VR-Link for Unity

Developers Guide

VT MÄK
A company of VT Systems

VR-Link for Unity

Developers Guide

VT MÄK
A company of VT Systems

For third-party license information, please see “Third Party Licenses,” on page x.

VT MÄK
150 Cambridge Park Drive, 3rd Floor
Cambridge, MA 02140 USA

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

info@mak.com

www.mak.com

Revision VRU-1.1-2-130802

Contents

Preface

Chapter 1 VR-Link for Unity SDK

Preface

This guide is a brief introduction to VR-Link for Unity to get you started. For developer documentation, please see the online API documentation.

How this Manual is Organized

The chapters are organized as follows:

Chapter 1, *VR-Link for Unity SDK*, describes the VR-Link for Unity SDK.

MÄK Products

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

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

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

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

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

- VR-Vantage™. VR-Vantage is a line of products designed to meet your simulation visualization needs. It includes three end-user applications (VR-Vantage Stealth, VR-Vantage PVD, and VR-Vantage IG), the VR-Vantage Toolkit, and VR-Vantage FreeView.
  - VR-Vantage Stealth displays a realistic, 3D view of your virtual world, a 2D plan view, and an exaggerated reality (XR) view. Together these views provide both situational awareness and the big picture of the simulated world. You can move your viewpoint to any location in the 3D world and can attach it to entities so that it moves as they do.
  - VR-Vantage IG is a configurable desktop image generator (IG) for out the window (OTW) scenes and remote camera views. It has most of the features of the Stealth, but is optimized for its IG function.

  - VR-Vantage PVD provides a 2D plan view display. It gives you the big picture of the simulated world.
  - The VR-Vantage Toolkit is a 3D visual application development toolkit. Use it to customize or extend MÄK's VR-Vantage applications, or to integrate VR-Vantage capabilities into your custom applications. VR-Vantage is built on top of OpenSceneGraph (OSG). The toolkit includes the OSG version used to build VR-Vantage.
  - VR-Vantage Free View is a terrain and 3D model viewer that introduces some of the features of VR-Vantage applications in a useful, free utility.
- MÄK Data Logger. The Data Logger, also called the Logger, can record HLA and DIS exercises and play them back for after-action review. You can play a recorded file at speeds above or below normal and can quickly jump to areas of interest. The Logger has a GUI and a text interface. The Logger API allows you to extend the Logger using plug-in modules or embed the Logger into your own application. The Logger editing features let you merge, trim, and offset Logger recordings.
- VR-Exchange™. VR-Exchange allows simulations that use incompatible communications protocols to interoperate. For example, within the HLA world, using VR-Exchange, federations using the HLA RPR FOM 1.0 can interoperate with simulations using RPR FOM 2.0, or federations using different RTIs can interoperate. VR-Exchange supports HLA, TENA, and DIS translation.
- VR-TheWorld™ Server. VR-TheWorld Server is a simple, yet powerful, web-based streaming terrain server, developed in conjunction with Pelican Mapping. Delivered with a global base map, you can also easily populate it with your own custom source data through a web-based interface. The server can be deployed on private, classified networks to provide streaming terrain data to a variety of simulation and visualization applications behind your firewall.
- VR-inTerra. VR-inTerra is a C++ API for adding terrain agility to applications. It can load, page, or stream terrain from a wide variety of formats or sources into a single, consistent run-time representation, consisting of a collision graph and vector network.

How to Contact Us

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

Telephone

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

E-mail

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

Internet

MÄK web site home page:	www.mak.com
License key requests:	www.mak.com/support/get-licenses.html
Product version and platform information:	www.mak.com/support/product-versions.html
For the free, unlicensed MÄK RTI:	www.mak.com/resources/bonus-material/cat_view/16-bonus-materials/24-mak-high-performance-rti.html
MÄK Community Forum:	www.mak.com/community-forum/1-forum.html

Post

Send postal correspondence to:	VT MÄK 150 Cambridge Park Drive, 3rd Floor Cambridge, MA, USA 02140

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

Document Conventions

This manual uses the following typographic conventions:

`Monospaced`	Indicates commands or values you enter.
`Monospaced Bold`	Indicates a key on the keyboard.
`Monospaced Italic`	Indicates command variables that you replace with appropriate values.
Blue text	A hypertext link to another location in this manual or another manual in the documentation set.
`{ }`	Indicates required arguments.
`[ ]`	Indicates optional arguments.
|	Separates options in a command where only one option may be chosen at a time.
`(	)`	In command syntax, indicates equivalent alternatives for a command-line option, for example, `(-h | --help)`.
/	Indicates a directory. Since MÄK products run on both Linux and Windows PC platforms, we use the / (slash) for generic discussions of pathnames. If you are running on a PC, substitute a \ (backslash) when you type pathnames.
Italic	Indicates a file name, pathname, or a class name.
sans Serif	Indicates a parameter or argument.
➤	Indicates a one-step procedure.
Menu → Option	Indicates a menu choice. For example, an instruction to select the Save option from the File menu appears as: Choose **File → Save**.
	Click the icon to run a tutorial video in the default browser.
	Indicates supplemental or clarifying information.
	Indicates additional information that you must observe to ensure the success of a procedure or other task.

Directory names are preceded with dot and slash characters that show their position with respect to the VR-Link for Unity home directory. For example, the directory *vrl1.1/doc* appears in the text as *./doc*.

Mouse Button Naming Conventions

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

Third Party Licenses

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

Boost License

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

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

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

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

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

libXML and libICONV

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

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

1

VR-Link for Unity SDK

VR-Link for Unity is a Unity Asset package that provides DIS/HLA interoperability for Unity.

1.1. Introduction

This guide will provide information on how to develop plug-ins to extend the behavior or VR-Link for Unity through its C++ SDK. It assumes that the reader is familiar with distributed networking concepts and VR-Link (the C++ API). This document will also reference the example Laser Designator plug-in that is included with the SDK to show how all the concepts covered in this guide are used to extend VR-Link for Unity.

1.1.1. Build Requirements

VR-Link for Unity is built with Microsoft Visual Studio 2010 (VC 10). It uses the 32 bit compiler. It is built against VR-Link 4.0.8 and uses the 32-bit libraries.

1.1.2. Debugging VR-Link for Unity Plug-ins

Since Unity does not release debug versions of its application, it is not possible to build and run a debug version of VR-Link for Unity. The real issue is that you cannot mix release and debug runtime libraries within an application. It is possible however to create "release with debug information" DLLs that can be debugged in the normal manner. Simply create a copy of a release build configuration and turn on generation of debug symbols and turn off any optimization while still linking to the release runtime libraries in the project settings. This will allow you to break and step through your plug-ins.

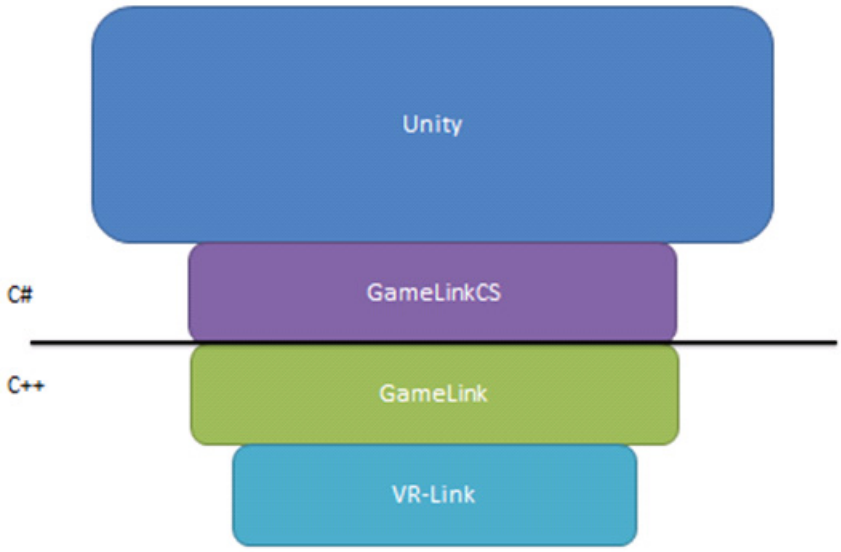
1.2. Architectural Overview

VR-Link for Unity is composed of several layers. The bulk of the heavy lifting is accomplished in a C++ library called Game Link, which adds Unity-specific coordinate conversions and entity management logic on top of our standard VR-Link toolkit. Above the C++ Game Link library is a set of C# bindings called GameLinkCS, which allows the Unity application to access the power of VR-Link and GameLink directly from C#, JavaScript, etc. GameLinkCS communicates with the underlying C++ libraries through a simple message-passing API.

The GameLinkCS layer is responsible for passing messages between Unity and VR-Link:

- For messages coming from DIS/HLA, GameLinkCS receives messages from VR-Link, and stores current state in Unity game object components. It automatically updates transforms to move objects around in the scene, based on VR-Link's dead-reckoning, smoothing, and ground clamping (if necessary). Users can extend the built-in message handler to fit the needs of their game design. For events (for example, Fire and Detonate), GameLinkCS provides a mechanism to register call-backs or custom handlers.

- For objects owned by Unity, GameLinkCS provides a "Publish" component which is responsible for passing current state across the language boundary to VR-Link. VR-Link then publishes your objects to the DIS/HLA network. For events/interactions, GameLinkCS provides a C# API to fill out and send the appropriate messages.



1.3. GameLink

The *DtGameLink* class is a singleton class that provides a centralized location to register message decoders, strategies, and resources used during execution. It is automatically created during startup and will load any plug-ins.

1.4. Loading Plug-ins

In order for a VR-Link for Unity plug-in to be loaded, the DLL must be in the plug-ins directory with the distributed VR-Link for Unity DLLs and it must export a function with the following name and signature:

```
extern "C" {
   DLL_EXPORT bool initGamelinkModule( DtGameLink* gl);
}
```

1.5. Initializing Connections

VR-Link for Unity is a protocol agile library. It allows for the protocol specific strategies to be loaded at runtime based on what the user asked for. Since many strategies may rely upon one another for shared resources, a connection initialization procedure is needed.

In an unconnected state, VR-Link for Unity only has 4 strategies loaded. These are the protocol specific initialization strategies. When a connection message is received the appropriate initialization strategy handles creating the remaining protocol specific strategies that are responsible for handling objects and interactions. Customer created strategies need to register themselves with the appropriate initialization strategy during plug-in loading so that they can be executed when a connection is made.

Since the order of initialization is important for dependent strategies a simple multi-pass initialization step is used to initialize all created strategies during a connection. A number of initialization passes are done to ensure that all strategies are able to create and find any shared resources or other strategies. During each pass, if a strategy is unable to fully initialize due to a missing resource, it will attempt to initialize in the next pass. This process is repeated until all strategies are initialized or until a deadlock is detected. This initialization step would fail if there is a circular dependency.

1.6. GameLink Concepts

This section discusses the GameLink concepts of strategies, resources, handles, and delegates.

1.6.1. Strategies

Strategies are the modular components within GameLink that process object state updates and interactions from VR-Link and creates the messages to send to Unity, and vice versa. There are several strategies included with VR-Link for Unity:

- Initialization
- Entity Object
- Fire Interaction
- Detonate Interaction
- Transmitter Object
- Signal Interaction
- Designator Object
- Terrain

There are copies of each of these strategies for each of the protocols supported by VR-Link (DIS, HLA 1.3, HLA 1516, HLA 1516-Evolved). The initialization strategy is responsible for creating the other strategies when a network connection is requested. It is possible to add new types of strategies or replace existing ones with updated behaviors through the use of plug-ins.

Strategies are given some CPU time every frame during a tick() function. During this function, any internal work may be done and is typically used to update any entity publishers that the strategy owns.

1.6.2. Resources

During execution certain data structures and classes need to be shared between different strategies. Typically shared resources are reflected objects, the simulation clock, the initialization strategy and the terrain strategy. Sharing objects between strategies is done by using a resource manager that is accessible from the *DtGameLink* class. Resources are generic pointers to objects wrapped in Handles, which are discussed below. The resource manager does not manage the memory of stored objects throughout their life-time, but does proved a generic means to find and access stored objects and data structures.

There are several default names used for commonly used object that are defined in *gamelink/utility.h*:

```
GAMELINK_DLL extern const char* EXERCISE_CONNECTION_NAME;
GAMELINK_DLL extern const char* CLOCK_NAME;
GAMELINK_DLL extern const char* REFLECTED_ENTITY_LIST_NAME;
GAMELINK_DLL extern const char* ENTITY_PUBLISHER_LIST_NAME;
GAMELINK_DLL extern const char* TERRAIN_STRATEGY_NAME;
```

These defined names provide a uniform way to access commonly used objects such as the VR-Link *DtExerciseConnection* or the *DtClock*.

There are also several utility functions to access commonly used objects:

```
GAMELINK_DLL DtExerciseConn* findExConnResource();
GAMELINK_DLL void setExConnResource(DtExerciseConn* conn);
GAMELINK_DLL DtReflectedEntity* findReflectedEntityResource(std::string
  id);
GAMELINK_DLL void setReflectedEntityResource(std::string id,
  DtReflectedEntity* ent);
GAMELINK_DLL DtEntityPublisher* findEntityPublisherResource(std::string
  id);
GAMELINK_DLL void setEntityPublisherResource(std::string id,
  DtEntityPublisher* pub);
GAMELINK_DLL DtClock* findClockResource();
GAMELINK_DLL void setClockResource(DtClock* clock);
GAMELINK_DLL DtReflectedEntityList* findReflectedEntityListResource();
GAMELINK_DLL void setReflectedEntityListResource(DtReflectedEntityList*
  rel);
GAMELINK_DLL DtTerrainStrategy* findTerrainStrategy();
GAMELINK_DLL void setTerrainStrategy(DtTerrainStrategy* ts);
```

1.6.3. Handles

Since the resource manager only stores simple pointers to classes and does not manage the memory of the object stored in the resource manager, a handle is used to provide a layer of protection to using raw pointers.

Handles are returned by the resource manager when requesting a resource. If the resource is changed without your plug-in knowing, typically by another strategy, it will automatically invalidate the handle and attempting to dereference it will result in a failure.

Creating handles is done through a static createHandle() function on the *DtHandle* class, which are then used to register the object with the resource manager. This is only necessary if the resource is intended to be shared between multiple strategies.

```
//create a new reflected laser designator list
myRelDx=new DtReflectedDesignatorList(myExconn);
//add some callbacks
myRelDx->addDesignatorAdditionCallback(
  &DtLaserDesignatorStrategy::vrlDesignatorDiscovered, this);
myRelDx->addDesignatorRemovalCallback(
  &DtLaserDesignatorStrategy::vrlDesignatorRemoved, this);
//create a handle to it
DtHandle h=DtHandle::createHandle(myRelDx);
//register the handle with the resource manager so other strategies can
  find it
myGameLink->setResource("REFLECTED_DESIGNATOR_LIST", h);
```

1.6.4. Delegates

Delegates are C++ classes that wrap a function to callback at a later time. These are used for message handlers that are registered with GameLink. When a strategy wants notification that a message arrives from Unity, it will create a delegate with one of the strategy's member functions and register that with the GameLink message handlers.

1.7. Messages

Messages are how VR-Link communicates with Unity. All messages are serialized into a byte buffer and then sent to Unity through a simple API. Messages are able to serialize and deserialize themselves. Adding new messages requires a C# implementation and a C++ implementation be created. This is done by using code generators and a message definition language.

1.7.1. Message Definition Language

The message definition language is written using the Lua language. Messages are declared using the "MESSAGE" keyword followed by the message layout and configuration in a key:value pair table. The following code is an example of a message:

```
MESSAGE{
  fileName="transmitterState";
```

```
className="TransmitterState";
includes={};
dllExport="GAMELINK_DLL";
enums={
  TransmitState=[[OFF, RADIO_ON_NOT_TRANSMITTING, ON]];
  InputSource=[[OTHER, PILOT, COPILOT, FIRST_OFFICER, DRIVER, LOADER,
GUNNER, COMMANDER, DIGITAL_DATA_DEVICE, INTERCOM]];
  AntennaPatternType=[[OMNIDIRECTIONAL=0, BEAM=1,
SPHERICALHARMONIC=2]];
  CryptoSystem=[[ OTHER, KY_28, VINSON, NSVE, WSVE, SINGARS_ICOM,
KY_75, KY_100]];
  CryptoMode=[[BASEBAND, DIPHASE]];
  ModulationMajorType=[[OTHER, AMPLITUDE,AMPLITUDE_ANGLE, ANGLE=3,
COMBINATION=4, PULSE=5, UNMODULATED=6, CARRIER_PHASE_SHIFT=7]];
  ModulationDetailAplitudeType=[[ OTHER, AFSK, AM, CW, DSB, ISB, LSB,
SSB_FULL, SSB_REDUC, USB, VSB]];
  ModulationDetailAplitudeAngleType=[[ OTHER, APLITUDE_ANGLE]];
  ModulationDetailAngleType=[[ OTHER, FM, FSK, PM]];
  ModulationDetailCombinationType=[[ OTHER, AMPLITUDE_ANGLE_PULSE]];
  ModulationDetailPulseType=[[ OTHER, PULSE, X_BAND, Y_BAND]];
  ModulationDetailUnmodulatedType=[[ OTHER, CONTINOUS_WAVE]];
  ModulationDetailCarrierPhaseShiftType=[[ OTHER]];
  ModulationSystemType=[[OTHER, GENERIC, HQ, HQII, HQIIA, SINCGARS,
CCTT_SINCGARS, EPLRS, JTIDS]];
  ReferenceSystem=[[WORLDCOORD=1, ENTITYCOORD=2]];
  StartOfMessage=[[NOT_START, START]];
  ClearChannel=[[NOT_CLEAR, CLEAR]];
  TransmitTerminalPrimaryMode=[[NTR, JTIDS]];
  TransmitTerminalSecondaryMode=[[NONE, NET_POSITION_REFERENCE,
PRIMARY_NAVIGATION_CONTROLLER, SECONDARY_NAVIGATION_CONTROLLER]];
  SyncState=[[COURSE=1, FINE=2]];

};
structs={
  ModulationType={
    {type="enum", name="major", enum="ModulationMajorType"};
    {type="UInt16", name="detail"}; --needs to be as the right enum
    {type="enum", name="system", enum="ModulationSystemType"};
  };
  OmniAntennaPattern={}; --this is a null pattern
  BeamAntennaPattern={
    {type="Vector3", name="orientation"};
    {type="float", name="AzimuthBeamWidth"};
    {type="float", name="ElevationBeamWidth"};
    {type="enum", name="referenceSystem", enum="ReferenceSystem"};
    {type="float", name="EZ"};
    {type="float", name="EX"};
    {type="float", name="phase"};
  };
  SphericalAntennaPattern={
    {type="byte", name="order"};
    {type="List<float>", name="coefficients"};
    {type="enum", name="referenceSystem", enum="ReferenceSystem"};
  };
};
attributes={
  {type="string", name="entityId"};
  {type="UInt16", name="radioId"};
  {type="EntityEnum", name="radioEntityType", new=true};
  {type="enum", name="transmitState", enum="TransmitState"};
```

```
        {type="enum", name="inputSource", enum="InputSource"};
        {type="Vector3", name="worldAntennaLocation"};
        {type="Vector3", name="relativeAntennaLocation"};
        {type="enum", name="antennaPatternType", enum="AntennaPatternType"};
        {type="UInt64", name="Frequency"};
        {type="float", name="transmitBandwidth"};
        {type="float", name="power"};
        {type="struct", name="modulationType", struct="ModulationType"};
        {type="enum", name="cryptoMode", enum="CryptoMode"};
        {type="enum", name="cryptoSystem", enum="CryptoSystem"};
        {type="UInt16", name="cryptoKey"};
        {type="List<byte>", name="modulationParameters", new=true};
        {type="List<byte>", name="antennaPaternParameters", new=true};
        {type="bool", name="freqHopInUse"};
        {type="bool", name="pseudoNoiseInUse"};
        {type="bool", name="timeHopInUse"};
    }
}
```

Message attributes can be simple data types, listed below, structures, enumerations, or arrays(lists) of acceptable attribute types.

1.7.2. Message Enums

Message Enumerations are defined in the enums table. Enumerations are given a name and then a list of enumerated values within double brackets [[ ]]. If a specific enumeration should have a specific value, then assign the enum=value in the definition. When declaring a message attribute as an enum, the type is "enum" and there is an additional field to declare which enum is used, for example:

```
{type="enum", name="transmitState", enum="TransmitState"};
```

1.7.3. Message Structures

Some messages need to package up several datafields into a single structure. These structures can be defined in the structs table. A structure follows the same rules as the attributes table and uses the same data types, which are defined below. When declaring a message attribute as a struct, the type is "struct" and there is an additional field to declare which struct is used, for example:

```
{type="struct", name="modulationType", struct="ModulationType"};
```

1.7.4. Message Attributes

Message attributes define the message layout. Attributes can be defined a simple data type, an enum, a structure, or a list of any of these types.

- string
- bool
- Int8
- Uint8

- Int16
- UInt16
- Int32
- UInt32
- Int64
- UInt64
- float
- double
- Vector2
- Vector3
- Vector4
- Quaternion
- EntityEnum
- List<DATA TYPE>

Using an EntityEnum, user defined struct or List<> datatype as an attribute requires that the attribute have the field new=true be defined as well.

```
{type="EntityEnum", name="munitionType", new=true};
```

1.7.5. Message Configuration

The message description has fields for the name of the file to generate (not including the extension) and the name of the class to build. The message definition also contains fields to describe any required include files, or DLL Export macro name, which are only used for the C++ code generator.

1.8. The Code Generators

Once the message definition is written, the included code generators generate C++ and C# source code to encode and decode the message data. Both code generators are command line tool and can be integrated into your build steps to generate message source code as a build step.

The C# code generator has the following command line parameters:

- `-i` The full path to the message definition file.
- `-o` The full path to where the generated C# files should go.

The C++ code generator has the following command line parameters:

- `-i` The full path to the message definition file.
- `-h` The full path to where the generated C++ header files should go.
- `-s` The full path to where the generated C++ source file should go.

1.9. Example Plug-in

The included example is the complete source code for a plug-in that adds support for Laser Designators in DIS and HLA. It uses the C++ VR-Link toolkit and the VR-Link for Unity SDK.

1.9.1. Project Description and Setup

The project files for the example are broken down into five projects. One project for the protocol independent message representations to be loaded into the message factory, then four protocol projects to create plug-ins that load a protocol specific strategy into the strategy factory and registers it with the initialization strategy so that it gets created on connection. Since VR-Link is protocol independent and uses compiler flags to determine which protocol to support, most of the source code for the example is shared between the protocol specific projects. Changes to files in project will show up as changes in other projects.

Since the project files are configured to use environment variables to find the C++ VR-Link SDK, it is recommended to edit the included *setupEnvironment.bat* file to match your system and use it to launch Visual Studio.

1.9.2. Laser Designator Messages

Laser designators are stateful objects within DIS and HLA and change their state when painting a target. This means that we will need a discovery, state update, and removed message to inform Unity about laser designators.

These three messages are defined in *laser.lua*. Several things to note is that entity ID's are transferred to Unity as strings and all positions are vectors in Unity's coordinate system.

1.9.3. Laser Designator Plug-in

The Laser Designator plug-in loads the laser designator messages into the message factory in its initGamelinkModule() function:

```
bool initGamelinkModule( DtGameLink* gl)
{
  // add the message types to the factory so that GameLink knows how to
  // decode them
  gl->messageFactory().registerCreator(
  LaserDesignatorDiscoveryMessage::theMessageName(),
  LaserDesignatorDiscoveryMessage::decode);
  gl->messageFactory().registerCreator(
    LaserDesignatorRemovedMessage::theMessageName(),
    LaserDesignatorRemovedMessage::decode);
  gl->messageFactory().registerCreator(
    LaserDesignatorStateMessage::theMessageName(),
    LaserDesignatorStateMessage::decode);
  return true;
}
```

1.9.4. Laser Designator Protocol Specific Plug-ins

The protocol specific laser designator plug-ins load their specific strategy into the strategy factory and register with the initialization strategy to be loaded during connection.

```
bool initGamelinkModule( DtGameLink* gl)
{
  // add the strategies to the factory
  // this is ok to do multiple times since it won't damage anything/or
  // leak resources
  gl->addStrategyCreator(DtLaserDesignatorStrategy::theName(),
  DtLaserDesignatorStrategy::create);

  // add the strategy to the correct init strategy so that it gets
  // created during initialization
  DtInitStrategy* initStr=dynamic_cast<DtInitStrategy*>(
    gl->findStrategy(DtInitStrategy::theName()));
  if(initStr!=NULL)
  {
    initStr->addInitChild(DtLaserDesignatorStrategy::theName());
  }
  else
  {
    DtWarn << "Unable to register Laser strategy with the Initialization
  strategy for DIS protocol" << std::endl;
  }

return true;
}
```

1.9.5. Laser Designator Strategy

When the laser designator strategy is created it registers interest in laser object messages with GameLink. It does this by wrapping its handler member functions with a delegate and passing this delegate to GameLink:

```
myDesignatorDiscoveredHandler=new DtMessageDelegate(this,
  &DtLaserDesignatorStrategy::handleUnityDesignatorDiscovered);
myDesignatorRemovedHandler=new DtMessageDelegate(this,
  &DtLaserDesignatorStrategy::handleUnityDesignatorRemoved);
myDesignatorStateHandler=new DtMessageDelegate(this,
  &DtLaserDesignatorStrategy::handleUnityDesignatorState);

myGameLink->addHandler(
  LaserDesignatorDiscoveryMessage::theMessageName(),
  myDesignatorDiscoveredHandler);
myGameLink->addHandler(LaserDesignatorRemovedMessage::theMessageName(),
  myDesignatorRemovedHandler);
myGameLink->addHandler(LaserDesignatorStateMessage::theMessageName(),
  myDesignatorStateHandler);
```

It is important to remember to remove these delegates from GameLink during shutdown so that they do not get called after the strategy is destroyed.

```
myGameLink->removeHandler(
  LaserDesignatorDiscoveryMessage::theMessageName(),
  myDesignatorDiscoveredHandler);
```

```
myGameLink->removeHandler(
  LaserDesignatorRemovedMessage::theMessageName(),
  myDesignatorRemovedHandler);
myGameLink->removeHandler(
  LaserDesignatorStateMessage::theMessageName(),
  myDesignatorStateHandler);
```

The initialization function of the laser designator strategy finds the exercise connection shared resource, creates a reflected laser designator list and installs some callbacks on it to listen for laser designators coming from the DIS/HLA network. It also creates a laser designator publisher list to hold publishers that it creates to mirror laser designators being controlled in Unity.

```
bool DtLaserDesignatorStrategy::init()
{
   if(myInitalized==true) return true;

   myExconn=findExConnResource();
   if(myExconn!=NULL)
   {
      //get the terrain strategy for coordinate conversions
      myTerrainStrategy=findTerrainStrategy();
      if(myTerrainStrategy==NULL)
      {
         //not ready yet, try again next pass
         return false;
      }

     //radio Designators
      myRelDx=new DtReflectedDesignatorList(myExconn);
      myRelDx-
  >addDesignatorAdditionCallback(&DtLaserDesignatorStrategy::vrlDesignat
  orDiscovered, this);
      myRelDx-
  >addDesignatorRemovalCallback(&DtLaserDesignatorStrategy::vrlDesignato
  rRemoved, this);
      DtHandle h=DtHandle::createHandle(myRelDx);
      myGameLink->setResource("REFLECTED_DESIGNATOR_LIST", h);

      myDxPubs=new DtPublisherList<DtDesignatorPublisher>();
      h=DtHandle::createHandle(myDxPubs);
      myGameLink->setResource("DESIGNATOR_PUBLISHER_LIST", h);

      myInitalized=true;
      DtDebug << "Laser Designator strategy initialized" << std::endl;
   }

   return myInitalized;
}
```

Since this initialization function will be run more than once, it protects itself against being initialized more than once and in the event it cannot get the shared resource exercise connection it may return false, causing it to be run again until is able to get the shared resource and fully initialize.

During the tick function, which is called every frame, the Laser Designator strategy updates any laser designator publishers that it created to mirror laser designators from Unity to the DIS/HLA network.

```
myDxPubs->tick();
```

The strategy contains several callbacks for registering with VR-Link as well as with the GameLink message handlers to handle discovery, removal and state update of laser designators from both the DIS/HLA network and within Unity. These callbacks marshall the data to and from the laser designator messages and VR-Link data structures ensuring to convert entity names and terrain coordinates using the conversion routines in the initialization strategy and terrain strategy respectively.

Convert the Unity name to the VR-Link name:

```
pub->dsr()->setDesignatedObject(DtInitStrategy::unityToVrlinkId(dxMsg-
  >getTargetId()));
```

Convert the VR-Link position to the the Unity position:

```
DtVector vec=myTerrainStrategy->geocentricToUnityPosition(dx->dsr()-
  >designatorSpotLocation());
```

1.9.6. Unity Changes

In order to receive and create laser designators within Unity we need to modify the simulation manager behavior to know how to handle laser designator messages and create the right type of publishers and reflected laser designators.

```
myMessageHandlers[LaserDesignatorDiscoveryMessage.theName]=new
  MessageDelegate(handleLaserDesignatorDiscoveryMessage);
    myMessageHandlers[LaserDesignatorRemovedMessage.theName]=new
  MessageDelegate(handleLaserDesignatorRemovedMessage);
    myMessageHandlers[LaserDesignatorStateMessage.theName]=new
  MessageDelegate(handleLaserDesignatorStateMessage);
```

Next we need implement the functions to handle the messages within the simulation manager. This includes created a reflected designator behavior when a new designator is discovered from the network, attaching it to the designated entity, as well as sending the designator state messages to handle. This is done through the following functions.

```
public void handleLaserDesignatorDiscoveryMessage(Message m)
  {
    LaserDesignatorDiscoveryMessage tdm=m as
  LaserDesignatorDiscoveryMessage;
    if(tdm!=null)
    {
      //find the game object to attach to
      GameObject instance=GameObject.Find(tdm.entityId);

      if(instance!=null)
      {
        //setup the reflected behavior
        ReflectedDesignatorBehavior
  rb=instance.GetComponentInChildren<ReflectedDesignatorBehavior>();
        if(rb==null)
        {
          rb=instance.AddComponent<ReflectedDesignatorBehavior>();
        }
      }
    }
```

```
}

public void handleLaserDesignatorRemovedMessage(Message m)
{
  LaserDesignatorRemovedMessage trm=m as
LaserDesignatorRemovedMessage;
  if(trm!=null)
  {
    GameObject instance=GameObject.Find(trm.entityId);
    if(instance!=null)
    {
      ReflectedDesignatorBehavior
rb=instance.GetComponent<ReflectedDesignatorBehavior>();
      Destroy (rb);
    }
  }
}

public void sendLaserDesignatorStateMessage(string id, Message m)
{
  GameObject instance=GameObject.Find(id);
  //update the entity
  if(instance!=null)
  {
    ReflectedDesignatorBehavior
rb=instance.GetComponentInChildren<ReflectedDesignatorBehavior>();
    if(rb==null)
    {
      throw new Exception("Error trying to update reflected
transmitter without reflected transmitter behavior component");
    }

    rb.handleMessage(m);
  }
  else
  {
    //discover that entity
    LaserDesignatorStateMessage sm=m as LaserDesignatorStateMessage;
    LaserDesignatorDiscoveryMessage dm=new
LaserDesignatorDiscoveryMessage();
    dm.entityId=sm.entityId;
    handleLaserDesignatorDiscoveryMessage(dm);
  }
}

public void handleLaserDesignatorStateMessage(Message m)
{
  LaserDesignatorStateMessage sm=m as LaserDesignatorStateMessage;
  if(sm!=null)
  {
    sendLaserDesignatorStateMessage(sm.entityId, m);
  }
}
```

Once we have the simulation manager modified, we also need to ensure that we have a reflected designator behavior and a designator publisher class that we can attach to Unity objects. These classes show a reference implementation of how to handle these messages and objects. It is expected that a customer would need to modify them to suit the needs of their application.

The following file handles the designator publisher:

```
/*****************************************************************************
  **********
** Copyright (c) 2013 MAK Technologies, Inc.
** All rights reserved.
*****************************************************************************
  *********/

using System;
using UnityEngine;
using System.Collections.Generic;


public class DesignatorPublishBehavior : MonoBehaviour
{
  public float myPower=0.0f;
  public float myWavelength=0.0f;
  public int myCode;
  public Vector3 myWorldSpot=Vector3.zero;

  public LaserDesignatorStateMessage myState;

  public GameObject SimManager;
  protected SimManagerBehavior mySimManager;

  // Use this for initialization
  void Start ()
  {
    if(SimManager==null)
    {
      GameObject[] gos=GameObject.FindGameObjectsWithTag("Simulation");
      foreach(GameObject go in gos)
      {
        SimManagerBehavior sm=go.GetComponent<SimManagerBehavior>();
        if(sm!=null)
        {
          mySimManager=sm;
        }
      }
    }
    else
    {
      SimManagerBehavior
  sm=SimManager.GetComponent<SimManagerBehavior>();
      if(sm!=null)
      {
        mySimManager=sm;
      }
    }

    if(mySimManager==null)
    {
      throw new Exception("Unable to find Simualtion Manager Behavior");
    }

    myState=new LaserDesignatorStateMessage();
    myState.entityId=gameObject.GetInstanceID().ToString(); //guaranteed
  to be unique
```

```
    myState.designatorCode=(LaserDesignatorStateMessage.DesignatorCode)myC
    ode;
      myState.power=myPower;
      myState.wavelength=myWavelength;
      myState.worldDesignatorSpot=myWorldSpot;

      //you can update more fields here depending on your simulation
    }

    void OnDisable()
    {

    }

    // Update is called once per frame
    void Update ()
    {
      if(mySimManager.connected==true)
      {
        myState.power=myPower;
        myState.wavelength=myWavelength;
        myState.worldDesignatorSpot=myWorldSpot;
        mySimManager.sendGamelinkMessage(myState);
      }
    }
}
```

The following file handles the reflected designator:

```
/*****************************************************************
  **********
** Copyright (c) 2013 MAK Technologies, Inc.
** All rights reserved.
*****************************************************************
  *********/

using System;
using UnityEngine;
using System.Collections.Generic;


public class ReflectedDesignatorBehavior : MonoBehaviour
{
  public LaserDesignatorStateMessage myState;

  Dictionary<string, MessageDelegate> myMessageHandlers=new
  Dictionary<string, MessageDelegate>();

  // Use this for initialization
  void Start ()
  {
    myMessageHandlers[LaserDesignatorStateMessage.theName]=new
  MessageDelegate(handleStateMessage);
  }

  // Update is called once per frame
  void Update ()
  {

  }
```

```
  public void handleMessage(Message m)
  {
    MessageDelegate md;
    if(myMessageHandlers.TryGetValue(m.name, out md)==true)
    {
      md(m);
    }
  }

  public void handleStateMessage(Message m)
  {
    LaserDesignatorStateMessage sm=m as LaserDesignatorStateMessage;
    if(sm!=null)
    {
      myState=sm;
    }
  }
}
```

Link - Simulate - Visualize

VRU-1.1-2-130802

150 CAMBRIDGE PARK DRIVE, 3RD FLOOR CAMBRIDGE, MA 02140 617.876.8085 www.mak.com