy®, and VR-Forces® are registered trade-
marks of MÄK Technologies, Inc.

Powered by Unreal Engine and Circle-U logo are a trademarks of Epic Games, Inc.
Unreal is a registered trademark of Epic Games, Inc.

All other trademarks are owned by their respective companies.

MÄK Technologies
10 Fawcett St.
Cambridge, MA 02138 USA

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

info@mak.com
www.mak.com

Revision GML-1.0-1-050519



Contents

Preface

How This Manual is Organized	vii
MÄK Products	viii
How to Contact Us	x
Document Conventions	xi
Hardware and Operating System Naming Conventions.....	xi
Mouse Button Naming Conventions	xi

Chapter 1 MÄK DIS/HLA Game-Link

1.1. Introduction	1-2
1.1.1. MÄK DIS/HLA Game-Link Components	1-3
1.1.2. Supported HLA and DIS Versions	1-3
1.1.3. Supported RTI Implementations	1-3
1.1.4. Supported Federation Object Models (FOMs)	1-4
1.1.5. Supported Unreal Engine Versions	1-4

Chapter 2 Installing MÄK DIS/HLA Game-Link

2.1. Installing MÄK DIS/HLA Game-Link	2-2
2.2. Installing the MÄK RTI	2-2
2.3. Setting Up and Using the FLEXlm License Manager	2-3
2.3.1. The FLEXlm Client-Server Architecture	2-3
2.3.2. Installing and Configuring the License Manager	2-4
2.3.3. Installing the License Manager Software	2-4
2.3.4. Requesting a License File	2-5
2.3.5. Putting the License File in the <i>flexlm</i> Directory	2-5
2.3.6. Setting the MAKLMGRD_LICENSE_FILE Variable	2-6
2.3.7. Adding Additional License Files	2-7
2.3.8. Running the License Server	2-7
2.3.9. Running FLEXlm as a Service on Windows	2-8

2.3.10. Troubleshooting the FLEXIm License Manager	2-9
2.4. Installing an RTI	2-11
2.4.1. Installing the MÄK RTI	2-11
2.4.2. Installing the DMSO RTI	2-12

Chapter 3 The Unreal Plug-in Interface

3.1. The Unreal Plug-in Interface	3-2
3.1.1. Modifying the Engine	3-3
3.1.2. Connecting to the Remote Control Module	3-4
3.2. Changes to the <i>AActor</i> Class	3-6
3.2.1. Types of Actors	3-6
3.2.2. Generating Signals	3-7
3.2.3. The Post-Create Event for Remote Actors	3-11
3.2.4. Changes in Vehicles	3-12
3.3. The <i>ULevel</i> Class	3-13
3.3.1. Specifying An Actor's Origin	3-13
3.3.2. Accessing Actors	3-13
3.4. The Remote Control Module	3-14
3.4.1. The Callback Manager	3-14
3.4.2. The Plug-in Manager	3-15
3.4.3. Plug-ins	3-15
3.4.4. Creating A Plug-in	3-16

Chapter 4 The MÄK Adapter Plug-in

4.1. MÄK Adapter Plug-in Architecture	4-2
4.2. Reflector Classes	4-3
4.2.1. Reflector Class Architecture	4-3
4.2.2. Factories	4-7
4.3. How <i>FPluginMakAdapter</i> Works	4-8
4.3.1. Connecting <i>FPluginMakAdapter</i> to <i>MakAdapter</i> - Exported Functions	4-8
4.3.2. Starting <i>MakAdapter</i>	4-9
4.3.3. Configuring <i>MakAdapter</i>	4-10
4.4. Terrain Correlation	4-18

Chapter 5 Integrating Game-Link into the Unreal Engine

5.1. Introduction	5-2
5.2. Copying Game-Link Classes to the Unreal Project	5-2
5.3. Editing Unreal Engine Files	5-3
5.3.1. Editing <i>AActor</i>	5-3
5.3.2. Editing <i>APawn</i>	5-4
5.3.3. Editing <i>ULevel</i>	5-6
5.3.4. Extending <i>UGameEngine</i>	5-7
5.4. Setting Compile Options	5-9

5.5. Linking with the MakAdapter	5-10
--	------

Appendix A MÄK Technologies Lisp

A.1. MÄK Technologies Lisp (MTL) Files	A-2
A.1.1. Using Environment Variables in an MTL File	A-2
A.1.2. Specifying Lists of Lists	A-3
A.1.3. Using Conditional Statements	A-3

Index



Preface

This manual is written for persons who will use the MÄK DIS/HLA Game-Link or develop software using the Game-Link API. The manual assumes that you are familiar with your operating system and development environment, the Unreal Engine development environment, and C++.

Please see release documentation for updates to this manual and the latest information about your version of Game-Link.

How This Manual is Organized

The chapters are organized as follows:

Chapter 1, *MÄK DIS/HLA Game-Link*, introduces the MÄK DIS/HLA Game-Link, describes its components, and describes various system and application compatibility issues.

Chapter 2, *Installing MÄK DIS/HLA Game-Link*, explains how to install the Game-Link and license management software.

Chapter 3, *The Unreal Plug-in Interface*, describes the Game-Link and how it works with the Unreal Engine.

Chapter 4, *The MÄK Adapter Plug-in*, describes the Game-Link plug-in, which allows Unreal games to interoperate with HLA or DIS simulations.

Chapter 5, *Integrating Game-Link into the Unreal Engine*, describes the procedures for integrating the Game-Link files with Unreal Engine source files.

Appendix A, *MÄK Technologies Lisp*, describes the MÄK Technologies Lisp format used in configuration files.

MÄK Products

MÄK DIS/HLA Game-Link is a member of the MÄK Technologies® line of software products designed to streamline the process of developing and using networked simulated environments. The MÄK Technologies product line includes the following:

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

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

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

- ♦ MÄK RTI. An RTI (Run-Time Infrastructure) is required to run applications using the High Level Architecture (HLA). The MÄK RTI is optimized for high performance. It has an API, RTIspy®, that allows you to extend the RTI using plug-in modules. It also has a graphical user interface for the MÄK RTI *rtiexec*.
- ♦ VR-Forces®. VR-Forces is a computer generated forces application and toolkit. It provides an application with a graphical user interface, that gives you a two dimensional view of a terrain database.

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

- ♦ MÄK Stealth and StealthXR. 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.

StealthXR (exaggerated reality) is a separately-licensed component of the Stealth. It combines the features of traditional 2D and 3D displays into one exaggerated reality display. It provides a big picture understanding of a battlefield along with an immersive sense of perspective.

- ♦ **MÄK Data Logger.** The Data Logger, also called the Logger, can record HLA and DIS exercises and play them back for after-action review. You can play a recorded file at speeds above or below normal, or in reverse, and locate areas of interest quickly. The Logger has a graphical user interface and a command line interface. The Logger API allows you to extend the Logger using plug-in modules or embed the Logger into your own application.

The Logger comes with a set of utility programs that let you inspect, combine, and edit Logger recordings.

- ♦ **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.
- ♦ **MÄK Gateway.** The Gateway is a translator application that allows DIS exercises to operate as members of HLA federation executions that use the RPR FOM, thereby preserving the value of simulation code written to support DIS. The MÄK Gateway is a standalone application and does not require you to write additional code for its use.
- ♦ **MÄK HLA/DIS Toolbox for MATLAB and Simulink** is a wrapper around VR-Link that allows MATLAB and Simulink users to analyze data from live or recorded HLA or DIS exercises. It also allows Simulink users to build models that can participate in HLA or DIS exercises.

How to Contact Us

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

Telephone

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

E-mail

Sales and upgrade information:	info@mak.com
Technical support:	support@mak.com

Internet

MÄK web site home page:	www.mak.com
License key requests:	www.mak.com/licenses.htm
Product version and platform information:	www.mak.com/product_version.htm
For the free MÄK RTI trial version:	www.mak.com/rti.htm
Support FAQ:	www.mak.com/faq.htm

Post

Send postal correspondence to:	MÄK Technologies 10 Fawcett St. Cambridge, MA, USA 02138
--------------------------------	--

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

Document Conventions

This manual uses the following typographic conventions:

Monospaced	Indicates commands or values you enter.
<i>Monospaced Italic</i>	Indicates command variables that you replace with appropriate values.
{ }	Indicates required arguments.
[]	Indicates optional arguments.
	Separates options in a command where only one option may be chosen at a time.
/	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.
<i>Italic</i>	Indicates a file name, pathname, or a class name.
sans Serif	Indicates a parameter or argument.
➤	Indicates a one-step procedure.
Menu → Option	Indicates a menu choice. For example, an instruction to select the Save option from the File menu appears as: Choose File → Save.
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 Game-Link home directory. For example, the directory *game-link/doc* appears in the text as *./doc*.

Hardware and Operating System Naming Conventions

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

Mouse Button Naming Conventions

An instruction to click the mouse button, refers to clicking the primary mouse button, 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.



MÄK DIS/HLA Game-Link

This chapter describes MÄK DIS/HLA Game-Link.

Introduction	1-2
MÄK DIS/HLA Game-Link Components	1-3
Supported HLA and DIS Versions.....	1-3
Supported RTI Implementations	1-3
Supported Federation Object Models (FOMs)	1-4
Supported Unreal Engine Versions	1-4

1.1. Introduction

MÄK DIS/HLA Game-Link (Game-Link) enables interoperability between applications built using the Unreal Engine and applications communicating via the High Level Architecture (HLA) or the Distributed Interactive Simulation (DIS) protocol. Game-Link is used with each Unreal server or stand-alone application to convert between Unreal Engine actors and interactions and HLA or DIS objects and interactions. [Figure 1-1](#) shows the results of using Game-Link. Dismounted infantry generated by MÄK VR-Forces (an HLA-compliant, computer generated forces simulation engine) are visible in the Unreal window and the Unreal actor is visible in the VR-Forces window.

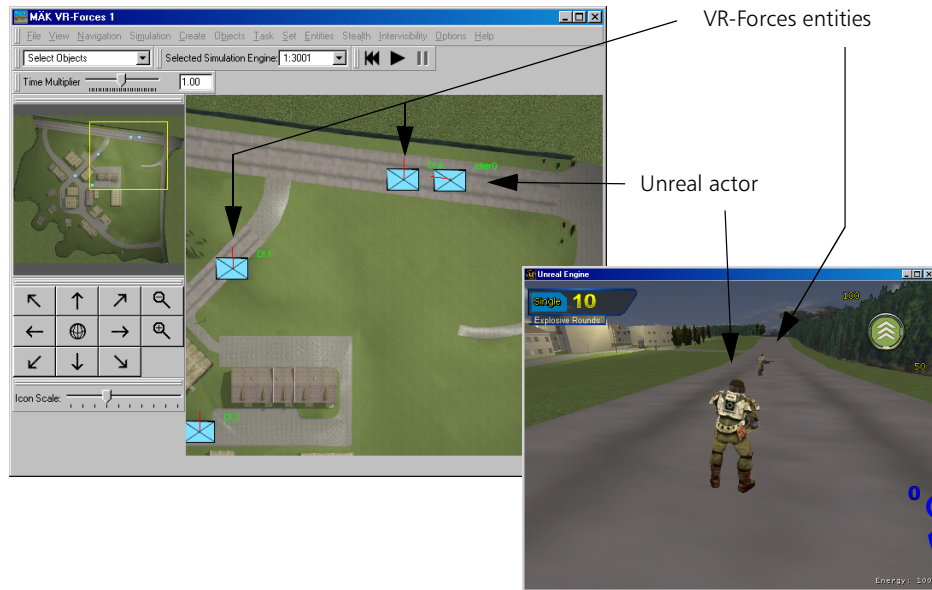


Figure 1-1. Unreal Engine interacting with MÄK VR-Forces

1.1.1. Game-Link Components

Game-Link has the following primary components:

- ♦ The Unreal Plug-in Interface (UPI), which is an extension to the Unreal Engine.
- ♦ The MÄK Adapter plug-in, which can send Unreal object states to the MÄK Adapter (an external application) and stimulate the Unreal Engine based on external simulation messages transmitted by the MÄK Adapter.
- ♦ The MÄK Adapter application, which uses the MÄK VR-Link networking toolkit to listen to HLA or DIS network messages and transmit them to the Adapter plug-in and which receives Unreal state updates from the plug-in and transmits them over the HLA or DIS network.
- ♦ Configuration files that define the mappings between Unreal actors and interactions and HLA or DIS objects and interactions.

This manual describes each of the Game-Link components and explains the process of incorporating them into your Unreal Engine extensions and application source code.

1.1.2. Supported HLA and DIS Versions

Game-Link supports the HLA 1.3 and IEEE 1516 standards. It also supports the DIS 2.0.4, IEEE 1278.1, 2.1.4, and IEEE 1278.1a. You can specify which version to use by specifying the appropriate DLL when you compile your Unreal application. For more details, see “[Linking with the MakAdapter](#),” on page 5-10.

1.1.3. Supported RTI Implementations

Game-Link includes the MÄK RTI (version 2.3.3) and uses it by default. However, you can easily use other link-compatible RTI implementations. To replace this version with any other version of the MÄK RTI or with another RTI, remove the following MÄK RTI libraries and replace them with libraries from the desired RTI:

- ♦ `libFedTime.dll`
- ♦ `libFedTime1516.dll`
- ♦ `libRTI-NG.dll`
- ♦ `librti1516.dll`

You should also remove the `rid.mtl` file and set the RTI configuration file associated with the new RTI.



If you have separately licensed the MÄK RTIs, you must comment out any DLL plug-ins referenced in `rid.mtl` to avoid name conflicts with Game-Link.

1.1.4. Supported Federation Object Models (FOMs)

Game-Link supports for the Real-Time Platform Reference (RPR) FOM versions 0.5, 0.7, 0.8, 1.0, and 2.0, drafts 6, 14, and 17. By default, it uses RPR FOM 1.0. You can specify which version to use in the *MakAdapter.mtl* configuration file. See “[The MakAdapter.mtl Configuration File](#),” on page 4-11 for more details. Please contact MÄK if you need support for other FOMs.

1.1.5. Supported Unreal Engine Versions

Game-Link includes edited versions of the Unreal Engine source and include files for Unreal Engine version 2.0 (drops 2226 and 2227) and version 2.5, drop 3236 and 3339. Please contact MÄK if you need support for older drops.



To use Game-Link, you must license the Unreal Engine from Epic Games, Inc. For licensing details, see <http://www.unrealtechnology.com> or contact licensing@epicgames.com.



Installing Game-Link

This chapter explains how to install the Game-Link and how to install license management software.

Installing MÄK DIS/HLA Game-Link	2-2
Installing the MÄK RTI.....	2-2
Setting Up and Using the FLEXlm License Manager	2-3
The FLEXlm Client-Server Architecture	2-3
Installing and Configuring the License Manager.....	2-4
Installing the License Manager Software	2-4
Requesting a License File.....	2-5
Putting the License File in the flexlm Directory.....	2-5
Setting the MAKLMGRD_LICENSE_FILE Variable.....	2-6
Adding Additional License Files.....	2-7
Running the License Server	2-7
Running FLEXlm as a Service on Windows	2-8
Troubleshooting the FLEXlm License Manager	2-9

2.1. Installing Game-Link

Game-Link is delivered as a self-extracting archive.

To install it:

1. Extract the archive to a directory on your computer.
2. Follow the procedures in Chapter 5, *Integrating Game-Link into the Unreal Engine*.
3. Follow the procedures in “[Setting Up and Using the FLEXlm License Manager](#),” on page 2-3.

2.2. Installing the MÄK RTI

Game-Link includes MÄK RTI, version 2.1a. It is installed automatically when you install Game-Link. You do not have to follow the instructions in “[Installing an RTI](#),” on page 2-11, unless you want to use a different RTI.

2.3. Setting Up and Using the FLEXlm License Manager

MÄK products use the FLEXlm license manager. Before you can use a MÄK product, you must obtain a valid license file and configure the license server and client machines. (For general information about FLEXlm, see <http://www.macrovision.com/products/flexlm/index.shtml>.) 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 server daemon, *lmgrd*, 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.

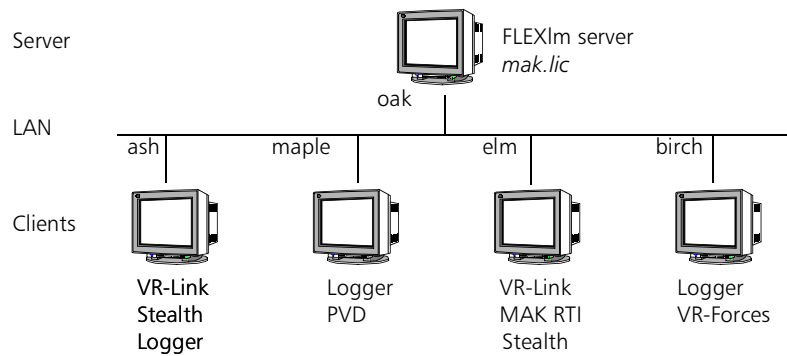
The license server executes vendor daemons (for MÄK Technologies, *maklmgrd*) and uses one or more license files, provided by individual vendors. The license file identifies the products for which you have licenses. On each client machine, the `MAKLMGRD_LICENSE_FILE` environment variable (which you must set) points to the server.

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



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

Figure 2-1 illustrates the client-server architecture for FLEXlm and MÄK products.



The MAKLMGRD_LICENSE_FILE environment variable is set to @oak on each client.

Figure 2-1. 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 2-7.

2.3.3. Installing the License Manager Software

As part of the installation of all MÄK products, the FLEXlm software is installed in the *flexlm92* directory under the product’s root directory.

1. Install your MÄK products on the machines on which you want to run them.
2. Copy the files in *./flexlm92* to *C:/flexlm* on the server machine.
3. Copy the files in *./flexlm92* to the server machine. For Windows, copy the files to *C:/flexlm*. For a UNIX computer, a typical location is */usr/local/flexlm*.



Note the difference between the directory name of the supplied files and the directory to which you copy them.

2.3.4. Requesting a License File

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

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

When you have this information, go to <http://www.mak.com/s1ss10p1.php> and complete the license key request form. MÄK will e-mail you a license file.

Identifying the Host ID and License Server Name in Windows

To find out the Host ID and license server name in Windows:

1. In the flexlm directory, run *lmtools.exe*. The LMTOOLS application opens.
2. Select the System Settings tab. The Host ID is the string in the Ethernet Address text box. The license server name is the name in the Computer/Hostname text box.

Identifying the Host ID and License Server Name on UNIX Computers

To find out your host ID:

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



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

- To find out the name of the server machine, at a command prompt, type `hostname`.

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

The license file you receive may be an attachment or embedded in an email.

1. If the attachment is not named *mak.lic*, rename it *mak.lic*.
If the license is embedded in an email, copy it into an empty text file. Name the file *mak.lic*.
2. Put the license file 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 2-7.

2.3.6. Setting the MAKLMGRD_LICENSE_FILE Variable

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



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

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

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

Windows 2000

To add the MAKLMGRD_LICENSE_FILE in Windows 2000:

1. Right-click the My Computer icon on the Desktop.
2. On the pop-up menu, select Properties. The System Properties dialog box opens.
3. Click the Advanced tab.
4. Click the Environment Variables button. The Environment Variables dialog box opens.
5. Click the New button. The New System Variable dialog box opens.
6. In the Variable Name field, enter MAKLMGRD_LICENSE_FILE.
7. In the Variable Value field, enter *@server_name*, where *server_name* is the name of the license server.
8. Click OK to back out of each dialog box and set the variable.

Windows XP

To add the MAKLMGRD_LICENSE_FILE in Windows XP:

1. On the Start menu, choose My Computer.
2. In the My Computer window, under System Tasks, click View System Information. The System Properties dialog box opens.
3. Follow steps 3 through 8 in the Windows 2000 procedure.

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 2-7.

2.3.7. Adding Additional License Files

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

- To request an additional license file, follow the same procedure as for your first license file. See [“Requesting a License File,”](#) on page 2-5.

To add a new license file to your license server:

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

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

2.3.8. Running the License Server

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

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



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`.

- To obtain the current status of the server, run `lmstat`.

Shutting Down the License Server

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

2.3.9. Running FLEXlm as a Service on Windows

You can configure FLEXlm to run as a service on Windows 2000, or XP.

To configure FLEXlm as a service:

1. Install the license management software and obtain a license as described previously in this chapter.
2. Run `C:/flexlm/lmtools.exe`.
3. On the Service/License File tab, select the Configuration Using Services option.
4. Select the Config Services tab.
5. In the Service Name box, enter a name for the service.
6. In the Path To The lmgrd.exe File box, enter the path of the directory where the `lmgrd.exe` file is located (for example, `C:/flexlm`), or use the Browse button to select the directory.
7. In the Path To The License File box, enter the path of the directory where the license file is located (for example, `C:/flexlm`), or use the Browse button to select the directory.
8. In the Path To The Debug Log File box, enter the path of the directory where you want FLEXlm to create the debug file, or use the Browse button to select a directory.
9. Check the User Services check box. The Start Server At Power Up check box becomes enabled.
10. Check the Start Server At Power Up check box.
11. Click the Save Service button and click Yes at the prompt.
12. Select the Start/Stop/Reread tab.
13. Click Start Server.

2.3.10. Troubleshooting the FLEXlm License Manager

For general troubleshooting information, refer to the FLEXlm FAQ at <http://www.macrovision.com/products/flexlm/index.shtml>. 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, check the Task Manager. (To open the Task Manager, press Ctrl-Alt-Delete and click Task Manager.)

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, select a process in the Task Manager, Processes tab and click End Process.

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

3. See if more than one *lmgrd* or *maklmgrd* process is running. Check the Task Manager. (To open the Task Manager, press Ctrl-Alt-Delete and click Task Manager.)
4. If there are old processes present that may be using a license, kill them. Select a process in the Task Manager, Processes tab and click End Process. It is important that you kill all the processes to ensure a clean restart for the license server.

5. 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:/flexlm/maklmgrd"
```



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

The License Manager Reports an Unsupported Product

If you receive an error message indicating an unsupported product, you are trying to use a MÄK product for which you do not have a license.

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

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

2.4. Installing an RTI

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

i

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

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

All MÄK products are tested against the MÄK RTI and the DMSO RTI. For the most recent information about the RTI versions supported by MÄK products, see the Product Versions page on the MÄK Technologies web site at: <http://www.mak.com/s1ss8p1.php>.

2.4.1. Installing the MÄK RTI

To install the MÄK RTI, follow the instructions in Chapter 2 of *MÄK RTI Reference Manual*. (If you do not have a printed version of the manual, you can view the PDF versions in *makrtix.x/doc*.)

Configuring Your System to Use the MÄK RTI

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

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

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

- ♦ The federation configuration file (*.fed*, *.fdd*, or *.xml*) for your federation execution (required)
- ♦ The RID file (*rid.mtl*) (optional).

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

Running Applications with the MÄK RTI

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

2.4.2. Installing the DMSO RTI

DMSO does not distribute the DMSO RTI any more, but you may be able to obtain it from other sources.

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

Configuring Your System to Use the DMSO RTI

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

Running Applications Using the DMSO RTI

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

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



The Unreal Plug-in Interface

This chapter describes the Unreal Plug-in Interface (UPI).

The Unreal Plug-in Interface.....	3-2
Modifying the Engine.....	3-3
Connecting to the Remote Control Module.....	3-4
Changes to the AActor Class.....	3-6
Types of Actors.....	3-6
Generating Signals.....	3-7
The Post-Create Event for Remote Actors	3-11
Changes in Vehicles.....	3-12
The ULevel Class.....	3-13
Accessing Actors	3-13
Specifying An Actor's Origin	3-13
The Remote Control Module	3-14
The Callback Manager	3-14
The Plug-in Manager	3-15
Plug-ins.....	3-15
Creating A Plug-in	3-16

3.1. The Unreal Plug-in Interface

The Unreal Plug-in Interface (UPI) is an extension to the Unreal Engine applicable to any drop. The UPI lets you write plug-in applications to query the state of the actors in the current level and to stimulate the level – modifying actors, adding new actors, or generating messages like the firing of weapons, detonation of munitions, or other messages, and so on. From a plug-in, you can export data about the state of the game and import data from other applications. You can use plug-ins to, for example, connect with other games or simulations, to create an AAR tool, to create a data logger tool, and so on.

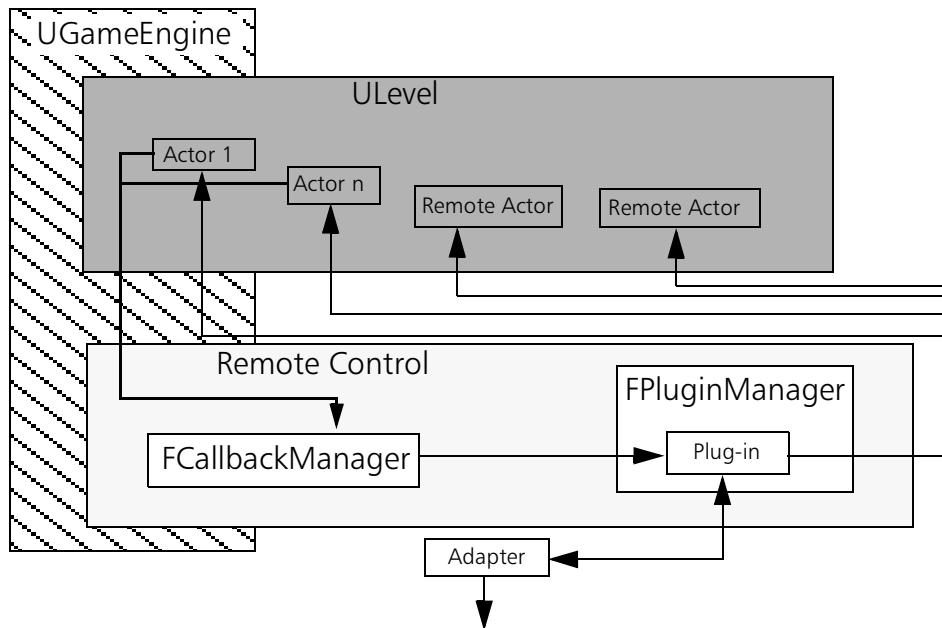


Figure 3-1. Unreal Plug-in Interface architecture

Figure 3-1 illustrates the UPI architecture. Unreal actors send signals that get received by the Remote Control module. Within the Remote Control module, the Callback Manager keeps track of signals for which plug-ins have registered interest. When such a signal is received, it is sent to the plug-in. The plug-in processes the signal in whatever way it is designed to respond. It forwards the data to an external application (an adapter) that processes the data per your needs.

The UPI is implemented by:

- ♦ Extensions to some Unreal Engine classes, mainly *AActor*, *APawn*, *ULevel*, and *UGameEngine*
- ♦ A module, named Remote Control, in which the plug-ins execute.

With the extensions, *AActor*-based classes generate signals for events like creation, deletion, and updates. These signals are sent to the Remote Control module. The Remote Control module passes the signals to any plug-ins that have registered an interest in them.

Extensions in *UGameEngine* enable access to the Remote Control module and keep the module connected with the current level. Plug-ins can affect the game by adding and removing actors, generating interactions, and so on.

Plug-ins are executed in the Remote Control after they are added to the Plug-in Manager. The decision about when to add a plug-in depends on its functionality. Deciding when to start a plug-in is up to you. You can use the mechanisms offered by Unreal, such as a new command line parameter, or a new entry in the Unreal Engine configuration files, and so on. The mechanism used must run the plug-in only where necessary (in the server, in each instance, in the clients, or in stand-alone mode).

3.1.1. Modifying the Engine

To incorporate the UPI in an Unreal Engine drop, you must implement the following major changes to the Unreal Engine:

- ♦ Add the Remote Control module to the Engine project
- ♦ Extend some Engine classes to connect with the Remote Control module
- ♦ Extend the *AActor* hierarchy to implement signals that get sent when basic events occur (creation, deletion, update, fire, detonation, and so on).

The code examples in this chapter show some of the extensions to Unreal Engine classes. Chapter 5, *Integrating Game-Link into the Unreal Engine*, has detailed instructions for integrating the UPI files with the Unreal Engine files and editing Unreal source files.

3.1.2. Connecting to the Remote Control Module

To connect to the Remote Control module, you must extend *UGameEngine* to provide functions to start and stop it, launch the plug-ins, and to connect it with the current level in Unreal. This is done in *RemoteControlUnGame.h* and *RemoteControlUnGame.cpp*.

RemoteControlUnGame.h:

```
// Initialize/destroy the Remote Control
virtual UBOOL initRemoteControl();
virtual UBOOL stopRemoteControl();

// connect/disconnect the Remote Control with level
virtual UBOOL connectRemoteControlToLevel(ULevel *level,
    const char *scenario=NULL);
virtual UBOOL disconnectRemoteControlFromLevel(ULevel *level);

// -----
// Plugin launcher
// -----

// Function available to add plugins into the plugin manager
virtual UBOOL startPlugins();
```

i

The plug-in launcher function is designed to make it easier to add plug-ins to the Plug-in Manager, but it is not required. You can add or remove a plug-in anywhere after the `initRemoteControl()` function and before `stopRemoteControl()`. Use the plug-in launcher function for plug-ins that run for the length of the simulation.

The implementation of the remote control functions (in *RemoteControlUnGame.cpp*) is as follows:

```
BOOL UGameEngine::initRemoteControl()
{
    return FRemoteControl::create();
}

UBOOL UGameEngine::stopRemoteControl()
{
    return FRemoteControl::shutdown();
}

// UPI: connect/disconnect the plugin manager with level
UBOOL UGameEngine::connectRemoteControlToLevel(ULevel *level,
    const char *scenario=NULL)
{
    if (!level)
        return false;
    FRemoteControl::remoteControl()->connectTo(level, scenario);
    return true;
}
```

```
UBOOL UGameEngine::disconnectRemoteControlFromLevel(ULevel *level)
{
    if (!level)
        return false;
    FRemoteControl::remoteControl()->connectTo(NULL);
    return true;
}
```

This version of startPlugins() shows how a plug-in is created and added into the Plug-in Manager:

i

The plug-in checks to see if it must start using a static class function. This mechanism is valid only if just one instance of the plug-in will run in the Engine instance.

```
// UPI: starting plugins that last until the end of the game
UBOOL UGameEngine::startPlugins()
{
    // HLA/DIS plugin Connection management. Only when usesMakAdapter
    // is specified
    if (FPluginMakAdapter::mustStart())
    {
        FPluginMakAdapter *plg = new FPluginMakAdapter();
        plg->setConfigFile("MakAdapter.mtl");
        plg->setPath("./MakData/");
        if (!FRemoteControl::remoteControl()->pluginManager()->
            addPlugin(plg)) delete plg;
    }

    if (GLevel)
    {
        for (int i = 0; i < GLevel->Actors.Num(); i++)
        {
            AActor *actor = GLevel->Actors(i);
            FRemoteControl::remoteControl()->publishCreate(actor);
        }
    }
    return true;
}
```

To activate the Remote Control, you must change the Init() function by adding a call to initRemoteControl(). See “[Extending UGameEngine](#),” on page 5-7.

To stop the Remote Control, you must change the Destroy() function by adding a call to stopRemoteControl(). See “[Extending UGameEngine](#),” on page 5-7.

To connect or disconnect the Remote Control from the current level, you must modify the LoadMap() function when the game is loaded. See “[Extending UGameEngine](#),” on page 5-7.

To launch the plug-ins, you must modify the `UGameEngine::Init()` function to call `startPlugins()`:

```
void UGameEngine::Init()
{
    guard(UGameEngine::Init);
    . . .

    // UPI: Remote Control: subscription of plugins
    startPlugins();

    debugf( NAME_Init, TEXT("Game engine initialized") );

    unguard;
}
```

Additionally, the Engine needs to tick the Remote Control by extending `UGameEngine::Tick()`. See “[Extending UGameEngine](#),” on page 5-7 for sample code.

This tick is passed into the following function for each plug-in listed in the Plug-in Manager:

```
void FPlugin::tick(double dt)
```

Each plug-in can use this periodic function to synchronize with the Engine timer and close the current frame before moving to the next.

3.2. Changes to the AActor Class

The *AActor* class contains the core changes implemented for the UPI. The changes include new state variables in Unreal Script and C++, a post-create event for use when the actor is remote, and new functions to send signals.

3.2.1. Types of Actors

In a stand-alone or client-server Unreal game, all actors are simulated by the Unreal Engine. However if you are running a game that uses the UPI, some actors may be simulated by external programs. Therefore, the Unreal Engine must be able to distinguish between local actors, which it simulates, and remote actors. Furthermore, it has to know whether or not to send information about local actors to the remote control module. To handle these distinctions among actors, the UPI introduces two Boolean variables in Unreal Script, `bIsLocal` and `bMustReflect`, with corresponding images in C++. To use the UPI, you must declare these variables in *Actor.uc*. See “[Editing AActor](#),” on page 5-3 for the specific code.

By default, the state of these variables is:

- ♦ `bIsLocal`: `true` (the actor is an Unreal object)
- ♦ `bMustReflect`: `false` (the actor does not connect the signals about its state with the Remote Control.)

If an actor is not an Unreal object (`bIsLocal = false`), Unreal Engine assumes it is a remote object and it cannot control the object.

3.2.2. Generating Signals

If an actor's `bMustReflect` variable is `true`, it generates signals to inform the Remote Control module about state changes or interactions. These signals are implemented as state signals or interaction signals.

State Signals

State signals are generated from C++, independently of the class of the actor, and directly connected with the Remote Control.

RemoteControlUnActor.cpp:

```
void AActor::remoteControlPostCreate(UBOOL isLocal)
{
    bIsLocal = isLocal;
    bMustReflect = false;
}

void AActor::remoteControlTick(FLOAT DeltaTime)
{
}

void AActor::remoteControlRemove()
{
}
```

State signals are generated when the actor is created (`ULevel::SpawnActor()`). Before `SpawnActor()` returns the actor and after all validations and function calls are done, the actor calls `remoteControlPostCreate()` with the `bLocal` variable received by `SpawnActor()` (see “[The ULevel Class](#),” on page 3-13):

```
AActor* ULevel::SpawnActor
(
    UClass*   Class,
    FName     InName,
    FVector   Location,
    FRotator   Rotation,
    AActor*   Template,
    UBOOL      bNoCollisionFail,
    UBOOL      bRemoteOwned,
    AActor*   Owner,
    APawn*     Instigator,
    UBOOL      bLocal // UPI: extension to track origin
)
{
    guard(ULevel::SpawnActor);
    . . .
    if ( !bRemoteOwned )
        Actor->eventPostNetBeginPlay();

    // UPI: sends the actor to the Remote Control, generating a creation
    // signal
    Actor->remoteControlPostCreate(bLocal);

    return Actor;
    unguardf(( TEXT("(%s)", Class->GetName() ));
}
```

When an actor is deleted (`AActor::Destroy()`), it generates a signal indicating that it is about to be destroyed (the pre-destroy signal). The signal is generated before any other action happens in the `Destroy()` function to ensure the actor's validity, in case any query is sent to it:

```
void AActor::Destroy()
{
    guard(AActor::Destroy);
    remoteControlRemove();

    . . .
}
```

The most frequent signal is the update, generated every time the actor ticks (`AActor::Tick()`), after anything else happens in the tick, and only if the actor is not being removed. In this way, the actor's state remains constant until the next tick:

```
UBOOL AActor::Tick( FLOAT DeltaSeconds, ELevelTick TickType )
{
    guard(AActor::Tick);

    . . .

    TickSpecial(DeltaSeconds); // perform any tick functions unique to an
                                // actor subclass
    // UPI: generating update signal to the Remote Control
    remoteControlTick(DeltaSeconds);
    return 1;
    unguard;
}
```

Subclasses of *AActor* that need to connect the signals with the Remote Control must redefine these functions. In general, any actor that is a pawn must connect its signals with the Remote Control. A pawn generates signals if it is local (must reflect) and if it finds a Remote Control:

```
void AActor::remoteControlPostCreate(UBOOL isLocal)
{
    bIsLocal = isLocal;
    if (!bIsLocal) bMustReflect = isLocal;
    if (bMustReflect && FRemoteControl::remoteControl())
    {
        FRemoteControl::remoteControl()->publishCreate(this);
    }
}

void AActor::remoteControlTick(FLOAT DeltaTime)
{
    if (bMustReflect && FRemoteControl::remoteControl())
    {
        FRemoteControl::remoteControl()->publishUpdate(this);
    }
}
```

```
void AActor::remoteControlRemove()
{
    if (bMustReflect && FRemoteControl::remoteControl())
    {
        bMustReflect = false;
        FRemoteControl::remoteControl()->publishRemove(this);
    }
}
```

Interaction Signals

An interaction is any asynchronous event generated by the Unreal Engine about which the plug-ins must be informed, such as messages, firing of weapons, detonation of munitions, radio signals, and so on.

Fire/Detonation Interaction Signals

As delivered, the UPI implements only two interaction signals – fire interactions and detonation interactions.

Definition in *Actor.uc*:

```
// UPI: Functions to generate fire/detonation interaction signals
// Functions to be called by munitions, reflecting fire
native final function bool ReflectFireInteraction(actor target, vector
    HitLocation);
native final function bool ReflectDetonationInteraction(actor target,
    vector HitLocation);
native final function bool ReflectFireInteractionWithResult(actor target,
    vector HitLocation, int Result);
native final function bool ReflectDetonationInteractionWithResult(actor
    target, vector HitLocation, int Result);
```

The functions with a result receive the type of damage as a value between 0 to 25 based on the following enumeration:

```
enum FFireDetonationResult
{
    FFireDetonationResultOther          = 0,
    FFireDetonationResultEntityImpact    = 1,
    FFireDetonationResultEntityProximate = 2,
    FFireDetonationResultGroundImpact    = 3,
    FFireDetonationResultGroundProximate = 4,
    FFireDetonationResultDetonation      = 5,
    FFireDetonationResultNone            = 6,
    FFireDetonationResultHE_HitSmall     = 7,
    FFireDetonationResultHE_HitMedium    = 8,
    FFireDetonationResultHE_HitLarge     = 9,
    FFireDetonationResultArmorPiercingHit = 10,
    FFireDetonationResultDirtBlastSmall  = 11,
    FFireDetonationResultDirtBlastMedium = 12,
    FFireDetonationResultDirtBlastLarge  = 13,
    FFireDetonationResultWaterBlastSmall = 14,
    FFireDetonationResultWaterBlastMedium = 15,
    FFireDetonationResultWaterBlastLarge = 16,
    FFireDetonationResultAirHit          = 17,
    FFireDetonationResultBuildingHitSmall = 18,
    FFireDetonationResultBuildingHitMedium = 19,
```

```
FFireDetonationResultBuildingHitLarge      = 20,  
FFireDetonationResultMineClearingLineCharge = 21,  
FFireDetonationResultEnvironmentObjectImpact = 22,  
FFireDetonationResultEnvironmentObjectProximate = 23,  
FFireDetonationResultWaterImpact          = 24,  
FFireDetonationResultAirBurst              = 25  
};
```

In C++ (*RemoteControlUnScript.cpp*)

```
void AActor::execReflectFireInteraction( FFrame& Stack, RESULT_DECL )  
{  
    guardSlow(AActor::execFireInteraction);  
  
    P_GET_ACTOR(Target);  
    P_GET_VECTOR(HitLocation);  
    P_FINISH;  
  
    FRemoteControl::remoteControl()->publishFire(this, Target,  
        HitLocation, FFireDetonationResultNone);  
    unguardexecSlow;  
}  
  
void AActor::execReflectDetonationInteraction(FFrame& Stack, RESULT_DECL)  
{  
    guardSlow(AActor::execFireInteraction);  
  
    P_GET_ACTOR(Target);  
    P_GET_VECTOR(HitLocation);  
    P_FINISH;  
  
    if (!HitLocation.IsZero())  
        FRemoteControl::remoteControl()->publishDetonation(this, Target,  
            HitLocation, FFireDetonationResultEntityImpact);  
    unguardexecSlow;  
}  
  
void AActor::execReflectFireInteractionWithResult( FFrame& Stack,  
    RESULT_DECL )  
{  
    guardSlow(AActor::execFireInteraction);  
  
    P_GET_ACTOR(Target);  
    P_GET_VECTOR(HitLocation);  
    P_GET_INT(Result);  
    P_FINISH;  
  
    FRemoteControl::remoteControl()->publishFire(this, Target,  
        HitLocation, Result);  
    unguardexecSlow;  
}  
  
void AActor::execReflectDetonationInteractionWithResult( FFrame& Stack,  
    RESULT_DECL )  
{  
    guardSlow(AActor::execFireInteraction);  
  
    P_GET_ACTOR(Target);  
    P_GET_VECTOR(HitLocation);  
    P_GET_INT(Result);  
    P_FINISH;
```

```

        if (!HitLocation.IsZero())
            FRemoteControl::remoteControl()->publishDetonation(this, Target,
                HitLocation, Result);
        unguardexecSlow;
    }

```

The fire and detonation interaction functions generate signals to the Remote Control from munitions, as in the following example from *AssaultRifleAmmo.uc*:

```

function ProcessTraceHit(Weapon W, Actor Other, Vector HitLocation,
    Vector HitNormal, Vector X, Vector Y, Vector Z)
{
    AmmoAmount -= 1;
    // UPI: generating fire/detonation signals
    ReflectFireInteraction(Other, HitLocation);
    ReflectDetonationInteraction(Other, HitLocation);
    . . .
}

```

The following example calculates the damage type:

```

function ProcessTraceHit(Weapon W, Actor Other, Vector HitLocation,
    Vector HitNormal, Vector X, Vector Y, Vector Z)
{
    AmmoAmount -= 1;

    // calculate damage to be published
    damage = ....

    // UPI: generating fire/detonation signals
    ReflectFireInteractionWithResult(Other, HitLocation, damage);
    ReflectDetonationInteractionWithResult(Other, HitLocation, damage);
    . . .
}

```

3.2.3. The Post-Create Event for Remote Actors

To make it easier to create remote objects, the `RemoteActorPostCreate()` event has been added to the base *AActor* class (*Actor.uc*):

```

// UPI: event called from plug-ins when they create a
// remote actor to set up additional components like inventory,
// attach bones, and so on.
event RemoteActorPostCreate()
{
    // Do something
}

```

Use this event if you need to do something after a remote actor gets added to the Unreal world to make it function properly in Unreal. For example, for soldiers (*CogLightInfantry.uc* in 2.0 or *xPawn* in 2.5), the event creates and assigns the default inventory to make an infantry actor look complete (helmet, weapon, ammo).

```

event RemoteActorPostCreate()
{
    AddDefaultInventory();
}

```

This event is fired from the plug-in when it creates remote actors, calling the C++ image of the Unreal Script event:

```
. . .
actor->eventRemoteActorPostCreate();
. . .
```

3.2.4. Changes in Vehicles

Actors can get into remote vehicles, but cannot drive them. The UPI enhances vehicles so that they know when they can be driven by another actor. In the Unreal Engine, edit the classes *KVehicle* and *SVehicle* (*Vehicle* in 2.5). These classes must check to see if the vehicle is local or remote before setting a driver. To check its origin, a vehicle uses the new `bIsLocal` variable added to *AActor*.

KVehicle.uc

```
// The pawn Driver has tried to take control of this vehicle
function TryToDrive(Pawn p)
{
    local Controller C;
    C = p.Controller;

    // UPI: verifying vehicle origin
    if (bIsLocal && (Driver == None) && (C != None) && C.bIsPlayer &&
        !C.IsInState('PlayerDriving') && p.IsHumanControlled() )
    {
        KDriverEnter(p);
    }
}
```

SVehicle.uc

```
// The pawn Driver has tried to take control of this vehicle
function TryToDrive(Pawn p)
{
    local Controller C;
    C = p.Controller;

    // UPI: verifying vehicle origin
    if (bIsLocal && (Driver == None) && (C != None) && C.bIsPlayer &&
        !C.IsInState('PlayerDriving') && p.IsHumanControlled() )
    {
        KDriverEnter(p);
    }
}
```

3.3. The ULevel Class

The level is the actor's game manager. It creates, deletes, and updates them. The Remote Control Module needs access to the actors list to track the origin of the actor (is the actor local or remote), and to learn about new actors.

3.3.1. Specifying An Actor's Origin

An actor needs to know if it is local or remote. The creator function has been modified to specify the origin and pass it to the actor. To do this, the UPI code adds a new boolean parameter, `bLocal`, to the end of the `SpawnActor()` parameters list. The default value of `bLocal` is `true`, meaning it is an Unreal (local) actor. With this as the default, all calls in existing code work without any change, but plug-ins can specify `bLocal` as `false`, to create remote actors. See “[Editing AActor](#),” on page 5-3 for sample code.

3.3.2. Accessing Actors

The *ULevel* class provides functions to find an actor in an actor list. Searching for an actor that does not exist anymore is normally considered to be an error. However, this is not the case when searching for remote actors. They can be removed when destroyed, but the original source may try to update them. For that reason, a new function `isAValidActor()` has been added to *ULevel*. It is similar to `GetActorIndex()`, but it does not consider it an error to search for an actor that does not exist (*RemoteControlUnLevel.h*):

```
// UPI: function added to search actors but avoiding error messages when
// missing
INT isAValidActor( AActor* Actor )
{
    guard(ULevel::GetActorIndex);
    for( INT i=0; i<Actors.Num(); i++ )
        if( Actors(i) == Actor )
            return i;
    return INDEX_NONE;
    unguard;
}
```

3.4. The Remote Control Module

The main module of the UPI is the Remote Control. The Remote Control has the following tasks:

1. It updates the plug-in's connection to the current level.

```
virtual void connectTo(ULevel *level, const char *scenario=NULL);
```

2. It listens for the signals coming from actors in the current level and it provides the plug-ins with access to the level.

```
virtual void publishCreate(AActor *actor);
virtual void publishUpdate(AActor *actor);
virtual void publishRemove(AActor *actor);
virtual void publishFire(AActor *munition, AActor *target,
    FVector hitLocation, int result);
virtual void publishDetonation(AActor *munition, AActor *target,
    FVector hitLocation, int result);
```

3. It maintains the Callback Manager (to receive the signals) and the Plug-in Manager (to handle plug-ins).

```
virtual FCallbackManager *callbackManager();
virtual FPluginManager *pluginManager();
```

4. It passes control down to each plug-in using the tick() function, called from UnGameEngine::Tick().

```
virtual void tick(float DeltaSeconds);
```

3.4.1. The Callback Manager

The Callback Manager manages the distribution of signals coming from local actors to the modules that have registered callback functions. It has a callback function subscription list for each type of available signal (creation, deletion, updates, and interactions) (*FCallbackManager.h*). You must subscribe and unsubscribe to any callback function you want to receive using one of the following functions:

```
virtual void addPostCreateCallback(FActorCallback cb, void *data);
virtual void removePostCreateCallback(FActorCallback cb, void *data);

virtual void addPreRemoveCallback(FActorCallback cb, void *data);
virtual void removePreRemoveCallback(FActorCallback cb, void *data);

virtual void addPostUpdateCallback(FActorCallback cb, void *data);
virtual void removePostUpdateCallback(FActorCallback cb, void *data);

virtual void addFireCallback(FFireDetonationCallback cb, void *data);
virtual void removeFireCallback(FFireDetonationCallback cb, void *data);

virtual void addDetonationCallback(FFireDetonationCallback cb,
    void *data);
virtual void removeDetonationCallback(FFireDetonationCallback cb,
    void *data);
```

The callbacks are fired when a signal arrives using one of these channels:

```
virtual void dispatchPostCreateCallbacks(AActor *actor);
virtual void dispatchPreRemoveCallbacks(AActor *actor);
virtual void dispatchPostUpdateCallbacks(AActor *actor);
virtual void dispatchFireCallbacks(AActor *munition, AActor *target,
    FVector hitLocation, int result);
virtual void dispatchDetonationCallbacks(AActor *munition, AActor *target,
    FVector hitLocation, int result);
```

3.4.2. The Plug-in Manager

The Plug-in Manager is a collector class that manages the plug-ins and maintains updates to the level connection in each one of them (*FPluginManager.h*).

```
virtual void addPlugin(FPlugin *plugin);
virtual void removePlugin(FPlugin *plugin);
virtual void connectTo(ULevel *level, const char *scenario=NULL);
```

The Plug-in Manager does not own the plug-ins, except when it itself is being destroyed. In that case, all plug-ins in the list are also destroyed. If you remove a plug-in from the Plug-in Manager, it is your responsibility to delete it.

3.4.3. Plug-ins

A plug-in is a standard interface to receive signals from actors and to access the level to stimulate them. A plug-in registers callback functions with the Callback Manager to listen to relevant signals. You can use the basic plug-in class, *FPlugin* (*FPlugin.h*), as a starting point to build other plug-ins:

```
virtual bool init(FPluginManager *mgr);
virtual bool connectTo(ULevel *level, const char *scenario=NULL);
virtual void tick(double dt);

// set of possible callbacks available to subscribe to
// uncomment those of interest in the plugin and discard the rest.

// static void postCreateCallback(AActor *actor, void *data);
// static void postUpdateCallback(AActor *actor, void *data);
// static void preRemoveCallback(AActor *actor, void *data);
// static void fireCallback(AActor *munition, AActor *target,
    FVector hitLocation, int result, void *data);
// static void detonationCallback(AActor *munition, AActor *target,
    FVector hitLocation, int result, void *data);
```

You can also examine the source code for the *FPluginMakAdapter* plug-in. See Chapter 4, *The MÄK Adapter Plug-in*, for details.

3.4.4. Creating A Plug-in

To create a plug-in:

1. Create a subclass from *FPlugin*.
2. Call the parent `init()` function in the `init()` function, before any other initialization code.
3. Implement and subscribe to the desired callback functions.
4. Add code to the engine to create the plug-in and add it to the Plug-in Manager.



You can use the plug-in launcher function to add this code, but it is not required.



The MÄK Adapter Plug-in

This chapter describes the MÄK Adapter plug-in, which lets Unreal games interact with HLA or DIS simulations.

MÄK Adapter Plug-in Architecture.....	4-2
Reflector Classes	4-3
Reflector Class Architecture.....	4-3
Factories	4-7
How FPluginMakAdapter Works.....	4-8
Connecting FPluginMakAdapter to MakAdapter - Exported Functions .	4-8
Starting MakAdapter	4-9
Configuring MakAdapter	4-10
Terrain Correlation	4-18

4.1. MÄK Adapter Plug-in Architecture

The MÄK Adapter plug-in (*FPluginMakAdapter*) is a Game-Link plug-in that connects an Unreal game with an HLA or DIS simulation using a library called the MÄK Adapter application (*MakAdapter*). *FPluginMakAdapter* listens to signals to create and remove actors published by the Unreal Engine and sends them to *MakAdapter*, which sends them out over the network. Similarly, *MakAdapter* listens to the HLA or DIS traffic and sends messages to *FPluginMakAdapter*, which stimulates the Unreal Engine.

The MÄK Adapter plug-in architecture is based on a master–slave structure, in which *MakAdapter* is the master and *FPluginMakAdapter* is a slave that reflects the external simulation and gets information from the Unreal world when *MakAdapter* requires it. Figure 4-1 illustrates the architecture.

This approach is not true for asynchronous events like creating and deleting actors and fire or detonation interactions coming from the Unreal world. These interactions generate asynchronous calls from *FPluginMakAdapter* to *MakAdapter*. In the same way, *MakAdapter* generates asynchronous calls to *FPluginMakAdapter* when a create/remove entity or fire/detonation interaction happens in HLA or DIS.

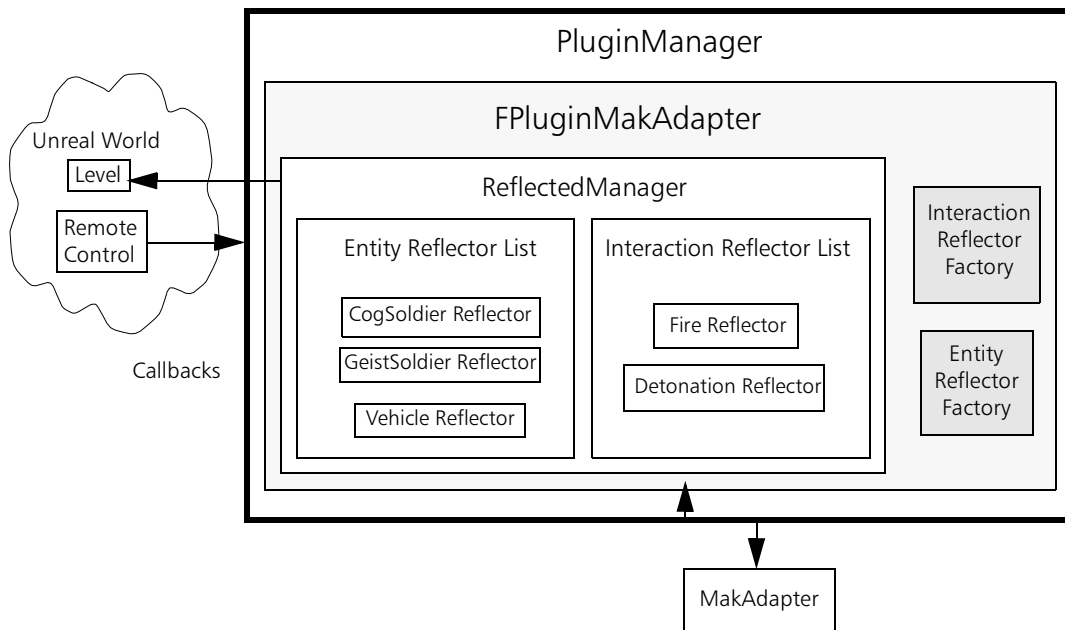


Figure 4-1. MÄK Adapter plug-in architecture

To give control to *MakAdapter* (given that Unreal Engine always believes that it is in control), when the `tick()` function in *FPluginMakAdapter* is called, control is passed to *MakAdapter* and it synchronously executes all procedures (updates of both sides of reflected entities/actors, and so on).

MakAdapter is a proprietary library that can be licensed from MÄK Technologies. It offers a limited number of reflectors, which can be expanded.

FPluginMakAdapter is an open source code module, which you must add to the Unreal Engine (see Chapter 5, *Integrating Game-Link into the Unreal Engine*).

4.2. Reflector Classes

The *MakAdapter*-to-*FPluginMakAdapter* connection is a component-based architecture. The components are modules that are specialized at dealing with specific actor types (Unreal side) and entity types (HLA or DIS side), for example, WarCOGSoldier, Bulldog, DI, M1A2, and so on, or M16 fire, AK47 detonation, and so on. These components are referred to as reflector classes. There are two types of reflector classes – entity reflector classes and interaction reflector classes.

4.2.1. Reflector Class Architecture

Reflector classes are based on:

- An API to get/set data from/to an actor. *FPluginMakAdapter* can use any reflector with any type of actor and it is the reflector's responsibility to access the actor correctly. For example, an infantry entity has posture or physics, but a vehicle does not.
- An API to share information between *MakAdapter* and *FPluginMakAdapter*. Due to memory model differences between the MÄK VR-Link toolkit and the Unreal Engine, data must be shared independently of the memory model. *MakAdapter* and *FPluginMakAdapter* have structures defined to share data, for example (from *FReplicationData.h*):

```
// FCreateData: structure class to pass data about new actors back and
// forth (created in Unreal or creation request coming from MakAdapter)
class FCreateData
{
public:
    FCreateData() {actor = 0; numAttachments = 0; appearance = 0;}

    double      location[3];
    double      orientation[3];
    double      velocity[3];
    double      acceleration[3];
    FActorP     actor;
    FAppearance appearance;
    char         numAttachments;
    char         force;
    char         unused[2];
    char         className[MAX_NAME_SIZE];
    char         objectName[MAX_NAME_SIZE];
    char         attachments[MAX_ATTACHMENTS][MAX_NAME_SIZE];
    char         available[16];
};
```

```
// FUpdateData: update basic information about an actor from/to Unreal.
// -2 <= Health <= 100
// isDestroyed {0, 1}
// numAttachments >= 0
// appearance is bit array used to represent different aspects based on
// type
// DI:          postures
// Vehicles: damage state
class FUpdateData
{
public:
    FUpdateData() {actor = 0; numAttachments = 0; isDestroyed = 0;
        health = 100; appearance = 0;}

    double      location[3];
    double      orientation[3];
    double      velocity[3];
    double      acceleration[3];
    double      attachmentRotations[MAX_ATTACHMENTS][3];
    FActorP     actor;
    FAppearance  appearance;
    char        health;
    char        isDestroyed;
    char        numAttachments;
    char        unused[1];
    char        available[16];
};

// Express damage amount (from/to Unreal)
class FDamageData
{
public:
    FDamageData() {actor = 0; isDestroyed = 0; damage = 0;}

    double      location[3];
    double      distance;
    double      range;
    FActorP     actor;
    char        className[MAX_NAME_SIZE];
    char        munitionType[MAX_NAME_SIZE];
    char        damage;
    char        isDestroyed;
    char        unused[6];
    char        available[16];
};

// Remove the actor from the simulation (from/to Unreal)
class FRemoveData
{
public:
    FRemoveData() {actor = 0;}

    FActorP     actor;
    char        available[16];
};
```

```
// Set/Request data for an actor
class FSetRequestData
{
public:
    FSetRequestData() {type = 0;}

    FActorP          actor;
    FSetRequestDataType type;
    char             unused[4];
    char             data[16];
    char             available[16];
};

// Fire interaction (from/to Unreal)
class FFireData
{
public:
    FFireData() {source = 0; target = 0; result = 0;}

    double    location[3];
    FActorP   source;
    FActorP   target;
    double    result;
    char      className[MAX_NAME_SIZE];
    char      munitionType[MAX_NAME_SIZE];
};

// Detonation interaction (from/to Unreal)
class FDetonationData
{
public:
    FDetonationData() {source = 0; target = 0; result = 0;}

    double    location[3];
    FActorP   source;
    FActorP   target;
    double    result;
    char      className[MAX_NAME_SIZE];
    char      munitionType[MAX_NAME_SIZE];
};

// Request to create a reflector and registered with reflector manager
// reflectorName: reflector's static name
// className: actor's class name to be handed by the reflector's instance
// makToUnreal / unrealToMak: signals of active directions
class FRegisterReflectorData
{
public:
    FRegisterReflectorData() {entityType = 0; makToUnreal = 0;
        unrealToMak = 0;}

    char entityType; // 1 - Entity, 0 - Interaction
    char makToUnreal;
    char unrealToMak;
    char unused[5];
    char reflectorName[MAX_NAME_SIZE];
    char className[MAX_NAME_SIZE];
};
```

A reflector class includes some basic static methods for identification and factory-based creation:

```
static const char *name();

// Entity Reflector class
static FEntityReflectorClass* create(const char *className,
    bool unrealToPlugin, bool pluginToUnreal);

// Interaction Reflector class
static FInteractionReflectorClass* create(const char *className,
    bool unrealToPlugin, bool pluginToUnreal);
```

It also includes basic instance methods for identifying an actor's class (actors of this class will be handled by the instance) and what direction is enabled (Unreal-to-MakAdapter or MakAdapter-to-Unreal):

```
virtual bool pluginToUnreal();
virtual bool unrealToPlugin();

virtual void setClassName(const char *name);
virtual const char *className();

virtual bool match(const char *className);
```

The Entity Reflector Class

An entity reflector class is designed to handle entities and actors that persist for a fixed amount of time in the simulation, like vehicles, infantry, and so on. Some type of munitions can fit in this category, such as missiles, rockets, flying bombs, minefields, and so on. An entity reflector class has member functions to manage reflected actors:

```
virtual bool createActor(ULevel *level, FCreateData *data);
virtual bool updateActor(ULevel *level, FUpdateData *data);
virtual bool removeActor(ULevel *level, FRemoveData *data);
```

It also has member functions to process and access Unreal actors based on signals or requests coming from *MakAdapter*:

```
virtual bool getPostCreateActor(AActor *actor, FCreateData *data);
virtual bool getPreRemoveActor(AActor *actor, FRemoveData *data);
virtual bool getStateActor(ULevel *level, FUpdateData *data);
virtual bool damageActor(ULevel *level, FDamageData *data);
```

The Interaction Reflector Class

An interaction reflector class reflects asynchronous interactions that occur in remote exercises. They do not persist and they do not update. The reflector permits interactions to be matched with one or more actions in the Unreal side, for example, play a sound, run an animation, or affect an entity state. The UPI includes reflector classes to reflect from or to Unreal fire and detonation interactions. The functions to send interactions from Unreal to HLA or DIS are:

```
virtual bool getFireInteractionData(AActor *munition, AActor *target,
    FVector location, FFireData *data);
virtual bool getDetonationInteractionData(AActor *munition,
    AActor *target, FVector location, FDetonationData *data);
```

The functions to reflect interactions from HLA or DIS to Unreal are:

```
virtual bool reflectFireInteraction(ULevel *level, FFireData *data);
virtual bool reflectDetonationInteraction(ULevel *level,
    FDetonationData *data);
```

Currently, only small arms are implemented in Unreal. A new effect, called RemoteRifleFireEffect, must be added to the WarEffects project. The effect is in *unreal-Sources\WarEffects\Classes\RemoteRifleEffect.uc*. This class could also be used for other types of fire and detonations classes already in Unreal, or you could implement similar new effects.

4.2.2. Factories

FPluginMakAdapter has a factory in which reflector creators are registered and instantiated:

```
virtual void registerEntityCreator(FEntityReflectorClassCreatorfn f,
    const char *name);
virtual void removeEntityCreator(FEntityReflectorClassCreatorfn f,
    const char *name);
virtual FEntityReflectorClass *entityReflectorCreate(const char *name,
    const char *className, bool unrealToPlugin, bool pluginToUnreal);

virtual void
    registerInteractionCreator(FInteractionReflectorClassCreatorfn f,
    const char *name);
virtual void removeInteractionCreator(
    FInteractionReflectorClassCreatorfn f, const char *name);
virtual FInteractionReflectorClass *interactionReflectorCreate(
    const char *name, const char *className, bool unrealToPlugin,
    bool pluginToUnreal);
```

4.3. How FPluginMakAdapter Works

FPluginMakAdapter:

1. Looks up the actor's availability.
2. Searches the actor's reflector based on its class name.
3. Calls the corresponding service from the reflector.
4. Sends the results to *MakAdapter*.

4.3.1. Connecting FPluginMakAdapter to MakAdapter - Exported Functions

Although *FPluginMakAdapter* connects the Unreal Engine and *MakAdapter*, memory model restrictions prevent connections by passing objects or object pointers to call their member functions. Therefore, *MakAdapter* exports a set of functions to enable communications with *FPluginMakAdapter*, passing data using class structures (with no virtual definitions). The API is published as exported functions (*exportedFunctions.h*):

```
// Public functions available from MakAdapter to be used in
// modules like FPluginMakAdapter.

// Callback types. FPluginMakAdapter sets callbacks to enable
// its use from inside MakAdapter
typedef bool (*DtCreateActorCb)(void *createData, void *data);
typedef bool (*DtUpdateActorCb)(void *updateData, void *data);
typedef bool (*DtGetStateActorCb)(void *updateData, void *data);
typedef void (*DtRemoveActorCb)(void *removeData, void *data);
typedef void (*DtDamageActorCb)(void *damageData, void *data);
typedef void (*DtReflectFireInteractionCb)(void *fireData, void *data);
typedef void (*DtReflectDetonationInteractionCb)(void *detonationData,
void *data);
typedef bool (*DtRegisterReflectorClassCb)(void *reflectorData, void
*data);

// Administrative functions to start/stop/init MAKAdapter
bool __declspec(dllexport) DtUStartAdapter();
bool __declspec(dllexport) DtUInitAdapter(const char *path, const char
*configFile);
void __declspec(dllexport) DtUShutdownAdapter();
bool __declspec(dllexport) DtURegisterReflectorClasses();
bool __declspec(dllexport) DtURegisterReflectorClasses(const char
*scenarioName);

// Tick function to give the control to MakAdapter, from this point
// MakAdapter behaves as a master and FPluginMakAdapter as slave
void __declspec(dllexport) DtUTick(double dt);

// Public functions to send asynchronous events to MakAdapter
void __declspec(dllexport) DtUCreateEntity(void *createData);
void __declspec(dllexport) DtURemoveEntity(void *removeData);
*void __declspec(dllexport) DtUSetDataEntity(void *setData);
void __declspec(dllexport) DtURequestDataEntity(void *requestData);
*void __declspec(dllexport) DtUCreateFireInteraction(void *fireData);
void __declspec(dllexport) DtUCreateDetonationInteraction(void
*detonationData);
```

```
// Callback subscriber functions
void __declspec(dllexport) DtUSetCreateActorCallback(DtCreateActorCb cb,
void *data);
void __declspec(dllexport) DtUSetUpdateActorCallback(DtUpdateActorCb cb,
void *data);
void __declspec(dllexport) DtUSetGetStateActorCallback(
DtGetStateActorCb cb, void *data);
void __declspec(dllexport) DtUSetRemoveActorCallback(DtRemoveActorCb cb,
void *data);
void __declspec(dllexport) DtUSetDamageActorCallback(DtDamageActorCb cb,
void *data);
void __declspec(dllexport) DtUSetReflectFireInteractionCallback(
DtReflectFireInteractionCb cb, void *data);
void __declspec(dllexport) DtUSetReflectDetonationInteractionCallback(
DtReflectDetonationInteractionCb cb, void *data);
void __declspec(dllexport) DtUSetRegisterReflectorClassCallback(
DtRegisterReflectorClassCb cb, void *data);
```

4.3.2. Starting *MakAdapter*

You start *MakAdapter* with a command line option `–UsesMakAdapter`. The option must be specified for the server when you run in client-server mode, or for each stand-alone session in stand-alone mode. To resolve the option, `UGameEngine::Init()` must include the following code:

```
void UGameEngine::Init()
{
    guard(UGameEngine::Init);
    check(sizeof(*this)==GetClass()->GetPropertiesSize());

    // Call base.
    UEngine::Init();

    // UPI: starting Remote Control
    initRemoteControl();

    . . .

    //=====
    // UPI: Remote Control: subscribing to use MakAdapter
    if (ParseParam(appCmdLine(),TEXT("UsesMakAdapter")))
        FPluginMakAdapter::start();

    //=====
    // UPI: Remote Control: subscription of plugins
    startPlugins();

    debugf( NAME_Init, TEXT("Game engine initialized") );

    unguard;
}
```

4.3.3. Configuring *MakAdapter*

To configure *MakAdapter*—to—*FPluginMakAdapter* behavior, you can edit two configuration files, an entity map file and a configuration file. These files map the HLA or DIS entities to Unreal actors and configure the connection. The files use the MÄK Technologies Lisp (MTL) syntax. See Appendix A, *MÄK Technologies Lisp* for a description of the MTL language.

The MÄK Adapter assumes that the entity map file is named for the VR-Forces scenario being loaded. For example, if you load a scenario named *myScenario*, the MÄK Adapter tries to load *myScenarioEntityMap.mtl*. This allows you to create entity map files that are specific to each scenario.

By default, the MÄK Adapter assumes that the configuration file is named *MakAdapter.mtl*. It also assumes that all data required by the adapter is in subdirectories of *unreal_game\System\MakData*. You can change the path and configuration file as follows:

```
virtual void setPath(const char *path);
virtual void setConfigFile(const char *name);
```

The MÄK Adapter searches for the following files in the path:

```
configuration_file.mtl
scenarios
├── scenario
│   ├── scenario.mtl
│   └── scenarioEntityMap.mtl
```

For example:

```
System
├── MakData
│   ├── MakAdapter.mtl
│   ├── scenarios
│   │   ├── Rocktown
│   │   │   ├── Rocktown.mtl
│   │   │   └── RocktownEntityMap.mtl
│   │   └── NTC
│   │       ├── NTC.mtl
│   │       └── NTCEntityMap.mtl
```

The *MakAdapter.mtl* Configuration File

This file contains parameters to configure the connection to DIS or HLA and basic simulation parameters.

```
;;=====
;;
;; DIS specific parameters
;;
;;=====

;; Default destination IP address for outgoing Pdus
(setqb destAddrString "0.0.0.0")

;; Port number for the IP connection to the exercise
(setqb disPort 3000)

;; The receive:send buffer size (this reduces the dropping of Pdus in
;; UDP) Here a negative value indicates that the system default should
;; be used; while a positive value is actually used as the size of the
;; respective buffer
(setqb recvBufferSize -1)
(setqb sendBufferSize -1)

;; "Use Asynchronous IO" flag
(setqb useAsyncIO 0)

;;=====
;;
;; HLA specific parameters
;;
;;=====

;; Federation name (may also serve as .fed file name)
(setqb execName "VR-Link")

;; Federate name
(setqb fedName "VR-Link Unreal")

;; Path to the .fed file to use. If none is specified, the federation
;; name is used also as the name of the .fed file (with .fed extension
;; appended)
(setqb fedFileName "")

;; Path to a shared library file defining a custom fom-mapper, and the
;; initialization data to be passed to the library upon initialization
(setqb fomMapperLib "")
(setqb fomMapperInitData "")

;; Version of RPR FOM to use. Allowed values are
;; 0.5 - RPR FOM 1.0, Draft 5
;; 0.7 - RPR FOM 1.0, Draft 7
;; 0.8 - RPR FOM 1.0, Draft 8
;; 1.0 - RPR FOM 1.0, Official (final) draft
;; 2.0006 - RPR FOM 2.0, Draft 6
;; 2.0014 - RPR FOM 2.0, Draft 14
(setqb rprFomVersion 1.0)

;;=====
;;
```

```
;; Protocol independent parameters
;;
;;=====

;; Diagnostic Configuration
(setqb DtNotifyLevel 2)

;; Exercise id
(setqb exerciseId 1)

;; Site id
(setqb siteId 8)

;; Host (application) id
(setqb appNum 2)

;; seconds
(setqb timeOutInterval 12.0)

;; seconds
(setqb timeThreshold 5.0)

;; minimum linear change to generate another PDU (meters)
(setqb translationThreshold 1.0)

;; minimum rotational change to generate another PDU (radians)
(setqb rotationThreshold 0.95492965855137201461330258023509)

;; Duration of one frame-cycle (in seconds)
(setqb dt 0.050000)

;; sleep time for the adapter (milliseconds)
(setqb sleepDeltaTime 30)

;; dead Reckoning Algorithm used in remote listeners.
;; Affects PDU's frequency
;;   DtDrOther   =      0,
;;   DtDrStatic  =      1,
;;   DtDrDrmFpw  =      2,
;;   DtDrDrmRpw  =      3,
;;   DtDrDrmRvw  =      4,
;;   DtDrDrmFvw  =      5,
;;   DtDrDrmFpb  =      6,
;;   DtDrDrmRpb  =      7,
;;   DtDrDrmRvb  =      8,
;;   DtDrDrmFvb  =      9
(setqb deadReckonAlgorithm 4)

;; 1 turn on smoothing
;; 0 turn off smoothing
(setqb smoothing 1)
```

The Scenario Directory

Each scenario has a directory that has a scenario description file named after the scenario and an entity map file. The scenario file has information about the difficulty level of the scenario, the location of the terrain and the entity map file to be used.

```
;; =====
;;
;;      Rocktown scenario description
;;
;; =====

(scenario-description

;; =====
;; Difficulty level to take damage. Scale 1 (minimum) to 10 (maximum)
;; =====

(difficulty-level 1)

;; =====
;; Terrain reference describes the properties to "translate" from this
;; terrain into a DIS/HLA simulation
;; The MAK reference system is right handed topographic (NED => y,x,-z)
;; The Unreal reference system is left handed (ESU => x,-y,z)
;; =====

(terrain-reference
;; database extent in topographic (NED)
  (extent-min -200.0 -200.0 -200.0)

  (extent-max 200.0 200.0 200.0)
  ;; =====
  ;; Latitude and longitude of the origin of the Topographic frame (in
  ;; degrees) you can use the format:
  ;; <dir deg min sec>
  ;; with dir = { DtEast, DtWest, DtNorth, DtSouth }
  ;; =====

  ;; (latitude DtNorth 35.0 23.0 36.096)
  ;; (longitude DtEast 116.0 54.0 0.0)
  ;; =====
  ;; or you can use the format:
  ;; number
  ;; =====

  (latitude 35.39336)
  (longitude 116.9)
)

;; =====
;; Entity map description
;; =====

(entity-map "rocktownEntityMap.mtl")
)
```

The Entity Map Configuration File

This file describes the mappings between entities and actors and interactions. The following example (from *rocktownEntityMap.mtl*) shows how to map an entity to an actor and how to map an interaction:

```
(entity-map

;;-----
;; global unit transformation. Applies to all entities and interactions
;;-----
  (group
    (group-entry
      (name "Global transformation")
      (unreal-to-mak
        (active-entry true)
        (position-transformation
          0.0    0.01    0.0
          -0.01   0.0    0.0
          0.0    0.0    -0.01
        )
        ;; PI/32768.0 from unreal units to radians
        (orientation-transformation
          0.0    0.0000958737992428525    0.0
          0.0000958737992428525    0.0    0.0
          0.0    0.0    0.0000958737992428525
        )
        ;; dead-reckoning algorithm
        ;; (dead-reckoning 4)
      )
      (mak-to-unreal
        (active-entry true)
        (position-transformation
          0.0    -100.0    0.0
          100.0    0.0    0.0
          0.0    0.0    -100.0
        )
        (orientation-transformation
          0.0    -10430.3783504705    0.0
          10430.3783504705    0.0    0.0
          0.0    0.0    -10430.3783504705
        )
        ;; smoothing
        ;; (smoothing true)
      )
    )
  )

;;-----
;; common values for infantry entries.
;;-----
  (group
    (group-entry
      (name "Infantry group")
      (mak-reflector-class "di-mak-reflector")
      (unreal-reflector-class "infantry-light-unreal-reflector")
      (unreal-to-mak
        (position-offset 30.0 -5.0 3.33)
        (orientation-offset -1.57079632679 0.0 0.0)
      )
    )
  )
)
```

```
(mak-to-unreal
  (position-offset -500.0 -3000.0 333.0)
  ;; 32768.0/2.0 => PI/2 in Unreal units
  (orientation-offset 0.0 16384.0 0.0)
)
)

(entity-entry
  (name "DI friendly")
  (mak-pattern "3:1:225:1:-1:-1:-1")
  (mak-type "3:1:225:1:1:0:0")
  (unreal-pattern "WarCOGSoldier")
  (unreal-class "WarClassLight.WarCOGSoldier")
  (unreal-to-mak
    (force-type DtForceFriendly)
  )
  (mak-to-unreal
    (force-type DtForceFriendly)
  )
)
)
)
)
```

Parameters

- ♦ **group** defines a group of entries with common parameters. Its first member must be a **group-entry**, which defines the common values for the group members. You can nest groups in parent groups. Child groups inherit the common values of the parent. They can redefine parent values for the members of the child group.
- ♦ **group-entry** defines common values for group members.
- ♦ **entity-entry** describes an entity association (entity-to-actor).
- ♦ **mak-pattern** specifies the enumeration of HLA or DIS entities that must be mapped with this entry.
- ♦ **mak-type** specifies the HLA or DIS entity type to be created when reflecting an actor into HLA or DIS.
- ♦ **mak-reflector-class** specifies the reflector's name in *MakAdapter*.
- ♦ **unreal-pattern** specifies actor's class name mapped to HLA or DIS using this entry.
- ♦ **unreal-class** specifies the actor's class to be created when reflecting an HLA or DIS entity into Unreal.
- ♦ **unreal-reflector-class** specifies the reflector's name in *FPluginMakAdapter*.
- ♦ **unreal-to-mak** specifies parameters associated with a conversion from Unreal to HLA or DIS.
- ♦ **mak-to-Unreal** specifies parameters associated with a conversion from HLA or DIS to Unreal.
- ♦ **active-entry** specifies a Boolean indicating if the entry (unreal-to-mak/mak-to-unreal) is active. If **false**, no reflection is done in that direction.
- ♦ **force-type** specifies type of force of the entity/actor (friendly, enemy, neutral, other, and so on).

- ♦ **position-transformation/orientation-transformation** specifies how to transform the associated axis from HLA/ DIS to Unreal. The position is expressed in topographic (NED) coordinates. The orientation is expressed as a TaitBryan (psi, theta, phi).
- ♦ **position-offset/orientation-offset** specifies additional units to add to each new axis.

With these entries, a complete matrix transformation is expressed for location and orientation.

- ♦ **smoothing** enables or disables smoothing for an entity.
- ♦ **dead-reckoning** specifies the dead-reckoning algorithm for an entity.

The following example (from *rocktownEntityMap.mtl*) shows how to map an interaction:

```
;;-----  
;; Common values for fire/detonation interaction entries  
;;-----  
(group  
  (group-entry  
    (name "Fire/detonation Interactions")  
    (mak-reflector-class "basic-mak-reflector")  
    (unreal-reflector-class "basic-unreal-reflector")  
    (unreal-fire-class "WarEffects.RemoteRifleFireEffect")  
    (unreal-detonation-class "WarEffects.RemoteRifleFireEffect")  
    (unreal-to-mak  
      (force-type DtForceFriendly)  
      (position-offset 30.0 -5.0 -1.0)  
      ;; PI/2 1.57079632679  
      (orientation-offset -1.57079632679 0.0 0.0)  
    )  
    (mak-to-unreal  
      (force-type DtForceOpposing)  
      ;; 32768.0/PI from radians to Unreal units  
      (orientation-offset 0.0 16384.0 0.0)  
    )  
  )  
  
  (interaction-entry  
    (name "rifle")  
    (mak-pattern "2:8:-1:2:2:-1:-1")  
    (mak-type "2:8:225:2:2:0:0")  
    (unreal-pattern "AssaultRifleAmmo")  
    (unreal-class "WarClassLight.AssaultRifleAmmo")  
    (mak-to-unreal  
      (damage-distance 2.0)  
      (damage-amount 10)  
      (position-offset -520.0 -3010.0 250.0)  
    )  
  )  
)
```

The parameters have the same meanings as those used for entity mapping, plus the following additional parameters:

- ♦ **unreal-fire-class** specifies the effect class name to reflect an HLA or DIS fire interaction.
- ♦ **unreal-detonation-class** specifies the effect class name to reflect an HLA or DIS detonation interaction.

If **unreal-fire-class** or **unreal-detonation-class** is not available, no fires or detonations are reflected in Unreal.

- ♦ **damage-distance** defines the munitions range radius in meters.
- ♦ **damage-amount** specifies how much an unreal-native actor will be affected for HLA or DIS interactions. Damage is measured between 0 – 100.

If the values for **damage-distance** and **damage-amount** are 0 (zero), detonations do not affect Unreal actors. In this case, the interaction reflector must produce the desired effect.

Offset Transformations

The transformation offsets in the entity map file compensate for correlation problems between terrains from different sources. Transformations are calculated as:

$$\text{NewPosition} = \text{Matrix} * \text{OldPosition} + \text{Offset}$$

where **Matrix** includes the scale. The **Offset** value must be in the units and coordinates used by the **NewPosition**.

As an example, consider the values in the sample entity map file on the previous pages:

```
;; unreal-to-mak transformation
(position-transformation
  0.0      0.01      0.0
 -0.01     0.0      0.0
  0.0      0.0     -0.01

;; mak-to-unreal transformation
(position-transformation
  0.0     -100.0     0.0
 100.0     0.0      0.0
  0.0      0.0    -100.0
```

Given the **OldPosition** (x, y, z), and an offset of 0, the unreal-to-mak transformation is:

$$\text{NewPos} = (y*0.01, -x*0.01, -z *0.01)$$

The mak-to-unreal transformation is:

$$(-(-x*0.01) * 100, y*0.01 * 100, -(-z *0.01)) => x, y, z$$

4.4. Terrain Correlation

Terrain correlation is done using the transformation described for each entity in the entity map file. The transformation describes how to convert from Unreal coordinates to from topographic coordinates and vice versa. The *MaKAdapter.mtl* configuration file has the reference longitude and latitude for the terrain in HLA or DIS. The entities in Unreal are placed using this point as origin and using the transformation to calculate an offset.

The same algorithm applies to conversions from HLA or DIS to Unreal. It calculates the topologic position using the same reference latitude and longitude and it converts the coordinate using the transformation described in the map file and uses it to place the entity. A similar mechanism is used for orientation. For interactions, the mapping is done using a topographic coordinate system, but it is the same concept.



Integrating Game-Link into the Unreal Engine

This chapter describes the changes you must make to the Unreal Engine to integrate it with Game-Link.

Introduction	5- 2
Copying Game-Link Classes to the Unreal Project.....	5- 2
Editing Unreal Engine Files	5- 3
Editing AActor	5- 3
Editing APawn	5- 4
Editing ULevel	5- 6
Extending UGameEngine.....	5- 7
Setting Compile Options	5- 9
Linking with the MakAdapter	5- 10

5.1. Introduction

The Game-Link distribution includes Game-Link-specific files and edited versions of Unreal Engine source and Engine, versions 2.0 (drops 2226, and 2227), and 2.5 (drops 3236 have not edited your Unreal files, you can simply replace them provided with Game-Link. If you have modified any of the files in your Unreal Engine drop, you must edit them by hand to add the Game-Link code.

The basic procedure for integrating Game-Link with Unreal Engine is:

1. Copy Game-Link files to the Unreal Engine directory tree.
2. Copy edited Unreal Engine files to the Unreal Engine directory following Unreal Engine script and C++ source files as required:
 - *AActor*
 - *APawn*
 - Vehicle classes
 - Munitions classes
 - *ULevel*
 - *UGameEngine*
3. Specify compile options.
4. Specify include directories.
5. Specify build dependencies.

5.2. Copying Game-Link Classes to the Unreal Project

Copy the files in the following directories to the Engine project in

- ♦ *install_dir\newEngineFiles\Engine\Src*
- ♦ *install_dir\newEngineFiles\Engine\Inc*

All new files are named *F** or *RemoteControl**. You can copy them into the Unreal source code or reference them directly from *install_dir\newEngineFiles*. Copy the files as follows:

1. Add the files *install_dir\newEngineFiles\Engine\Src\F** to the *Engine\Src* project.
2. Add the files *install_dir\newEngineFiles\Engine\Inc\F** to the *Engine\Inc* project.
3. Add the files *install_dir\newEngineFiles\Engine\Src\RemoteControl** to the *Engine\Src* project.
4. Add the files *install_dir\newEngineFiles\Engine\Inc\RemoteControl** to the *Engine\Inc* project.
5. Verify that the *.cpp* files are not using precompiled headers.

5.3. Editing Unreal Engine Files

This section explains how to edit the Unreal Engine scripts and Game-Link. If you have not previously edited a given Unreal Engine file just copy the edited version from the Game-Link distribution to the Unreal Engine directory. You only need to edit files edited for some other purpose (so that you do not lose your

5.3.1. Editing *AActor*

You must edit the *AActor* class in C++ and Unreal Script.

In *Actor.uc*:

1. Declare the following variables:

```
//  
// UPI: variables to track actor's origin and to determine if  
// must be emitted
```

```
var          bool          bIsLocal;  
var          bool          bMustReflect;
```

2. Declare the following functions:

```
//  
// UPI: Functions to generate fire/detonation interaction  
  
// Functions to be called by munitions, reflecting fire  
native final function bool ReflectFireInteraction(actor target,  
    vector HitLocation);  
native final function bool ReflectDetonationInteraction(actor vector  
    HitLocation);  
native final function bool ReflectFireInteractionWithResult(actor vector  
    HitLocation, int Result);  
native final function bool actor target, vector HitLocation, int Result);  
  
// UPI: event called from remoteControl's plugins when they  
// remote actor to set up additional components like inventory,  
// bones, and so on.  
event RemoteActorPostCreate()  
{  
}
```

3. Run UCC make before you make any changes in C++.

In C++:

1. Define the functions for signal generation by including the *AActor.h*:

```
//UPI: functions to generate signals about creation, deletion and  
#include "RemoteControlAActor.h"
```

2. Edit the `AActor::Tick()` function to generate the update signal and edit the `AActor::Destroy()` function to generate the destroy signal, as follows:

```
void AActor::Destroy()
{
    guard(AActor::Destroy);
    // UPI: generates remove signal into the Remote Control
    remoteControlRemove();

    . . .
}

UBOOL AActor::Tick( FLOAT DeltaSeconds, ELevelTick TickType )
{
    guard(AActor::Tick);

    . . .

    TickSpecial(DeltaSeconds); // perform any tick functions unique to an
                                // actor subclass
    // UPI: generating update signal to the Remote Control
    remoteControlTick(DeltaSeconds);
    return 1;
    unguard;
}
```

5.3.2. Editing *APawn*

APawn must define the signals in C++, to implement them and connect them with the Remote Control. To include these redefinitions, add the *APawn.h*:

```
// UPI: functions to generate signals about creation, deletion
// updates. Redefines those in remoteControlAActor.h
#include "RemoteControlAPawn.h"
```

Modifying *APawn* Subclasses

You can redefine any of the functions in *APawn* subclasses. In particular, the `RemoteActorPostCreate()` event must be modified to complete the creation process for actors. For example, in *KCar* (*Kcar.uc*), `RemoteActorPostCreate()` makes the vehicle structure rigid to enable moving without driving:

```
event RemoteActorPostCreate()
{
    frontLeft.SetBase(self);
    frontRight.SetBase(self);
    rearLeft.SetBase(self);
    rearRight.SetBase(self);
}
```

The directories *unrealSources\WarClassLight\Classes* and *unrealSources\Vehicles\Classes* have the following examples of *APawn* subclasses that redefine events:

- ♦ *CogLightInfantry.uc*
- ♦ *GeistLightInfantry.uc*
- ♦ *Kcar.uc*.

Modifying Vehicle Classes

At a minimum, you must edit the basic vehicle classes if they will be used for remote vehicles.

KCar must add a `RemoteActorPostCreate()` redefinition to make the vehicle structure rigid, attaching the wheels to the body. See the example in [“Modifying APawn Subclasses,”](#) on .

KVehicle and *SVehicle* must check their origin in the `tryToDrive()` function. Add a check to `bIsLocal` before setting the driver.

```
if (bIsLocal && (Driver == None) . . .
```

Editing Munitions Classes

If you want to send signals for fires or detonations, in munitions classes, edit the `processTraceHit()` function to call `reflectFireInteraction()` or `reflectDetonationInteraction()`, as in the following example:

```
function ProcessTraceHit(Weapon W, Actor Other, Vector Vector HitNormal,
    Vector X, Vector Y, Vector Z)
{
    AmmoAmount -= 1;
    // UPI: generating fire/detonation signals
    ReflectFireInteraction(Other, HitLocation);
    ReflectDetonationInteraction(Other, HitLocation);
    . . .
}
```

The following example shows how to calculate the damage type:

```
function ProcessTraceHit(Weapon W, Actor Other, Vector Vector HitNormal,
    Vector X, Vector Y, Vector Z)
{
    AmmoAmount -= 1;

    // calculate damage to be published
    damage = ....

    // UPI: generating fire/detonation signals
    ReflectFireInteractionWithResult(Other, HitLocation, damage);
    ReflectDetonationInteractionWithResult(Other, HitLocation,
    . . .
}
```

The directory `unrealSources\WarClassLight\Classes` has the following examples of edited munitions classes:

- ♦ *AssaultRifleAmmo.uc*
- ♦ *AssaultRifleAmmo_Exploding.uc*
- ♦ *CogP9Ammo.uc*.

5.3.3. Editing *ULevel*

In *ULevel*, you must edit the actor's creator function to track origin generate creation signals. The level needs a new function to look failing if the actor is not found. (For more information, see [“Accessing Actors,”](#) on .)

Editing the `SpawnActor()` Function

Edit the `SpawnActor()` function to enable origin tracking for actors and to generate the creator signal.

Define the `bLocal` parameter with default value true. (For more information, see [“Specifying An Actor's Origin,”](#) on .)

```
// UPI: bLocal additional parameter to track origin
virtual AActor* SpawnActor( UClass* Class, FName InName=NAME_None,
    FVector Location=FVector(0,0,0), FRotator AActor* Template=NULL, UBOOL
    bNoCollisionFail=0, UBOOL AActor* Owner=NULL, APawn* Instigator=NULL,
    UBOOL bLocal=1 );
```

Edit the function to pass the `bLocal` value to the actor and generate the creation signal.

```
AActor* ULevel::SpawnActor
(
    UClass*      Class,
    FName        InName,
    FVector      Location,
    FRotator     Rotation,
    AActor*      Template,
    UBOOL        bNoCollisionFail,
    UBOOL        bRemoteOwned,
    AActor*      Owner,
    APawn*       Instigator,
    . . .
    UBOOL        bLocal // UPI: extension to track origin
)
{
    guard(ULevel::SpawnActor);

    . . .

    if ( !bRemoteOwned )
        Actor->eventPostNetBeginPlay();

    // UPI: sends the actor to the Remote Control, generating a
    // signal
    Actor->remoteControlPostCreate(bLocal);

    return Actor;
    unguardf(( TEXT("(%s)", Class->GetName() ));
}
```

Add the isAValidActor() Function

The isAValidActor() function has been added to *ULevel* to enable searching for actors without generating errors if they are missing. To define this code at the end of the *ULevel* class definition (*UnLevel.h*):

```
class ENGINE_API ULevel : public ULevelBase
{
    DECLARE_CLASS(ULevel, ULevelBase, 0, Engine)
    . . .

    // UPI: defining new function to search actors
    #include "RemoteControlUnLevel.h"
};
```

5.3.4. Extending UGameEngine

The *UGameEngine* class is extended to create the Remote Control module and to connect the Remote Control to the current level. To define the new following to *UnGame.h*:

```
class ENGINE_API UGameEngine : public UEngine
{
    DECLARE_CLASS(UGameEngine, UEngine, CLASS_Config|CLASS_Transient, Engine)

    // UPI: including Remote Control related function definitions
    #include "RemoteControlUnGame.h"
};
```

Edit the functions *UnGameEngine::Init()*, *UnGameEngine::Destroy()*, and *UnGameEngine::LoadMap()* in *unrealSources\Engine\Src\UnGame.cpp*. (You may want to copy the new code from the edited version provided with the

Usually, the *LoadMap()* function changes for each game when and where the main level is loaded.

```
void UGameEngine::Init()
{
    guard(UGameEngine::Init);
    check(sizeof(*this) == GetClass()->GetPropertiesSize());

    // Call base.
    UEngine::Init();

    // UPI: starting Remote Control
    initRemoteControl();

    . . .

    //=====
    // UPI: Remote Control: subscription of plugins
    startPlugins();

    debugf( NAME_Init, TEXT("Game engine initialized") );

    unguard;
}
```

```
void UGameEngine::Destroy()
{
    guard(UGameEngine::Destroy);

    // UPI: stopping Remote Control. The plugins will be deleted
    stopRemoteControl();

    . . .
}

ULevel* UGameEngine::LoadMap( const FURL& URL, UPendingLevel*
    const TMap<FString,FString>* TravelInfo, FString& Error )
{
    guard(UGameEngine::LoadMap);
    Error = TEXT("");
    GInitRunaway();

    // Remember current level's stack level.
    INT SavedHubStackLevel = GLevel ? GLevel->GetLevelInfo()-: 0;

    // Get network package map.
    UPackageMap* PackageMap = NULL;
    if( Pending )
        PackageMap = Pending->GetDriver()->ServerConnection-

    // Verify that we can load all packages we need.
    UObject* MapParent = NULL;
    guard(VerifyPackages);
    try
    {
        . . .
    }
    #if UNICODE
    catch( TCHAR* CatchError )
    #else
    catch( char* CatchError )
    #endif
    {
        . . .
    }
    unguard;

    // Display loading screen.
    guard>LoadingScreen);
    if( Client && Client->Viewports.Num() && GLevel &&
        !URL.HasOption(TEXT("quiet")) )
    {
        . . .
    }
    unguard;

    // Notify of the level change, before we dissociate Viewport
    guard(NotifyLevelChange);
    if( GLevel )
        NotifyLevelChange();
    unguard;

    // Dissociate Viewport actors.
    guard(DissociateViewports);
    if( Client )
```

```

{
    . . .
}
unguard;

// Clean up game state.
guard(ExitLevel);
if( GLevel )
{
    . . .
}
unguard;

// Load the level and all objects under it, using the proper
guard(LoadLevel);

// UPI: disconnecting old level and connecting the new one
disconnectRemoteControlFromLevel(GLevel);
char scenarioFile[1025];
appToAnsi(*URL.Map, scenarioFile);
GLevel = LoadObject<ULevel>( MapParent, TEXT("MyLevel"), LOAD_NoFail,
NULL );
connectRemoteControlToLevel(GLevel, scenarioFile);
unguard;
. . .
}

```

Finally, pass the tick to the Remote Control. To do this, in `UGameEngine::Tick()`, adding the call to the tick:

```

void UGameEngine::Tick( FLOAT DeltaSeconds )
{
    guard(UGameEngine::Tick);

    . . .

    ClientCycles=LocalClientCycles;
    unguard;

    // UPI: Remote Control ticks passing it down to the plugins
    FRemoteControl::remoteControl()->tick(DeltaSeconds);

    unlock(LocalTickCycles);
    TickCycles=LocalTickCycles;
    GTicks++;
    unguard;
}

```

5.4. Setting Compile Options

In your development environment, add whichever of the following pre-is appropriate for the engine you are using:

- ♦ `__DEF_UE25` (For Unreal Engine 2.5)
- ♦ `__DEF_UE20` (For Unreal Engine 2.0).

5.5. Linking with the MakAdapter

To link with *GameLink.dll*, in your development environment, add the following to your Engine project:

- ♦ In additional include directories, add *install_dir\Inc*
- ♦ In additional dependencies, add
install_dir\Lib\GameLinkXXX.lib

Where *GameLinkXXX.lib* could be:

- *GameLinkHLA13.lib* for HLA 1.3
- *GameLinkHLA1516.lib* for HLA 1516
- *GameLinkDIS.lib* for DIS.



MÄK Technologies Lisp

This appendix describes the MÄK Technologies Lisp (MTL) format.

MÄK Technologies Lisp (MTL) Files.....	A-2
Using Environment Variables in an MTL File	A-2
Specifying Lists of Lists.....	A-3
Using Conditional Statements	A-3

A.1. MÄK Technologies Lisp (MTL) Files

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

Most parameter entries take the format:

```
(setq parameter_name value)
```

For example:

```
(setq EnableSound 1)
```

Or:

```
(setqb parameter_name value)
```

The `setqb` syntax indicates a bound variable.

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

In most cases, the parameter names are known to the MÄK application and the MTL commands simply assign values to them. However, you can use the `setq` command to create arbitrary symbolic names or aliases that are then used in later commands. For example, in the following two commands (from the MÄK Stealth), the symbolic name `VaporTrail` is assigned to an `OpenFlight` file. Then the symbolic name is used in the `trail-map` command. The name `VaporTrail` has no intrinsic meaning to the Stealth. It is given meaning by the `setq` command, which allows it to be used later in the `trail-map` command.

```
(setq VaporTrail (list "../data/models/makTrails/Vapor-  
Trail" "VaporTrail.flt" 6.0 1.5 1.5 4.0 1.5 0.5))  
(trail-map (list 1 2 -1 0 -1 -1 -1)(list VaporTrail 0.0 0.0  
0.0))
```

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

```
(load mtl-path "params.mtl")
```

A.1.1. Using Environment Variables in an MTL File

If you want to use an environment variable in an MTL file, use the following syntax:

```
(setqb parameter_name (getenv (quote env_var)))
```

For example:

```
(setqb disDestAddr (getenv DEST_ADDR))
```

A.1.2. Specifying Lists of Lists

The example of setting a symbolic name in a previous section uses a list. You can also specify a list of lists, for example:

```
(trail-map (list 1 2 -1 -1 -1 -1 -1)
  (list vaporTrail -8.0 -3.0 0.0)
  (list vaporTrail -8.0 3.0 0.0)
)
```

A parameter entry can also have multiple individual lists:

```
(trail-map (list 1 2 -1 0 -1 -1 -1) (list VaporTrail 0.0 0.0
0.0))
```

A.1.3. Using Conditional Statements

MTL commands can be conditional. Conditional statements are used frequently when parameters are different for DIS and HLA. A simple conditional statement has the format:

```
(if (condition) (statement_to_evaluate))
```

If you have more than one statement to evaluate, use a sequence block:

```
;; HLA specific parameters
(if (equal HLA 1)
  (sequence
    (setqb fomMapperLib "")
    (setqb fomMapperInitData "")
    (setqb rprFomVersion 1.0)
    (setqb pathToSublistFile "sublist.mtl")
    (setqb ignoreAdvisories 0)
    (setqb fedLookahead 1.0)
  )
)
```




Index

A

- activating
 - Remote Control 3-5
- actor
 - deletion 3-8
 - getting and setting data 4-3
 - origin 3-13, 5-6
 - remote 3-11
 - searching list of searching
 - actor list 3-13
 - types 3-6
- Actor.uc 3-9, 3-11
 - editing 5-3
- adapter 3-2
- adding
 - new license file 2-7
- architecture
 - MÄK Adapter plug-in 4-2
 - reflector class 4-3
 - UPI 3-2
- AssaultRifleAmmo.uc 3-11
- asynchronous event 3-9

B

- bIsLocal parameter 3-6
- bLocal 5-6
- bLocal parameter 3-7, 3-13
- bMustReflect parameter 3-6

C

- callback function 3-15
- Callback Manager 3-2, 3-14
- class

- reflector 4-3
- CogLightInfantry.uc 3-11
- comment
 - MTL parameter A-2
- compile options 5-9
- configuration file
 - entity map 4-14
 - MakAdapter.mtl 4-11
 - RTI 2-11
- configuring
 - MakAdapter 4-10
 - system for MÄK RTI 2-11
- connecting
 - FPluginMakAdapter and MakAdapter 4-8
 - Remote Control 3-5
 - Remote Control module 3-4
- conventions
 - manual i-xi
- correlating
 - terrain 4-18
- creating
 - plug-in 3-16
- creator function
 - registering 4-7
- creator 5-6

D

- data
 - getting and setting 4-3
 - sharing between plug-in and adapter 4-3
- deleting
 - actor 3-8
- destroy signal 5-4
- DIS 1-2, 4-7
 - versions supported 1-3

- disconnecting
 - Remote Control 3-5
- Distributed Interactive Simulation 1-2
- DMSO RTI 2-12

E

- editing
 - Actor.uc 5-3
 - munitions classes 5-5
 - vehicle classes 5-5
- entity
 - reflector class 4-3, 4-6
- entity map file 4-14
- environment variable
 - MAKLMGRD_LICENSE_FILE 2-6
 - RTI_CONFIG 2-11

F

- factory 4-6, 4-7
- FDD file 2-11
- FED file 2-11
- file
 - FED 2-11
 - license 2-5
 - MTL A-2
 - RID 1-3, 2-11
 - rti.mtl* 2-11
 - RTI.RID* 2-11
- FLEXlm
 - directory location 2-4
 - installing 2-4
 - license file 2-5
 - license manager 2-3
 - license server name 2-5
 - multiple licenses 2-7
 - troubleshooting 2-9
- FOM 2-11
 - versions supported 1-4
- FPluginMakAdapter 3-15, 4-2, 4-8
 - connecting to MakAdapter 4-8

G

- Game-Link Plug-in Interface 1-3
- generating
 - signals 3-7
- getting

- actor data 4-3

H

- header 5-2
- High Level Architecture 1-2
- HLA 1-2, 4-7
 - versions supported 1-3
- host ID
 - getting 2-5
- host name 2-5

I

- installing
 - DMSO RTI 2-12
 - FLEXlm software 2-4
 - MAK Unreal Adapter 2-2
 - RTI 2-11
- interaction
 - reflector class 4-3, 4-7
 - signal 3-7, 3-9

K

- KCar 5-5
- Kcar.uc 5-4
- KVehicle 3-12, 5-5

L

- launching
 - plug-in 3-5
- libFedTime1516.dll 1-3
- libFedTime.dll 1-3
- librti1516.dll 1-3
- libRTI-NG.dll 1-3
- license file 2-5
 - adding new 2-7
- license server
 - name 2-5
 - running 2-7
 - status 2-8
 - stopping 2-8
 - troubleshooting 2-9
- lmdown program 2-8
- lmhostid 2-5
- lmstat program 2-8
- local

actor 3-13
local actor 3-6, 5-6

M

MÄK Adapter application 1-3, 4-2
MÄK Adapter plug-in 1-3, 4-2
 architecture 4-2
MÄK RTI 1-3
 configuring 2-11
MÄK Technologies LISP A-2
MÄK VR-Link 1-3
MakAdapter 4-2
 configuring 4-10
 connecting to FPluginMakAdapter 4-8
 source files 5-2
 starting 4-9
MakAdapter.mtl 4-11
MAKLMGRD_LICENSE_FILE environment
 variable 2-6
manual
 conventions i-xi
memory model 4-3
message, munition 3-9
module
 Remote Control 3-2
MTL file A-2
munition 4-6
munitions class
 editing 5-5

O

origin
 actor 5-6

P

parameter
 blsLocal 3-6
 bLocal 3-7, 3-13
 bMustReflect 3-6
 commenting out A-2
path
 specifying for RTI 2-11
pawn 3-8
plug-in 3-2, 3-15
 creating 3-16
 launching 3-5

Plug-in Manager 3-3, 3-15
post-create event 3-6, 3-11
pre-destroy signal 3-8
publishing data 3-6

R

radio signal 3-9
Real-Time Platform Reference FOM 1-4
reflector class 4-3, 4-6
registering
 callback function 3-15
 creator functions 4-7
remote
 actor 3-11, 3-13
remote actor 3-6, 5-6
Remote Control
 starting, stopping, and connecting to 3-5
Remote Control module 3-2, 3-14
 connecting to 3-4
RemoteControlUnActor.cpp 3-7
RemoteControlUnGame.cpp 3-4
RemoteControlUnScript.cpp 3-10
RemoteRifleFireEffect 4-7
RID file 1-3, 2-11
rid.mtl 1-3
rid.mtl file 2-11
RPR FOM 1-4
RTI 2-11
 definition of 2-11
 DMSO, installing 2-12
 versions supported 1-3
RTI_CONFIG environment variable 2-11
rtiexec 2-12
RTI.rid file 2-11
runLm program 2-8
running
 license server 2-7

S

setting
 actor data 4-3
shutting down
 license server 2-8
signal
 creator 5-6
 generating 3-7
 interaction 3-9
 pre-destroy 3-8

Index

- state and interaction 3-7
- update 3-8
- source file
 - UPI 5-2
- specifying
 - bLocal** 5-6
- starting
 - MakAdapter 4-9
 - plug-in 3-5
- state
 - signal 3-7
- status
 - license server 2-8
- stopping
 - license server 2-8
 - Remote Control 3-5
- SVehicle 3-12, 5-5
- synchronizing
 - terrain 4-18

T

- terrain
 - correlating between simulations and Unreal 4-18
- topographic coordinates 4-18
- troubleshooting
 - FLEXIm 2-9

U

- UnGame.cpp 5-7
- Unreal Engine
 - versions supported 1-4
- Unreal Plug-in Interface
 - architecture 3-2
- update signal 3-8, 5-4
- UPI 3-2
 - source and header files 5-2

V

- variable
 - environment 2-6
- vehicle
 - modifying 3-12
- vehicle class
 - editing 5-5
- version

- DIS 1-3
- HLA 1-3
- RTI 1-3
- versions
 - Unreal Engine 1-4
- VR-Link 1-3

W

- weapons fire 3-9

X-Y-Z

- XML
 - RTI configuration file 2-11
- KVehicle.uc 3-12
- SVehicle.uc 3-12



Link - Simulate - Visualize

GML-1.0-1-050519