# CONTENTS FOR VOLUME 1

## RELEASE NOTES
- Updates for Version r5 ................................................................. 1
- Documentation Updates ................................................................. 1
- Sample Additions ............................................................................. 3
- Updates for Version r4 ................................................................. 3
- Documentation Updates ................................................................. 3
- Sample Additions ............................................................................. 4
- Problems, Restrictions, and Workarounds ........................................ 5

## INTRODUCTION
- Using the SDK ................................................................. 7
- Header Files ............................................................................. 7
- Sample Files ............................................................................. 7
- Helix Universal Server ................................................................. 7
- RealPlayer .............................................................................. 8
- TestPlay .............................................................................. 8
- SDK Notes ............................................................................. 8
- Audience for This Guide ......................................................... 8
- How This Guide is Organized .................................................. 9
- Conventions in This Manual ..................................................... 11
- Additional Helix Resources ....................................................... 11
- Helix Community Web Site ....................................................... 12
- Technical Support ..................................................................... 12

## PART I: COMMON PLATFORM FEATURES

### 1 INSTALLATION AND DEVELOPMENT TOOLS
- Installing and Configuring Helix Universal Server ................................ 15
- Installing and Configuring a Helix Client .......................................... 15
- TestPlay Installation .................................................................. 16
- Windows Component Locations .................................................. 16
- Using the RTP Packet Format .................................................. 17
- Macintosh Component Locations ............................................. 17
- UNIX Component Locations .................................................... 18
- Windows Development Tools ..................................................... 18
- UNIX Development Tools ......................................................... 19
- Macintosh Development Tools .................................................. 19
- Helix Client and Server Software Development Kit Interface Name Change ........................................ 19

### 2 HELIX DNA COMPONENTS
- Helix Components ..................................................................... 23
- Helix Universal Server ............................................................... 24
- Helix Client ............................................................................. 24
### DEVELOPING PLUG-INS

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Designing a Plug-in</td>
<td>35</td>
</tr>
<tr>
<td>Plug-in Basics</td>
<td>36</td>
</tr>
<tr>
<td>Creating a Plug-in Instance</td>
<td>36</td>
</tr>
<tr>
<td>Freeing Plug-in Resources</td>
<td>37</td>
</tr>
<tr>
<td>Defining Plug-in Attributes</td>
<td>37</td>
</tr>
<tr>
<td>Exporting Plug-in Interfaces</td>
<td>38</td>
</tr>
<tr>
<td>Creating a Helix Object</td>
<td>39</td>
</tr>
<tr>
<td>Plug-in Initialization</td>
<td>40</td>
</tr>
<tr>
<td>Additional Plug-in Initialization</td>
<td>41</td>
</tr>
<tr>
<td>Maintaining Plug-in Reference Counters</td>
<td>41</td>
</tr>
<tr>
<td>Plug-in Object Constructor and Destructor</td>
<td>42</td>
</tr>
<tr>
<td>Enumerating Plug-ins</td>
<td>42</td>
</tr>
<tr>
<td>Building a Sample Plug-in</td>
<td>43</td>
</tr>
<tr>
<td>&quot;Hello World&quot; Sample</td>
<td>43</td>
</tr>
<tr>
<td>Debugging a Plug-in</td>
<td>44</td>
</tr>
<tr>
<td>Testing a Plug-in</td>
<td>44</td>
</tr>
<tr>
<td>Installing Plug-ins for Testing</td>
<td>44</td>
</tr>
<tr>
<td>Testing with RealPlayer</td>
<td>45</td>
</tr>
<tr>
<td>Testing with TestPlay</td>
<td>45</td>
</tr>
<tr>
<td>Testing with Multimedia</td>
<td>46</td>
</tr>
<tr>
<td>Testing with HTTP</td>
<td>46</td>
</tr>
<tr>
<td>Plug-in Examples</td>
<td>47</td>
</tr>
</tbody>
</table>

### FILE HANDLING

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>File System and File Format Plug-in Interaction</td>
<td>49</td>
</tr>
<tr>
<td>Helix Universal Server Interaction</td>
<td>49</td>
</tr>
<tr>
<td>Helix Client Interaction</td>
<td>49</td>
</tr>
<tr>
<td>File System Plug-ins</td>
<td>50</td>
</tr>
<tr>
<td>Interfaces</td>
<td>50</td>
</tr>
<tr>
<td>Coding the Plug-in</td>
<td>51</td>
</tr>
<tr>
<td>Starting Up</td>
<td>52</td>
</tr>
<tr>
<td>Initializing</td>
<td>52</td>
</tr>
<tr>
<td>Creating File Objects</td>
<td>53</td>
</tr>
</tbody>
</table>
Supporting Relative File Access ................................................................. 54
Closing File Objects .................................................................................. 54
Multiload File Systems and Streamers ....................................................... 54
Sequence of Calls .................................................................................... 55
Modifying the File System Plug-in Sample Code ........................................ 56
File Format Plug-ins ................................................................................ 56
Design Considerations ............................................................................ 57
Interfaces .................................................................................................. 57
Coding the Plug-in .................................................................................. 58
Starting Up ............................................................................................... 59
Initializing ............................................................................................... 59
Creating Stream Headers ......................................................................... 61
Creating Stream Packets ......................................................................... 62
Handling User Seeks .............................................................................. 63
Closing ..................................................................................................... 63
Supporting Multiple Packet Formats ....................................................... 63
RTP Stream Header Properties ................................................................. 64
SDPData Header Property ........................................................................ 65
MIME Type ............................................................................................. 65
RTP Timing ............................................................................................. 65
Reporting Pending Status ........................................................................ 66
Modifying the Response Headers ............................................................ 66
Interacting with a File Object ................................................................. 67
Retrieving File Information ..................................................................... 67
Seeking File Data ................................................................................... 67
Reading File Data .................................................................................. 68
Closing a File Object ............................................................................... 68
Creating Relative File Objects ............................................................... 69
Using Multiple Relative Files ................................................................. 69
Modifying the File Format Plug-in Sample Code ..................................... 70

5 DATA CONVERSION AND REVERSION PLUG-INS 73
Interfaces .................................................................................................. 73
Starting Up the Conversion Plug-in .......................................................... 74
Initializing the Conversion Plug-in .......................................................... 75
Converting Data ..................................................................................... 76
Conversion and Multicast Support .......................................................... 76
Coding the Reversion Plug-in ................................................................. 77
Starting Up the Reversion Plug-in ........................................................... 77
Initializing the Reversion Plug-in ........................................................... 78
Restoring Data ....................................................................................... 78
Supporting Client and Server Communication ....................................... 79
Modifying the Conversion and Reversion Plug-in Sample Code .......... 79

6 NETWORK SERVICES ................................................................. 81
Interfaces .................................................................................................. 81
Creating a Listen Socket ........................................................................ 82
Using a TCP Socket ............................................................................... 82
Using a UDP Socket ............................................................................... 83
PART II: SERVER PLUG-INS AND SERVICES

7 STATUS CODES AND ERRORS

Interference

Status Codes

Reporting Errors

Receiving Error Messages

8 LIVE BROADCASTING

Broadcast Plug-in

Interference

Coding the Plug-in

Starting Up

Initializing

Creating Stream Headers

Creating Stream Packets

Modifying the Broadcast Plug-in Sample Code

Remote Broadcast Services

Using Remote Broadcast Services

Remote Broadcast Components

Interference

Initializing Remote Broadcast Services

Sending a New Media Stream

Sending the Media Stream

Closing Remote Broadcast Services

Getting the Current Time

Simulated Live Transfer Agent

Design Considerations

Initializing the SLTA

Setting the Title, Author, and Copyright

Beginning Transmission

Ending Transmission

Closing the SLTA

Setting Target Bandwidth

RealText Broadcast

Interference

Coding the Remote Text Broadcast Application

Initializing the RealText Broadcast Library

Defining the RealText Window

Sending Text to the Library

Communicating with the Library

Modifying the RealText Broadcast Sample Code

Creating a Broadcast RealText Clip

Tips for Creating Broadcast RealText Files

RealPix Broadcast

Interference
# MANAGING THE SERVER REGISTRY

## Registry Basics
- Custom Properties
- RTP Marker Bit Property
- Directives
- Informational Properties
- Priority
- Design Considerations

## Predefined Server Configuration Properties
- Predefined Server Properties
- Predefined Client Properties
- Predefined License Properties
- Predefined Monitor Properties
- Custom or Predefined Server Configuration Properties

## Coding the Remote Pix Broadcast Application
- Modifying the RealPix Broadcast Sample Code

## Initializing the LiveRealPix Broadcast Library

## Sending Images to the Library

## Stopping the LiveRealPix Broadcast Library

## Modifying the RealPix Broadcast Sample Code

## Server HTTP and RTSP Authentication
- Implementing Authentication
- Authorization Types

## Starting Up

## Sending Images to the Library

## Initializing the LiveRealPix Broadcast Library

## Stopping the LiveRealPix Broadcast Library

## Modifying the RealPix Broadcast Sample Code

## Using the Scheduler

## Watching Properties

## Starting Up

## Modifying the Monitor Plug-in Sample Code

## Implementing Authentication

## Authorization Types

## Coding the Plug-in

## Design Considerations

## Interfaces

## Starting Up

## Modifying the Allowance Plug-in Sample Code

## Example 1: A File System Plug-in

## Example 2: A Monitor Plug-in

## Example 1: Adding a Simple Property to the Registry

## Example 2: Adding Statistics to the Registry

## 10 RESTRICTING ACCESS

## Server HTTP and RTSP Authentication
- Interfaces
- Implementing Authentication
- Example 1: A File System Plug-in
- Example 2: A Monitor Plug-in

## Authorization Types

## Coding the Allowance Plug-in

## Design Considerations

## Interfaces

## Starting Up

## Modifying the Allowance Plug-in Sample Code

## 11 ADAPTIVE STREAM MANAGEMENT

## Interfaces

## Example of ASM Interfaces in Use

## ASM Rules and Properties

## Example 1: Adding a Simple Property to the Registry

## Example 2: Adding Statistics to the Registry

## Examples

## Example 1: Adding a Simple Property to the Registry

## Example 2: Adding Statistics to the Registry

## Examples

## Example 1: Adding a Simple Property to the Registry

## Example 2: Adding Statistics to the Registry

## Examples
PART III: CLIENT PLUG-INS AND SERVICES

12 TOP-LEVEL CLIENT
   Interfaces .................................................................................................................. 153
   Building a Client ..................................................................................................... 154
   Using the Client Core .............................................................................................. 154
     Using the Select Method on UNIX ...................................................................... 154
   Using a Player Object ............................................................................................. 155
     Getting Information About a Presentation ......................................................... 157
   Manipulating Groups in a Presentation .................................................................. 158
   Adding Site Support to a Client ............................................................................. 158
   Handling Password Authentication ....................................................................... 158
   Modifying the Top-level Client Sample Code ....................................................... 159

13 RENDERING PLUG-IN
   Developing Basic Rendering Plug-ins .................................................................. 161
   Design Considerations ......................................................................................... 161
   Interfaces ............................................................................................................. 162
   Coding the Plug-in ................................................................................................ 163
     Starting Up ......................................................................................................... 163
     Initializing .......................................................................................................... 163
     Rendering Streams ............................................................................................ 164
     Seeking .............................................................................................................. 165
     Pausing ............................................................................................................... 166
     Hypernavigating .................................................................................................. 166
     Sending Back-Channel Packets ......................................................................... 166
     Terminating a Presentation ................................................................................ 167
   Accessing Stream and Player Objects .................................................................. 167
     Using the Stream Object .................................................................................... 167
     Using the Stream Source Object ....................................................................... 168
   Rendering Images on the Client Display ................................................................ 168
   Timing and Synchronization ................................................................................ 169
   Modifying the Basic Rendering Plug-in Sample Code ........................................ 170
   Audio Services .................................................................................................... 171
     Interfaces ........................................................................................................... 172
     Supported Input Formats .................................................................................... 173
       Mixing Audio Streams ....................................................................................... 173
       Quality of Mixed Audio Streams .................................................................... 174
     Coding a Rendering Plug-in for Audio ............................................................ 174
     Controlling Volume ........................................................................................... 176
   Implementing Midstream Playback and Instant Sounds ........................................ 177
   Using Post-Processed Audio Data ....................................................................... 178
     Getting Pre-Mix Audio Data .............................................................................. 178
RELEASE NOTES

Updates for Version r5

The Helix Client and Server Software Development Kit represents a new, expanded replacement for the previously released RealSystem SDK. The following release notes provide a brief look at the changes made to the SDK to support the new capabilities of the Helix platform.

Documentation Updates

Specific documentation updates include:

• The Helix SDK Developer’s Guide has been split into two volumes. The first volume contains installation and compilation instructions, as well as tutorial material explaining how to modify various components of Helix Universal Server and the Helix client. The second volume contains reference information for the interfaces, methods, structures, and functions supplied with the Helix SDK.

• The basic organization of the tutorial material found in the first third of the previous version of this guide has been updated. Whereas the previous version organized this material by overview material, plug-ins, features and services, and building on the core, this version of the guide organizes this material by server, client, and common server and client tasks.

• Basic information about designing a plug-in has been added. See “Chapter 3: Developing Plug-ins” beginning in Volume 1, on page 35.

• The information originally found in “Chapter 5: File Format Plug-in” and “Chapter 7: File System Plug-in” has been incorporated in a single chapter. See “Chapter 4: File Handling” beginning in Volume 1, on page 49.

• The information originally found in “Chapter 8: Broadcast Plug-in”, “Chapter 17: Remote Broadcast”, “Chapter 18: RealText Broadcast”, and “Chapter 19: RealPix Broadcast” has been incorporated in a single chapter. See “Chapter 8: Live Broadcasting” beginning in Volume 1, on page 91.

• A chapter that describes converting data on the server and restoring the data on the client was added. See “Chapter 5: Data Conversion and Reversion Plug-Ins” beginning in Volume 1, on page 73.

• In previous releases of the Helix Client and Server Software Development Kit, broadcast plug-ins used the remote broadcast library to convey a real-time media stream from an encoder for transmission to a host server. The current release uses the new remote broadcast services to perform this task. See “Remote Broadcast Services” in Volume 1, on page 97.

  Note: RealText and RealPix broadcast continue to use the remote broadcast library.
• Previous releases of the Helix Client and Server Software Development Kit provided the IHXSLTA interface for simulated live transfer agent (SLTA) applications. The current release of the Helix Client and Server Software Development Kit replaces the obsolete IHXSLTA interface with a new IHXiQSLTA interface. The primary difference between the core implementation of these two interfaces is the underlying broadcast transport technology. See “Simulated Live Transfer Agent” in Volume 1, on page 101.

• The information originally found in “Appendix F: RealServer Property Registry” and “Chapter 9: Monitor Plug-in” has been incorporated in a single chapter. See “Chapter 9: Managing the Server Registry” beginning in Volume 1, on page 119.

• The information originally found in “Chapter 10: Allowance Plug-in” and “Chapter 20: Authentication” has been incorporated in a single chapter. See “Chapter 10: Restricting Access” beginning in Volume 1, on page 131.

• The information originally found in “Chapter 6: Rendering Plug-in”, “Chapter 14: Audio Services”, and “Chapter 13: Sites (Windowing)” has been incorporated in a single chapter. See “Chapter 13: Rendering Plug-In” beginning in Volume 1, on page 161.

• The following interface information was added to “Appendix A: Interface List” beginning in Volume 2, on page 191:
  - IHXAudioHookManager
  - IHXAuthenticationManager2
  - IHXDrawFocus
  - IHXFastFileStats
  - IHXFileMove
  - IHXFocusNavigation
  - IHXGroupSink2
  - IHXiQSLTA
  - IHXKeyboardFocus
  - IHXKeyValueList
  - IHXKeyValueListIter
  - IHXKeyValueListIterOneKey
  - IHXLiveRealPixResend
  - IHXMutex
  - IHXNetworkInterfaceEnumerator
  - IHXNetworkServices2
  - IHXPlayerControllerProxyRedirect
  - IHXPlayerNavigator
  - IHXPrefetch
  - IHXPrefetchSink
  - IHXProcessEntryPoint
  - IHXRegistryAltStringHandling
  - IHXRegistryIID
  - IHXServerReconfigNotification
• IHXSiteEnumerator
• IHXSiteEvent
• IHXSyncFileFormatObject
• IHXThreadSafeMethods
• IHXThreadSafeScheduler
• IHXTrack
• IHXTrackSink
• IHXUpdateProperties
• IHXVideoHook
• IHXVideoHookSink
• IHXViewDRMRights
• IHXWantServerReconfigNotification
• IHXXMLParser
• IHXXMLParserResponse

• The appendix on configuration file variables was deleted. You can find this information at http://service.real.com/help/library/guides/helixserverconfig/ServerConfig.htm.

• “Appendix G: RealMedia File Format (RMFF)” in the previous version of this documentation has been moved to “Appendix E: RealMedia File Format (RMFF) Reference” beginning in Volume 2, on page 717.

Sample Additions

The Helix Client and Server Software Development Kit provides numerous sample files from which you can develop your plug-ins and applications. For a complete list of samples supplied in this version of the SDK, see “Sample Files” in Volume 2, on page 738. The following samples have been added to this version of the SDK:

XML Parser Application Sample Added
A new XML parser application sample has been added in the advanced samples directory of the SDK. This application parses XML components from the specified file, breaking the XML document to its component parts and displaying them on the console.

Updates for Version r4

Documentation Updates

Specific documentation updates include:

• “Appendix A: Interface List” has been updated to include all of the interfaces and methods included in all of the header files released with the SDK.

• An appendix that describes all of the Helix structures has been added. See “Appendix B: Structure List”.

3
• An appendix that describes all of the Helix functions has been added. See “Appendix C: Function List”.
• An appendix that describes all of the Helix PNR_return values has been added. See “Appendix D: Return Values”.
• “Appendix B: RealServer Configuration” in the previous version of this documentation has been moved to “Appendix E: RealServer Configuration”.
• “Appendix C: RealServer Property Registry” in the previous version of this documentation has been moved to “Appendix F: RealServer Property Registry”.
• “Appendix D: RealMedia File Format (RMFF)” in the previous version of this documentation has been moved to “Appendix G: RealMedia File Format (RMFF)”.

Sample Additions

The following samples have been added to this version of the SDK:

Authenticating File System Plug-in Sample Added
A new authenticating file system plug-in sample has been added. This plug-in closely parallels the standard file system plug-in sample, but verifies the user’s identity using realms.

Audio Device Top-level Client Sample Added
A new audio device top-level client sample has been added. This top-level client shows how to add low-level audio.

RTP-audio Remote Broadcast Encoder Application Sample Added
A new RTP-audio remote broadcast encoder application sample has been added. This application demonstrates how to stream various types of live audio.

Pay-per-view Allowance Plug-in Sample Added
A new pay-per-view allowance plug-in sample has been added. This plug-in manages pay-per-view authentication by password protecting all appropriate URLs and authenticating clients that attempt to access the protected URLs.

Video Capture Top-level Client Sample Added
A new video capture top-level client sample has been added. This top-level client demonstrates how to capture or intercept video frames.

Data Conversion and Reversion Plug-in Samples Added
New data conversion and reversion samples have been added. These samples provide a means of converting and restoring a stream from the server to a client. As an example, these samples could be used for encrypting data on a server (the converter always resides on the server) and decrypting the data on the client (the reverter always resides on the client).

Slide Show Sample Added
A new live RealPix slide show sample has been added. This sample demonstrates how to make a slide show application using IHXLiveRealPix.
Problems, Restrictions, and Workarounds

The following are known restrictions on the use of this release of the Helix SDK:

Internet Explorer 3.0 and SMIL
Internet Explorer 3.0 does not properly read SMIL files that use the `<head>` or `<title>` tag. It ignores the file's MIME type and tries to display the file as HTML. Avoid this problem by omitting these tags in your SMIL files. This problem does not occur with Netscape Navigator or Internet Explorer 4.0 and later.
INTRODUCTION

Welcome to the Helix Client and Server Software Development Kit (SDK), which RealNetworks has created for developers working with Helix media streaming. Helix provides the most sophisticated system available for streaming multimedia files across a network. This developer's guide will help you use the SDK to produce server-side and client-side plug-ins, and to create media clients based on the Helix client engine.

Using the SDK

Because Helix is based on the Component Object Model (COM) binary standard, you can develop Helix components using virtually any programming language. Using the SDK sample files, however, requires using C++. It is important to familiarize yourself with COM before you begin developing Helix components. Note, however, that Helix diverges from the COM standard to simplify cross-platform development.

**For More Information:** For details, see “Chapter 3: Developing Plug-ins” beginning in Volume 1, on page 35.

Header Files

The Helix header files define the Helix interfaces. When you are ready to begin developing Helix components, refer to the header files along with this documentation. The header files may contain information about function variables and return values not listed in the documentation.

Sample Files

You can use the sample files included with this SDK as templates for building your own Helix components. Using the sample code requires a knowledge of C or C++. Except for interfaces that interact with file systems explicitly, all sample code is platform-independent.

**For More Information:** RealNetworks recommends using specific compilers for compiling code based on the sample files. For more information, see “Building a Sample Plug-in” in Volume 1, on page 43.

Helix Universal Server

Use Helix Universal Server to test your Helix components. Helix Universal Server binary files are not included with this SDK, but free downloads of the basic version of Helix Universal Server are available at [http://www.realnetworks.com/products/](http://www.realnetworks.com/products/).
RealPlayer

Use RealPlayer to play streamed media. The commercial version of RealPlayer is available at http://www.real.com/. If RealPlayer is not available for your platform, you can use TestPlay.

TestPlay

This SDK includes source code for TestPlay, which offers the core RealPlayer functionality without a graphical user interface. You can test your plug-ins with TestPlay, or you can use the utility as a basis for building your own Helix clients.

SDK Notes

Before you begin using the Helix Client and Server Software Development Kit, you should be aware of the following:

- The Helix Client and Server Software Development Kit does not contain any binary files. Helix executable files and libraries are installed with RealPlayer and Helix Universal Server. These files can be found in different locations, depending on the operating system being used. For more information, see Appendix F, as well as Chapter 3.

- The Helix Client and Server Software Development Kit APIs are supported only in versions 6.0 and later of the RealSystem product line, and are not compatible with 5.0 or earlier versions of RealSystem. As denoted in “Appendix F: SDK Organization”, some of the samples supplied with this SDK require software versions higher than RealSystem 6.0.

Audience for This Guide

This guide is intended for experienced software developers who are interested in extending the capabilities of the Helix Universal Server or a Helix client (such as RealPlayer) by adding or modifying various components. These components may include server or client plug-ins, or services such as audio or network services.

The Helix Client and Server Software Development Kit is aimed at two different audiences of developers: proprietary component developers and Helix community developers.

Proprietary component developers are typically designing custom, proprietary solutions around Helix architecture products, in many cases for commercial use. The only tools provided by RealNetworks for this type of component development is the Helix Client and Server Software Development Kit.

In contrast, Helix community developers are designing components through the Helix community Web site (www.helixcommunity.org). Components designed using the Helix community’s RealNetworks Public Source License or RealNetworks Community Source License might contain some proprietary parts, but other portions will become part of the Helix DNA architecture. The tools provided by RealNetworks for these types of components include the Helix Client and Server Software Development Kit (for reference and a possible leverage point), the Helix DNA source, and the Helix community mailing lists and Internet relay chat (IRC).

**For More Information:** See your Helix license agreement for information on developing and distributing your components.
How This Guide is Organized

This developer’s guide contains the following chapters and appendices.

Chapter 1: Installation and Development Tools
Find out how to install the required Helix Universal Server and Helix client binaries that contain entry functions used by the Helix Client and Server Software Development Kit. In addition, this chapter lists the tools needed to compile the SDK samples on Windows, UNIX, or a Macintosh.

Chapter 2: Helix DNA Components
This chapter discusses the underlying architecture of Helix Universal Server and Helix clients, and the different ways the server and client communicate.

Chapter 3: Developing Plug-ins
If you are designing new plug-ins for either Helix Universal Server or a Helix client, this chapter provides you with a detailed description of how to construct a plug-in for the Helix DNA architecture.

Chapter 4: File Handling
This chapter describes how to create file system and file format plug-ins for either Helix Universal Server or a Helix client. A file system plug-in handles low-level I/O operations on a file, and a file format plug-in converts a native file into a stream of packets.

Chapter 5: Data Conversion and Reversion Plug-ins
This chapter demonstrates how to perform a simple encryption scheme on a stream. The encryption takes place using the data conversion plug-in on Helix Universal Server, and decryption is done using the data reversion plug-in on a Helix client.

Chapter 6: Network Services
Network services supplies the underlying network capabilities of Helix Universal Server and the Helix client. Use the information in this chapter to enhance and modify the network capabilities of your component design.

Chapter 7: Status Codes and Errors
Both Helix Universal Server and Helix client generate status codes and error messages. This chapter describes how to use the SDK to modify your components to handle these status and errors messages.

Chapter 8: Live Broadcasting
Refer to this chapter if you are interested in modifying the live broadcasting features of Helix Universal Server. This chapter covers broadcast plug-ins, remote broadcast services, and the RealText and RealPix broadcast features.

Chapter 9: Managing the Server Registry
This chapter discusses how to design components that add properties to Helix Universal Server’s registry, and describes how to create a monitor plug-in to monitor activity in the server’s registry.
Chapter 10: Restricting Access
There are several ways to restrict user access to content located on Helix Universal Server. This chapter describes how to provide authentication for clients requesting content, and shows how to use allowance plug-ins to restrict specific types of content.

Chapter 11: Adaptive Stream Management
This chapter describes how to use Adaptive Stream Management rules that provide Helix Universal Server with a means of making intelligent decisions about how to deliver data packets efficiently and robustly.

Chapter 12: Top-Level Client
Use the information in this chapter to modify the Helix client’s user interface or its communication with the client core and other components, such as rendering plug-ins Helix.

Chapter 13: Rendering Plug-In
With a rendering plug-in, you can change the way the client interprets a stream of Helix packets. You can also modify audio services and manage client sites from within a rendering plug-in.

Chapter 14: Client Authentication
This chapter describes how to retrieve user name and password information from a Helix client user.

Appendix A: Interface List
This appendix provides descriptions of the COM interfaces and methods found in the header files supplied with the Helix Client and Server Software Development Kit.

Appendix B: Structure List
This appendix provides descriptions of the structures found in the header files supplied with the Helix Client and Server Software Development Kit.

Appendix C: Function List
This appendix provides descriptions of the functions found in the header files supplied with the Helix Client and Server Software Development Kit.

Appendix D: Return Values
This appendix provides descriptions of the return values specifically used by Helix Universal Server and Helix clients.

Appendix E: RealMedia File Format (RMFF) Reference
This appendix discusses the RealMedia File Format (RMFF), a standard tagged file format that uses four-character codes to identify file elements.

Appendix F: SDK Organization
This appendix describes each of the header files and samples included with the Helix Client and Server Software Development Kit.
Conventions in This Manual

The following table explains the key notational conventions used in this manual.

<table>
<thead>
<tr>
<th>Convention</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>emphasis</td>
<td>Bold text is used for in-line headings, user-interface elements, URLs, and e-mail addresses.</td>
</tr>
<tr>
<td>terminology</td>
<td>Italic text is used for technical terms being introduced, and to lend emphasis to generic English words or phrases.</td>
</tr>
<tr>
<td>syntax</td>
<td>This font is used for fragments or complete lines of programming syntax (markup).</td>
</tr>
<tr>
<td>syntax emphasis</td>
<td>Bold syntax character formatting is used for program names, and to emphasize specific syntax elements.</td>
</tr>
<tr>
<td>variables</td>
<td>Italic syntax character formatting denotes variables within fragments or complete lines of syntax.</td>
</tr>
<tr>
<td>[options]</td>
<td>Square brackets indicate values that you may or may not need to use. As a rule, when you use these optional values, you do not include the brackets themselves.</td>
</tr>
<tr>
<td>choice 1</td>
<td>choice 2</td>
</tr>
<tr>
<td>...</td>
<td>Ellipses indicate nonessential information omitted from examples.</td>
</tr>
</tbody>
</table>

Additional Helix Resources

In addition to this manual, you may need the following RealNetworks documentation resources, available at [http://service.real.com/help/library/index.html](http://service.real.com/help/library/index.html):

- **RealNetworks Production Guide**
  
  This manual covers the basics of creating a streaming multimedia presentation. It explains how to do the following:
  
  - Determine the appropriate streaming bandwidth for a presentation.
  - Organize a presentation with Synchronized Multimedia Integration Language (SMIL).
  - Write RealPix and RealText markup.
  - Produce Macromedia Flash animation for streaming.
  - Play a presentation in a Web page instead of in RealPlayer.
  - Optimize audio and video.

- **Helix Universal Server Administration Guide**
  
  The basic reference for the Helix Universal Server administrator, this manual explains how to set up, configure, and run Helix Universal Server to stream multimedia presentations. You need this manual only if you are running Helix Universal Server yourself.

- **Tools manuals**
  
  RealNetworks tools enable you to encode RealAudio and RealVideo files and to broadcast content live. For background information on recording high-quality source files, refer to *RealNetworks*
Production Guide. Then follow the instructions provided with your encoding tool to convert your source files to RealAudio or RealVideo.

**Helix Community Web Site**

The Helix community is a collaborative effort between RealNetworks, independent developers, and leading companies to create and extend the Helix DNA platform, the first open and comprehensive platform for digital media delivery. This community enables companies, institutions, and individual developers to access and license the Helix platform source code to build Helix-powered encoder, server, and client products and other media applications for both commercial and non-commercial use. To learn more about the Helix community, visit this Web page:

- [https://www.helixcommunity.org/](https://www.helixcommunity.org/)

**Technical Support**

For technical support with the Helix Client and Server Software Development Kit, please e-mail the following address:

supportsdk@real.com

For general information about RealNetworks technical support, visit the following Web site:

[http://service.real.com/](http://service.real.com/)
COMMON PLATFORM FEATURES

The following chapters introduce you to Helix concepts. They explain the basics of using this SDK to develop client or server plug-ins. They also discuss common platform services that a Helix server or client can use.
The Helix Client and Server Software Development Kit contains a set of header files and samples that demonstrate how to develop the plug-ins and services described in this guide. Before you begin compiling and using these samples, you must install some required binary files that contain the entry functions to either Helix Universal Server or the Helix client. This chapter describes how to install and configure the required components, and includes information on the tools you use to compile the SDK samples.

### Installing and Configuring Helix Universal Server

You need to install Helix Universal Server when developing server plug-ins and services with this SDK. Helix Universal Server binary files are not included with this SDK, but free downloads of the basic Helix Universal Server are available at [http://www.realnetworks.com/products/](http://www.realnetworks.com/products/). Helix Universal Server requires a license key to run. You receive a license key by e-mail based on the information you provided when downloading Helix Universal Server.

**For More Information:** For instructions on configuring and running Helix Universal Server, see *Helix Universal Server Administration Guide*, which is available in HTML format with your downloaded Helix Universal Server. Documentation is also available at [http://service.real.com/help/library/servers.html](http://service.real.com/help/library/servers.html).

Helix Universal Server works with any Web server that supports configurable MIME types. Setting the correct MIME types in the Web server causes a Web browser to play a media clip with RealPlayer rather than download the file to the browser. In addition to defining your own data types in your Web server, make sure that the following are defined:

- `application/smil` for files with a `.smil` extension
- `audio/x-pn-realaudio` for files with a `.ra`, `.rm`, or `.ram` extension
- `audio/x-pn-realaudio-plugin` for files with a `.rpm` extension

**For More Information:** The procedure for configuring MIME types varies from one Web server to another. For more information, refer to your Web server documentation.

### Installing and Configuring a Helix Client

Helix includes two applications that you can use to test your streaming data type plug-ins: RealPlayer and TestPlay. By testing plug-ins using both RealPlayer and TestPlay, you may find problems in one environment that do not appear in the other.

RealPlayer is a standard Helix client available on Macintosh, Windows, and UNIX platforms. RealPlayer is not included with this SDK, but free downloads are available at [http://www.real.com](http://www.real.com).
Use RealPlayer to test your streaming data type. It includes a Netscape plug-in and an ActiveX control (Windows only) that enable you to play presentations in the Web browser without opening RealPlayer as a separate application.

Note: Because Helix Universal Server can stream data to applications other than RealPlayer, this documentation uses “Helix client” to refer generically to RealPlayer, a media player based on the Helix client core, a Web browser, or any other application that receives Helix Universal Server streams.


TestPlay Installation

TestPlay consists of the RealPlayer core with a command line interface. Use it to test streaming data types in a client environment other than RealPlayer. You can use TestPlay, for example, to mimic the delivery of streaming data directly within a Web browser. Because TestPlay demonstrates how a client application uses the Helix Transport Engine (client “core”), you can use TestPlay as the basis for embedding Helix client functionality in an application.

TestPlay is provided as source code in the Helix Client and Server Software Development Kit, and is located in helixsdk/source/samples/clientapps/testplay. Because TestPlay uses the same transport engine, plug-ins, and codecs as RealPlayer, it can be installed in the same location as RealPlayer. After you have installed the latest version of RealPlayer for your platform, compile TestPlay in the RealPlayer directory.

For More Information: For general build instructions, see “Building a Sample Plug-in” in Volume 1, on page 43.

Windows Component Locations

Running the RealPlayer for Windows installer does the following:

• Installs RealPlayer at the following default location:
  c:\Program Files\Common Files\Real\n
• Installs common DLLs in the following directory:
  c:\Program Files\Common Files\Real\Common

• Installs plug-ins in the following directory:
  c:\Program Files\Common Files\Real\Plugins

• Installs codecs in the following directory:
  c:\Program Files\Common Files\Real\Codecs

• Installs the run-time version of RealPlayer in the following directory:
  C:\Program Files\Real\RealPlayer

• Configures Web browsers to open RealPlayer whenever they receive a file with the .ram or .rpm extension.
• Stores the RealPlayer subdirectory preference settings in the Windows registry, under the following keys:

HKEY_CLASSES_ROOT
\Software
\RealNetworks
\Preferences
\DT_Common
\DT_Plugins
\DT_Codecs

Make sure that RealPlayer was installed correctly by choosing the File>Open command and then opening (and playing) any .rm or .ra files you have available.

Using the RTP Packet Format

When receiving streams from Helix Universal Server, RealPlayer uses the RealNetworks RDT packet format, which enables features such as SureStream RealVideo. RealPlayer fully supports the standards-based RTP packet format, however, and uses that format when communicating with an RTP-based server. Optionally, you can configure RealPlayer to use the RTP format (rather than RDT) when receiving files from Helix Universal Server. To do so, add the following registry key, setting the value to 1:

HKEY_CLASSES_ROOT
\Software
\RealNetworks
\RealPlayer
\6.0
\Preferences
\UseRTP

Note: RealPlayer can use the RTP data transport format only when a file’s data type has a public RTP payload type. For example, .rm files do not have public RTP payload types. These files are streamed using RDT, even if the RTP registry key default value is set to 1.

Macintosh Component Locations

Running the RealPlayer for Macintosh installer (from http://www.real.com/products/) does the following:

• Installs plug-ins in the following directory:

/RealPlayer.app/Contents/MacOS/Library/Plugins

• Installs codecs in the following directory:

/RealPlayer.app/Contents/MacOS/Library/Codecs

• Installs common files in the following directory:

/RealPlayer.app/Contents/MacOS/Library/Common

If you have an earlier version of RealPlayer installed, the installation overwrites older versions of the RealPlayer files, but it does not delete plug-ins or codecs you have placed in the folder.
Make sure that RealPlayer was installed correctly by choosing the **File>Open** command and then opening (and playing) any .rm or .ra files you have available.

### UNIX Component Locations

Running the RealPlayer 8 for UNIX installer does the following, assuming RealPlayer is installed in its default location:

- Installs plug-ins in the following directory:
  
  ~/RealPlayer8/Plugins

- Installs codecs in the following directory:
  
  ~/RealPlayer8/Codecs

- Installs common files in the following directory:
  
  ~/RealPlayer8/Common

If you have an earlier version of RealPlayer for UNIX installed, the installation overwrites older versions of files in the RealPlayer8 directory, but it does not delete plug-ins or codecs you have placed in the directory.

Make sure that RealPlayer for UNIX was installed correctly by choosing the **File>Open File** command and then opening (and playing) any .rm or .ra files you have available.

### Windows Development Tools


➤ **To compile files using Visual C++:**

1. Use an unzipping utility (such as WinZip) to uncompressed either the debug or release versions of the makefiles for your computer. All of the compressed makefiles are located in `\helixsdk\makefiles`.

2. Open Microsoft Developer Studio.

3. On the **File** menu, click **Open Workspace**.

4. In the **Files of type** box, click **Makefiles (*.mak)**. Note that Visual C++ “make” files always have the .mak extension, as in the following example:

    `\helixsdk\source\samples\datatype\audio_renderer\Makefile.mak`

5. Click the .mak file in the desired `\samples` subfolder, and then click **Open**.

6. In the **Select Default Project Configuration** box, select a Release build or a Debug build.

7. Click the **Build** button, or click **Build filename** on the **Build** menu.

   Or, to build the file by using Visual C++ from the command line, enter the following command:

   ```
   nmake /f "win32-i386.mak" CFG="configuration"
   ```

   For example:

   ```
   nmake /f "win32-i386.mak" CFG="audio_renderer - Win32 Debug"
   nmake /f "win32-i386.mak" CFG="audio_renderer - Win32 Release"
   ```
Note: You can also compile all of the source code in the Helix Client and Server Software Development Kit at once by using the makefiles located in the \helix\sdk\source directory.

UNIX Development Tools

Helix SDK sample files were test-compiled with GNU’s gcc for UNIX. The UNIX makefile for each sample file is located in the main directory of each sample. To build the sample with gcc:

1. Use an unzipping utility (such as unzip) to uncompress either the debug or release versions of the makefiles for your computer. All of the compressed makefiles are located in /helixsdk/makefiles.

2. Use the following command in the directory of the sample you want to compile:

   ```make -f Makefile```

Note: You can also compile all of the source code in the Helix Client and Server Software Development Kit at once by using the makefiles located in the /helix/sdk/source directory.


Macintosh Development Tools

For the Macintosh, Helix SDK sample files were test-compiled with Metrowerk's CodeWarrior Pro 8.0 with the CodeWarrior 8.2 update.

For More Information: [http://www.metrowerks.com](http://www.metrowerks.com)

To compile files using CodeWarrior Pro:

1. Use an unzipping utility to uncompress either the debug or release versions of the makefiles for your computer. All of the compressed makefiles are located in /helixsdk/makefiles.

2. Open the CodeWarrior project file located in the individual sample subfolder within the /samples folder.

3. Build the project. Be sure to include the files MacHXMem.cp and HXMM.lib in the project for all Macintosh plug-ins.

4. Move the shared library into the /RealPlayer.app folder.

Helix Client and Server Software Development Kit Interface Name Change

To make all the APIs in the Helix DNA platform consistent, many of the interface names, result codes, structures, and constants originally used in the RealSystem SDK were changed to reflect the new Helix architecture. To help in updating existing applications from the RealSystem SDK to the Helix Client and Server Software Development Kit, a set of tools have been added to the SDK that aid in the process of filtering existing source code to use the new Helix architecture names.
The tools provided in the /helixsdk/tools directory are written in Python script. Python is a language used by the build system so you probably have it installed already. If you don’t, it can be obtained at [http://www.python.org](http://www.python.org). Python 2.2 is recommended to run the scripts. The Python scripts included in the /helixsdk/tools directory search your source code for RealSystem SDK interface names and update them to Helix Client and Server Software Development Kit interface names.

The /helixsdk/tools directory contains a set of tools that provide an automated means of updating to the Helix architecture names. The utilities included in this directory are:

- **filtermod**
  A Python script that acts as a small wrapper script around namereplace.py to provide a standard set of file editing operations on a set of directories and files.

- **namereplace.py**
  A Python script that traverses a directory tree and edits files matching specified file masks according to old to new mappings from a “dictionary” file. Run namereplace.py --help for more information.

  **Note:** When running this script against the standard dictionaries, it is normal to get some “Dup substitution value” warnings. This just means that there are two different “old strings” that map to the same “new string.” This can happen if, for instance, the name for something changed a second time (so there is a mapping to the new name for both the original name and the second name) or if multiple old copies of the same name were collapsed into one.

- **findumake.py**
  A Python script that traverses a directory tree looking for files used as “umake files” for the `umake` command that generates make files. This includes files whose names match specific patterns (for example, `Umakefil` or `*.pcf`) and specifically excludes files whose names match specific patterns (for example, `Makefile` or `*.cpp`). If files aren’t specifically included or excluded based on their names, the first part of the file is scanned for frequently-occurring strings (for example, “project.”). Then it outputs a list of the files it finds. It is used here to locate all the umake files for editing related to renamed header or source files. Run python findumake.py for details.

The /helixsdk/tools directory also contains a pair of text files that include lists of old RealSystem SDK names and their replacements. The text files included in this directory are:

- **dictionary.txt**
  Maps the old RealSystem SDK names to the new Helix Client and Server SDK names.

- **renames.txt**
  Maps the old RealSystem SDK file names and directory locations to the new Helix Client and Server SDK file names and directory locations.

All of these Python scripts and text files must be on your hard drive before they are run.

To execute the name change, change directories to the RealSystem SDK home directory and use the following syntax:

```bash
python <path>filtermod
```

where `<path>` is the absolute path name to the /helixsdk/tools directory.
**Note:** If Python is not in your path, you will need to type the full path to the location of the Python executable.
Chapter 2: HELIX DNA COMPONENTS

Helix technology provides a complete platform for streaming data types and building streaming media applications. Helix can stream virtually any data type, including audio, video, images, text, and animation, from any data source, whether a local file system, a database, or a live broadcast. This chapter introduces you to Helix components, and explains the basic programming concepts required to use the Helix Client and Server Software Development Kit.

Helix Components

Helix includes Helix Universal Server and the Helix client engine, which is the basis for RealPlayer. The open Helix architecture enables you to develop plug-ins that extend the capabilities of Helix Universal Server or a Helix client. Helix Universal Server and Helix clients also support open streaming protocols, making it possible for them to interoperate with other standards-based data-streaming systems.

The Helix Client and Server Software Development Kit (SDK) provides developers with the public interfaces with which to extend and customize Helix. RealNetworks designed this SDK for developers who need to do any of the following:

• Stream a new data type
  To stream a data type to RealPlayer, Web browsers, or other applications, you build plug-ins that enable Helix Universal Server to stream the data type and enable Helix clients to render it. RealNetworks will help you make your plug-ins available for download by anyone using RealPlayer.

• Build a Helix client
  The public interfaces for the Helix client enable you to incorporate Helix capabilities into your application. Then, with the appropriate plug-ins installed, your application can render streaming data types just like RealPlayer.


• Customize Helix Universal Server
  The Helix Client and Server Software Development Kit enables you to create logging and monitoring tools for Helix Universal Server, or to implement data streaming from sources such as databases. You simply build the appropriate plug-in.

Helix Universal Server

Helix Universal Server runs under UNIX or Windows 32-bit operating systems. It supports broadcasting by multicast or unicast, using TCP/IP or UDP/IP. The open architecture of Helix Universal Server provides useful features such as the following:

- The XML-based configuration file enables you to control basic Helix Universal Server properties and create your own customized properties loaded into the Helix Universal Server registry. See “Chapter 9: Managing the Server Registry” beginning in Volume 1, on page 119.
- Authentication lets you customize Helix Universal Server to verify any connection or file request against an encrypted password list. See “Chapter 10: Restricting Access” beginning in Volume 1, on page 131.

Tip: A sample server monitor plug-in is provided in source code as /source/samples/server/monitor/monitor.cpp. For more information, see “Modifying the Monitor Plug-in Sample Code” in Volume 1, on page 127.

Helix Universal Server Architecture

Helix Client

As with Helix Universal Server, the Helix client is based on an open architecture that enables you to customize client features or add client functionality to an application. In “Chapter 12: Top-Level
Client” beginning in Volume 1, on page 153, you will find sample code and documentation that explain how to add Helix client functionality to an application. Helix clients also provide the following functionality:

- Helix clients include Audio Services, RealNetworks’ state-of-the-art technology for manipulating and mixing audio streams. See “Audio Services” in Volume 1, on page 171.
- Site windowing enables clients to render visual data with a minimum of platform-specific code. See “Sites (Windowing)” in Volume 1, on page 183.
- With the Network Services described in Chapter 6, clients can make network connections.

**Helix Client Architecture**

![Helix Client Architecture Diagram]

**Client/Server Protocol Support**

When communicating with RealPlayer, Helix Universal Server by default uses Real-Time Streaming Protocol (RTSP) as its control protocol and RealNetworks RDP as its proprietary packet protocol. Helix also supports the standards-based RTP packet protocol, enabling Helix Universal Server and Helix clients to interoperate with RTP-based servers or clients. Both Helix Universal Server and the RealPlayer client also support RealNetworks PNA proprietary protocol for backwards compatibility.
with earlier versions of RealSystem. The following table shows which client types and protocols you use with particular versions of Helix Universal Server and with RTP-based servers.

<table>
<thead>
<tr>
<th>Server</th>
<th>Client</th>
<th>Control protocol</th>
<th>Packet protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td>RealServer G2 and later</td>
<td>RealPlayer G2 and later</td>
<td>RTSP</td>
<td>RDP</td>
</tr>
<tr>
<td>RealServer G2 and later</td>
<td>RTP-based client</td>
<td>RTSP</td>
<td>RTP</td>
</tr>
<tr>
<td>RTP-based server</td>
<td>RealPlayer G2 and later</td>
<td>RTSP</td>
<td>RTP</td>
</tr>
<tr>
<td>RealServer G2 and later</td>
<td>RealPlayer 3.0 through 5.0</td>
<td>PNA</td>
<td>PNA</td>
</tr>
<tr>
<td>RealServer 3.0 through 5.0</td>
<td>RealPlayer G2 and later</td>
<td>PNA</td>
<td>PNA</td>
</tr>
</tbody>
</table>

SMIL Markup Language

Multiclip presentations are organized with Synchronized Multimedia Integration Language (SMIL), a standards-based markup language. A SMIL file lists the locations of the media clips (video, audio, animation, text, and so on) that make up the presentation, specifying how and when each media file plays. You can use a SMIL file to stream multiple tracks simultaneously, stream tracks in a specific sequence, and stream different clips based on feedback from the client. The World Wide Web Consortium (W3C) has released the SMIL 2.0 specification as a W3C Recommendation.


Plug-ins

The heart of Helix is the plug-in architecture, which enables Helix Universal Server to stream any data type. It also enables you to customize Helix Universal Server features. On Windows, Helix plug-ins are 16-bit or 32-bit DLLs. On UNIX and Macintosh, they are shared libraries. Because Helix provides several cross-platform services, the plug-ins you develop require little or no code specific to any given operating system. With this SDK, you can build the following types of plug-ins:

- File format plug-in
  This plug-in converts a certain data type to packets that Helix Universal Server can stream. Although Helix Universal Server uses file format plug-ins primarily for streaming, Helix clients also use them to play local files. See Chapter 4 for more information.

  Note: Because Helix Universal Server and Helix clients share the same Helix architecture, plug-ins developed for Helix Universal Server work for Helix clients as well, with no additional coding required.

- Rendering plug-in
  This plug-in receives streamed packets and renders them for playback on the Helix client computer. Every file format plug-in has a corresponding rendering plug-in. For more on rendering, see Chapter 13.
CHAPTER 2: Helix DNA Components

• File system plug-in
Helix Universal Server and Helix clients use a file system plug-in to read files from a data source. Because both the server and the clients include plug-ins for reading from local disks, you do not need to build a file system plug-in unless you want to customize the server or the clients to read from an alternative data source such as a database. For details, see Chapter 4.

• Broadcast plug-in
A broadcast plug-in converts a live data source to a Helix packet stream. Although you can build your own broadcast plug-in, Helix includes remote broadcast libraries that tie an encoder application to Helix Universal Server’s standard broadcast plug-in. Thus, you can easily broadcast live audio or video data, as well as RealText or RealPix.

For More Information: See “Broadcast Plug-in” in Volume 1, on page 91. For more on remote broadcast features, see “Remote Broadcast Services” in Volume 1, on page 97, “RealText Broadcast” in Volume 1, on page 105 and “RealPix Broadcast” in Volume 1, on page 110.

• Monitor plug-in
As described in “Creating a Monitor Plug-in” in Volume 1, on page 124, a monitor plug-in can retrieve system information from the Helix Universal Server registry. This includes, for example, the number of clients currently connected and the URL requested by each client. You can also define new properties stored in the Helix Universal Server registry.

• Allowance plug-in
Allowance plug-ins, which Chapter 10 covers, allow only certain requests or actions from a Helix client.

Streaming Files from Helix Universal Server
The most common example of Helix in action is the streaming of files from Helix Universal Server to RealPlayer when the user clicks a link in a Web page. The following illustration explains the steps that take place to make this happen.

File Streaming from Helix Universal Server to RealPlayer
Note: All communication to Helix Universal Server comes from RealPlayer. Neither the Web browser nor the Web server communicates directly with Helix Universal Server.

1. In a Web page, a user clicks a link to a media clip. The Web page author has configured the link to point to Helix Universal Server, use Helix Universal Server’s Ramgen utility, and communicate through HTTP.


2. The Helix Universal Server Ramgen utility generates a Ram file with the extension .ram (for presentations played in RealPlayer) or .rpm (for presentations played in the Web page). The Ram file specifies the location of the media clip.

3. The Ram file is passed back to the Web browser.

4. The Ram file causes the Web browser to open RealPlayer.

5. RealPlayer uses the URL in the Ram file to request the media clip from Helix Universal Server. (During product installation, RealPlayer is automatically configured to behave as a helper application that opens whenever the Web browser receives a Ram file.)

Note: All communication to Helix Universal Server comes from RealPlayer; neither the Web browser nor the Web server communicates with Helix Universal Server directly.

6. Based on the URLs of the requested media files, Helix Universal Server determines which file system plug-ins to use. In most cases, the files are on a local disk and Helix Universal Server uses its standard file system plug-in. If a file is in a data source such as a database, however, Helix Universal Server loads a different file system plug-in. Because of its open, extensible file system plug-in architecture, Helix Universal Server can easily serve files stored in any type of data source in any location.

7. The file system plug-in locates the requested media clip.

8. The file system plug-in creates file objects that can read the files’ data. These are standard system objects that use Helix interfaces.

9. Based on the files’ Multipurpose Internet Mail Extensions (MIME) types, Helix Universal Server determines the file format plug-ins to use. Each file format plug-in is designed to create streaming packets for a specific data type.

10. The file format plug-ins interact with the file objects to get file data, passing stream headers and stream packets to Helix Universal Server.

11. Helix Universal Server streams the packets to RealPlayer using RTSP/RDP.

12. Based on the streams’ MIME types, RealPlayer loads the appropriate rendering plug-ins (the counterparts to the file format plug-ins) and passes them the data to be rendered. Using Helix interfaces, rendering plug-ins display visual data in RealPlayer windows or play back audio data using Audio Services.

The flexibility of Helix leaves room for many variations on this basic scenario:
• RealPlayer includes a Netscape plug-in and ActiveX control that can play presentations in the browser without opening RealPlayer as a separate application.

• Instead of streaming to RealPlayer, Helix Universal Server can stream to any application that has built-in Helix client functionality.

• Helix Universal Server can stream to any RTP-based client, not just clients built on the Helix architecture.

• Because Helix clients have built-in support for RTP, they can receive streams from any RTP-based server.

**Delivering Multiple Streams**

Multimedia presentations streamed by Helix Universal Server typically consist of more than one data type. Each data type can have more than one stream. The Audio Video Interleaved (AVI) format, for example, uses separate streams for video and audio data. The sources for streamed data can be separate files specified within a SMIL file. Or they can be combined in a single file of a container data type, such as RealMedia File Format (RMFF).

The following illustration shows three files that make up a multimedia presentation. Files A and C each contain a visual data type rendered in a client window. File B contains audio data played on the client computer’s sound system. The Helix client opens a separate rendering plug-in for each data type, ensuring that the three parts stay synchronized with their single timeline during playback.

**Communicating with HTTP**

Helix clients can receive files from a Web server. As shown in the following illustration, clients have an HTTP file system plug-in that enables them to communicate with Web servers. Clients also use the file format plug-in for the streamed data type. Because Helix Universal Server and Helix clients share the same architecture, file format plug-ins work on either the server or client.
HTTP delivery provides a reasonable method of downloading short clips from a Web server using the HTTP protocol. It is not as robust as Helix Universal Server streaming, however, and it lacks features such as seeking. When the user seeks forward in the presentation timeline, for example, the client stops rendering the data type but continues to process the data as it comes in at its normal rate. Rendering resumes as soon as the streamed data reaches the timeline position designated by the seek.

Playing Local Files

A Helix client can also play back files on a user’s local computer. The client uses its standard file system plug-in to access the local computer, and it uses file format and rendering plug-ins for the specific file data type to play each file. As noted previously, file format plug-ins work on either the server or client side.

Client Local File Playback

COM Development Model

Helix is based on the Component Object Model (COM) jointly developed by Microsoft Corporation and Digital Equipment Corporation. Helix components use the COM IUnknown::QueryInterface method to expose their interfaces. The Helix COM-style interfaces begin with the prefix “IHX.” Helix does not employ all aspects of COM, however. It implements a subset of COM functions to provide cross-platform operation without requiring Windows libraries or Windows-emulation code on UNIX and Macintosh platforms. Helix thus eliminates the need for heavyweight Windows components like the registry and the COM and object linking and embedding (OLE) run-time libraries. The following sections describe how Helix diverges from COM.


Using IUnknown::AddRef and IUnknown::Release

Because Helix components are often utilized by other Helix components, components must correctly implement reference counting. The COM methods IUnknown::AddRef and IUnknown::Release control reference counting and therefore determine each object’s lifetime. The following rules describe when Helix components need to call IUnknown::AddRef and IUnknown::Release on objects:

- The following functions call an IUnknown::AddRef on objects before returning them. When a component finishes with the object, it must perform an IUnknown::Release on that object:
  - HXCreateInstance
  - IUnknown::QueryInterface
• IHXFileSystemObject::CreateFile
• IHXFileSystemObject::CreateDir
• IHXCommonClassFactory::CreateInstance

• If a component receives an object as a parameter of a function call, the component must perform an IUnknown::AddRef on the object. When the component finishes with the object, it must perform an IUnknown::Release on the object.

• If a component creates an object using the C++ new operator, the component must call IUnknown::AddRef on the object and then release the object using IUnknown::Release when it is finished.

Here are some examples:

• File format plug-ins must perform an IUnknown::AddRef on the file object they are passed by IHXFileFormatObject::InitFileFormat. They must release the file object by calling IUnknown::Release when they are shut down or when they are finished playing the stream.

• Users of IHXFileSystemManager must perform an IUnknown::AddRef on the object passed to them by IHXFileSystemManagerResponse::FileObjectReady. They must release the object using IUnknown::Release when they are finished with it.

• Users of IHXFileSystemManager that call IUnknown::QueryInterface to get a different interface inside IHXFileSystemManagerResponse::FileObjectReady do not need to call IUnknown::AddRef on the IUnknown object they are passed except to save it for later use.

Using IHXBuffer to Create Data Buffers

The IHXBuffer interface, implemented by the Helix core and defined in ihxpcks.h, enables Helix components to create data buffers managed through COM reference counting. Typically, Helix components use these buffers to pass data. For example, a file system plug-in passes stream header data to a file format plug-in through a buffer object. (See “Creating Stream Headers” in Volume 1, on page 61 for more information.)

➤ To create a buffer object, a Helix component:

1. Creates the IHXBuffer interface through IHXCommonClassFactory as described in “Creating a Helix Object” in Volume 1, on page 39.

2. Calls IHXBuffer::SetSize to set the buffer size. This performs the memory allocation.

3. Calls IHXBuffer::GetBuffer to obtain a pointer to the buffer object.

4. Uses IHXBuffer::Set to write data to the buffer object.

   Any Helix component can call IHXBuffer::Set to modify buffer data (the IUnknown::AddRef used during object creation means only that the object is not deleted). It is intended, however, that only the buffer creator sets the buffer’s contents. Other components that need to modify the buffer should create a new buffer object. If the file format plug-in needs to modify the contents of a stream header buffer passed to it by the file system plug-in, for example, it should create a new buffer object for the modified data.

5. Passes a buffer pointer to other components as necessary.
6. Releases the buffer object with IUnknown::Release.

When a component receives a pointer to a buffer object, it calls IHXBuffer::GetSize to retrieve the buffer size, and IHXBuffer::Get to retrieve the buffer data.

Using IHXValues to Create Indexed Lists

Like IHXBuffer, the IHXValues interface, also implemented by the Helix core architecture and defined in ihxpackts.h, is widely used. The IHXValues interface enables Helix components to create general lists that pair names with values. For example, when a file format plug-in receives a buffer of stream header data from a file system plug-in, it creates an IHXValues interface that manages the buffer data along with other data needed by the rendering plug-in. Each value in an IHXValues name/value pair is one of the following:

- unsigned long
- pointer to a buffer of arbitrary data
- pointer to a buffer of null-terminated C-string data

To create an IHXValues object, a Helix component:

1. Creates the IHXValues interface through IHXCommonClassFactory as described in “Creating a Helix Object” in Volume 1, on page 39.

2. Calls the IHXValues::SetPropertyULONG32, IHXValues::SetPropertyBuffer, or IHXValues::SetPropertyCString method as many times as necessary to add name/value pairs to the object.

   Note: As with IHXBuffer, it is intended that only the creator of the IHXValues interface sets its contents.

3. Passes an object pointer to other components as necessary.

4. Releases the interface with IUnknown::Release.

The following example, taken from an SDK sample file, illustrates how to create and populate an IHXValues interface that contains an unsigned long and a pointer to an arbitrary buffer:

```cpp
IHXValues* pHeaderObj = NULL;
m_pClassFactory->CreateInstance(CLSID_IHXValues, (void**)&pHeaderObj);
if (pHeaderObj != NULL)
{
    pHeaderObj->SetPropertyULONG32("StreamCount", 1);
    pHeaderObj->SetPropertyBuffer("OpaqueData", pHeaderDataReadFromFile);
    ...
}
```

After receiving a pointer to the IHXValues interface, a Helix component can retrieve a value by name with IHXValues::GetProperty<type>, where <type> is ULONG32, Buffer, or CString. A component can go through the name/value list with IHXValues::GetFirstProperty<type> and IHXValues::GetNextProperty<type>. To remove specific name/value pairs from an IHXValues object, use the IHXValuesRemove interface. The key used by the IHXValuesRemove methods corresponds to the property name used by IHXValues.
CHAPTER 2: Helix DNA Components

Using IHXPacket to Create Stream Packets

Helix components use the IHXPacket interface, defined in ihxpckts.h, to create data packets streamed between Helix Universal Server and its clients. For example, a file format plug-in prepares packets that Helix Universal Server streams to the client. As well, a client’s rendering plug-in can use the system’s back channel to send packets of information back to its file format plug-in. (For more information, see “Sending Back-Channel Packets” in Volume 1, on page 166).

➤ To create a packet, a Helix component:

1. Creates the IHXPacket interface through IHXCommonClassFactory as described in “Creating a Helix Object” in Volume 1, on page 39.

2. Calls IHXPacket::Set to pass the packet a pointer to an IHXBuffer interface and define the packet properties, such as delivery time and Adaptive Stream Management (ASM) rules. The packet properties help Helix stream packets efficiently. See ihxpckts.h for more information on packet properties.

   Note: As with IHXBuffer, it is intended that only the creator of the IHXPacket interface sets its contents. This is critical with packets because more than one renderer may use a packet object.

3. Passes a packet pointer to other components as necessary.

4. Releases the packet interface with IUnknown::Release.

The following example, taken from an SDK sample file, illustrates how to create an IHXPacket interface:

```cpp
IHXPacket* pPacketObj = NULL;
if (m_pClassFactory->CreateInstance(CLSID_IHXPacket, (void**)&pPacketObj))
{
    UINT32 deliveryTime = m_NextPacketDeliveryTime;
    UINT16 streamNo = MY_STREAM_NO;
    UINT8 ASMFlags = HX_ASM_SWITCH_ON;
    UINT16 ASMRuleNo = 0;
    pPacketObj->Set(pPacketDataReadFromFile, deliveryTime, streamNo, ASMFlags, ASMRuleNo);
    ...
}
```

After receiving a pointer to the packet interface, a Helix component, typically a rendering plug-in, retrieves the packet data with IHXPacket::Get. Other methods enable a component to retrieve specific packet properties. A component can also call IHXPacket::IsLost to determine whether the packet has been lost. See “Chapter 13: Rendering Plug-In” beginning in Volume 1, on page 161 for more information.

Operating Asynchronously

Helix components function asynchronously, using response interfaces to return data for calls made to them. For example, a file format plug-in may request file data from a file system plug-in through IHXFileObject. The file system plug-in then uses IHXFileResponse, which the file format plug-in implements, to return the requested data. This asynchronous architecture enables the file format plug-in to perform other actions while the file system plug-in prepares the requested data. Given Helix’s
asynchronous nature, a plug-in must be able to handle any call made to it while it is processing data or waiting for a response from another component.

Sample File Coding Conventions

All Helix sample files use the conventions denoted in the following table.

<table>
<thead>
<tr>
<th>Convention</th>
<th>Designation</th>
</tr>
</thead>
<tbody>
<tr>
<td>pName</td>
<td>Pointer</td>
</tr>
<tr>
<td>bName</td>
<td>Boolean</td>
</tr>
<tr>
<td>nName</td>
<td>Integer</td>
</tr>
<tr>
<td>unName</td>
<td>Unsigned Integer</td>
</tr>
<tr>
<td>ulName</td>
<td>Unsigned Long</td>
</tr>
<tr>
<td>m_Name</td>
<td>Member variable</td>
</tr>
<tr>
<td>zm_Name</td>
<td>Static member variable</td>
</tr>
<tr>
<td>g_Name</td>
<td>Global variable</td>
</tr>
</tbody>
</table>
DEVELOPING PLUG-INS

A plug-in is a shared library that provides a specific service in Helix Universal Server or the Helix client. Examples of services provided by plug-ins in Helix Universal Server include access control, client monitoring and file system access. Examples of services provided by plug-ins in the Helix client include rendering, viewing source, and file system access. In general, plug-ins can be used to extend or customize Helix functionality. This chapter provides guidelines on how to design, build, and test a plug-in.

Designing a Plug-in

Keep the following general points in mind as you develop your plug-ins:

• Keep your plug-ins as simple, self-contained, and universal as possible. Carefully consider the impact on channel opportunities and audience size if, for example, you add requirements for special client APIs or hardware.

• Keep the asynchronous nature of Helix in mind. Your plug-in must be able to handle any call made to it while it is processing data or waiting for a response from another object. Do not code your plug-in so that it expects a specific sequence of events to occur as it interacts with Helix.

• Try to keep plug-in file size under 300 KB uncompressed.

• Design your data type delivery to satisfy all users, from 28.8 Kbps modem users on up. Either keep the streaming bit rate low overall, or design solutions that deliver an acceptable experience to low-end users, and scale with increasing sophistication for users with high-speed connections.

• Design code to run on multiple platforms, including the Windows, Macintosh, and UNIX platforms. Providing such cross-platform solutions makes your product available to the broadest range of customers, allowing you to fully leverage the many platforms on which Helix products are deployed.

• Do not rely on Multimedia Extensions (MMX) or other technologies to improve performance of client playback.

• If you are developing plug-ins for an audio data type, use Helix’s Audio Services, which are described in Chapter 13. These powerful services handle audio mixing, resampling, volume control, and instantaneous sounds.

• Because RealPlayer and Helix Universal Server are localized in many languages, we recommend using a resource table and designing your data type’s interface and documentation for localization. For example, use double-byte characters or Unicode where appropriate.

• Whenever possible, plug-ins should create dynamic strings using the IHXBuffer interface.
• Provide good documentation on your data type and plug-ins. Online format is ideal, as long as users can print pages easily. To minimize client component sizes, though, do not include documentation with the client download. Make it available on a Web site instead.

• Design your content creation tools to consider the bit rate of the stream over time, and to enable designers to make decisions on how to produce the content. A well-designed data type will use the Adaptive Stream Management features to level the bit rate of its stream. If you want to interact with other data types, such as streaming images, text, RealAudio or RealVideo, check to see whether other SDKs enable you to add content creation functionality for a data type to your tools.

**Note:** In this guide, look in each section that describes building a plug-in for any other design considerations specific to the type of plug-in being discussed.

**For More Information:** Before you start to build your plug-in, see “Creating a Plug-in Instance” in Volume 1, on page 36.

### Plug-in Basics

In a standard installation, all plug-ins are stored in the “Plugins” directory under the Helix Universal Server or Helix client installation root directory (this plug-in storage location is configurable). During initialization, the server or client core modules scan this directory and perform the following operations for each installed plug-in:

1. Immediately after the plug-in library is loaded, the plug-in’s entry point function, `HXCreateInstance`, is invoked.

2. Next, the plug-in’s `IHXPlugin::GetPluginInfo` method is executed to obtain the plug-in’s attributes.

3. Finally, the plug-in’s initialization method, `IHXPlugin::InitPlugin`, is invoked.

Once the plug-in is loaded and initialized, it is able to receive requests from the core to perform specific operations. The type of operations performed depends entirely on the interfaces implemented by the plug-in. At a minimum, every plug-in must implement the `IHXPlugin` interface. This interface provides the basic hooks required to enable communication between the plug-in and the core. Plug-ins usually implement at least one other interface in addition to `IHXPlugin`.

The following sections describe the plug-in initialization steps in detail and provide examples on how to implement each of the required interface methods. These sections are presented here in the general order in which they appear in a plug-in.

### Creating a Plug-in Instance

All plug-ins use globally unique identifiers (GUIDs) to determine an object’s interfaces. When you use `IUnknown::QueryInterface` to determine whether an object supports a particular interface, you use the GUID to identify the interface. To use GUIDs in your plug-ins, you must include the following line at the beginning of your code:

```c
#define INITGUID 1 /* Interface ID's */
```

Helix does not use the Windows `CoCreateInstance` function to create plug-in objects. Instead, each Helix plug-in must implement `HXCreateInstance`, a “C-style” entry point that exposes the `IHXPlugin` interface.
The following, taken from a Helix Client and Server Software Development Kit (SDK) sample file, illustrates the entry point:

```c
STDAPI
HXCreateInstance(IUnknown** ppExFileFormatObj)
{
    *ppExFileFormatObj = (IUnknown*)(IHXPlugin*)new CMyExampleFileFormat();
    if (*ppExFileFormatObj != NULL)
    {
        (*ppExFileFormatObj)->AddRef();
        return HXR_OK;
    }
    return HXR_OUTOFMEMORY;
}
```

The code illustrated in this sample returns a pointer to the `IHXPlugin` interface, which is used by the core to determine the plug-in’s characteristics. The `IUnknown::AddRef` method increments the plug-in’s reference counter (described more fully in “Exporting Plug-in Interfaces” in Volume 1, on page 38). A plug-in must increment its reference counter any time it returns an interface pointer. The code then returns `HXR_OK`, which is the standard return code that indicates success.

**Note:** On Windows, the plug-in must export `HXCreateInstance` in its `.def` file.

### Freeing Plug-in Resources

To complete any required cleanup and release any global resources held by the plug-in, the plug-in should implement the `HXShutdown` function. `HXShutdown` is the last function invoked just before the plug-in is unloaded. Generally, the server or client core never unloads any plug-ins that are loaded on application startup. The following example illustrates the shutdown point:

```c
STDAPI
ENTRYPOINT(HXShutdown)(void){
    // Add cleanup code here as required.
    return HXR_OK;
}
```

### Defining Plug-in Attributes

Once the plug-in library is loaded, the Helix core uses the `IHXPlugin::GetPluginInfo` method to retrieve the plug-in’s attributes. These attributes indicate whether the plug-in can be loaded more than once, and include any descriptive information about the plug-in. The descriptive information is displayed in the About box in the application’s user interface, and may also appear in the application’s log file. The following illustrates an implementation of this method:

```c
STDMETHODIMP
CMyExampleFileFormat::GetPluginInfo
(
    REF(BOOL)         bLoadMultiple,
    REF(const char*) pDescription,
    REF(const char*) pCopyright,
    REF(const char*) pMoreInfoURL,
    REF(UINT32)      versionNumber
```
The `bLoadMultiple` variable indicates whether this is a multiload or non-multiload plug-in. The Helix core creates a separate process/thread for each non-multiload plug-in. In addition to this, only a single plug-in instance is created for non-multiload plug-ins. In contrast, multiload plug-ins can be instantiated multiple times. In this example, the plug-in is created as a multiload plug-in.

### Exporting Plug-in Interfaces

After loading a plug-in and obtaining its attributes, the Helix core then tries to determine the plug-in type using the `IUnknown::QueryInterface` method. This method defines all interfaces that are exported or implemented by a plug-in.

Server plug-ins are classified into one of the five following categories based on the type of interface implemented:

- File system plug-ins implement the `IHXFileSystemObject` interface.
- Datatype (that is, file format) plug-ins implement the `IHXFileFormatObject` interface.
- Allowance plug-ins implement the `IHXPlayerConnectionAdviseSink` interface.
- Live broadcast plug-ins implement the `IHXBroadcastFormatObject` interface.
- All other plug-ins are classified as generic plug-ins.

Client plug-ins are classified into one of the four following categories:

- File system plug-ins implement the `IHXFileSystemObject` interface.
- Datatype (that is, file format) plug-ins implement the `IHXFileFormatObject` interface.
- Rendering plug-ins implement the `IHXRenderer` interface.
- All other plug-ins are classified as generic plug-ins.

The following plug-in code segment demonstrates an implementation of the `IUnknown::QueryInterface` method:

```c++
STDMETHODIMP CMyExampleFileFormat::QueryInterface( 
    REFIID interfaceID, 
    void** ppInterfaceObj 
)
{
    // File Format plug-ins MUST be able to be multiply instantiated
    bLoadMultiple = TRUE;

    pDescription  = zm_pDescription;
    pCopyright  = zm_pCopyright;
    pMoreInfoURL  = zm_pMoreInfoURL;
    versionNumber = MY_PLUGIN_VERSION;

    return HXR_OK;
}
```
// By definition all COM objects support the IUnknown interface
if (IsEqualIID(interfaceID, IID_IUnknown))
{
    AddRef();
    *ppInterfaceObj = (IUnknown*)(IHXPlugin*)this;
    return HXR_OK;
}

// IHXPlugin interface is supported
else if (IsEqualIID(interfaceID, IID_IHXPlugin))
{
    AddRef();
    *ppInterfaceObj = (IHXPlugin*)this;
    return HXR_OK;
}

// IHXFileResponse interface is supported
else if (IsEqualIID(interfaceID, IID_IHXFileResponse))
{
    AddRef();
    *ppInterfaceObj = (IHXFileResponse*)this;
    return HXR_OK;
}

// IHXFileFormatObject interface is supported
else if (IsEqualIID(interfaceID, IID_IHXFileFormatObject))
{
    AddRef();
    *ppInterfaceObj = (IHXFileFormatObject*)this;
    return HXR_OK;
}

// No other interfaces are supported
*ppInterfaceObj = NULL;
return HXR_NOINTERFACE;

This code segment exports the IHXPlugin, IHXFileResponse, and IHXFileFormatObject interfaces. It also implicitly exports the IUnknown interface as well, since every interface inherits from IUnknown. Each time an interface pointer is returned, the plug-in’s reference counter is incremented using the IUnknown::AddRef method. In this sample, if any other interfaces are requested, the standard return value HXR_NOINTERFACE is returned to the caller, indicating that the interface is not implemented by the plug-in.

Creating a Helix Object

A Helix component can use the C++ new operator to create objects that it alone manipulates. To create objects passed to other Helix components, however, a component should use IHXCommonClassFactory. When Helix initializes a component, it passes the component a pointer to the system context. The component can then use this pointer to call IHXCommonClassFactory::CreateInstance and create new Helix
objects. The following extract from a Helix Client and Server Software Development Kit sample file illustrates a call to this method:

\[
\text{IHXBuffer}^* \ pMyBuffer = \text{NULL};
\]
\[
\text{m\_pClassFactory\->CreateInstance(\text{CLSID\_IHXBuffer}, (void**)\&pStringObj)};
\]

Plug-ins can obtain a reference to an object implementing the \texttt{IHXCommonClassFactory} interface through the context. For example:

\[
\text{IHXCommonClassFactory}^* \ pMyClassFactory = \text{NULL};
\]
\[
\text{pContext\->QueryInterface(\text{IID\_IHXCommonClassFactory},}
\]
\[
(\text{void**})\&\text{m\_pClassFactory});
\]

**Plug-in Initialization**

Plug-in initialization logic is always implemented in the \texttt{IHXPlugin::InitPlugin} method. In non-multiload plug-ins, the \texttt{IHXPlugin::InitPlugin} method is invoked immediately after the plug-in library is loaded during the server or client initialization. Execution of this method is delayed in multiload plug-ins. In multiload plug-ins, the initialization method is invoked only when an instance of the plug-in is created to service a particular request.

**Note:** Multiload plug-ins can force the core to invoke the \texttt{IHXPlugin::InitPlugin} method immediately after the plug-in is loaded by implementing the \texttt{IHXGenericPlugin} interface and setting the \texttt{bIsGeneric} parameter to \texttt{TRUE}.

The following code segment shows an implementation of \texttt{IHXPlugin::InitPlugin} method:

```cpp
STDMETHODIMP CMyHelloWorld::InitPlugin(IUnknown* pHXCore)
{
    // Determine if the IHXErrorMessages interface is available
    IHXErrorMessages* pErrorMessages = NULL;
    pHXCore->QueryInterface(IID_IHXErrorMessages, (void**)&pErrorMessages);

    // If the interface is supported
    if (pErrorMessages != NULL)
    {
        // Print out "Hello World"
        pErrorMessages->Report(PNLOG_INFO, 0, 0, "Hello World", 0);
        
        /*
        * Note that QueryInterface() takes care of adding a reference to the
        * interface for us. You are however responsible for releasing the
        * reference to the interface when you are done using it by calling
        * the Release() routine.
        */
        pErrorMessages->Release();
    }

    return PNR_OK;
}
```

40
In this sample, the plug-in stores a reference to the context, then queries (IUnknown::QueryInterface) the context to obtain a reference to the error messages interface. The plug-in does not explicitly increment the pErrorMessages reference counter since this is done implicitly by the IUnknown::QueryInterface method. The plug-in then uses the error messages interface to output “Hello World” to the server’s console and error log file (error.log) or, if implemented in a client plug-in, to a popup dialog box. Once the plug-in is finished with the error messages interface, it decrements the object’s reference counter using the IUnknown::Release method.

**Additional Plug-in Initialization**

The Helix Client and Server Software Development Kit also contains the IHXObjectConfiguration interface, an additional interface for initializing plug-ins. This interface is primarily used by authentication plug-ins that require information from the configuration file, and uses services outside the scope of a normal plug-in.

The IHXObjectConfiguration::SetConfiguration method passes in a set of IHXValues mappings. Typically in an authentication plug-in, these values represent the Realm, DatabaseID, and other standard authentication information. Different types of authentication plug-ins may contain special fields, such as Windows NT authentication plug-ins that pass in a Provider.

For example, the server core uses the IHXObjectConfiguration::SetConfiguration method to determine the realm and database ID for authenticating a particular audio clip. If the clip’s realm uses basic authentication, the server core creates an object that supports basic authentication. It then calls IHXObjectConfiguration::SetConfiguration on that object so the object can access the proper realm and database ID for that audio clip.

The IHXObjectConfiguration::SetContext method passes in a global factory object to a plug-in so the plug-in can use services from the main program (that is, the program that uses the plug-in, such as the server or client core) that are not generally available to a plug-in. To access these services, your implementation of the IHXObjectConfiguration::SetContext method must first query (IUnknown::QueryInterface) the context for IHXCommonClassFactory.

The /source/samples/server/authbasic/authbasic.cpp sample file demonstrates one possible implementation of the IHXObjectConfiguration interface.

**Maintaining Plug-in Reference Counters**

Every plug-in must implement the IUnknown::AddRef and IUnknown::Release methods to keep track of the number of interfaces being used in the plug-in. The IUnknown::AddRef method increments the plug-in’s reference counter by one each time a pointer to an interface implemented by the plug-in is requested. The IUnknown::Release method decrements the reference counter when the interface is no longer in use. When the reference counter is decremented to 0, the plug-in object is deleted.

The following code segment demonstrates a standard implementation of the IUnknown::AddRef and IUnknown::Release methods from one of the Helix Client and Server Software Development Kit samples. Virtually all plug-ins implement these methods the same way.
STDMETHODIMP_(UINT32)
CMyExampleFileFormat::AddRef(void)
{
    return InterlockedIncrement(&m_ulRefCount);
}

STDMETHODIMP_(UINT32)
CMyExampleFileFormat::Release(void)
{
    if (InterlockedDecrement(&m_RefCount) > 0)
    {
        return m_RefCount;
    }
    delete this;
    return 0;
}

Note: The InterlockedIncrement and InterlockedDecrement functions ensure that the reference counters are incremented or decremented atomically.

Plug-in Object Constructor and Destructor

In addition to all the methods associated with interfaces implemented by a plug-in, two other methods are generally found in a plug-in—the plug-in object’s constructor and destructor. The following code segment provides an example implementation of these methods as found in one of the Helix Client and Server Software Development Kit samples:

CMyExampleFileSystem::CMyExampleFileSystem(void)
: m_ulRefCount (0),
  m_pClassFactory (NULL)
{
    m_BasePath[ 0 ] = '\0';
}

CMyExampleFileSystem::~CMyExampleFileSystem(void)
{
    if (m_pClassFactory != NULL)
    {
        m_pClassFactory->Release();
        m_pClassFactory = NULL;
    }
}

Enumerating Plug-ins

Your application can use the either the server or client core’s plug-in enumerator interfaces to find all plug-ins or all plug-ins that support a specific interface. This includes rendering, file format, and file system plug-ins. To return the total number of plug-ins supported by the core, call the
IHXPluginEnumerator::GetNumOfPlugins method. Your application can then call
IHXPluginEnumerator::GetPlugin to return an instance of a plug-in.

To determine how many plug-ins expose a specific interface, the call the IHXPluginGroupEnumerator::Init method to pass the enumerator object an interface ID. You can then call
IHXPluginEnumerator::GetNumOfPlugins to get the number of plug-ins that support that interface, and
then IHXPluginEnumerator::GetPlugin to return an instance of a plug-in.

Building a Sample Plug-in

The following sections give pointers on compiling your plug-in. Before you start building your own plug-in, you should test-compile some Helix Client and Server Software Development Kit sample files. The simplest sample file is the “Hello World” sample.

“Hello World” Sample

The Helix Client and Server Software Development Kit includes many sample files you can use as a basis for writing your own plug-ins. The simplest is the “Hello World” plug-in, which prints “Hello World” on the Helix Universal Server console. RealNetworks recommends that, before you start building your plug-ins, you learn the basic Helix features by modifying, compiling, and testing the “Hello World” sample located at /source/samples/client/hello_world/hello_world.cpp.

This sample introduces the IUnknown::QueryInterface, IUnknown::AddRef, and IUnknown::Release methods of COM. It also shows how Helix Universal Server instantiates a plug-in and discovers its features through the IHXPlugin interface. When Helix Universal Server is started, it loads the plug-in as follows:

1. Helix Universal Server calls HXCreateInstance to create a new instance of the plug-in. See “Creating a Plug-in Instance” in Volume 1, on page 36 for more on this method.

2. Helix Universal Server calls IHXPlugin::GetPluginInfo, which returns descriptive information about the plug-in, including its copyright and “more information” URL. The LoadMultiple attribute must be set to TRUE for Helix Universal Server to open multiple instances of the plug-in. See “Defining Plug-in Attributes” in Volume 1, on page 37 for more on this method.

3. Helix Universal Server calls the plug-in’s IHXPlugin::InitPlugin method, passing it a pointer to the Helix Universal Server context. The plug-in uses this pointer to query for IHXErrorMessages on Helix Universal Server and, if it finds that interface, prints “Hello World” on the system console and in the log file. See “Plug-in Initialization” in Volume 1, on page 40 for more on this method.

The sample file contains comments that explain each function. You will want to test the sample file before trying to use the plug-in, as described in the following procedure.

➤ To modify and test a sample:

1. Copy the sample code to a working directory. Optionally, change the sample’s file names and class names.

2. Modify the plug-in description, copyright information, and “more info” URL stored in zm_pDescription, zm_pCopyright, and zm_pMoreInfoURL.

3. Compile and install the plug-in.
For More Information: See “Installing Plug-ins for Testing” in Volume 1, on page 44.

Debugging a Plug-in

You should be able to debug your Helix plug-ins by using your standard debugger. In Visual C++, you can configure the debugging settings in your DLL's Makefile or project to use realplay.exe as your application for debugging client plug-ins. To debug Helix Universal Server plug-ins, include printf statements in your plug-in, and send debug messages to the console through standard out (stdout).

Testing a Plug-in

When you test your plug-ins, do as much “real-world” empirical testing of your data type as possible. Be sure to test under conditions that are less than ideal. For example, try dialing up on 56 Kbps modems, and playing your data type over the Internet from a Helix Universal Server.

The following are general guidelines for testing how your plug-ins work. This is not a comprehensive list, as testing strategies vary depending on the data type.

➤ To test your plug-ins:
1. Verify that all of the file types you are concerned with and the streamed MIME types play correctly.
2. Test to make sure that files of each data type play all the way through successfully.
3. Verify that the window is in the correct default location inside or outside the player console.
4. Ensure that the pause and resume functions work properly.
5. If you implemented the seek function, make sure seek, pause, and resume are all working properly.
6. Ensure that stopping midstream works properly.
7. Verify that everything goes as it should during data buffering.
8. Determine whether running-out-of-memory situations are handled gracefully.
9. Verify that overall performance is acceptable on slower computers.
10. See whether RealPlayer can play several streams at the same time on the same computer. (Using a SMIL layout is the best way to test this.)
11. Test for per-packet memory leaks by playing a long stream (two hours or longer) and verifying that player memory usage does not steadily rise. You can do this by making a Ram file with a sequence of many short streams.
12. Ensure your plug-in can appropriately handle packet loss.

Installing Plug-ins for Testing

Before you begin testing your plug-ins, you must first install the plug-ins in the proper directories.
To install plug-ins for testing:

1. Copy any server-side plug-ins to the plug-ins directory defined by the server configuration file. This includes the file format plug-in for your data type, and may include file system and broadcast plug-ins as well.


2. On the client side, copy your data type’s rendering plug-in and, if testing with a local file, the file format plug-in to the plug-ins directory on the client computer:

   • On Windows, this directory is either C:\Windows\System\Real or C:\Program Files\Common Files\Real\RealPlayer\plugins, depending on which version of client is installed.

   For More Information: See “Windows Component Locations” in Volume 1, on page 16 for more on installation paths.

   • On UNIX, the plug-ins directory is the /RealPlayer8/Plugins subdirectory of the Home directory.

   • On the Macintosh, this is the /RealPlayer.app/Contents/MacOS/Library/Plugins folder.

   For More Information: For more on this default folder, see “Macintosh Component Locations” in Volume 1, on page 17.

3. Copy a file of your data type to a content directory on the Helix Universal Server to test with the server release. Also, copy the file to the client computer to open as a local file.

4. Test your plug-in by playing a simple file of your data type in RealPlayer and TestPlay. Then create more extensive multimedia presentations that display your data type along with other data types in a timed presentation.

Testing with RealPlayer

The following steps describe how to test your plug-ins using RealPlayer.

To test plug-ins with RealPlayer:

1. Start RealPlayer.

2. Use the File>Open File command to open the client’s local file.

3. Use the File>Open File to specify the file’s URL to stream the file from Helix Universal Server.

Testing with TestPlay

With TestPlay, you can test streaming data types in a client environment other than RealPlayer. For example, you can use TestPlay to mimic the delivery of streaming data directly within a Web browser. After installing TestPlay as described in “TestPlay Installation” in Volume 1, on page 16, you can run it from a command line with this syntax:

testplay <URL>

where <URL> is the network or local file to play. Here are some examples:

testplay rtsp://real.myserver.com/clips/newvideo.rm
testplay file:c:\real\rmplayer\welcome.rm

TestPlay opens any required windows and plays the specified presentation. It writes the data from the IHXClientAdviseSink interface to the console to demonstrate the information available to a custom client application.

Testing with Multimedia

After you have successfully tested your plug-ins by streaming and reading a simple file of your data type, create multimedia presentations that include your data type. Then play your presentations using RealPlayer and TestPlay. You define the content of the presentation using the Synchronized Multimedia Integration Language (SMIL).

Testing with HTTP

You should also test to ensure that your data type is delivered properly from a Web server. Although HTTP is not as robust as Real-Time Streaming Protocol (RTSP), it provides a reasonable method to delivery short clips. As shown in the following illustration, RealPlayer must have the file format plug-in and rendering plug-in installed. The HTTP file system plug-in comes with RealPlayer. No plug-ins are used with the Web server, however.

HTTP Delivery

➤ To deliver your data type using HTTP:

1. Define the Multipurpose Internet Mail Extensions (MIME) type of your data type in the Web server.
2. Copy your files, including a SMIL file, to the Web server.
3. In your HTML document, create a hyperlink to the SMIL file. You can use relative or complete paths. If you use complete paths, you must include the host name, as in the following example:
   `<A HREF="http://hostname/file.smil">`
4. Click the HTML hyperlink to start the HTTP delivery.

Plug-in Examples

The examples presented in this chapter contain all the essential elements of a plug-in. However, they are highly simplified and provide little useful functionality. The sample files included with the Helix Client and Server Software Development Kit in the \source\samples directories provide more realistic examples.
Both Helix Universal Server and Helix clients use file system plug-ins and file format plug-ins to access native files and convert the original file format to a series of packets. File system plug-ins handle low-level file I/O operations such as reading and writing, and act as an abstract interface that allows a file format plug-in to access a file. File format plug-ins convert a native file into streams of packets that flow through the Helix platform.

**File System and File Format Plug-in Interaction**

Helix Universal Server and Helix clients use different means of loading and coordinating file system plug-ins and file format plug-ins. The following sections describe these interactions.

**Helix Universal Server Interaction**

The following actions occur when Helix Universal Server receives a request for a file:

1. Helix Universal Server parses the request URL for the mount point, which is the directory or directory alias preceding the file name.

2. Based on its `FSMount` parameter, which pairs mount points to file system plug-ins, Helix Universal Server determines which file system plug-in to use.

   **For More Information:** For information on how Helix Universal Server handles multiple plug-ins for the same mount point, see the `FS Mount` parameter in Helix Universal Server Configuration File Reference.

3. Helix Universal Server initializes the selected file system plug-in, creates a file object, and checks that the file exists on that file system.

4. Helix Universal Server uses the file object to determine the file’s Multipurpose Internet Mail Extensions (MIME) type. If it cannot determine the MIME type through the interfaces, it uses information it received from its file format plug-ins at startup to cross-reference the file extension with the MIME type.

5. Based on the file MIME type, Helix Universal Server loads the appropriate file format plug-in and passes it the file object for the requested file.

6. The file format plug-in then initializes and uses the file object to retrieve file data. If file authentication is used, the file object verifies user access for the file during its initialization stage.

**Helix Client Interaction**

The following actions occur when a Helix client receives a play request:
1. The Helix client parses the URL to determine whether it is a local or network URL request.

2. If it is a local playback request, then:
   a. The Helix client loads the file system plug-in and creates a file object.
   b. The Helix client assumes it is a RAM meta file and validates it by loading the RAM file format and rendering plug-ins, since some web browsers can misname the file extension for RAM files.
   c. If it is not a RAM meta file, then the file format plug-in of the corresponding data type is instantiated based on the MIME types or file extension.

3. If it is a network playback request, then:
   a. The Helix client loads the appropriate protocol object (for either PNA or RTSP).
   b. The client performs the protocol handshaking with the server and sends the play request to the server.

4. After receiving the stream headers, either locally or from the server, the Helix client loads the appropriate rendering plug-ins based on the MIME type specified in the stream headers.

5. The Helix client then passes the data to the rendering plug-in and playback starts.

File System Plug-ins

Helix Universal Server and the Helix client both use file system plug-ins to access the files they serve. Each file system plug-in provides access to a different type of data storage mechanism, such as the computer's local disks or a database. On each request, a file system plug-in creates a file object that components such as file format plug-ins use to access the requested file's data. These system-standard file objects thereby create a virtual file system for accessing file data without regard to data location or storage format.

Helix Universal Server contains a plug-in for accessing files on its local disks. It also includes an HTTP plug-in for accessing files on a Web server and a broadcast plug-in for streaming files from a live source. If you need to access data from a source not accessible through these plug-ins, you need to build a file system plug-in or a broadcast plug-in that handles standard file operations for that data source. You need to write a file system plug-in, for example, to stream files stored in a database.

For More Information: For a list of plug-ins included with the system, see “Appendix F: SDK Organization”. Before you begin building, read the design considerations discussed in “Designing a Plug-in” in Volume 1, on page 35.

Helix clients also use file system plug-ins to access files locally or on a network through their File>Open... commands. Clients contain plug-ins for accessing data on their local disks as well as on HTTP servers. They typically do not need to have other file system plug-ins installed. Because Helix makes file handling identical for server and clients, however, any file system plug-in developed for Helix Universal Server will also work on clients.

Interfaces

A file system plug-in implements the following interfaces:
• IHXPlugin. Header file: hxplugin.h.

Every plug-in implements this interface, which Helix uses to determine the plug-in’s characteristics.

• IHXFileSystemObject. Header file: hxfiles.h.

A file system plug-in must implement this interface, which provides the basic methods that Helix Universal Server uses to instruct it to create file objects.

• IHXPendingStatus. Header file: hxpends.h.

The file system plug-in can implement this interface to provide the Helix client with the status of its pending operations.

File objects created by file system plug-ins typically implement the following interfaces:

• IHXFileObject. Header file: hxfiles.h.

All file objects implement this interface, which gives the system read and write access to a single file or resource. The response interface implemented by file format plug-ins is IHXFileResponse.

• IHXFileExists. Header file: hxfiles.h.

The file object must implement this interface, which Helix Universal Server uses to see whether the requested file exists on the file system. The response interface is IHXFileExistsResponse.

• IHXFileMimeMapper. Header file: hxfiles.h.

Helix Universal Server uses this optional interface to request the MIME type of a file. The response interface is IHXFileMimeMapperResponse.

• IHXFileStat. Header file: hxfiles.h.

This optional interface provides information about a specific file, such as file length and creation time. The response interface is IHXFileStatResponse.

• IHXGetFileFromSamePool. Header file: hxfiles.h.

This interface instructs the file object to create a relative file object. The response interface is IHXGetFileFromSamePoolResponse.

• IHXRequestHandler. Header file: hxfiles.h.

If the file object implements this interface, it can access the request object and modify the request response headers.

Coding the Plug-in

The following sections explain how Helix Universal Server or a Helix client, a file system plug-in, and a file format plug-in use the Helix interfaces to create and use file objects. The sample files included with this software development kit (SDK) illustrate many of these features. Refer to “Appendix A: Interface List” beginning in Volume 2, on page 191 and the Helix Client and Server Software Development Kit header files for more information on function variables and return values.

Note: The order of function calls listed in the following sections provides a generalized explanation and is for illustrative purposes only. Because Helix is asynchronous, your plug-in must be able to handle any call made to it while it is processing data or waiting
Starting Up

When Helix Universal Server or a Helix client is started, it loads each file system plug-in. The server or client performs a series of calls to your plug-in's functions and methods in a specific order.

➤ To provide start-up support in your file system plug-in:

1. Create a new instance of the file system plug-in by implementing `HXCreateInstance`. The system calls this function at startup and each time it receives a request for a file handled by the plug-in.

   For More Information: See “Creating a Plug-in Instance” in Volume 1, on page 36.

2. Return descriptive information about the plug-in, including its copyright and “more information” URL, by implementing `IHXPlugin::GetPluginInfo`. Set the `bLoadMultiple` attribute to `TRUE` to open multiple instances of the plug-in in separate processes. Set it to `FALSE` to open all instances in the same process.


3. Return the following functional information about the plug-in by implementing the `IHXFileSystemObject::GetFileSystemInfo` method:
   - `pShortName` Provides the plug-in short name, which is listed in Helix Universal Server's `FSMount` configuration parameter. Helix Universal Server uses this short name to identify which file system plug-in to use for an URL request.
   - `pProtocol` Lists the local file access protocol supported by the plug-in. It is used only by the Helix client to select a plug-in to use for local files. In the URL `file://myfile.txt`, for example, the protocol is `file`. Helix Universal Server plug-ins do not need to include this parameter.

Initializing

When Helix Universal Server or the Helix client receives a request for a file, it begins by initializing the file system plug-in (see “File System and File Format Plug-in Interaction” in Volume 1, on page 49).

➤ To provide plug-in initialization support in your file system plug-in:

1. Perform any necessary initialization procedures by implementing the `IHXPlugin::InitPlugin` method. Helix Universal Server or the Helix client uses this method to pass a pointer to the system context. Minimally, this method can use the context pointer to store a reference to `IHXCommonClassFactory` so the plug-in can later create Helix objects.

   For More Information: See “Creating a Plug-in Instance” in Volume 1, on page 36.

2. Perform any initialization procedures required by your plug-in by implementing the `IHXFileSystemObject::InitFileSystem` method. On Helix Universal Server, this method passes the plug-in a pointer to an `IHXValues` interface that contains the plug-in's `FSMount` parameters (except for the short name). For example, the values interface pairs the name `MountPoint` with a pointer to an
IHXBuﬀer interface containing the plug-in’s mount point value, and BasePath with a pointer to a
buffer containing its base path value.

Note: The ﬁle system plug-in does not have a response interface through which it
indicates that its initialization is complete. Only the ﬁle object uses a response interface.

Creating File Objects

Once you have started and initialized your ﬁle system plug-in, you create a ﬁle object to retrieve ﬁle
data for streaming.

➤ To create and initialize a ﬁle object in your ﬁle system plug-in:

1. Create a new IHXFileObject interface by implementing the IHXFileSystemObject::CreateFile method.
   When Helix Universal Server or the Helix client calls this method, the method must pass back to
   the server or client a pointer to the ﬁle object. The system then “owns” the object and is responsible
   for releasing it.

2. Determine if a given ﬁle exists by implementing the IHXFileExists::DoesExist method. Helix
   Universal Server or the Helix client calls this method, passing the ﬁle object the path from the
   URL request and a pointer to the response object (Helix Universal Server or the Helix client). The
   ﬁle object converts the path as necessary to ﬁnd the ﬁle on its ﬁle system. It returns a Boolean
   expression with IHXFileExistsResponse::DoesExistDone.
   If IHXFileExistsResponse::DoesExistDone returns FALSE, Helix Universal Server or the Helix client
   releases the ﬁle object and closes the plug-in.

3. If IHXFileExistsResponse::DoesExistDone returns TRUE, determine the ﬁle’s MIME type by
   implementing the IHXFileMimeMapper::FindMimeType method. When Helix Universal Server or the
   Helix client calls this method, the method passes the ﬁle object a pointer to the response object
   (Helix Universal Server or the Helix client).

4. The ﬁle object calls the server or client’s IHXFileMimeMapperResponse::MimeTypeFound method,
   returning a status code and the ﬁle’s MIME type. If the ﬁle object does not implement the
   IHXFileMimeMapper interface or cannot determine the MIME type, Helix Universal Server or the
   Helix client uses information it received from its ﬁle format plug-ins at startup to cross-reference
   the ﬁle extension with the MIME type.

   For More Information: See “Status Codes” in Volume 1, on page 85.

   Based on the requested ﬁle’s MIME type, Helix Universal Server or the Helix client loads the
   appropriate ﬁle format plug-in and passes it a pointer to the ﬁle object.

5. Initialize the ﬁle object by implementing the IHXFileObject::Init method. The ﬁle format plug-in
calls this method to pass the ﬁle object a pointer to the ﬁle response interface and ﬂags to indicate
that it is opening the ﬁle as read/write/binary (see hxfiles.h for the ﬂag deﬁnitions).

   Note: Within the IHXFileObject::Init method, the ﬁle object should check the ﬁle validity
by opening the ﬁle. If ﬁle authentication is implemented, the ﬁle object performs
authentication during initialization. For more information, see “Chapter 10: Restricting
Access”.


The file object passes the file format plug-in a status code through \texttt{IHXFileResponse::InitDone}.

\textbf{For More Information:} For information on how that plug-in uses the file object to get file data, see “Interacting with a File Object” in Volume 1, on page 67. “Modifying the Response Headers” in Volume 1, on page 66 explains how to modify the request response headers to send data to the client. See also “Status Codes” in Volume 1, on page 85.

\textbf{Supporting Relative File Access}

As described in “Creating Relative File Objects” in Volume 1, on page 69, file format plug-ins can use relative file objects to access files related to a requested file, or open multiple file objects for a single file. To support this feature, file objects must implement \texttt{IHXGetFileFromSamePool}.

\begin{itemize}
\item \textbf{To support relative file access:}
\begin{enumerate}
\item A file system manager (\texttt{IHXFileSystemManager}) created by the file format plug-in calls \texttt{IHXGetFileFromSamePool} on the file object to indicate that it needs a new file object and to identify itself as the response object.
\item The file object creates a new, uninitialized file object. It can do this by calling \texttt{IHXFileSystemObject::CreateFile} on the file system plug-in.
\item Using \texttt{IHXGetFileFromSamePoolResponse::FileObjectReady}, the original file object passes the file system manager a status code and a pointer to the new file object.
\end{enumerate}
\end{itemize}

\textbf{For More Information:} See “Status Codes” in Volume 1, on page 85.

4. The file format plug-in and file system manager then initialize and use the new file object as needed.

\textbf{Closing File Objects}

Before destroying a file object, the component using the file object calls \texttt{IHXFileObject::Close}. This routine should release all resources associated with the file object. Helix Universal Server calls \texttt{IUnknown::Release} when it finishes with a file system plug-in.

\textbf{Multiload File Systems and Streamers}

The Helix Universal Server core provides streamers, which are processes or threads that stream multimedia content to clients. Streamers typically read this multimedia content from file systems. Even though the shared object (or DLL) containing your file system plug-in is loaded only once by Helix Universal Server, one instance of your file system object will be created per streamer-mountpoint combination. For example, if the server is running on a four-processor machine with four streamers and the server configuration file lists your file system plug-in as the handler for a single content mountpoint, you will get four instances of your file system object. If it were listed as the handler for two mountpoints you would get eight instances.

A file system instance is never passed between streamers or mountpoints. The streamer creating the instance of your file system for a specific mountpoint will be the only user of that instance and it will only use it to access the mountpoint it was created to handle. This model allows for scalability while
minimizing data contention. A side effect of this structure is that, depending on platform (threaded
versus process-based), globals may or may not be shared between all instances of your objects. You
should therefore avoid the use of globals.

**Sequence of Calls**

As with any product based on the Helix architecture, the order in which calls are made to the file
system plug-in cannot be assumed. Events can occur in any order, and a call to any method can be
issued while any callback is outstanding. Even for the most careful of programmers, this can lead to
hard-to-reproduce CAs, memory leaks, or other problems. Although this chapter documents certain
rules that govern what calls are legal while certain callbacks are outstanding, you cannot expect all
callers to follow these rules. When this occurs (for example, if a caller issues an IHXFileObject::Read
with an IHXFileResponse::ReadDone outstanding) you need to return an error to the caller and gracefully
handle the situation.

For example, take the case of the server having issued an IHXFileObject::Read call. If your file system
plug-in is actually asynchronous (that is, your implementation of IHXFileObject::Read does not call
IHXFileResponse::ReadDone immediately), then it is easily possible for an IHXFileObject::Seek to be issued
while the IHXFileResponse::ReadDone is pending. The file system interface indicates that if an
IHXFileObject::Seek is issued while a IHXFileResponse::ReadDone is pending, the IHXFileResponse::ReadDone
is cancelled and the IHXFileResponse::Seek is issued normally. At this point, the
IHXFileResponse::ReadDone callback should still continue, but with an HXR_CANCELED status. You should
also endeavour to have the IHXFileResponse::ReadDone callback execute before calling the
IHXFileResponse::SeekDone callback, since that is the order in which they were originally issued. This is,
strictly speaking, not a requirement, since the calling module should not make assumptions about the
order in which its callbacks will be invoked. However, defensive programming such as this can help
prevent problems when it comes time to integrate your new file system plug-in with random datatypes
and servers.

In general, Helix Universal Server follows a pattern when making calls to a file system plug-in:

- IHXFileSystemObject::CreateFile
- IHXFileExists::DoesExist
- IHXFileStat::Stat
- IHXFileObject::Init
- Any combination of IHXFileObject::Seek and IHXFileObject::Read
- IHXFileObject::Close

Once IHXFileObject::Close is invoked, the server core will issue an IHXFileObject::Init again before calling
IHXFileObject::Seek or IHXFileObject::Read again. However, the IHXFileObject::Init method does not need
to be invoked before IHXFileExists::DoesExist, IHXFileStat::Stat, or IHXFileObject::Close are called. If
IHXFileObject::Seek or IHXFileObject::Read are invoked without an IHXFileObject::Init, then an error
should be returned to the caller.

You should also make sure to either call IHXFileObject::Close from your file object destructor or clean up
your file object resources in a different manner, as an explicit IHXFileObject::Close might never be
invoked from the caller. Instead, all the references holders of your file object might just release
(IUnknown::Release) all their references, in which case your file object destructor is invoked.
Modifying the File System Plug-in Sample Code

The Helix Client and Server Software Development Kit includes source code for a file system plug-in based on the standard file system plug-in that ships with Helix Universal Server. Use these sample files to learn basic concepts about Helix file handling. You can also use the samples as templates for building your own file system plug-in:

- `/source/samples/server/filesystem/filesystem.cpp`
  This sample file provides the basics for creating the file system plug-in.

- `/source/samples/server/filesystem/fileobject.cpp`
  This sample file creates the basic file objects that give system components access to file data.

▶ To modify the sample files:

1. Copy the source code from the samples directory to a working directory.
2. Change the file names and class names to match your file system name.
3. In filesys1.cpp, change the plug-in description, copyright, “more information” URL, file system short name, and protocol stored in zm_pDescription, zm_pCopyright, zm_pMoreInfoURL, zm_pShortName, and zm_pProtocol.
4. By using the comments following this procedure, modify the sample methods to handle file access as necessary for your platform and data source.
5. Compile, debug, and test your plug-in.


Another file system plug-in sample supplied with the SDK includes a simple form of authentication. The authenticating file system plug-in sample for Helix Universal Server contains the following source files:

- `/source/samples/server/authfilesys/authfilesys.cpp`
  This sample file provides the basics for creating the file system plug-in.

- `/source/samples/server/authfilesys/authfileobj.cpp`
  This sample file creates the basic file objects that give system components access to file data.

File Format Plug-ins

The first step in integrating your data type into Helix is building a file format plug-in that converts your data type into a stream of packets. Helix Universal Server uses your plug-in to stream your data type directly from its native file format. The Helix client then assembles the packets and displays the data through a rendering plug-in, which is described in Chapter 13. Clients and tools can also use the file format plug-in to read files locally. RealPlayer, for example, uses the plug-in to read a file of your data type saved to the client’s local disk. The following illustration shows how the file format plug-in interacts with a native file to produce streaming data packets.
File Format Plug-in

A file format plug-in does not read files directly from disk or any other data source. It simply interacts with a file object given to it by Helix. The file object, which is created by a file system plug-in (see Chapter 4), provides a virtual file system with standard interfaces that the file format plug-in can use to perform read and seek operations without regard to the platform or storage device that holds the file.

Design Considerations

In addition to the general plug-in design considerations described in “Designing a Plug-in” in Volume 1, on page 35, keep the following points in mind as you develop your file format plug-in:

• Optimize the file format plug-in’s CPU usage. Helix Universal Server creates a file format plug-in instance for each connected client. Plug-ins that use too much of a CPU’s resources can quickly degrade Helix Universal Server performance.

• Design your data type to use Adaptive Stream Management (ASM), which Chapter 11 covers, to adapt to changing network conditions dynamically. Your data type should report as much information about itself as possible to ASM (for example, report the Average Bandwidth Standard Deviation). Also, you should use ASM to assign priorities to your packets so that Helix Universal Server can deliver them effectively and robustly over the Internet.

• If serving a file request requires accessing multiple files, your file format plug-in can implement file cross-referencing in a catalog of its own design. It can then use the file object interfaces, particularly the relative file object methods, to load the necessary files for a requested URL.


The alternative method is to make your file format a container data type for the other data types. Content producers do not then need to keep track of multiple files for each presentation and the file format plug-in does not need to cross-reference URLs with file sets. Using a container file format eliminates the advantages of streaming from native file formats, however. Helix supports either implementation. This design choice is your decision.

Interfaces

A file format plug-in typically implements the following interfaces:

• IHXPlugin. Header file: hxplugn.h.

  Every plug-in implements this interface, which Helix uses to determine the plug-in’s characteristics.

• IHXFileFormatObject. Header file: hxformt.h.
A file format plug-in must implement this interface, which provides the basic methods that Helix uses to instruct the plug-in to send it file headers and packets.

- **IHXFileResponse**. Header file: `hxfiles.h`.
  This response interface to `IHXFileObject` enables the file format plug-in to receive asynchronous notification that the file object has finished an operation.

- **IHXASMSource**. Header file: `hxasm.h`.
  The file format plug-in implements this interface to receive rule subscription and unsubscription information for ASM.

The file format plug-in may also implement the following interfaces, depending on which Helix features it needs to support or use:

- **IHXFileStatResponse**. Header file: `hxfiles.h`.
  The file format plug-in implements this response interface to `IHXFileStat` to receive the asynchronous response data about requested file statistics from a file object.

- **IHXFileSystemManagerResponse**. Header file: `hxfiles.h`.
  This response interface to `IHXFileSystemManager` enables a file format plug-in to receive file object pointers for files related to a requested file.

- **IHXDirHandlerResponse**. Header file: `hxfiles.h`.
  This response interface to `IHXDirHandler` enables a plug-in to receive directory object pointers for directories related to a requested file.

- **IHXPendingStatus**. Header file: `hxpends.h`.
  The file format plug-in can implement this interface to provide the Helix client with the status of its pending operations.

- **IHXPacketFormat**. Header file: `hxformt.h`.
  The file format plug-in can implement this interface to support multiple packet data types.

- **IHXBackChannel**. Header file: `hxasm.h`.
  The file format plug-in can implement this interface to receive data from its corresponding rendering plug-in through the back channel.

**Coding the Plug-in**

The following sections explain how Helix (server or client) and a file format plug-in use the Helix interfaces to stream data to a client. The sample files included with this software development kit (SDK) illustrate many of these features. You can use these sample files as a starting point for building your own plug-in. Refer to “Appendix A: Interface List” beginning in Volume 2, on page 191 and the Helix Client and Server Software Development Kit header files for more information on function variables and return values.

**Note:** The order of function calls listed in the following sections provides a generalized explanation and is for illustrative purposes only. Because Helix is asynchronous, your plug-in must be able to handle any call made to it while it is processing data or waiting...
Starting Up

When Helix Universal Server or a Helix client is started, it loads each file format plug-in. The server or client performs a series of calls to your plug-in’s functions and methods in a specific order.

➤ To provide start-up support in your file format plug-in:

1. Create a new instance of the file format plug-in by implementing HXCreateInstance. The system calls this function at startup and each time it receives a request for a file type handled by the plug-in.

   For More Information: See “Creating a Plug-in Instance” in Volume 1, on page 36.

2. Return descriptive information about the plug-in, including its copyright and “more information” URL, by implementing IHXPlugin::GetPluginInfo. Set the bLoadMultiple attribute to TRUE. This causes Helix Universal Server to open multiple instances of the plug-in in separate processes. A Helix client using the file format plug-in ignores this attribute.


3. Use IHXFileFormatObject::GetFileFormatInfo to return the following functional information about the plug-in to the server or client core:
   - The pFileMimeTypes parameter indicates which Multipurpose Internet Mail Extensions (MIME) type or types the file format plug-in handles. The corresponding rendering plug-in or plug-ins must have the same MIME type or types defined for them.
   - The pFileExtensions parameter gives the file extensions for the files the plug-in handles.
   - The pFileOpenNames parameter gives the file type descriptions and file extensions that appear in the “Files of type” pull-down in the client’s Open File... dialog box.

Initializing

When Helix Universal Server or a Helix client receives a request for a file, it identifies the appropriate file format plug-in based on the requested file’s extension and the pFileExtensions values returned by the file format plug-ins during startup. If two or more plug-ins handle the same file extension, the system uses the first plug-in for that file type that it loaded during startup. The system then initializes the chosen plug-in.

➤ To provide plug-in initialization support in your file format plug-in:

1. Perform any necessary initialization procedures by implementing the IHXPlugin::InitPlugin method. Helix Universal Server uses this method to pass a pointer to the system context. Minimally, this method can use the context pointer to store a reference to IHXCommonClassFactory so the plug-in can later create Helix objects to send data to the system.


2. Initialize the file by implementing the IHXFileFormatObject::InitFileFormat method. This method should also store references to the following interfaces:
• The IHXFormatResponse interface.
This interface is used by the system to receive notification of plug-in actions. This is typically
the interface that instantiated the file format plug-in (for either Helix Universal Server or the
client).
• The IHXRequest interface.
The plug-in can use this interface to access the request response headers.

    For More Information: See “Modifying the Response Headers” in Volume 1, on page 66.

• The IHXFileObject interface.
This interface manages the file object that the file format plug-in uses to get file information.
The plug-in should call IHXFileObject::Init to pass this file object a pointer to the
IHXFileResponse interface, which receives asynchronous notification when file actions complete.
The response object is typically the file format plug-in itself.

The IHXFileObject::Init call also passes flags to indicate the file type. For example, the file
format plug-in could use the following code to initialize the file object for reading a binary file:
m_pFileObject->Init(HX_FILE_READ | HX_FILE_BINARY, this);

    For More Information: See “IHXFileObject::Init” in Volume 2, on page 334 in Appendix
A: Interface List for the flag definitions. Note, too, that a file format plug-in can open
multiple file objects to access multiple points in the file simultaneously. This is useful if
the file contains separate audio and video tracks, for example. For more information,
see “Creating Relative File Objects” in Volume 1, on page 69.

3. Return a status code when the initialization finishes by implementing the
IHXFormatResponse::InitDone method. Within this method, your plug-in can notify the system through
IHXFormatResponse::InitDone whether its initialization has completed successfully or not.

    For More Information: See “Status Codes” in Volume 1, on page 85.

Optionally, your file format plug-in can gather file information from the file object.


4. Include IHXFileFormatObject::GetFileHeader so the system can get an object that contains the file
header data. Your plug-in can then use the file object interface to retrieve an IHXBuffer of this data.
Because of its asynchronous nature, Helix uses several routines to complete this operation.


5. With the retrieved IHXBuffer data, your file format plug-in then needs to create an IHXValues
interface that will contain the file header data. This interface must contain a value for the required
property StreamCount, as well as any other opaque data the plug-in needs to send the renderer
before sending the file streams.

6. Pass the system a pointer to the file header object by calling the IHXFormatResponse::FileHeaderReady
method. Then use IUnknown::Release to release the interface.
Creating Stream Headers

After sending the system the file header for the requested file, your file format plug-in must send the system the stream headers for each stream in the file.

➤ To send the stream headers:

1. Get the stream header information by implementing the IHXFileFormatObject::GetStreamHeader method. For each stream, this method uses the file object to get an IHXBuffer interface that contains the stream header data. Because of its asynchronous nature, Helix uses several routines to complete this operation.


2. With the IHXBuffer of the retrieved stream header data, your file format plug-in can create an IHXValues interface to contain the stream header data. This interface also contains any opaque data the plug-in needs to send to its renderer, as well as values for the properties listed in the following table.

   **Stream Header Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>StreamNumber</td>
<td>Number of this stream.</td>
<td>Yes</td>
</tr>
<tr>
<td>AvgBitRate</td>
<td>Average number of bits streamed per second.</td>
<td>Yes</td>
</tr>
<tr>
<td>Duration</td>
<td>Duration of this stream in milliseconds.</td>
<td>Yes</td>
</tr>
<tr>
<td>MimeType</td>
<td>The stream MIME type, which the Helix client uses to associate the stream with the right renderer.</td>
<td>Yes</td>
</tr>
<tr>
<td>Preroll</td>
<td>Number of milliseconds to buffer before starting the stream.</td>
<td>No</td>
</tr>
<tr>
<td>ASMRuleBook</td>
<td>The rule book for ASM.</td>
<td>No</td>
</tr>
<tr>
<td>MaxBitRate</td>
<td>Maximum bit rate for the stream.</td>
<td>No</td>
</tr>
<tr>
<td>MaxPacketSize</td>
<td>Maximum packet size for the stream.</td>
<td>No</td>
</tr>
<tr>
<td>AvgPacketSize</td>
<td>Average packet size for the stream.</td>
<td>No</td>
</tr>
<tr>
<td>StreamName</td>
<td>Name of the stream.</td>
<td>No</td>
</tr>
</tbody>
</table>

   Note: These stream header properties are used for both the RDT and PNA packet formats. A plug-in that supports other packet formats such as RTP sets additional stream header properties. For more on this, see “Supporting Multiple Packet Formats” in Volume 1, on page 63.

3. Pass the system a pointer to the header object for each stream using IHXFormatResponse::StreamHeaderReady. Then use IUnknown::Release to release each object.

4. If ASM is used, implement the IHXASMSource::Subscribe method for the system to call to indicate which rules for which streams the client has subscribed to.

Creating Stream Packets

After receiving the stream header or headers, the system requests packets for each stream.

➤ To handle packet requests:

1. Retrieve a packet for a stream by implementing the `IHXFileFormatObject::GetPacket` method. Because of the asynchronous nature of Helix, retrieving the packet data requires several routines.


2. With the retrieved packet data from each `IHXFileFormatObject::GetPacket` call, your file format plug-in can create an `IHXPacket` interface that contains the opaque data passed to the renderer, as well as values for the properties listed in the following table.

   **Stream Packet Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>deliveryTime</td>
<td>Time stamp for the packet delivery.</td>
<td>Yes</td>
</tr>
<tr>
<td>streamNo</td>
<td>Number of this stream.</td>
<td>Yes</td>
</tr>
<tr>
<td>ASMFlags</td>
<td>When set to ASM_SWITCH_ON, indicates that an ASM rule change can occur on this packet.</td>
<td>Only for packets where rule changes may occur.</td>
</tr>
<tr>
<td>ASMRuleNo</td>
<td>ASM rule number associated with this packet.</td>
<td>Yes, when ASM is used.</td>
</tr>
</tbody>
</table>

   For More Information: For the basics of packet creation, see “Using IHXPacket to Create Stream Packets” in Volume 1, on page 33. See also “Chapter 11: Adaptive Stream Management” beginning in Volume 1, on page 141.

3. Call `IHXFormatResponse::PacketReady` to pass the system a pointer to the packet interface. Your file format plug-in then releases the packet interface with `IUnknown::Release`.

   When the system needs another packet, it calls your plug-in’s `IHXFileFormatObject::GetPacket` method again, and the preceding procedure is repeated until your plug-in has delivered all packets.
   At that point your plug-in must call `IHXFormatResponse::StreamDone` to notify the system that the stream has finished.

   A file format plug-in should generally create packets of 430 to 500 bytes for the opaque data. Staying under 500 bytes decreases the likelihood of packet fragmentation. RealNetworks also recommends that you write code that enables the plug-in to quickly change the size of the packets it sends.

   For each stream, a file format plug-in should not accept requests for another packet or header until the previous request has been satisfied. The plug-in should return the status code `HXR_UNEXPECTED` if it receives an `IHXFileFormatObject::GetPacket`, `IHXFileFormatObject::GetFileHeader`, or `IHXFileFormatObject::GetStreamHeader` call before returning `IHXFormatResponse::PacketReady`, `IHXFormatResponse::FileHeaderReady`, or `IHXFormatResponse::StreamHeaderReady` from the previous request. If the plug-in supports multiple streams, however, the plug-in must be able to support a `IHXFileFormatObject::GetPacket` call for one stream while it is preparing a packet for another stream.

   For More Information: See “Status Codes” in Volume 1, on page 85.
Handling User Seeks

During a Helix presentation, the user might move the client slider to begin playing the presentation back from a different place in the timeline. By implementing `IHXFileFormatObject::Seek`, your file format plug-in can move to the packet nearest to the requested time in the file. The system calls the your plug-in’s `IHXFileFormatObject::Seek` method, passing the plug-in the requested time in milliseconds. Your file format plug-in then notifies the system of success or failure using `IHXFormatResponse::SeekDone`.

In performing the seek, your file format plug-in needs to determine what packet to send and perform the necessary actions to prepare and send it. A plug-in for an audio data type may need only to send the packet that corresponds to the current place in a new timeline. A plug-in for a video data type, however, may have to locate the keyframe before the new timeline.

For More Information: For more on seeking and reading, see “Interacting with a File Object” in Volume 1, on page 67. “Creating Stream Packets” in Volume 1, on page 62 explains the packet creation procedures.

Closing

When the file playback has finished or is stopped, your file format plug-in handle any clean up required before closing the plug-in. By implementing the `IHXFileFormatObject::Close` method, your plug-in can release all references to objects and deallocate memory when the system calls this method. The `IHXFileFormatObject::Close` method must include a call to `IHXFileObject::Close` for all file objects in use. When it releases the file resources, the file object returns a status code through `IHXFileResponse::CloseDone`.

For More Information: See “Status Codes” in Volume 1, on page 85.

Supporting Multiple Packet Formats

A file format plug-in can support packet formats in addition to the Helix default packet format of RDT. This enables the plug-in to deliver its data type to a client based on an standardized packet format. Helix currently supports the RTP packet standard. Supporting RTP also requires using the ASM Marker property.

For More Information: See “RTP Marker Bit Property” in Volume 1, on page 146.

If your file format plug-in implements `IHXPacketFormat::GetSupportedPacketFormats`, Helix Universal Server can call this method to determine which formats the plug-in supports. Your method can then return a null-terminated list of supported formats ("rdt", “rtp”, and so on). Helix Universal Server then calls your plug-in’s implementation of `IHXPacketFormat::SetPacketFormat` to inform the plug-in of the format to use. If your plug-in does not implement `IHXPacketFormat`, Helix Universal Server assumes that the plug-in supports only RDT.
RTP Stream Header Properties

In addition to the regular stream header properties, file format plug-ins supporting RTP need to set the header properties listed in the following table.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTPPayloadType</td>
<td>Payload type number. See the following table.</td>
<td>Yes</td>
</tr>
<tr>
<td>SamplesPerSecond</td>
<td>Number of samples per second.</td>
<td>With dynamic RTP payload audio codecs only.</td>
</tr>
<tr>
<td>Channels</td>
<td>Number of channels.</td>
<td>With dynamic RTP payload audio codecs only.</td>
</tr>
</tbody>
</table>

The audio codec properties facilitate the construction of an interoperable stream description called SDP, which enables RTP clients to play streams from dynamic payload types. The following table summarizes the supported RTP payload types.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>RTP_PAYLOAD_PCMU</td>
</tr>
<tr>
<td>1</td>
<td>RTP_PAYLOAD_1016</td>
</tr>
<tr>
<td>2</td>
<td>RTP_PAYLOAD_G721</td>
</tr>
<tr>
<td>3</td>
<td>RTP_PAYLOAD_GSM</td>
</tr>
<tr>
<td>4</td>
<td>RTP_PAYLOAD_G723</td>
</tr>
<tr>
<td>5</td>
<td>RTP_PAYLOAD_DVI4_8</td>
</tr>
<tr>
<td>6</td>
<td>RTP_PAYLOAD_DVI4_16</td>
</tr>
<tr>
<td>7</td>
<td>RTP_PAYLOAD_LPC</td>
</tr>
<tr>
<td>8</td>
<td>RTP_PAYLOAD_PCMA</td>
</tr>
<tr>
<td>9</td>
<td>RTP_PAYLOAD_G722</td>
</tr>
<tr>
<td>10</td>
<td>RTP_PAYLOAD_L16_2CH</td>
</tr>
<tr>
<td>11</td>
<td>RTP_PAYLOAD_L16_1CH</td>
</tr>
<tr>
<td>14</td>
<td>RTP_PAYLOAD_MPA</td>
</tr>
<tr>
<td>15</td>
<td>RTP_PAYLOAD_G728</td>
</tr>
<tr>
<td>16</td>
<td>RTP_PAYLOAD_DVI4_11</td>
</tr>
<tr>
<td>17</td>
<td>RTP_PAYLOAD_DVI4_22</td>
</tr>
<tr>
<td>26</td>
<td>RTP_PAYLOAD_JPEG</td>
</tr>
<tr>
<td>31</td>
<td>RTP_PAYLOAD_H261</td>
</tr>
<tr>
<td>32</td>
<td>RTP_PAYLOAD_MPV</td>
</tr>
<tr>
<td>33</td>
<td>RTP_PAYLOAD_MP2T</td>
</tr>
<tr>
<td>34</td>
<td>RTP_PAYLOAD_H263</td>
</tr>
</tbody>
</table>

SDPData Header Property

The SDPData property in the file or stream header enables a file format or encoder to provide a direct input to the Session Description Protocol (SDP). This property also enables a renderer to receive the non-universal SDP attributes.

A file format plug-in can specify the SDPData String property as part of the stream and/or file header. The content of the SDPData must be a series of properly formatted and properly delimited SDP attribute records (“a=”) — refer to RFC2327 for formatting requirements. The attribute records a=rtpmap and a=control are the exception, and must not be specified as part of SDPData. The SDPData information is integrated in the SDP describe request response verbatim. If specified in the header file, it is appended to the session description. If specified in the stream header, it is appended to the media description.

On the client side, the reversal of this process occurs. A media player or server, when receiving packets from the encoder, places any nonuniversal (and thus not recognized by the transport layer) SDP attributes in the SDPData, passing it on in the file or stream header, depending on where the nonuniversal attributes are located — in the session or media description.

MIME Type

Even though a MIME type could conceivably assume any string, to enable interoperability it should be in the following format:

<media>/<encoding name>

For More Information: Refer to RFC2327 for more information.

RTP Timing

The Helix architecture uses milliseconds as the integral unit of time. In RTP streaming, the time stamp might contain a nonintegral number of milliseconds. This results in an aliasing effect when RTP time stamps are converted to milliseconds. The IHXRTPPacket interface can be used to avoid the aliasing effect, and preserve the accuracy of RTP time stamps.

The IHXRTPPacket interface that is derived from IHXPacket contains the IHXRTPPacket::GetRTP, IHXRTPPacket::GetRTPTime, and IHXRTPPacket::SetRTP methods. These methods are analogous to IHXPacket::Get, IHXPacket::GetTime, and IHXPacket::Set, and provide for the addition of RTP time to the packet. The RTP time is scaled by Samples/Second, which a file format plug-in supporting RTP is required to specify using the SamplesPerSecond property:

<time in milliseconds> = 1000.0 * RTPTime / SamplesPerSecond

The RTP time stamps the RTP packet sent out by the server. The media player forms packets supporting the IHXRTPPacket interface whenever a packet is received through RTP. Thus the renderers can take advantage of the raw RTP time stamp if needed.

Tip: The server is not using the RTP time for scheduling purposes. The standard packet time in milliseconds is the time the server uses as delivery time.

A packet supporting the IHXRTPPacket interface is required to implement IHXPacket. Thus, the additional functionality of the IHXRTPPacket interface can be ignored if it is not needed. However, if a packet supporting IHXRTPPacket is set using the IHXRTPPacket::Set method, the RTP time of that packet
is set to the same value as the standard time (in milliseconds). This is significant because when the server is using RTP, it uses packet RTP times stamps verbatim whenever the packet supports the IHXRTPPacket interface.

**Note:** If a stream contains a packet supporting the IHXRTPPacket interface, then all packets in the stream must support IHXRTPPacket.

### Reporting Pending Status

Optionally, file format and file system plug-ins can implement IHXPendingStatus to inform the Helix client of their pending operations. To learn the component’s status, the client calls IHXPendingStatus::getStatus, which returns three parameters. These three parameters provide a status code, a description of the current status, and a percentage completed value.

The status code (`uStatusCode`), which is not related to the system status codes, indicates the plug-in’s current activity. The file format (or file system) plug-in can currently be initializing, buffering data, or be ready for operation. In addition, the file system plug-in can be contacting the host.

**For More Information:** See “Status Codes” in Volume 1, on page 85 and “IHXPendingStatus::getStatus” in Volume 2, on page 450.

The description of the current status (`pStatusDesc`) provides an optional pointer to an IHXBuffer interface that contains a text description of the current status. This buffer might contain, for example, the address of a host being contacted.

The percentage complete value (`ulPercentDone`) is an integer value that represents the percentage of the task complete. This is used primarily for buffering. If the plug-in is contacting a host or initializing, the returned value is 0 when the operation is under way, 100 when complete. A percentage completed value for the HX_STATUS_READY state is ignored.

### Modifying the Response Headers

When a client requests a file, the system generates a request object that contains the URL and the request headers. This object exists as long as the request is being served. A file format plug-in or a file object can send the client any information about the file by attaching response headers to the request object before Helix Universal Server begins to stream the file.

**To attach response headers to the request object:**

1. During initialization, Helix Universal Server passes a file format plug-in a pointer to the request object. For a file object, Helix Universal Server calls IHXRequestHandler::SetRequest to pass the pointer to the file object.

2. If necessary, the plug-in or file object can call IHXRequest::GetURL to get the fully qualified path for the requested file.

3. The plug-in or file object creates an IHXValues interface, and adds the necessary response header values to the object.
4. The plug-in or file object calls IHXRequest::SetResponseHeaders to give the request object a pointer to the values interface containing the response headers. (Multiple values interfaces can be associated with the request object.)

5. If the connection is over HTTP, Helix Universal Server includes the response headers in the HTTP header section. Over RTSP, it gives the response headers to the client in an IHXValues interface.

**Interacting with a File Object**

Instead of reading files directly, a file format plug-in accesses data through file objects created by a file system plug-in. Helix determines which file system plug-in to use, and hands the file format plug-in a file object during initialization. The file format plug-in can then get file data through IHXFileObject, receiving responses through IHXFileResponse without needing to know the location or format of the file data.

**Retrieving File Information**

To retrieve the file object’s name without any path information, the file format plug-in can call IHXFileObject::GetFilename, which returns a pointer to the file name. The plug-in can use the request object to get the fully qualified path of the request URL.

*For More Information:* See “Modifying the Response Headers” in Volume 1, on page 66.

If the file object implements IHXFileStat, a file format plug-in can call the IHXFileStat::Stat method, which passes the file object a pointer to the file statistics response object. The file object then gathers file statistics and returns them to the response object, which is typically the file format plug-in, through IHXFileStatResponse::StatDone. Statistics include the following, which are defined in hxfiles.h:

- System status
- File size, in bytes
- File creation time
- File last access time
- File modification time
- Mode

** Seeking File Data**

When the file format plug-in receives an IHXFileFormatObject::Seek, IHXFileFormatObject::GetFileHeader, IHXFileFormatObject::GetStreamHeader, or IHXFileFormatObject::GetPacket call, it uses the IHXFileObject interface to seek for and read the file through the file object.

➤ To perform a file seek:

1. Record the state of your file format plug-in before implementing the seek so it can respond to the proper context (IHXFileFormatObject::GetFileHeader, IHXFileFormatObject::GetStreamHeader, or IHXFileFormatObject::GetPacket) when it receives notification that the seek has finished.
2. Call the IHXFileObject::Seek method. This method takes a file offset value and a bRelative parameter that, when set to TRUE, specifies that the value is relative to the last seek. When set to FALSE, the bRelative parameter makes the seek an absolute offset.

3. Get the status of the seek operation by implementing the IHXFileResponse::SeekDone method. The file object responds to your plug-in with the status code HXR_OK or HXR_FAIL. Do not assume, however, that when a call to IHXFileObject::Seek returns, IHXFileResponse::SeekDone has been called.

**For More Information:** See “Status Codes” in Volume 1, on page 85.

Reading File Data

Once a seek has completed successfully, you can then read the file data.

➤ To read the file data:

1. Ensure that your file format plug-in records its state so that it can respond to the proper context (IHXFileFormatObject::GetFileHeader, IHXFileFormatObject::GetStreamHeader, or IHXFileFormatObject::GetPacket) when the subsequent read action completes.

2. Call the IHXFileObject::Read method, which takes a value for the length of the data buffer to read.

   **Tip:** If the file header is extremely complex and requires several read calls to process, you might want to read the maximum size of the header in one read, and perform your parsing logic on the returned memory buffer. This way you use far fewer states in your state computer.

   The file object calls your plug-in’s IHXFileResponse::ReadDone method, returning a pointer to the IHXBuffer interface that contains the data. Do not assume, however, that when a call to IHXFileObject::Read returns, IHXFileResponse::ReadDone has been called.

3. Call your plug-in’s method that creates the file header, stream header, or packet object from the returned IHXBuffer interface.

   Note the following about using IHXFileObject::Read:

   • If you call IHXFileObject::Read a second time without having received an IHXFileResponse::ReadDone from the first call, this second IHXFileObject::Read returns the status code HXR_UNEXPECTED. The first read will still be pending.

   **For More Information:** See “Status Codes” in Volume 1, on page 85.

   • If you call IHXFileObject::Seek but have not yet received an IHXFileResponse::ReadDone callback from a previous read, IHXFileResponse::ReadDone returns with the status code HXR_CANCELLED. The seek will still be pending.

Closing a File Object

Before releasing a file object, your file format plug-in must call IHXFileObject::Close to release resources associated with the object. The plug-in can then call IUnknown::Release.
Creating Relative File Objects

During initialization, the file format plug-in initializes a file object for the requested file. In addition, it can access another file or set of files in locations relative to that specified by the requested URL. It can even create additional file objects for the same file to, for example, access multiple streams in the file. Using the relative file object methods greatly reduces the amount of overhead the system performs when creating multiple file objects.

By implementing IHXFileSystemManagerResponse, your file format plug-in can create relative file objects.

> To create relative file objects:

1. Create an IHXFileSystemManager interface from the common class factory. For example:
   ```
   m_pCommonClassFactory->CreateInstance(CLSID_IHXFileSystemManager,
   (void**)&m_pFileSystemManager);
   ```
2. Call IHXFileSystemManager::Init to identify itself as the response object.
3. Call IHXFileSystemManager::GetRelativeFileObject, passing a pointer to the original file object and the relative path to the new file. For example, if the requested URL is for:
   ```
   /home/rmserver/rafiles/alanis.ra
   ```
   and the new file is:
   ```
   /home/rmserver/rafiles/extra/alanis.rae
   ```
   the relative path passed in IHXFileSystemManager::GetRelativeFileObject is
   ```
   extra/alanis.rae
   ```
   The relative path must contain a file name and, optionally, a subdirectory path. Hence the relative file must reside at or below the directory level of the original file. To create a new file object for the same file, the plug-in simply passes in the file name, which it can retrieve through IHXFileObject::GetFilename, as the relative path.

   The file manager object then calls your plug-in’s IHXFileSystemManagerResponse::FileObjectReady method, returning a pointer to the new file object.

   Tip: Do not try to create a second file object with the file manager object before the first file object is ready.

4. Initialize the new file object by calling IHXFileObject::Init as described in “Initializing” in Volume 1, on page 59.

Using Multiple Relative Files

A file format plug-in can create a file object relative to another relative file object. With the simple file system (the basic file system used to locate files on Helix Universal Server or the client’s local disks), all relative file objects are relative to the file object passed as a pointer in IHXFileSystemManager::GetRelativeFileObject. The following example illustrates this:

1. The original file object points to the following file:
   ```
   /home/rmserver/rafiles/alanis.ra
   ```
2. A relative object to the file extra/alanis.rae is then created to point to the following path:
   ```
   /home/rmserver/rafiles/extra/alanis.rae
   ```
3. A third file object defined as alanis.bork is then created relative to the second file object. (That is, the IHXFileSystemManager::GetRelativeFileObject method used to create the third object passes a pointer to the relative file object defined in the preceding step, as well as the path alanis.bork.) Because this file object is relative to the second file object, it points to the following file:
/home/rmserv/raf/extra/alanis.bork

Note, however, that this is not necessarily true for file systems other than the simple file system. Other file system plug-ins supported by Helix's virtual file system architecture may create the third file object relative to the first file object:
/home/rmserv/raf/alanis.bork

Modifying the File Format Plug-in Sample Code

The Helix Client and Server Software Development Kit includes sample file format plug-ins that you can use as a starting point for building your own plug-in:

- /source/samples/server/fileformat2/fileformat2.cpp
  A basic file format plug-in that streams data to a corresponding, basic rendering plug-in. Build these sample files to learn basic concepts about streaming and rendering Helix packets.

- /source/samples/server/fileformat/fileformat.cpp
  An intermediate sample file that creates a single data stream.

- /source/samples/datatype/pre_post_fileformat/pre_post_fileformat.cpp
  An intermediate sample file that creates pre-mix and post-mix audio streams.

Perform the following steps to modify either of the intermediate sample files. This procedure assumes that your company name is “Foo Bar, Inc.”, your file extension is “.foo”, and the MIME type of your file format is “application/x-foobar”.

➤ To modify one of the sample files:

1. Copy the source code from the Samples directory to a working directory.

2. Change the file name and class names to match your file format name. For example, if you are implementing a Foo file format you might replace all occurrences of CExampleFileFormat with CFooFileFormat, and rename the files fooffpln.cpp and fooffpln.h.

3. Change the plug-in description, copyright, and “more information” URL stored in zm_pDescription, zm_pCopyright, and zm_pMoreInfoURL. For the Foo example, you could change the values as follows:
   char* CFooFileFormat::zm_pDescription = “Foo File Format Plug-in”;
   char* CFooFileFormat::zm_pCopyright = “(c)1997 Foo Bar”;
   char* CFooFileFormat::zm_pMoreInfoURL = “http://www.foobar.com”;

4. Change the file MIME type, extension, and Open File info stored in zm_pFileMimeTypes, zm_pFileExtensions, and zm_pFileOpenNames. For the Foo example, you could change the values as follows:
   char* CFooFileFormat::zm_pFileMimeTypes = {“application/x-foobar”, NULL};
   char* CFooFileFormat::zm_pFileExtensions = {“foo”, NULL};
   char* CFooFileFormat::zm_pFileOpenNames = {“Foo File Format (*.foo)”, NULL};
5. Change the stream MIME type from application/x-hx-examplesstream to the new value for your stream. The stream MIME type is usually the same as the file MIME type.

6. If your file format is a container file format, such as QuickTime, it needs to create multiple streams of data packets. Change the StreamCount value in pHdr to the number of streams your plug-in creates.

7. Process your input data as needed to create Helix packets.

8. Compile, debug, and test your plug-in.

Data conversion and reversion plug-ins provide a means of converting and restoring a stream from the server to a client. As an example, a set of these plug-ins could encrypt data on a server (the converter always resides on the server) and decrypt the data on a client (the reverter always resides on the client). In addition, file and stream headers can be converted. The data conversion and data reversion plug-ins can also send messages back and forth to each other. These messages can be used, for example, as a key exchange between an encryptor and decryptor.

When a client plays a stream from a DataConvertMount list name, that stream goes through the corresponding IHXDataConvert interface on the server. An IHXDataRevert interface residing on the client and matching the data conversion MIME type on the client gets instantiated, and handles the stream reversion. The converter and reverter communicate with each other through IHXDataConvertResponse::SendControlBuffer and IHXDataRevertResponse::SendControlBuffer.

### Interfaces

A server’s conversion plug-in typically implements the following interfaces:

- **IHXPlugin.** Header file: hxplugn.h. Every plug-in implements this interface, which Helix uses to determine the plug-in’s characteristics.

- **IHXDataConvert.** Header file: hxdtcvt.h. All conversion plug-ins on the server must implement this interface, which handles header and packet conversion. This interface also receives the control channel information from the IHXDataRevert interface on the client.

- **IHXDataConvertResponse.** Header file: hxdtcvt.h. This interface provides an asynchronous response to the server core that the data conversion that had been requested has been completed. The data returned to the server can be modified or can be returned unchanged. This interface can also send the location of control channel information to the client.

- **IHXDataConvertSystemObject.** Header file: hxdtcvt.h. All data conversion plug-ins must implement this interface. Helix Universal Server uses this interface to intialize the data conversion plug-in.
A client’s reversion plug-in typically implements the following interfaces:

- **IHXPlugin**. Header file: `hxplugn.h`. Every plug-in implements this interface, which Helix uses to determine the plug-in’s characteristics.

- **IHXDataRevert**. Header file: `hxdtcvt.h`. All reversion plug-ins on the client must implement this interface, which restores any information that was converted on the server. This interface also receives the control channel information from the **IHXDataConvert** interface on the server.

- **IHXDataRevertResponse**. Header file: `hxdtcvt.h`. The client reversion plug-in uses this interface to acknowledge the restoration of header and packet information asynchronously.

Coding the Conversion Plug-in

The following sections explain how the conversion plug-in uses the Helix interfaces to convert data. The sample files included with this SDK illustrate many of these features. You can use these sample files as a starting point for building your own plug-ins. Refer to “Appendix A: Interface List” beginning in Volume 2, on page 191 and the Helix Client and Server Software Development Kit header files for more information on function variables and return values.

**Note:** The order of function calls listed in the following sections provides a generalized explanation and is for illustrative purposes only. Because Helix is asynchronous, your plug-in must be able to handle any call made to it while it is processing data or waiting for a response from another object. Do not code your plug-in so that it expects a specific sequence of events to occur as it interacts with Helix.

**Starting Up the Conversion Plug-in**

When Helix Universal Server is started, it loads each conversion plug-in. Helix Universal Server performs a series of calls to your plug-in’s functions and methods in a specific order.

➢ To provide start-up support in your conversion plug-in:

1. Create a new instance of the converter system object by implementing **HXCreateInstance**. The server calls this function at start-up and each time it receives a stream to be converted by the plug-in.
   
   **For More Information:** See “Creating a Plug-in Instance” in Volume 1, on page 36.

2. Return descriptive information about the plug-in, including its copyright and “more information” URL, by implementing **IHXPlugin::GetPluginInfo**. Set the **bMultipleLoad** attribute to **TRUE** to enable the client to launch multiple instances of the plug-in in separate processes.
   
   **For More Information:** See “Defining Plug-in Attributes” in Volume 1, on page 37.

3. Return the short name of the data converter to use, as defined by the appropriate **DataConvertMount** list name in the server configuration file, by implementing the **IHXDataConvertSystemObject::GetDataConvertInfo** method. The server uses this information to determine which data converter to instantiate based on the mount point.
   
   The following example shows one possible use of the **DataConvertMount** list name in the server configuration file:
![... DATA CONVERTERS -->
</List>
</List>
</List>
</List>
</List>

The ShortName and MountPoint variables are required. Any other variables, such as the ConvertHeaders variable in the Second data convert, are optional; you can choose any variable name to be used for whatever purpose in your plug-in code.

The MountPoint variable in the DataConvertMount list name does not create the mount point; it is only used for reference. This mount point must be created elsewhere in the configuration file. For example, the mount point could have been created under the FSMount list name. For more information, see the MountPoint and BasePath entries in the FS Mount parameter section in Helix Universal Server Configuration File Reference.

Initializing the Conversion Plug-in

When Helix Universal Server is ready to initialize your plug-in, it performs a series of calls to your plug-in’s methods in a specific order.

To provide plug-in initialization support in your conversion plug-in:

1. Perform any necessary plug-in initialization procedures by implementing the IHXPlugin::InitPlugin method. Helix Universal Server uses this method to pass a pointer to the system context. Minimally, this method can use the context pointer to store a reference to IHXCommonClassFactory so the plug-in can later create Helix objects used in converting data.

   For More Information: See “Plug-in Initialization” in Volume 1, on page 40

2. Perform any conversion initialization procedures in the plug-in by implementing the IHXDataConvertSystemObject::InitDataConvertSystem method. This method passes the plug-in a pointer to an IHXValues interface that manages the plug-in's parameters, specifically any parameters under the second level lists in the DataConvertMount list name contained in the server configuration file. For example, in the sample DataConvertMount list name shown above, the ShortName and MountPoint variables for the First data convert, Second data convert, and Data convert for live list names, along with the ConvertHeaders variable for the Second data convert list name are managed by this IHXValues interface.
Converting Data

Once your data conversion plug-in is initialized, it can then begin converting data. Helix Universal Server performs a series of calls to your plug-in to begin converting data.

➤ To provide data stream conversion support in your plug-in:

1. Create a data conversion interface through which file headers, stream headers, and data packets can be converted by implementing the `IHXDataConvertSystemObject::CreateDataConvert` method.

2. Call the server’s `IHXDataConvert::DataConvertInit` to instantiate the `IHXDataConvertResponse` interface on the server and associate it with the current `IHXDataConvert` interface.

3. Get the conversion type that is going to be used for this stream by implementing the `IHXDataConvert::GetConversionMimeType` method. Because this method doesn’t get called until after initialization, a single conversion plug-in can support either a single conversion type or several different conversion types.

   The server then begins sending the headers to your conversion plug-in.

4. Convert the file header by implementing the `IHXDataConvert::ConvertFileHeader` method. If no conversion is required, your plug-in can call `IHXDataConvertResponse::ConvertedFileHeaderReady` with the `status` parameter set to `HXR_OK` and the `pFileHeader` parameter set to `NULL`.

   Note: You can also use specific conversion information, such as a key sent by the client, to convert the headers. For more information, see “Supporting Client and Server Communication” in Volume 1, on page 79.

5. Convert any stream headers by implementing the `IHXDataConvert::ConvertStreamHeader` method. If no conversion is required, your plug-in must call the server’s `IHXDataConvertResponse::ConvertedStreamHeaderReady` with the `status` parameter set to `HXR_OK` and the `pStreamHeader` parameter set to `NULL`.

   The server now begins sending the actual stream packets to be converted.

6. Convert the incoming packets by implementing the `IHXDataConvert::ConvertData` method. If no conversion is required, your conversion plug-in must call the server’s `IHXDataConvertResponse::ConvertedDataReady` method with the `status` parameter set to `HXR_OK` and a `NULL` buffer. In response, the server then sends the original, unconverted packet to the client.

   Note: You can also use specific conversion information, such as a key sent by the client, to convert the data packets. For more information, see “Supporting Client and Server Communication” in Volume 1, on page 79.

Conversion and Multicast Support

Back channel multicasting provides a means of passing header information across a control channel to each of the clients connected to the network. Each client requires its own `IHXDataConvert` interface for the transfer of this header information. The actual live stream requires only one `IHXDataConvert` interface for all clients connected to the network.
To provide back channel multicasting support in your plug-in:

1. Add a conversion interface to handle the file and stream header conversion by implementing the `IHXDataConvert::AddMulticastControlConverter` method. The `IHXDataConvert` interface created by this method will send headers to only one client through its control channel to the server. Therefore, the server must call this method multiple times — once for each client — if more than one client is to receive the multicast broadcast.

2. Add a conversion interface to handle the data conversion by implementing the `IHXDataConvert::SetMulticastTransportConverter` method. In this case, only one `IHXDataConvert` interface is required since all clients receive the broadcast across a single multicast-enabled network.

   Note: Scalable multicasting cannot support data conversion since no control channel exists to pass the header information.

Coding the Reversion Plug-in

The following sections explain how the reversion plug-in uses the Helix interfaces to restore data. The sample files included with this SDK illustrate many of these features. You can use these sample files as a starting point for building your own plug-ins. Refer to “Appendix A: Interface List” beginning in Volume 2, on page 191 and the Helix Client and Server Software Development Kit header files for more information on function variables and return values.

   Note: The order of function calls listed in the following sections provides a generalized explanation and is for illustrative purposes only. Because Helix is asynchronous, your plug-in must be able to handle any call made to it while it is processing data or waiting for a response from another object. Do not code your plug-in so that it expects a specific sequence of events to occur as it interacts with Helix.

Starting Up the Reversion Plug-in

When the Helix client is started, it loads each reversion plug-in. The Helix client performs a series of calls to your plug-in's functions and methods in a specific order.

To provide start-up support in your reversion plug-in:

1. Create a new instance of the reverter system object by implementing `HXCreateInstance`. The client calls this function at start-up and each time it receives a stream to be restored by the plug-in.

   For More Information: See “Creating a Plug-in Instance” in Volume 1, on page 36.

2. Return descriptive information about the plug-in, including its copyright and “more information” URL, by implementing the `IHXPlugin::GetPluginInfo` method. Set the `bMultipleLoad` attribute to `TRUE` to enable the client to launch multiple instances of the plug-in in separate processes.

Initializing the Reversion Plug-in

When the Helix client is ready to initialize your plug-in, it performs a series of calls to your plug-in’s methods in a specific order.

➤ To provide plug-in initialization support in your reversion plug-in:

1. Perform any necessary initialization procedures by implementing the IHXPlugin::InitPlugin method. The client uses this method to pass a pointer to the system context. Use the context pointer to store a reference to IHXCommonClassFactory so that it can later create Helix objects used in reverting data to its original format.

   For More Information: See “Plug-in Initialization” in Volume 1, on page 40

2. Call IHXDataRevert::DataRevertInit to instantiate the IHXDataRevertResponse interface on the client and associate it with the current IHXDataRevert interface.

Restoring Data

Once the data reversion plug-in is initialized, it can then begin restoring data. The client calls a series of reversion plug-in methods to restore a data stream.

➤ To provide data reversion support in your reversion plug-in:

1. Get the data conversion MIME types that were used to convert the stream on the server by implementing the IHXDataRevert::GetDataRevertInfo method. This information consists of a NULL-terminated array of one or more data conversion MIME types.

2. Restore the MIME type file header by implementing the IHXDataRevert::RevertFileHeader method. If no restoration is required, your plug-in can call IHXDataRevertResponse::RevertedFileHeaderReady with the status parameter set to HXR_OK and the pFileHeader parameter set to NULL.

   Note: You can also use specific restoration information, such as a key that was originally sent by the client to the server, to restore the headers. For more information, see “Supporting Client and Server Communication” in Volume 1, on page 79.

3. Restore any MIME type stream headers by implementing the IHXDataRevert::RevertStreamHeader method. If no restoration is required, your plug-in must call the client’s IHXDataRevertResponse::RevertedStreamHeaderReady method with the status parameter set to HXR_OK and the pStreamHeader parameter set to NULL.

4. Restore the incoming packet by implementing the IHXDataRevert::RevertData method. The client calls this method when it begins sending the actual stream packets to be restored. If no restoration is required, your reversion plug-in must call the client’s IHXDataRevertResponse::RevertedDataReady method with the status parameter set to HXR_OK and a NULL buffer, after which the client does not attempt to restore the packet.

   Note: You can also use specific restoration information, such as a key that was originally sent by the client to the server, to restore the data packets. For more information, see “Supporting Client and Server Communication” in Volume 1, on page 79.
Supporting Client and Server Communication

You can provide bidirectional communication between the conversion and reversion plug-ins. This communication could be used, for example, to provide encryption key negotiation and transmission between the conversion and reversion plug-ins. The information passed between the conversion and the reversion plug-ins is transferred using the two-way TCP control connection between the server and client and is, therefore, lossless. You can pass this information between the conversion and reversion plug-ins at any time — this provides a means of passing the encryption key, for instance, before encoding the file and stream headers.

➤ To send information from the reversion plug-in on the client to the conversion plug-in on the server:

1. The client starts up and initializes the reversion plug-in.
2. The reversion plug-in calls the client’s IHXDataRevertResponse::SendControlBuffer method, supplying the method with a buffer containing the data to be sent to the conversion plug-in on the server. Once the server has received the data, it calls the conversion plug-in’s IHXDataConvert::ControlBufferReady method, which supplies the conversion plug-in with the information passed from the client.

➤ To send information from the conversion plug-in on the server to the reversion plug-in on the client:

1. The server starts up and initializes the conversion plug-in.
2. The conversion plug-in calls the server’s IHXDataConvertResponse::SendControlBuffer method, supplying the method with a buffer containing the data to be sent to the reversion plug-in on the client. Once the client has received the data, it calls the reversion plug-in’s IHXDataRevert::ControlBufferReady method, which supplies the reversion plug-in with the information passed from the server.

Modifying the Conversion and Reversion Plug-in Sample Code

The Helix Client and Server Software Development Kit includes sample conversion and reversion plug-ins that you can use as a starting point for building your own plug-ins:

- /source/samples/server/converter/converter.cpp
  A basic conversion plug-in that converts data streamed from the server core. Build this sample file to learn basic concepts about converting Helix packets.

- /source/samples/server/reverter/reverter.cpp
  A basic reversion plug-in that restores data streamed from the client core. Build this sample file to learn basic concepts about restoring converted Helix packets.

Perform the following steps to change the conversion or reversion code. These steps assume your company name is “Foo Bar, Inc.”, your file extension is .foo, and the MIME type of your data stream is application/x-foobar.

➤ To change conversion or reversion code:

1. Copy the sample code from the samples directory to a working directory. Change the file name and class names to match your file format name. For example, if you are implementing a Foo data type
you might replace all occurrences of Converter with FooConverter, and rename the files fooconvr.cpp and fooconvr.h.

2. Change the plug-in description, copyright information, and more info URL stored in pDescription, pCopyright, and pMoreInfoURL. For the Foo example, you could change the values as follows:

```cpp
char* FooConverter::pDescription = "Foo Conversion Plug-in";
char* FooConverter::pCopyright = "(c)2000 Foo Bar";
char* FooConverter::pMoreInfoURL = "http://www.foobar.com";
```

3. Change the stream MIME types stored in zm_pConvertTypes. For the Foo example, you could change the values as follows:

```cpp
char* FooConverter::zm_pConvertTypes = "application/x-foobar";
```

4. Process the Helix packets and convert or restore the data as necessary.

5. Compile, debug, and test your plug-ins.

**For More Information:** See “Building a Sample Plug-in” in Volume 1, on page 43.
Chapter 6: NETWORK SERVICES

Helix’s Network Services provides cross-platform methods for managing network communications. Any server-side or client-side Helix component can use Network Services to create Transmission Control Protocol (TCP) or User Datagram Protocol (UDP) connections for reading and writing data. Network Services also provides interfaces that enable components to resolve Domain Name System (DNS) host names and listen for TCP connections on specified ports.

Interfaces

A Helix component uses the following interface to access Network Services:

- IHXNetworkServices. Header file: hxengin.h.
  The Helix core architecture implements this interface, which Helix components use to generate network socket interfaces.

A Helix component implements the following asynchronous response interfaces to use Network Services:

- IHXListenResponse. Header file: hxengin.h.
  This response interface to IHXListenSocket enables the component to receive information about TCP connections on a port.

- IHXResolverResponse. Header file: hxengin.h.
  This response interface to IHXResolver enables the component to resolve host names.

- IHXTCPResponse. Header file: hxengin.h.
  The Helix component implements this response interface to IHXTCPSocket if it needs to create a TCP socket.

- IHXUDPResponse. Header file: hxengin.h.
  The Helix component implements this response interface to IHXUDPSocket if it needs to create a UDP socket.

Network Services also includes the following platform-specific interface:

- IHXAsyncI0Selection. Header file: hxengin.h.
  Implemented by the Helix core architecture, this interface enables UNIX components to receive callbacks through IHXCallback based on input/output (I/O) events normally handled by select().
Creating a Listen Socket

A Helix component performs the following actions to set up a TCP listening socket and receive notice of a TCP connection on a local port:

1. The component calls IHXNetworkServices::CreateListenSocket and receives a pointer to the new socket interface.

2. The component calls IHXListenSocket::Init on the socket interface to identify itself as the response interface. This method also passes the local IP address and port number on which to listen.

3. When a TCP connection is made on the specified port, Helix creates an IHXTCPSocket interface and calls the component’s IHXListenResponse::NewConnection method, which returns a status code and a pointer to the TCP socket interface. The component can then use the TCP socket methods described in the following section.

For More Information: See “Status Codes” in Volume 1, on page 85.

Using a TCP Socket

A Helix component can create or inherit an existing TCP socket by performing one of the following actions, accordingly:

• If it needs to create the TCP socket interface, the component calls IHXNetworkServices::CreateTCPSocket and receives a pointer to the new socket interface. The component then calls IHXTCP::Init to initialize the socket and identify itself as the response interface.

• If the component inherits an existing, initialized TCP socket, it uses IHXTCP::SetResponse to identify itself as the response interface. It can also use the following IHXTCP methods to return in local host order the addresses and ports of the TCP socket:
  • IHXTCP::GetForeignAddress
  • IHXTCP::GetLocalAddress
  • IHXTCP::GetForeignPort
  • IHXTCP::GetLocalPort

After it has set up the TCP socket, the component can perform the following actions:

• The component can call IHXTCP::Bind to bind the TCP socket to a local address and port. It specifies the port and address in native byte order.

• To connect to an address, the component can call IHXTCP::Connect, passing a host name in the form www.myserver.com or an IP address in the form nnn.nnn.nnn.nnn. Helix then returns a status code through IHXTCPResponse::ConnectDone.

For More Information: See “Status Codes” in Volume 1, on page 85.

• The component can call IHXTCP::Read to read the specified number of bytes from the TCP source. Through IHXTCPResponse::ReadDone the system returns a status code and a pointer to an IHXBuffer interface containing the data.
• The component can write data to the TCP socket through \texttt{IHXTCP\textbackslash{}Socket::Write}, which passes Network Services a pointer to an \texttt{IHXBuffer} interface.

\textbf{Note:} To write a large amount of data (more than 500 Kbps or so), the component should first call \texttt{IHXTCP\textbackslash{}Socket::WantWrite}. When the channel is ready to write the data, the TCP interface returns a status code with \texttt{IHXTCP\textbackslash{}Response::WriteReady}.

### Using a UDP Socket

To use a UDP socket, an object calls \texttt{IHXNetwork\textbackslash{}Services::CreateUDPSocket} and receives a pointer to the new socket interface. It then calls \texttt{IHXUDPSocket::Init} to initialize the socket and identify itself as the response interface. This method also passes the IP connection address as a four-byte address in native byte order. (The component can use the resolver to get this address.) After it has set up the UDP socket, the component can perform the following actions with it (note that all addresses are in native byte order):

• The component can call \texttt{IHXUDPSocket::GetLocalPort} to retrieve the local port and \texttt{IHXUDPSocket::Bind} to bind the UDP socket to a local address and port.

• The component can call \texttt{IHXUDPSocket::Read} to read a specified number of bytes from the UDP source. The UDP interface uses \texttt{IHXUDP\textbackslash{}Response::Read\textbackslash{}Done} to return a status code, the local address, the local port, and a pointer to an \texttt{IHXBuffer} interface containing the data.

\textbf{For More Information:} See “Status Codes” in Volume 1, on page 85.

• The component can write data to the UDP socket through \texttt{IHXUDPSocket::Write}, which passes a pointer to an \texttt{IHXBuffer} interface. The component can also write to a different UDP address with \texttt{IHXUDPSocket::WriteTo}, which specifies an address and a port as it passes the \texttt{IHXBuffer} pointer.

• In some cases, the component may need to connect through Helix’s UDP multicast. It then uses \texttt{IHXUDPSocket::Join\textbackslash{}Multicast\textbackslash{}Group}, specifying the multicast address (the address it is sending to) and the interface address (the address it is sending from). It uses \texttt{IHXUDPSocket::Leave\textbackslash{}Multicast\textbackslash{}Group} to leave the multicast.

### Resolving a DNS Host Name

A Helix component performs the following actions to get the IP address for a DNS host name:

1. The component calls \texttt{IHXNetwork\textbackslash{}Services::CreateResolver} and receives a pointer to the new resolver interface.

2. The component calls \texttt{IHXResolver::Init} to identify itself as the response interface.

3. The component calls \texttt{IHXResolver::GetHost\textbackslash{}By\textbackslash{}Name}, passing Helix a DNS name in the form \texttt{www.myserver.com}. The system responds with \texttt{IHXResolver\textbackslash{}Response::GetHost\textbackslash{}By\textbackslash{}Name\textbackslash{}Done}, which returns a status code and a four-byte IP address in native byte order.

\textbf{For More Information:} See “Status Codes” in Volume 1, on page 85.
Getting an I/O Callback on UNIX

On UNIX, Helix (server or client) handles I/O events through the UNIX `select()` function. UNIX plugins do not have access to this function directly. If a plug-in needs to handle an I/O event, it sets up a callback through which Helix informs it that a specified file descriptor is ready for reading or writing, or that it has an exception. During this process, the following actions occur:

1. The plug-in implements `IHXCallback`.
2. The plug-in calls `IHXAsyncI0Selection::Add`, identifying itself as the callback interface and passing the system the file descriptor and type.
3. When the file descriptor is ready for reading or writing, or has an exception, the system calls `IHXCallback::Func`.
4. Within `IHXCallback::Func`, the plug-in handles the event as necessary.
5. The plug-in can cancel a callback with `IHXAsyncI0Selection::Remove`. 
Any Helix component can report errors. On a Helix client, these error messages display in a pop-up dialog box. On Helix Universal Server, the errors are recorded in the system error log file, the location and name of which is determined by the `ErrorLogPath` parameter in the Helix Universal Server configuration file. Through the Helix interfaces, a component can also monitor system error messages.

**Note:** A monitor plug-in can monitor the Helix Universal Server registry for client connections and other conditions that are not errors. See “Chapter 9: Managing the Server Registry”.

### Interfaces

A Helix component implements or uses the following error-reporting interfaces:

- **IHXErrorMessages.** Header file: `hxerror.h`.
  
  The Helix core architecture implements this interface, which any Helix component can use to report errors.

- **IHXErrorSinkControl.** Header file: `hxerror.h`.

  The Helix core architecture implements this interface, which any Helix component can use to register as an error sink and receive notice of system errors.

- **IHXErrorSink.** Header file: `hxerror.h`.

  A Helix component implements this interface to receive notice of system errors.

### Status Codes

The header file `hxresult.h` defines the status codes used in many Helix function calls as well as the system error messages. In normal operation, all Helix components need to be able to handle the following basic codes that may be returned by the Helix system:

- **HXR_FAIL**  
  Operation failed.

- **HXR_OK**  
  Operation succeeded.

- **HXR_UNEXPECTED**  
  Call was unexpected or method is not implemented.

A component can call `IHXErrorMessages::GetErrorText` to get a text description of a status code. The method takes a status code as a parameter and returns a pointer to an `IHXBuffer` interface that contains the status code text.

**For More Information:** See “Appendix D: Return Values” beginning in Volume 2, on page 707.
Reporting Errors

Any Helix component can report an error to Helix (Helix Universal Server or Helix client). Helix Universal Server logs errors in its system error log. Helix clients report errors in a pop-up dialog box but typically do not record error messages on the client computer or upload them to the system log. To report errors, a component calls `IHXErrorMessages::Report`, which takes the following parameters:

- **unSeverity**
  Relative error severity, which affects how the system reacts. For any level of error, the Helix client may pop up a dialog box, depending on how the top-level client is implemented.

<table>
<thead>
<tr>
<th>Level</th>
<th>Condition</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Panic</td>
<td>Error potentially causing a system failure. Helix takes actions necessary to correct the problem. This may include shutting down the presentation.</td>
</tr>
<tr>
<td>1</td>
<td>Severe</td>
<td>Error requiring immediate user intervention to prevent a problem. Helix will shut down the presentation if necessary.</td>
</tr>
<tr>
<td>2</td>
<td>Critical</td>
<td>Error that may require user intervention to correct. Helix will shut down the presentation if necessary.</td>
</tr>
<tr>
<td>3</td>
<td>General</td>
<td>Error that does not cause a significant problem with normal system operation.</td>
</tr>
<tr>
<td>4</td>
<td>Warning</td>
<td>Warning about a condition that does not cause system problems but may require attention.</td>
</tr>
<tr>
<td>5</td>
<td>Notice</td>
<td>Notice about a condition that does not cause system problems but should be noted.</td>
</tr>
<tr>
<td>6</td>
<td>Informational</td>
<td>Informational message only.</td>
</tr>
<tr>
<td>7</td>
<td>Debug</td>
<td>Information of use only when debugging a program.</td>
</tr>
</tbody>
</table>

- **ulHXCode**
  A Helix status code, which is translated to a text reason for display in the client error dialog box or Helix Universal Server log file. A component can call `IHXErrorMessages::GetErrorText` to get the code’s text description. This method, which takes a status code such as `HXR_OK` as a parameter, returns a pointer to an `IHXBuffer` interface containing the status code text.

- **ulUserCode**
  A component-specific error code that is not translated to a text representation. It can have any value that is logged (server) or displayed (client) without further interpretation.

- **pUserString**
  A component-specific error string that gives a text explanation of the component-specific error code. The string is logged (server) or displayed (client) without further interpretation.

- **pMoreInfoURL**
  A component-specific “more information” URL string that typically is used only on client computers. Pop-up error messages on RealPlayer, for example, contain a URL that directs the user to a [http://www.real.com](http://www.real.com) page that explains the error.
Receiving Error Messages

A Helix component on Helix Universal Server or on a client computer can receive notice of system errors by registering as an error sink.

➤ To receive error messages:

1. The component calls IHXErrorSinkControl::AddErrorSink to register itself as an error sink. The method passes to the system context a pointer to the component as the IHXErrorSink interface. It also passes two integer values, unLowSeverity and unHighSeverity, which have error severity levels as values. If, for example, the method call designates low as 0 and high as 2, the component receives messages only for errors of severity 0, 1, or 2.

2. When an error that falls within the registered severity range occurs, Helix Universal Server or the client calls IHXErrorSink::ErrorOccurred, passing the component the values for IHXErrorMessages::Report described previously in the error-reporting section.

3. When the component does not need to receive error messages, it calls IHXErrorSinkControl::RemoveErrorSink.

The sample logging plug-in /source/samples/server/logging/logging.cpp demonstrates how to get error messages using IHXErrorSink and write them to a log file named exlog.log.

For More Information: For details about creating plug-ins, such as error-logging plug-ins, see “Chapter 3: Developing Plug-ins” beginning in Volume 1, on page 35.
SERVER PLUG-INS AND SERVICES

The following chapters explain how to build and use plug-ins and features for Helix-based servers. Topics include broadcasting, the server registry, and authentication. See also the chapters in Part I for information about features common to both Helix servers and clients.
CHAPTER 8

LIVE BROADCASTING

Helix Universal Server provides both on-demand streaming and live broadcasting capabilities. Live broadcasting is similar to a live television event; a user can tune into a live Internet or intranet broadcast to join the presentation in progress. You can modify many of the live broadcasting services of Helix Universal Server to provide access to your live content. For example, you can create a broadcast plug-in to stream your own unique file formats, or modify the remote broadcast services to pass buffers of encoded data from your application to the broadcast plug-in. You can also use the RealText and RealPix features to broadcast live text and images.

Broadcast Plug-in

To stream your data type from a live source, you can use Helix's remote broadcast feature, which ties a broadcast application into the standard Helix broadcast plug-in. If this feature does not provide the required functionality, however, you can build your own broadcast plug-in to stream live content to clients. Live content can come from an encoder or other digital source on a network, or directly from digitizing hardware on the server computer.

Broadcast Plug-in

As shown in the following illustration, a broadcast plug-in creates two objects used by Helix Universal Server. Helix Universal Server uses the file object to verify URL requests to the broadcast. It uses the broadcast object to get the packets. The broadcast plug-in can use Network Services to communicate with the live source over a TCP/IP network. If the live content is created locally in hardware, the plug-in communicates directly with the hardware.

File and Broadcast Objects

Note: Because it delivers packets to Helix Universal Server, the broadcast plug-in performs the functions of both a file system plug-in and a file format plug-in.

Interfaces

A broadcast plug-in typically implements the following interfaces:

- **IHXPlugin**. Header file: `hxplugn.h`.
  Every plug-in implements this interface, which Helix Universal Server uses to determine the plug-in's characteristics.

- **IHXBroadcastFormatObject**. Header file: `hxformat.h`.
  A broadcast plug-in must implement this interface, which provides the basic methods that Helix Universal Server uses to instruct the plug-in to send it file headers and packets. The response interface is `IHXFormatResponse`, the same response interface used by a file format plug-in.

- **IHXFileSystemObject**. Header file: `hxfiles.h`.
  Helix Universal Server uses this interface to create a file object that validates requests to join the live broadcast.

- **IHXFileExists**. Header file: `hxfiles.h`.
  Helix Universal Server uses this interface to verify that the broadcast source for a URL request exists.

- **IHXBroadcastMapper**. Header file: `hxfiles.h`.
  Helix Universal Server uses this interface to discover the type of file system provided by the plug-in and to get the plug-in's file system short name.

- **IHXCallback**. Header file: `hxengin.h`.
  If the plug-in uses `IHXScheduler`, it implements this interface to receive Scheduler callbacks at designated intervals.

- **IHXRequestHandler**. Header file: `hxfiles.h`.
  The broadcast object can implement this interface to access the broadcast response headers.

Coding the Plug-in

The following sections explain how Helix Universal Server and a broadcast plug-in use the Helix interfaces to stream data to a client. The sample files included with this software development kit (SDK) illustrate many of these features. You can use these sample files as a starting point for building
your own plug-in. For more information on function variables and return values, refer to the Helix Client and Server Software Development Kit header files.

**Note:** The order of function calls listed in the following sections provides a generalized explanation and is for illustrative purposes only. Because Helix is asynchronous, the plug-in must be able to handle any call made to it while it is processing data or waiting for a response from another object. Do not code the plug-in such that it expects a specific sequence of events to occur as it interacts with Helix Universal Server.

### Starting Up

When Helix Universal Server is started, it loads each broadcast plug-in. Helix Universal Server performs a series of calls to your plug-in’s functions and methods in a specific order.

**➤** **To provide start-up support in your broadcast plug-in:**

1. Create a new instance of the broadcast format object by implementing `HXCreateInstance`.

   **For More Information:** See “Creating a Plug-in Instance” in Volume 1, on page 36.

2. Return descriptive information about the plug-in, including its copyright and “more information” URL, by implementing `IHXPlugin::GetPluginInfo`. The `bLoadMultiple` attribute must be set to `FALSE` to ensure that all clients can connect to the same global broadcast object.

   **For More Information:** See “Defining Plug-in Attributes” in Volume 1, on page 37.

3. Return the plug-in short name, a unique string that identifies the plug-in by implementing `IHXBroadcastFormatObject::GetBroadcastFormatInfo`. This short name is the same as the one used by the plug-in in Helix Universal Server’s `FSMount` configuration parameter.

   **For More Information:** See the FS Mount parameter in *Helix Universal Server Configuration File Reference*.

4. Get the information required to determine which file system plug-in to use by implementing the `IHXFileSystemObject::GetFileSystemInfo` method. This method returns the plug-in short name and protocol. The plug-in can return any value for the protocol variable, which is used only when a Helix client plays from its local file system and is ignored for broadcast plug-ins.

### Initializing

When Helix Universal Server receives the first request for a live feed, it selects the broadcast plug-in based on the requested URL’s mount point. The `FSMount` configuration parameter associates plug-ins with URL mount points. (See “File System and File Format Plug-in Interaction” in Volume 1, on page 49.) Helix Universal Server then initializes the plug-in.

**➤** **To provide initialization support for your broadcast plug-in:**

1. Initialize the plug-in by implementing `IHXPlugin::InitPlugin`. Helix Universal Server uses this method to pass a pointer to the system context to your broadcast plug-in. Your plug-in can then initialize a broadcast object from the context. For example:
if(!g_example_broadcast_format)
{
    // Initialize the global object
    g_example_broadcast_format = new ExampleBroadcastFormat(pContext);

    // Set our local pointer to the context in case this instance gets
    // used for anything else
    m_pContext = pContext;
    m_pContext->AddRef();
}

Note: The global broadcast object exists until the live stream ends, whether or not any clients are connected to it.


2. Initialize the broadcast format by implementing IHXBroadcastFormatObject::InitBroadcastFormat. Helix Universal Server calls this method, passing the broadcast plug-in a pointer to the live stream URL. Helix Universal Server also sets itself up as the broadcast response object that receives notification of plug-in actions through IHXFormatResponse.

The plug-in should use the context pointer to store a reference to IHXCommonClassFactory for later use. It should also use the context pointer to query for IHXScheduler if it will use the Scheduler to receive callbacks.

For More Information: See “Using the Scheduler” in Volume 1, on page 127.

3. Notify Helix Universal Server that the initialization has been completed successfully by returning the status code HXR_OK through IHXFormatResponse::InitDone.

For More Information: See “Status Codes” in Volume 1, on page 85.

4. Return the status code HXR_OK when Helix Universal Server calls IHXFileSystemObject::InitFileSystem. Each time a client requests a connection to the live stream, Helix Universal Server performs a series of steps to verify the connection. These are the standard steps that Helix Universal Server uses to verify the existence of any requested file. Your broadcast plug-in must verify that the “file” is an available live stream. After the client connection is validated, the file object is destroyed and the client is connected to the global broadcast object.

➤ To verify the connection to the live stream:

1. Create a file object for the live broadcast by implementing the IHXFileSystemObject::CreateFile method. This method must create an IHXFileObject interface. In turn, the IHXFileObject interface must implement the following interfaces:
   • IHXFileExists
   • IHXBroadcastMapper
   • IHXBroadcastFormatObject

2. Determine if the file exists by implementing the IHXFileExists::DoesExist method. Helix Universal Server uses this method to pass the file object a pointer to the file path. Your plug-in’s file object
calls IHXFileExistsResponse::DoesExistDone to return TRUE if the path is for a valid broadcast “file.” Otherwise, it must return FALSE.

3. Determine the broadcast format implementing the IHXBroadcastMapper::FindBroadcastType method. Helix Universal Server once again passes the file object a pointer to the file path. Your plug-in’s file object must return a status code and the broadcast plug-in short name through IHXBroadcastMapperResponse::BroadcastTypeFound.

   **For More Information:** See “Status Codes” in Volume 1, on page 85.

4. Get the source header data by implementing the IHXBroadcastFormatObject::GetFileHeader method. When Helix Universal Server calls this method, your plug-in retrieves the stream count and any “opaque” data (see the definition in the following section) it needs to send its rendering plug-ins, encapsulating this information in an IHXValues interface and calling IHXFormatResponse::FileHeaderReady to pass Helix Universal Server a status code and a pointer to the values interface.

**Creating Stream Headers**

After your plug-in sends Helix Universal Server the file header for the requested URL, your broadcast plug-in sends Helix Universal Server the stream header for each stream by implementing the IHXBroadcastFormatObject::GetStreamHeader method.

➤ **To send the stream header for each stream:**

1. Create an IHXBuffer interface for each of the following:
   - Stream Name.
     Identifies the stream to the user. The stream name appears in the RealPlayer window.
   - Stream MIME type.
     Identifies the stream to the browser. For example:
     application/x-hx-livestream
   - Opaque Data.
     Any data that the plug-in must pass to its renderer.

2. Create an IHXValues interface that points to the buffer interfaces and that contains the standard stream header properties.

   **For More Information:** See “Creating Stream Headers” in Volume 1, on page 61. For information on stream start times, preroll, and so on, see “Timing and Synchronization” in Volume 1, on page 169.

3. Call IHXFormatResponse::StreamHeaderReady to pass Helix Universal Server a status code and a pointer to the IHXValues stream header object. It then uses IUnknown::Release to release the buffer and values interfaces.

   **For More Information:** See “Status Codes” in Volume 1, on page 85.
If Adaptive Streaming Management (ASM) is used, Helix Universal Server calls your plug-in’s
IHXASMSource::Subscribe method to subscribe to all rules. Your broadcast plug-in must then send all
packets for all ASM rules to Helix Universal Server, which determines which packets to stream to which
clients, based on each client’s rule subscription.

For More Information: See “Chapter 11: Adaptive Stream Management” beginning in
Volume 1, on page 141.

Creating Stream Packets

After receiving the stream header or headers, Helix Universal Server requests packets for each live
stream. (This step is similar to packet creation in a file format plug-in, as described in “Creating Stream
Packets” in Volume 1, on page 62.) Your broadcast plug-in sends Helix Universal Server a packet by
implementing the IHXBroadcastFormatObject::StartPackets method.

➤ To send specified packets for each live stream:

1. Use the broadcast object to start sending Helix Universal Server IHXPacket components that
contain the opaque data passed to the renderer, as well as values for the Helix stream properties.
Typically, your plug-in should send packets to Helix Universal Server at regular intervals, as
prompted by the scheduler.

For the basics of packet creation, see “Using IHXPacket to Create Stream Packets” in
Volume 1, on page 33. For information on receiving callbacks, see “Using the
Scheduler” in Volume 1, on page 127.

2. Continue to send packets until either Helix Universal Server calls your plug-in’s
IHXBroadcastFormatObject::StopPackets method, or until the stream source stops, at which point
your plug-in calls IHXFormatResponse::StreamDone.

A broadcast plug-in should generally create packets of 430 to 500 bytes for the opaque data. Staying
under 500 bytes decreases the likelihood of packet fragmentation. RealNetworks also recommends
that you write code that enables you to quickly change the size of the packets the plug-in sends.

Modifying the Broadcast Plug-in Sample Code

You can use the sample broadcast plug-in as a starting point for building your own plug-in, as in the
following examples:

• /source/samples/server/broadcast/bcastformat.cpp
  This sample file stores the context. For most uses, you do not need to modify this file.

• /source/samples/server/broadcast/broadcast.cpp
  This sample file creates the broadcast object.

You need to create one broadcast object for each stream being broadcast, using the following
procedure.

➤ To modify the sample code:

1. Copy the sample files.
2. Change the file name and class names to match your broadcast format name.

3. In exlivobj.cpp, change the plug-in description, copyright, and “more information” URL stored in zm_pDescription, zm_pCopyright, and zm_pMoreInfoURL.

4. In exlivobj.cpp, change the Multipurpose Internet Mail Extensions (MIME) type and description of each stream in ExampleBroadcastObject::GetStreamHeader to match the type of data in the stream. The stream’s MIME type determines which rendering plug-in is loaded to play that stream.

5. In exlivobj.cpp, use the IHXNetworkServices interface to open a connection to the live data source. (Note that the example code does not currently demonstrate how to use Network Services.)


6. In exlivobj.cpp, read the live source information and create packets to be delivered to Helix Universal Server using ExampleBroadcastObject::StartPackets.

7. Define the plug-in’s mount point in the Helix Universal Server FSMount parameter.

   For More Information: See the FS Mount parameter in Helix Universal Server Configuration File Reference.

8. Compile, debug, and test your plug-in.


Remote Broadcast Services

With Helix, you can stream any data type from a live source. The remote broadcast feature ties a broadcast application into a Helix broadcast plug-in. This minimizes overhead on the broadcast application because the broadcast plug-in manages the connection to Helix Universal Server. As shown in the following illustration, the broadcast application passes buffers of encoded data to remote broadcast services, which then connects to a Helix broadcast plug-in and delivers the stream to Helix Universal Server.

Remote Broadcast Services

![Remote Broadcast Services Diagram]

Suppose that the live broadcast has the following URL:
rtsp://www.real.com/live/newvids

When the broadcast begins, the broadcast application connects to remote broadcast services, passing it the name of the live broadcast “file” (newvids), as well as identifying which broadcast plug-in to use. When the first client requests the live feed, Helix Universal Server determines which broadcast plug-in
to use based on the URL’s mount point (/live/). Helix Universal Server’s FSMount configuration parameter ties the mount point to a specific broadcast plug-in.

For More Information: See the FS Mount parameter in Helix Universal Server Configuration File Reference.

Using Remote Broadcast Services

Helix Producer is an example of an application that uses remote broadcast services. Helix Producer uses its encoding engine to produce a real-time media stream. The real-time media stream is then conveyed to the remote broadcast services module for transmission to a host server.

An application should use remote broadcast services only if Helix DNA Producer or the Helix DNA Producer SDK will not allow the required extent of broadcast functionality. This is because remote broadcast services assumes that the application has fully prepared the media stream for transmission to the end user. Remote broadcast services is simply a data transport interface; it will not encode media nor will it prepare a media session for presentation to the rest of the system.

Remote broadcast services uses a proprietary data transport and control protocol called BCNG. BCNG is a connectionless, stateless data transport that can use UDP, TCP, or Multicast as an underlying IP transport. A single remote broadcast services instance can transmit a single stream to multiple servers using either unicast or multicast, affording redundancy and distribution of client load across recipient reflecting servers. Further, the connectionless state of the BCNG session allows for servers to dynamically enter and exit broadcast sessions without a change of state at the transmitter. This flexibility is important to live deployments in which the recipient servers are clustered.

Use of the remote broadcast services plug-in (rembrdcst.dll in Windows, rembrdcst.so in UNIX) requires that the user make available the XML configuration plug-in (xmlcfg.dll in Windows, xmlcfg.so in UNIX) by setting the path for the plug-ins and support libraries.

Remote Broadcast Components

The following table lists the components you use to broadcast data with Helix. Helix components are included with a Helix Universal Server installation.

<table>
<thead>
<tr>
<th>Data type</th>
<th>Component</th>
<th>On Windows</th>
<th>On UNIX</th>
</tr>
</thead>
<tbody>
<tr>
<td>RealAudio 3.0,</td>
<td>Broadcast library</td>
<td>RealAudio or RealVideo Encoder</td>
<td>RealAudio or RealVideo Encoder</td>
</tr>
<tr>
<td>RealVideo 4.0</td>
<td>Broadcast plug-in (short name: pn-live3)</td>
<td>liv3plin.dll</td>
<td>liv3plin.so</td>
</tr>
<tr>
<td>RealSystem G2 and later</td>
<td>Remote broadcast services</td>
<td>rembrdcst.dll</td>
<td>rembrdcst.so</td>
</tr>
<tr>
<td>data types</td>
<td>Broadcast plug-in (short name: pn-encoder)</td>
<td>brcvplin.dll</td>
<td>brcvplin.so</td>
</tr>
<tr>
<td>RealText</td>
<td>See “RealText Broadcast” in Volume 1, on page 105.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RealPix</td>
<td>See “RealPix Broadcast” in Volume 1, on page 110.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Note: If the remote broadcast feature does not provide all the functionality you require, you can build your own broadcast plug-in as described in “Broadcast Plug-in” in Volume 1, on page 91. As well, if you want to encode directly on the Helix Universal Server computer, you may get better performance by developing a broadcast plug-in.

Interfaces

A Helix component uses the following interfaces to access remote broadcast services:

- **IHXRemoteBroadcastConfiguration**. Header file: hxbrdcst.h.

  A Helix component uses this interface to add or remove a host to which remote broadcast services are being transmitted at any point either before or during a remote broadcast session. The response interface is IHXRemoteBroadcastConfigurationResponse.

- **IHXRemoteBroadcastServices**. Header file: hxencod.h.

  A Helix component uses this interface to control and transmit real-time data streams to Helix Universal Server.

Initializing Remote Broadcast Services

The IHXRemoteBroadcastServices::InitRemoteBroadcast method initializes remote broadcast services, allocating required underlying interfaces and creating the BCNG transport for transmitting the real-time data stream to the recipient. IHXRemoteBroadcastServices::InitRemoteBroadcastServices takes as its parameter a pointer to an IHXBuffer interface that contains the character string representing the path to a configuration file. A remote broadcast services configuration file provides information to the remote broadcast services that describes how to configure the BCNG transport that it uses. The following example shows one possible configuration:

```xml
<!-- Example Configuration file -->
<!-- Change these entries to support your configuration -->
<List Name="BroadcastDistribution">
  <Var SourceName="ExampleSourceName"/>
  <List Name="Destinations">
    <List Name="ExampleName">
      <Var PathPrefix="*"/>
      <Var PortRange="30001-30020"/>
      <Var AcquisitionDataInterval="30"/>
      <Var FECLevel="0"/>
      <Var SureStreamAware="0"/>
      <Var BufferlessTransport="1"/>
      <Var LocalAddress="0.0.0.0"/>
      <Var Address="127.0.0.1"/>
      <Var TTL="16"/>
      <Var ResendSupported="0"/>
      <Var Protocol="udp/unicast"/>
      <List Name="Security">
        <Var Type="Basic"/>
        <Var Password="ExamplePassword"/>
      </List>
    </List>
  </List>
</List>
```
There must also be a corresponding receiver configuration entry on the server(s) to which remote broadcast services will be transmitting. The following example shows a server receiver configuration entry:

```xml
<!--              R E C E I V E R               -->
<List Name="BroadcastReceiver">
  <List Name="Receivers">
    <List Name="Anyone">
      <Var FECLevel="20"/>
      <Var UseTCPForPullBackchannel="0"/>
      <Var OriginSpec="0.0.0.0/0"/>
      <Var AcquisitionDataInterval="30"/>
      <List Name="Security">
        <Var Password="secret"/>
        <Var Type="Basic"/>
      </List>
      <Var Protocol="udp/unicast"/>
      <Var PullSplitEnabled="0"/>
      <Var PortRange="30001-30020"/>
      <Var ResendSupported="0"/>
    </List>
  </List>
</List>
```

BCNG can operate in one of two major modes, either push or pull. To learn more about these modes, consult the “Transmitters and Receivers” chapter of the Helix Universal Server Administration Guide. In essence, the remote broadcast services behaves in the same way as a broadcast transmitter does in a Helix Universal Server broadcast splitting network. The meaning of the configuration values are the same, and are documented in the Helix Universal Server Configuration Reference found at [http://service.real.com/help/library/servers.html#server](http://service.real.com/help/library/servers.html#server).

### Sending a New Media Stream

The IHXRemoteBroadcastServices::NewRemoteBroadcastSession method is invoked when a new media stream is to be sent to the recipients specified in the configuration file. The pSessionName parameter is a
pointer to an IHXBuffer interface that contains a character string representing the name the system uses to identify the broadcast media session. Helix clients use this name to access the stream, and the remote broadcast services use the session name to match against the PathPrefix variable specified in the configuration file to determine which hosts will receive the specified stream. The IUnknown pointer used in this method must point to the source of the media stream. The source of the media stream must implement IHXRawSourceObject, through which remote broadcast services will interact to request the transmission of packets. A majority of the session interaction will occur through the IHXRawSourceObject interface.

Sending the Media Stream

To drive the transmission of packets, and update the schedule timeline maintained by remote broadcast services, the user of remote broadcast services must call IHXRemoteBroadcastServices::Process at regular intervals. The frequency with which IHXRemoteBroadcastServices::Process is called determines the granularity of the scheduler, and therefore the granularity of the packet transmission schedule. A typical use of IHXRemoteBroadcastServices::Process will invoke this method in an application's main loop with a specified yield time to the system for processing.

Closing Remote Broadcast Services

The IHXRemoteBroadcastServices::Close method allows remote broadcast services to deallocate all resources used for transmitting active sessions.

Getting the Current Time

The IHXRemoteBroadcastServices::GetTime method returns the current time of the scheduler timeline maintained by remote broadcast services.

Simulated Live Transfer Agent

For replaying a prerecorded stream as if it were live, Helix exposes the simulated live transfer agent (SLTA) interface. During a simulated live broadcast, viewers who watch a presentation join the event in progress; no matter when visitors connect, they all see the same thing at the same time. Examples of SLTA applications include automated streaming radio presentations, or the continuous playback of previously-recorded broadcast sessions.

Helix Universal Server comes with the SLTA application, which can be used for most simulations. However, you can create your own customized version of this application if you need a particular play list format, a special interface, remote control, or other individual requirement. The iQSLTA library, shipped in the \Lib subdirectory of the Helix Universal Server installation, provides the IHXtQSLTA interface for this purpose.

An implementation of the simulated live transfer agent uses the file format and file system plug-ins in the host server installation to access the pre-encoded files from the specified disk. The file format plug-in appropriate to the format of the pre-encoded media file is used to read abstract packets of data from the file. The SLTA transposes these packets to the required format for transmission over the network, then sends the packets to the recipient server.
Previous releases of the Helix Client and Server Software Development Kit provided the IHXSLTA interface for SLTA applications. The current release of the Helix Client and Server Software Development Kit adds a new IHXiQSLTA interface. The primary difference between the core implementation of these two interfaces is the underlying broadcast transport technology. IHXSLTA uses an RTSP control channel and an RDT data channel. RTSP is an IETF standardized control protocol for multimedia applications. RDT is a proprietary data transport protocol developed by RealNetworks. In contrast, IHXiQSLTA uses the iQ, or BCNG, broadcast transport protocol. BCNG (Broadcast Next Generation) is a proprietary data transport developed by RealNetworks for the transmission of live data between servers and encoders in the Helix system. BCNG is presented as a service through the IHXRemoteBroadcastServices interface.

The BCNG data transport uses a set of semantics that differs from RTSP and RDT. BCNG assumes no connection state between end points in a transmission session. This semantic allows the recipient end point to silently stop receiving data for any reason, including a fault, and to resume session state as soon as it is ready to receive data again. BCNG also supports a number of configurable aspects including the underlying IP transport to use, forward error correction, and retransmission.

The recipient end point uses a broadcast reception plug-in to receive data sent by the SLTA. The broadcast reception plug-in must be configured to receive live streams. In the legacy SLTA, this configuration was done using the RTSP control channel. In the current SLTA, this configuration is done either manually, by adding a specific entry to the Helix Universal Server configuration file, or using an automatic configuration mode that employs HTTP and is provided as a service by the IHXRemoteBroadcastServices interface.

Design Considerations

If you are creating your own simulation application using the IHXiQSLTA interface, keep the following design considerations in mind:

- The iQSLTA library uses the generic file format interface to read the static content files, so it can use any file format plug-in, but has currently only been tested with the RealVideo and RealAudio file format (rmffplin.so on UNIX, rmfformat.dll on Windows). SLTA does not work well with time stamp delivery or “sparse” data types. A sparse data type is one whose presentation duration is larger than its delivery duration. For example, RealPix can send one image, with five seconds as the delivery duration, but move that image around for an hour (presentation duration).

- You must properly set up your application to locate plug-ins and other libraries. Your application must do this by setting the DT_Plugins and DT_Common directories, then calling the SetDLLAccessPath function in the iQSLTA library to pass in these paths.

A file format plug-in can implement some interfaces that help SLTA properly turn the contents from a static file into a live presentation (SLTA currently only supports RealAudio and RealVideo formats). These interfaces are as follows:

- IHXPacketTimeOffsetHandler
- IHXPacketTimeOffsetHandlerResponse
- IHXLiveFileFormatInfo

The file format plug-in implements the IHXPacketTimeOffsetHandler interface. This interface enables the file format plug-in to adjust the time stamps on packets that are being sent in the live stream. Whereas
a group of static files contain their own sets of time stamps for the packets in the file, SLTA expects a set of time stamps that are in logical order. This interface enables the file format plug-in to change the time stamp of the packet being sent from the static file, using a time offset. This way, the time stamp conforms with logical order required by SLTA.

The iQSLTA library supplied with Helix Universal Server implements the IHXPacketTimeOffsetHandlerResponse interface. This interface returns the packet with the time offset.

The file format plug-in implements the IHXLiveFileFormatInfo interface. This interface enables the plug-in to ask a few miscellaneous questions to make live streaming work better for the requested data type. For example, some data types contain packets whose playback durations are fairly long. If a player joins the live stream after one of these packets has been sent, they will be unable to make use of the data in that packet, because they joined too late to receive it. To solve this problem, the IHXLiveFileFormatInfo interface enables the file format plug-in to specify that a stream requires resends. The file format plug-in can then determine how long each packet should be resent (duration) and how often to resend it (bit rate), as well as how those resend packets are constructed. This enables a player who joins after the original packet has already been sent to receive a resent “copy” of that packet at a later time and still be able to make use of some of the packets contents.

**For More Information:** See the section on simulating a live broadcast in the *Helix Universal Server Administration Guide*.

### Initializing the SLTA

The IHXioSLTA::Init method has two function signatures: one for an advanced mode initialization, and the other for a basic mode initialization. The following samples show advanced mode and basic mode function signatures:

**Advanced mode**

```cpp
STDMETHOD(Init) ( 
    THIS_ 
    IHXBuffer* pConfigFilePath, 
    IHXBuffer* pSessionName 
) PURE;
```

**Basic mode**

```cpp
STDMETHOD(Init) ( 
    THIS_, 
    const char* host, 
    UINT16 httpPort, 
    const char* user, 
    const char* password, 
    IHXBuffer* pSessionName, 
    BOOL bTCP 
) PURE;
```

The difference between the modes is the way in which the session is configured. In advanced mode, a special configuration file in XML format is provided to IHXSLTA through the initialization method (IHXioSLTA::Init—Advanced Mode). In basic mode, no XML configuration file needs to be provided since the SLTA will configure itself according to default parameters.
Note: Advanced mode is not discussed in this document. If your application requires specific configuration of BCNG transport parameters, please contact SDK support at RealNetworks at supportsdk@real.com.

The basic mode initialization method (IHXIQiSLTA::Init—Basic Mode) takes as its parameters a string that specifies the host to which to send the live session. The host specification can either be in dotted decimal notation or a fully-qualified domain name. The HTTP port of the recipient host must be specified, along with two strings that represent the username and password for HTTP connections on the recipient host. The final two parameters are the name of the live media session, which will be the name used by client to access the live stream you create, and a boolean parameter that tells the SLTA to use TCP as the underlying transport if set to TRUE.

Setting the Title, Author, and Copyright

The IHXIQiSLTA::SetTAC method sets the title, author, and copyright information for the live stream being transmitted to the server. This information is updated in the end client user interface using the events data channel. The events data channel is updated at a very low data rate. It is therefore suggested that the title, author, and copyright information be set at the beginning of the session, before calling IHXIQiSLTA::BeginTransmission, so that some initial values are transmitted to connecting clients. The rate at which these values are updated is 100 bits per second (bps).

Beginning Transmission

The IHXIQiSLTA::BeginTransmission method tells the SLTA module to begin sending the media stream to the recipient server. For IHXIQiSLTA::BeginTransmission to have meaning IHXIQiSLTA::Init must have already been called successfully. IHXIQiSLTA::BeginTransmission takes as its parameters the path to the pre-encoded media stored on disk, or the MediaName, and the name of the session that will represent the live stream of the pre-encoded media on the recipient server, or SessionName. The SessionName is used by clients to request the stream from the recipient server.

IHXIQiSLTA::BeginTransmission first attempts to find the specified media file. If the media file is found, it begins packetizing and sending the media to the recipient server using the BCNG protocol. If the media file is not found, it returns an HXR_FILE_NOT_FOUND error. IHXIQiSLTA::BeginTransmission will not return until the entire media is transmitted or IHXIQiSLTA::EndTransmission is called.

IHXIQiSLTA::BeginTransmission will loop, yielding processing time to the host system, while it is transmitting data. An application that uses IHXIQiSSLTA may require that IHXIQiSLTA::BeginTransmission be called from a separate thread from a user interaction thread, since IHXIQiSLTA::BeginTransmission will block execution until it returns.

Ending Transmission

The IHXIQiSLTA::EndTransmission method tells the IHXIQiSLTA implementation to stop sending packets, clean up the existing live session, and terminate the transmission loop initiated by the IHXIQiSLTA::BeginTransmission method. This method can be called at any point after IHXIQiSLTA::BeginTransmission is called, and must be called after all files have been sent by IHXIQiSLTA::BeginTransmission.
Closing the SLTA

The IHXiQSLTA::Close method completes deallocation of all resources allocated by the IHXiQSLTA implementation. An application of IHXiQSLTA should invoke IHXiQSLTA::Close to insure that the application does not cause memory leaks after it has completed its use of IHXiQSLTA.

Setting Target Bandwidth

The IHXiQSLTA::SetTargetBandwidth method allows the user to select a target encoding rate from a multirate SureStream encoded file. For example, if the pre-encoded media consists of SureStream encodings at 150 Kbps, 256 Kbps, and 340 Kbps, then the user can select which of these encodings to send by specifying the target bandwidth through IHXiQSLTA::SetTargetBandwidth. If this method is not invoked, then all data rates are sent simultaneously and clients are allowed to select which data rate to receive.

RealText Broadcast

The RealText broadcast feature enables you to stream live broadcasts of RealText over a network. You can use it, for example, to broadcast a live stock ticker feed. To set up a text broadcast, you create a broadcast application that adds RealText markup to a text feed. You then integrate the broadcast application with the RealText Broadcast Library through a single system interface. That library takes care of the overhead of integrating with the standard Helix Remote Broadcast Library and a Helix broadcast plug-in, which manages the connection to Helix Universal Server.


RealText Broadcast

The following table lists the components you use to broadcast RealText content with Helix. These components are included with a Helix Universal Server installation.

<table>
<thead>
<tr>
<th>RealText Broadcast Components</th>
<th>On Windows</th>
<th>On UNIX</th>
</tr>
</thead>
<tbody>
<tr>
<td>RealText Broadcast Library</td>
<td>rtlive.dll</td>
<td>rtlive.so</td>
</tr>
<tr>
<td>Remote Broadcast Library</td>
<td>encnet.dll</td>
<td>encnet.so</td>
</tr>
<tr>
<td>Broadcast plug-in (short name: hx-encoder)</td>
<td>encoplin.dll</td>
<td>encoplin.so</td>
</tr>
<tr>
<td>SDP stream description plug-in</td>
<td>sdpplin.dll</td>
<td>sdpplin.so</td>
</tr>
<tr>
<td>Authentication manager</td>
<td>authmgr.dll</td>
<td>authmgr.so</td>
</tr>
<tr>
<td>RealSystem 5.0 encryption manager</td>
<td>rn5auth.dll</td>
<td>rn5auth.so</td>
</tr>
</tbody>
</table>
Suppose that the live broadcast has the following URL:
rtsp://www.real.com/encoder/ticker

When the broadcast begins, the broadcast application connects to the RealText Broadcast Library, passing it the name of the live broadcast “file” (ticker), as well as identifying which broadcast plug-in to use. When the first client requests the live feed, Helix Universal Server determines which broadcast plug-in to use based on the URL’s mount point (/encoder/). Helix Universal Server’s FSMount configuration parameter ties the mount point to a specific broadcast plug-in.

**For More Information:** See the FS Mount parameter in *Helix Universal Server Configuration File Reference.*

### Interfaces

The RealText broadcast application uses the following interface:

**IHXLiveText. Header file:** hxlvtxt.h.

The RealText Broadcast Library implements this interface. The broadcast application uses this interface to send window parameters, text, and markup to the library. There is no response interface.

### Coding the Remote Text Broadcast Application

The following sections explain how the text broadcast application interacts with the RealText Broadcast Library. The sample files included with this SDK illustrate many of these features. You can use the sample files as a starting point for building your own broadcast application.

#### Initializing the RealText Broadcast Library

When a RealText broadcast application is started, it performs the following actions:

1. The application loads the RealText Broadcast Library and the Remote Broadcast Library listed in the table “RealText Broadcast Components” in Volume 1, on page 105.
2. The application gets the RealText library’s CreateLiveText process address, as shown in this extract from the sample file:
   ```
m_fpCreateLiveText = (FPRMCREATELIVETEXT) GetProcAddress(hDllRtlive, “CreateLiveText”);
   ```
3. The application uses the CreateLiveText function to create a live text encoding object:
   ```
   if(HXR_OK == m_fpCreateLiveText((IHXLiveText**) &pLiveTextEncoder))
   ```
4. The application uses IHXLiveText::InitLiveText to pass the live text object the following parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Defines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host</td>
<td>The host name or IP address of Helix Universal Server.</td>
</tr>
<tr>
<td>Port</td>
<td>The server port the broadcast plug-in listens on. This plug-in’s FSMount setting defines this.</td>
</tr>
</tbody>
</table>

(Table Page 1 of 2)
5. The application calls IHXLiveText::EncoderIsInitialized, which returns a Boolean value, to determine whether the library is initialized and ready to receive text.

Defining the RealText Window

Before sending data, the broadcast application can define RealText <window> tag properties. Because this tag is typically not included in a live broadcast, the application calls the following IHXLiveText methods to set the properties:

- IHXLiveText::SetBackgroundColor—Controls the window background color.
- IHXLiveText::SetDoLooping—Determines whether looping occurs.
- IHXLiveText::SetHyperlinkInfo—Specifies whether hyperlinks are underlined and sets the hyperlink color.
- IHXLiveText::SetTextMotion—Sets the window scroll rate and crawl rate.
- IHXLiveText::SetType—Determines the window type (generic, tickertape, and so on).
- IHXLiveText::SetWindowDimensions—Controls the window width and height in pixels.
- IHXLiveText::UseWordwrap—Sets word wrap on or off.

Note that for any one of these methods that the application does not use, the default values for the associated properties are used.

Sending Text to the Library

After initializing the RealText Broadcast Library, the broadcast application passes text to the library, which then creates the Helix packets transmitted by the broadcast plug-in. The broadcast application uses the IHXLiveText methods to pass the text.

The IHXLiveText::AddData method passes the text (char) and includes a Boolean value that, when set to TRUE, causes the library to broadcast all text in its buffer immediately. If the value is FALSE, the library holds the data until either there is enough of it to make a packet approximately 500 bytes in size, or 2 seconds have elapsed since the last packet was sent.

The text passed by IHXLiveText::AddData can include any RealText markup. If the text does not include <time begin> tags, the broadcast library time stamps the text as it is received. So if the broadcast application sends text string B to the library one second after it sends string A, the clients' RealText renderers display string B one second after displaying string A.
IHXLiveText::AddTickerItem is used only with tickertape windows, and passes parameters (char) for both the upper ticker value and the lower ticker value. Like IHXLiveText::AddData, it includes a Boolean value for sending all buffered text immediately. When you use this method, you do not need to enclose the text in <tu> and <tl> markup tags.

Calling IHXLiveText::flush causes the library to broadcast and clear all data in its buffer. An IHXLiveText::AddData or IHXLiveText::AddTickerItem call with the Boolean parameter set to TRUE also flushes the buffer.

The IHXLiveText::GetTime method returns the number of milliseconds that have elapsed since the broadcast application started.

An application calls IHXLiveText::Process after calling IHXLiveText::AddData, IHXLiveText::AddTickerItem, or IHXLiveText::flush to enable the library to perform necessary processing. When not sending text, the application should also call this method approximately every five seconds. This method is required due to the asynchronous nature of Helix.

Communicating with the Library

The IHXLiveText interface does not have a corresponding response interface that the broadcast application implements to receive feedback from the RealText Broadcast Library. The application can call specific methods, however, to determine the library’s status.

The IHXLiveText::EncoderIsDone method, called after IHXLiveText::SetEncoderDone, returns a Boolean value that indicates if the library has broadcast all text it has received. The IHXLiveText::EncoderIsInitialized method returns a Boolean value indicating that the library has received the initialization parameters that enable it to connect to the Helix Universal Server broadcast plug-in. After initialization, the library is ready to receive text. The Boolean value returned for IHXLiveText::PacketsHaveStarted indicates whether the library has started sending packets to the broadcast plug-in. The broadcast application calls IHXLiveText::SetEncoderDone to notify the library that it has finished sending text.

Modifying the RealText Broadcast Sample Code

Interfaces to the RealText Broadcast Library are defined in hxlvtxt.h. Sample code for streaming text that uses the library is provided in the following files:

- /source/samples/datatype/live_realtext/main.cpp
  This sample file builds a simple encoder that sends text every 5.5 seconds to a tickertape window. This text contains the amount of time that has elapsed since the encoder started and has a hyperlink to a United States government atomic clock Web page to demonstrate how to add hyperlinks to live text.

- /source/samples/datatype/live_realtext/main2.cpp
  The more complex main2.cpp streams the entire contents of an input text file every time it detects that the file has been updated (which it does by seeing if the file has been saved since the last check). It also prompts for window type, window size, and window background color.

The sample code includes a simple command-line interface for gathering the required initialization parameters. Both main.cpp and main2.cpp contain comments telling you how to modify the
applications, including where text is added to the stream. The comments in hxlvtxt.h tell how to alter the stream’s window attributes, such as background color, height, and width.

➤ To modify the sample code and implement remote broadcasting for your application:

1. Install the broadcast libraries in a directory on your path (LD_LIBRARY_PATH for UNIX). These libraries are listed in the table “RealText Broadcast Components” in Volume 1, on page 105.
2. Copy the sample code to a working directory. Change the file names to match your application name.
3. Provide a way for your application to get the required initialization parameters, and then initialize the broadcast library. For example:
   ```c
   pLiveTextEncoder->InitLiveText(szHost, atoi(szPort), szUsername, szPassword, szFilename);
   ```
4. Write the application-specific processing code. Be sure to call IHXLiveText::Process frequently to enable the library to perform its necessary processing.
5. Compile your application. The application does not need to link to the broadcast libraries as long as it calls LoadLibrary as shown in the main.cpp sample file.

   For More Information: For general build instructions, see “Building a Sample Plug-in” in Volume 1, on page 43.

➤ To test broadcasting with your application:

1. Make sure that the broadcast plug-in is installed in Helix Universal Server’s plug-in directory.
2. Edit the Helix Universal Server FSMount parameter to specify the broadcast plug-in’s mount point, short name, port, and, optionally, its password. If the parameter does not specify a password, passwords passed to the broadcast library are ignored and any broadcast application can connect to the plug-in.

   For More Information: See the FS Mount parameter in Helix Universal Server Configuration File Reference.
   Start the broadcast application, connecting it to the port defined in the plug-in’s FSMount parameter. The following example shows how to do this from the command line for an application connecting to Helix Universal Server port 7072 and broadcasting a file named ticker (the syntax may differ for your application):
   ```bash
   % exlvtext helixserver.example.com 7072 <user name> <password> ticker
   ```
4. Debug the process of receiving a live text stream in RealPlayer. Using File>Open Location, enter the URL for receiving the live stream. The URL should be in the following form:
   ```plaintext
   rtsp://serverhost.domain.com/mountpoint/filename
   ```
   Thus, in the preceding ticker broadcast example, the URL would be as follows:
   ```plaintext
   rtsp://rmaserver.site.com/encoder/ticker
   ```
   RealPlayer should buffer the broadcast and then play it. Note that the scroll bar is disabled when RealPlayer receives live content.
5. Debug the process of broadcasting multiple streams—one of which is live—to RealPlayer. Do this by creating a Synchronized Multimedia Integration Language (SMIL) file that groups one live track of your data type with another data type.

Creating a Broadcast RealText Clip

Create a RealText clip for broadcast as a simple text file. Use the RealText mark-up tags described in RealNetworks Production Guide to format the display text. The file should not have <window> and </window> tags. You set window attributes such as type, width, and height when you start the broadcast application. Helix Universal Server sends these attributes to each media player when it connects to the broadcast.


Tips for Creating Broadcast RealText Files

Here are some pointers on using RealText mark-up when broadcasting a file:

- Start a new input file with the <clear/> tag to clear the existing text in the RealPlayer window. Do not use a <clear/> tag in the middle of an input file, though. In a broadcast, a <clear/> tag does not need to follow a <time begin.../> tag.
- Avoid using <time.../> tags to specify when text appears. Timing elements specify offsets from the start of the broadcast, not from the receipt of the RealText update that includes the elements. Using timing tags can therefore cause unexpected results.
- Keep RealText broadcast files small and broadcast them in a “real time” manner. Do not send large files that use RealText timing tags to delay when the text appears in the RealPlayer window.

RealPix Broadcast

The RealPix broadcast feature makes it possible for you to stream “live” RealPix presentations over a network. You can use it, for example, to broadcast JPEG images captured at regular intervals by a camera. To set up a RealPix broadcast, you create a broadcast application that adds RealPix markup to streaming images. You then integrate the broadcast application with the RealPix Broadcast Library through a single system interface. That library takes care of the overhead of integrating with the standard RealSystem Remote Broadcast Library and a Helix broadcast plug-in, which manages the connection to Helix Universal Server.

The following table lists the components you use to broadcast LiveRealPix with Helix. These components are included with a Helix Universal Server installation.

<table>
<thead>
<tr>
<th>LiveRealPix Broadcast Components</th>
<th>On Windows</th>
<th>On UNIX</th>
</tr>
</thead>
<tbody>
<tr>
<td>LiveRealPix Broadcast Library</td>
<td>pxlive.dll</td>
<td>pxlive.so</td>
</tr>
<tr>
<td>Remote Broadcast Library</td>
<td>encnet.dll</td>
<td>encnet.so</td>
</tr>
<tr>
<td>Broadcast plug-in (short name: hx-encoder)</td>
<td>encoplin.dll</td>
<td>encoplin.so</td>
</tr>
</tbody>
</table>

Suppose that the live broadcast has the following URL:

rtsp://www.real.com/encoder/image

When the broadcast begins, the broadcast application connects to the LiveRealPix Broadcast Library, passing it the name of the live broadcast “file” (image), as well as identifying which broadcast plug-in to use. When the first client requests the live feed, Helix Universal Server determines which broadcast plug-in to use based on the URL’s mount point (/encoder/). Helix Universal Server’s FSMount configuration parameter ties the mount point to a specific broadcast plug-in.

For More Information: See the FS Mount parameter in Helix Universal Server Configuration File Reference.

Interfaces

The LiveRealPix broadcast application uses the following interfaces:

- **IHXLiveRealPix.** Header file: hxlvpix.h.
  
The LiveRealPix Broadcast Library implements this interface. The broadcast application uses this interface to send initialization parameters, images, effects, and markup to the library.

- **IHXLiveRealPixResponse.** Header file: hxlvpix.h.

  A LiveRealPix broadcast application implements this interface and uses it to respond to various IHXLiveRealPix methods.

Coding the Remote Pix Broadcast Application

The following sections explain how the RealPix broadcast application interacts with the LiveRealPix Broadcast Library. The sample file included with this SDK illustrates many of these features. You can use the sample file as a starting point for building your own broadcast application.

Initializing the LiveRealPix Broadcast Library

When a LiveRealPix broadcast application is started, it performs these actions:

1. It loads the LiveRealPix Broadcast Library and the Remote Broadcast Library listed in the table “RealText Broadcast Components” in Volume 1, on page 105.

2. It gets the LiveRealPix library’s CreateLiveRealPix process address, as shown in this extract from the sample file:
FPRMCREateliveRealPix fpCreateLiveRealPix = (FPRMCREateliveRealPix) GetProcAddress(hRPLiveDLL, "CreateLiveRealPix");

3. It can now create an instance of the IHXLiveRealPix interface, as in the following example:

```c
IHXLiveRealPix *pLiveRealPix = NULL;
HX_RESULT        retVal       = fpCreateLiveRealPix(&pLiveRealPix);
```

4. Because most of the IHXLiveRealPix methods return responses, the application must also create an instance of the IHXLiveRealPixResponse interface, as in the following example:

```c
IHXLiveRealPixResponse *pLiveRealPixResponse = NULL;
HX_RESULT retVal = fpCreateLiveRealPixResponse(&pLiveRealPixResponse);
```

5. It uses IHXLiveRealPix::StartEncoder to pass the broadcast library a pointer to the IHXLiveRealPixResponse interface, along with a PixInitInfo structure containing the parameters listed and described in the following table.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Defines</th>
</tr>
</thead>
<tbody>
<tr>
<td>ServerAddress</td>
<td>The host name or IP address of Helix Universal Server.</td>
</tr>
<tr>
<td>ServerPort</td>
<td>The server port that the broadcast plug-in listens on. This plug-in’s FSMount setting defines this.</td>
</tr>
<tr>
<td>Username</td>
<td>The name the application uses to connect to the broadcast plug-in. It should be encoder if a password is used.</td>
</tr>
<tr>
<td>Password</td>
<td>The password that the application uses to connect to the broadcast plug-in. The plug-in’s FSMount setting defines this.</td>
</tr>
<tr>
<td>Filename</td>
<td>The “file name” that follows the broadcast plug-in’s mount point in the URL. For example, in rtsp://www.real.com/encoder/image, the broadcast plug-in has a mount point of /encoder/ and the file name is image.</td>
</tr>
<tr>
<td>Title</td>
<td>The title of the image.</td>
</tr>
<tr>
<td>Author</td>
<td>The name of the person or organization that created the image.</td>
</tr>
<tr>
<td>Copyright</td>
<td>The copyright information for the image.</td>
</tr>
<tr>
<td>Bit rate</td>
<td>The rate (in kilobits per second, or Kbps) at which the specified image is to be sent.</td>
</tr>
<tr>
<td>MaxFPS</td>
<td>The upper limit on the number of effects frames to be sent per second.</td>
</tr>
<tr>
<td>DisplayWidth</td>
<td>The width of the presentation window, in pixels.</td>
</tr>
<tr>
<td>DisplayHeight</td>
<td>The height of the presentation window, in pixels.</td>
</tr>
<tr>
<td>PreserveAspect</td>
<td>The Boolean value that states whether the aspect ratio of image is to be preserved. If TRUE, the image aspect ratio is preserved.</td>
</tr>
<tr>
<td>DefaultURL</td>
<td>The default URL that the browser will send the client to when the image is clicked.</td>
</tr>
</tbody>
</table>

(Table Page 1 of 2)
6. The application calls IHXLivelRealPix::Process to give the library time to initialize. The library returns a value indicating whether the initialization was successful.

Sending Images to the Library

After initializing the LiveRealPix Broadcast Library, the broadcast application passes images to the library, which then creates the Helix packets transmitted by the broadcast plug-in. The broadcast application uses the following IHXLivelRealPix methods to pass the images:

- **IHXLiveRealPix::InitImage**
  Prepares an image for sending. It passes the live pix object a PixImageInfo structure containing the following parameters:
  - **ImageBuffer** The buffer that contains the image data.
  - **ImageBufferSize** The size of the image buffer.
  - **ImageCodec** The image codec to be used.
  - **Handle** The handle of the image; used in IHXLivelRealPix::SendImage.
  - **NumPackets** The number of packets the image will comprise.
  - **TimeToSend** The amount of time (in milliseconds) required to send the image.

Remember that IHXLivelRealPix::InitImage simply breaks the image up into packets, and does not send anything to the server. Note also that IHXLivelRealPix::InitImage is not an asynchronous call—all processing has finished by the time this call returns.

- **IHXLiveRealPix::SendImage**
  Transfers all of the image’s packets (the image referenced by the handle returned by IHXLivelRealPix::InitImage) into the packet send queue. Subsequent calls to IHXLivelRealPix::Process will cause these packets to be sent to the Helix Universal Server.

When all the packets for this image have been sent, the encoder will respond with IHXLivelRealPixResponse::ImageSent.

- **IHXLiveRealPix::SendEffect**
  Creates an effects packet with the information contained in the PixEffectInfo structure and immediately adds this packet to the packet send queue. The packet contains the parameters listed

### Live Pix Initialization Parameters (continued)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Defines</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumImageCodecs</td>
<td>The number of image codecs used in the specified stream.</td>
</tr>
<tr>
<td>ImageCodec</td>
<td>The string names of codecs. The codec string names currently supported are &quot;image/vnd.rn-realpix.jpeg&quot; (for JPEG images) and &quot;image/vnd.rn-realpix.gif&quot; (for GIF images).</td>
</tr>
<tr>
<td>NumEffectsPackages</td>
<td>The number of external effect packages used.</td>
</tr>
<tr>
<td>EffectPackage</td>
<td>The string names of effect packages.</td>
</tr>
</tbody>
</table>

(Table Page 2 of 2)
and described in the following table. Further calls to `IHXLiveRealPix::Process` result in this packet being sent to the server. A handle is returned in the `PixEffectInfo` structure by which this effect can later be identified.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Defines</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>EffectType</strong></td>
<td>The effect type: EFFECT_FILL, EFFECT_FADEIN, etc. See hxlvpix.h for more information.</td>
</tr>
<tr>
<td><strong>Start</strong></td>
<td>The start time for the effect, in milliseconds.</td>
</tr>
<tr>
<td><strong>Duration</strong></td>
<td>The duration of the effect, in milliseconds.</td>
</tr>
<tr>
<td><strong>Target</strong></td>
<td>The handle of the image that the effect is to be performed on (returned by <code>IHXLiveRealPix::InitImage</code>).</td>
</tr>
<tr>
<td><strong>SrcX</strong></td>
<td>The horizontal offset of the source rectangle. This parameter, and the three source parameters that follow, can be used to display just a portion of the original image. To display the entire image, set these all to 0 (zero).</td>
</tr>
<tr>
<td><strong>SrcY</strong></td>
<td>The vertical offset of the source rectangle.</td>
</tr>
<tr>
<td><strong>SrcW</strong></td>
<td>The width of the source rectangle.</td>
</tr>
<tr>
<td><strong>SrcH</strong></td>
<td>The height of the source rectangle.</td>
</tr>
<tr>
<td><strong>DstX</strong></td>
<td>The horizontal offset of the destination rectangle. This parameter, and the three destination parameters that follow, are used to position and size the image within the presentation space.</td>
</tr>
<tr>
<td><strong>DstY</strong></td>
<td>The vertical offset of the destination rectangle.</td>
</tr>
<tr>
<td><strong>DstW</strong></td>
<td>The width of the destination rectangle.</td>
</tr>
<tr>
<td><strong>DstH</strong></td>
<td>The height of the destination rectangle.</td>
</tr>
<tr>
<td><strong>MaxFps</strong></td>
<td>The maximum number of frames per second for this effect.</td>
</tr>
<tr>
<td><strong>AspectFlag</strong></td>
<td>TRUE to preserve the aspect ratio for this effect.</td>
</tr>
<tr>
<td><strong>Red</strong></td>
<td>The red component of the fill or fadeout color.</td>
</tr>
<tr>
<td><strong>Green</strong></td>
<td>The green component of the fill or fadeout color.</td>
</tr>
<tr>
<td><strong>Blue</strong></td>
<td>The blue component of the fill or fadeout color.</td>
</tr>
<tr>
<td><strong>WipeDirection</strong></td>
<td>WIPE_DIRECTION_xxx, where xxx is UP, DOWN, LEFT, or RIGHT.</td>
</tr>
<tr>
<td><strong>WipeType</strong></td>
<td>WIPE_TYPE_NORMAL or WIPE_TYPE_PUSH.</td>
</tr>
<tr>
<td><strong>URL</strong></td>
<td>The URL for the Web location that the client will be sent to when the image is clicked.</td>
</tr>
<tr>
<td><strong>ExtPackage</strong></td>
<td>The name of the external effect package.</td>
</tr>
<tr>
<td><strong>ExtName</strong></td>
<td>The name of the external effect within the package.</td>
</tr>
<tr>
<td><strong>ExtData</strong></td>
<td>The opaque string data for the external effect.</td>
</tr>
<tr>
<td><strong>ExtFile</strong></td>
<td>The file contents of external effect data.</td>
</tr>
</tbody>
</table>

(Table Page 1 of 2)
When the effect has been sent to the server, the encoder will respond with IHXLiveRealPixResponse::EffectSent.

• IHXLiveRealPix::GetTime

Returns the amount of time (in milliseconds) that has elapsed since the encoder library was initialized. This value can be used in conjunction with the Start effect parameter and the TimeToSend parameter returned by IHXLiveRealPix::InitImage to control the timing of the presentation.

• IHXLiveRealPix::Process

Must be called to give the library time to send packets to the server. It should be called frequently between IHXLiveRealPix::SendImage and IHXLiveRealPixResponse::ImageSent, as well as between IHXLiveRealPix::SendEffect and IHXLiveRealPixResponse::EffectSent. At all other times, it should be called every 3 to 5 seconds.

When designing your LiveRealPix broadcast application, keep the following points in mind:

• Sending an image with IHXLiveRealPix::SendImage does not cause an image to be displayed; IHXLiveRealPix::SendEffect does this. You can send several images to the client in advance; you can then control the placement and timing of the display by using IHXLiveRealPix::SendEffect.

• IHXLiveRealPix::SendEffect can do more than just display an image. For instance, to change the target URL for an image already displayed, you send a fade-in effect with a duration of 0 (zero) and the new URL, and the link will change without any visible effect.

• Using the effects parameters that control the size and placement of the images, you can display numerous images in the presentation space, and you can control each of them independently of the others.

Stopping the LiveRealPix Broadcast Library

When the broadcast application finishes sending images to the broadcast library, it calls IHXLiveRealPix::StopEncoder, which causes the library to shut down the connection with the Helix Universal Server. The encoder will respond with IHXLiveRealPixResponse::EncoderStopped.

Modifying the RealPix Broadcast Sample Code

Interfaces to the RealText Broadcast Library are defined in hxlvpix.h. Sample code for a LiveRealPix application that uses the library is provided in the following file:

/source/samples/datatype/live_realpix/live_realpix.cpp
This simple LiveRealPix broadcast application loads the LiveRealPix Library, initializes it with some passed-in arguments, then sends an image from a file on disk, along with a fade-in effect. It then watches the file for changes to its size or last modification time, and resends the image, using IMXLiveRealPix::SendImage whenever it detects a change to the file. It also sends a fade-in effect, which causes the image just sent to display. The application calls IMXLiveRealPix::Process frequently to give processor time to the Broadcast Library.


➤ To modify the sample code and implement remote broadcasting for your application:
Install the broadcast libraries in a directory on your path (LD_LIBRARY_PATH on UNIX). These libraries are listed in the table “RealText Broadcast Components” in Volume 1, on page 105.

7. Copy the sample code to a working directory. Change the file names to match your application name.

8. Provide a way for your application to get the required initialization parameters, then initialize the broadcast library. For example:
   ```
   pLiveRealPix->StartEncoder(&cInitInfo, this)
   ```
   where cInitInfo is a structure of the type PixInitInfo. See hxlvpix.h for more information.

9. Write the application-specific processing code. Be sure to call IMXLiveRealPix::Process frequently to enable the library to perform its necessary processing.

10. Compile your application. The application does not need to link to the broadcast libraries as long as it calls LoadLibrary as shown in exlivpix.cpp.


➤ To test broadcasting with your application:

1. Make sure that the broadcast plug-in is installed in Helix Universal Server’s plug-in directory.

2. Edit the Helix Universal Server FSMount parameter to specify the broadcast plug-in’s mount point, short name, port, and, optionally, its password. If the parameter does not specify a password, passwords passed to the broadcast library are ignored and any broadcast application can connect to the plug-in.

     For More Information: See the FS Mount parameter in Helix Universal Server Configuration File Reference.

   Start the broadcast application, connecting it to the port defined in the plug-in’s FSMount parameter. The following example shows how to do this from the command line for an application connecting to Helix Universal Server port 7072 and broadcasting a file named image (the syntax may differ in your application):
   ```
   % exlivpix rmaserver.site.com 7072 <user name> <password> image
   ```
4. Debug the process of receiving a live image stream in RealPlayer. Using **File>Open Location**, enter the URL for receiving the live stream. The URL should be in the following form:

```
rtsp://serverhost.domain.com/mountpoint/filename
```

Thus, in the preceding image broadcast example, the URL would be as follows:

```
rtsp://rmaserver.site.com/encoder/image
```

RealPlayer should buffer the broadcast and then play it. Note that the scroll bar is disabled when RealPlayer receives live content.

5. Debug the process of sending multiple streams—one of which is live—to RealPlayer. Do this by creating a Synchronized Multimedia Integration Language (SMIL) file that groups one live track of your data type with another data type.
Chapter 9: MANAGING THE SERVER REGISTRY

Helix Universal Server maintains a property registry that contains a variety of information about the server’s configuration and current load, connected clients, and each client’s sessions. You can also use the Helix Universal Server property registry to store new, user-defined properties, which can be changed or deleted by the process that created them. You can use server plug-ins to retrieve all of these properties, whether predefined or user-defined, using interfaces designed for this purpose. You can also have server plug-ins be notified when specific properties change.

For More Information: See “Creating a Monitor Plug-in” in Volume 1, on page 124 to learn how to create a plug-in that monitors properties in the Helix Universal Server property registry.

Registry Basics

The Helix Universal Server property registry is a hierarchical structure of name/value pairs (properties) that is capable of storing many different types of data including strings, buffers, and integers. The registry provides various types of information including server and client statistics, configuration information, and system status. Each property in the registry has a name, ID, data type, and value. The registry interfaces provided by Helix Universal Server include methods for the setting and retrieval of properties by name or by ID.

Registry Data Types

The following table lists the registry data types currently available for use. The Methods column lists the methods supported for each data type.

<table>
<thead>
<tr>
<th>Data type</th>
<th>Method</th>
<th>Method description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer Signed 32-bit</td>
<td>IHXRegistry::AddInt</td>
<td>Adds a new property.</td>
</tr>
<tr>
<td></td>
<td>IHXRegistry::GetIntByName</td>
<td>Retrieves a value using the property name.</td>
</tr>
<tr>
<td></td>
<td>IHXRegistry::GetIntById</td>
<td>Retrieves a value using the property ID.</td>
</tr>
<tr>
<td></td>
<td>IHXRegistry::SetIntByName</td>
<td>Sets a value using the property name.</td>
</tr>
<tr>
<td></td>
<td>IHXRegistry::SetIntById</td>
<td>Sets a value using the property ID.</td>
</tr>
<tr>
<td>String null-terminated unsigned char* within an IHXBuffer</td>
<td>IHXRegistry::AddStr</td>
<td>Adds a new property.</td>
</tr>
<tr>
<td></td>
<td>IHXRegistry::GetStrByName*</td>
<td>Retrieves a value using the property name.</td>
</tr>
<tr>
<td></td>
<td>IHXRegistry::GetStrById*</td>
<td>Retrieves a value using the property ID.</td>
</tr>
<tr>
<td></td>
<td>IHXRegistry::SetStrByName</td>
<td>Sets a value using the property name.</td>
</tr>
<tr>
<td></td>
<td>IHXRegistry::SetStrById</td>
<td>Sets a value using the property ID.</td>
</tr>
</tbody>
</table>

* Takes as a parameter an IHXBuffer interface in which it passes back the retrieved string. (Table Page 1 of 2)
As you build your monitor plug-in, keep in mind the following about registry data types:

- Property names are not case sensitive.
- Because a composite property is a combination of other property types, it does not have a value.
- An integer reference property is for values that vary rapidly, perhaps several times a second. It has as its value a pointer to an integer variable that you have “malloced” from the heap. Only the property creator can modify its value, which he or she can accomplish without going through the registry. All other users can access this integer reference property with IHXRegistry::GetIntByName or IHXRegistry::GetIntById.

**Predefined Server Properties**

The following table lists the server properties predefined in the Helix Universal Server registry.

<table>
<thead>
<tr>
<th>Property name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>server</td>
<td>Composite</td>
<td>Helix Universal Server connection statistics.</td>
</tr>
<tr>
<td>server.ClientCount</td>
<td>Integer</td>
<td>Number of clients connected to Helix Universal Server.</td>
</tr>
<tr>
<td>server.PNAClientCount</td>
<td>Integer</td>
<td>Number of clients connected using the PNA protocol.</td>
</tr>
<tr>
<td>server.RTSPClientCount</td>
<td>Integer</td>
<td>Number of clients connected using RTSP protocol.</td>
</tr>
<tr>
<td>server.HTTPClientCount</td>
<td>Integer</td>
<td>Number of clients connected using HTTP protocol.</td>
</tr>
<tr>
<td>server.UDPTransportCount</td>
<td>Integer</td>
<td>Number of clients connected using UDP.</td>
</tr>
<tr>
<td>server.TCPTransportCount</td>
<td>Integer</td>
<td>Number of clients connected using TCP.</td>
</tr>
<tr>
<td>server.MulticastTransportCount</td>
<td>Integer</td>
<td>Number of clients connected through Multicast.</td>
</tr>
<tr>
<td>server.PercentCPUUsage</td>
<td>Integer</td>
<td>Current server CPU use (0 to 100 percent).</td>
</tr>
<tr>
<td>server.BytesMemoryUsage</td>
<td>Integer</td>
<td>Number of bytes of memory the server is currently using.</td>
</tr>
</tbody>
</table>

* Takes as a parameter an IHXBuffer interface in which it passes back the retrieved string. (Table Page 2 of 2)
Predefined Client Properties

The following table lists the predefined registry properties for each connected client. The \( n \) in the property name is a placeholder for the client’s ID. Note that only the client property is present at all times. The client.\( n.* \) properties are present only as long as client.\( n \) is connected. Helix Universal Server deletes these properties from the registry as clients disconnect. Also note that the Session attributes are not available when the client connects, but become available as the client/server control negotiation progresses.

<table>
<thead>
<tr>
<th>Property name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>server.Platform</td>
<td>String</td>
<td>Server operating system.</td>
</tr>
</tbody>
</table>

Predefined Server Registry Properties (continued)

<table>
<thead>
<tr>
<th>Property name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>client</td>
<td>Composite</td>
<td>Client statistics.</td>
</tr>
<tr>
<td>client.( n )</td>
<td>Composite</td>
<td>Statistics for client.( n ).</td>
</tr>
<tr>
<td>client.( n ).Addr</td>
<td>String</td>
<td>IP address of the client computer.</td>
</tr>
<tr>
<td>client.( n ).Bandwidth</td>
<td>Integer</td>
<td>Connection bandwidth.</td>
</tr>
<tr>
<td>client.( n ).ClientID</td>
<td>String</td>
<td>Client information, including platform, Player version, and whether client is RealPlayer Plus.</td>
</tr>
<tr>
<td>client.( n ).ConnID</td>
<td>Integer</td>
<td>Connection ID, which is the same as ( n ) (for example, client.54.54).</td>
</tr>
<tr>
<td>client.( n ).GUID</td>
<td>String</td>
<td>Unique ID of each RealPlayer.</td>
</tr>
<tr>
<td>client.( n ).InterfaceAddr</td>
<td>String</td>
<td>Server IP address client request came in on.</td>
</tr>
<tr>
<td>client.( n ).isUDP</td>
<td>Integer</td>
<td>Whether UDP is used. 1=true, 0=false.</td>
</tr>
<tr>
<td>client.( n ).Language</td>
<td>String</td>
<td>Player’s preferred language.</td>
</tr>
<tr>
<td>client.( n ).PlayerStarttime</td>
<td>String</td>
<td>Player’s local time at start of connection.</td>
</tr>
<tr>
<td>client.( n ).Port</td>
<td>Integer</td>
<td>Incoming port of Helix Universal Server to which the client is connected.</td>
</tr>
<tr>
<td>client.( n ).Protocol</td>
<td>String</td>
<td>Protocol used by client to connect to Helix Universal Server (PNA or RTSP).</td>
</tr>
<tr>
<td>client.( n ).SessionCount</td>
<td>String</td>
<td>Keeps track of the number of sessions the client has open.</td>
</tr>
<tr>
<td>client.( n ).StartTime</td>
<td>String</td>
<td>Server’s system time at start of client connection.</td>
</tr>
<tr>
<td>client.( n ).SupportMulticast</td>
<td>Integer</td>
<td>Whether connection supports multicasting. 1=true, 0=false.</td>
</tr>
<tr>
<td>client.( n ).User-Agent</td>
<td>String</td>
<td>The User-Agent string that the client sent to the server.</td>
</tr>
<tr>
<td>client.( n ).Version</td>
<td>String</td>
<td>Protocol (PNA or RTSP) version.</td>
</tr>
</tbody>
</table>
### Client Session Statistics

<table>
<thead>
<tr>
<th>Property name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>client.n.Session</td>
<td>Composite</td>
<td>Client session information.</td>
</tr>
<tr>
<td>client.n.Session.n.StreamCount</td>
<td>Integer</td>
<td>Number of client streams.</td>
</tr>
<tr>
<td>client.n.Session.n.URL</td>
<td>String</td>
<td>The URL for this session.</td>
</tr>
<tr>
<td>client.n.Session.n.Stream.n</td>
<td>Composite</td>
<td>Session stream information.</td>
</tr>
<tr>
<td>client.n.Session.n.TransportCount</td>
<td>Integer</td>
<td>Number of transport methods used for session.</td>
</tr>
<tr>
<td>client.n.Session.n.Transport.n</td>
<td>Composite</td>
<td>Client session information for a specific transport type.</td>
</tr>
<tr>
<td>client.n.Session.n.Transport.n.ResendFailure</td>
<td>Integer</td>
<td>Number of times resend requests were not satisfied.</td>
</tr>
<tr>
<td>client.n.Session.n.Transport.n.ResendSuccess</td>
<td>Integer</td>
<td>Number of times resend requests were satisfied.</td>
</tr>
<tr>
<td>client.n.Session.n.Transport.n.SendingTime</td>
<td>Integer</td>
<td>Amount of time (in seconds) that this transport session has operated.</td>
</tr>
</tbody>
</table>

* Only supported for PNA at present.

### Predefined License Properties

The following table lists the license properties predefined in the Helix Universal Server registry.

<table>
<thead>
<tr>
<th>Property name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>license</td>
<td>Composite</td>
<td>License file variables.</td>
</tr>
<tr>
<td>license.NumLicenses</td>
<td>Integer</td>
<td>The number of individual license files that were read in by the server at startup time (the number of valid license files in the server’s License/ subdirectory).</td>
</tr>
</tbody>
</table>
Predefined Monitor Properties

The following table lists the monitor properties predefined in the Helix Universal Server registry.

<table>
<thead>
<tr>
<th>Property name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monitors</td>
<td>Composite</td>
<td>Information about all connected monitors.</td>
</tr>
<tr>
<td>license.NumLicenses</td>
<td>Integer</td>
<td>The number of monitors currently connected to the server.</td>
</tr>
</tbody>
</table>

Custom or Predefined Server Configuration Properties

In addition to the predefined and user-defined properties described so far, the Helix Universal Server property registry contains a number of server configuration parameters, which are loaded from a configuration file at server startup time. A plug-in can use the methods listed earlier in this section to retrieve these customized or predefined Helix Universal Server configuration parameters from the registry. (See Helix Universal Server Configuration File Reference.) Because the parameters entered in the configuration file do not specify the registry data types, Helix Universal Server sets the data types appropriately as integers or strings when it reads the file. Parameters that have true or false values (case does not matter) are added as integer values with true equal to 1 and false equal to 0. The monitor plug-in then accesses the properties by name according to conventions such as these:

config.Variable_Name
config.List_Name.Variable_Name
config.List_Name.List_Name.Variable_Name

**Note:** Helix Universal Server supports arbitrarily complex nested lists for configuration parameters. The registry property naming conventions may therefore be more complex than the preceding examples.

You can also access variables within lists and lists within lists by their order within the list one level higher. (They must be accessed this way if they are not named.) Instead of using the variable or list name, the plug-in uses elemx, where x is the list entry number, starting with 0 (zero). For example, suppose that the configuration file specifies the following parameter:

```xml
<List Name="FSMount">
  <List Name="hx-live3">
    <Var MountPoint="/live/"/>
    <Var Port="7090"/>
  </List>
  <List Name="hx-local">
    <Var MountPoint="/"/>
    <Var BasePath="/home/helixserver/rafiles"/>
    <Var Authentication="True"/>
  </List>
</List>
```

The plug-in can access the /live/ value for the first MountPoint variable with any of the following:
Creating a Monitor Plug-in

Helix Universal Server plug-ins can monitor resources in the Helix Universal Server property registry, which is a dynamic repository for a variety of server and client properties. These properties include such values as the number of clients currently connected, the total bandwidth being utilized, and a comprehensive set of statistics for each of these clients. A monitor plug-in can monitor any registry property, receiving notification when Helix Universal Server updates the property. Monitor plug-ins can also add their own properties to the registry and receive notifications when those properties change.

For More Information: For a detailed description of the Helix Universal Server property registry, including descriptions of the various server, client, and session properties, see “Registry Basics” in Volume 1, on page 119.

Interface

A monitor plug-in implements the following interfaces:

- **IHXPlugin**. Header file: hxplugn.h. Every plug-in implements this interface, which Helix uses to determine the plug-in’s characteristics.
- **IHXPropWatchResponse**. Header file: hxmon.h. When it updates monitored resources in its registry, Helix Universal Server uses this interface to notify the plug-in.
- **IHXCallback**. Header file: hxengin.h. If the plug-in uses Helix Scheduler, it implements this interface to receive callbacks at designated intervals.

The monitor plug-in uses the following interfaces:

- **IHXRegistry**. Header file: hxmon.h. A monitor plug-in uses this interface to access and watch information in the Helix Universal Server registry.
• IHXPropWatch. Header file: hxmon.h. This interface allows the monitor plug-in to set and clear watches on properties within the Helix Universal Server registry. Helix Universal Server sends notifications to the plug-in through the IHXPropWatchResponse interface.

• IHXScheduler. Header file: hxengin.h. The plug-in uses this interface to schedule callbacks from the Helix Scheduler.

  **Note:** Helix Universal Server interfaces do not provide methods for logging or writing information to disk or remote destinations. Monitor plug-ins are responsible for managing any needed data handling.

**Coding a Monitor Plug-in**

The following sections explain how Helix Universal Server and a monitor plug-in use the Helix interfaces. The sample code included with this software development kit (SDK) illustrates many of these features. You can use the sample code as a starting point for building your own plug-in. Refer to the Helix Client and Server Software Development Kit header files for more information on function variables and return values.

  **Note:** The order of function calls listed in the following sections provides a generalized explanation and is for illustrative purposes only. Because Helix is asynchronous, your plug-in must be able to handle any call made to it while it is processing data or waiting for a response from another object. Do not code your plug-in so that it expects a specific sequence of events to occur as it interacts with Helix Universal Server.

**Starting Up**

When Helix Universal Server is started, it loads each monitor plug-in. Helix Universal Server performs a series of calls to your plug-in’s functions and methods in a specific order.

  ➤ **To provide start-up support in your monitor plug-in:**

1. Create a new instance of the monitor plug-in by implementing HXCreateInstance.

   **For More Information:** See “Creating a Plug-in Instance” in Volume 1, on page 36.

2. Return descriptive information about the plug-in by implementing IHXPlugin::GetPluginInfo. A plug-in used only for monitoring can return NULL for all descriptive values except pDescription, which should be Monitor. Set bLoadMultiple to FALSE to open only a single instance of the monitor plug-in in the Helix Universal Server process.

   **For More Information:** See “Defining Plug-in Attributes” in Volume 1, on page 37.

3. Initialize the plug-in by implementing the IHXPlugin::InitPlugin method. Helix Universal Server uses this method to pass a pointer to the system context to your monitor plug-in. You must design this method to perform any necessary initialization, minimally doing the following for a monitor plug-in:

   a. With the Helix Universal Server context pointer, store a reference to IHXCommonClassFactory so the plug-in can create Helix objects.

   b. Use IHXCommonClassFactory::CreateInstance to get an instance of the IHXRegistry interface.
c. Use IHXRegistry::CreatePropWatch to create an IHXPropWatch interface. With the IHXPropWatch interface, the monitor plug-in can set watch points on properties.

d. Use IHXPropWatch::Init to initialize the property watch interface with the response interface, IHXPropWatchResponse, which is the monitor plug-in itself. Through the response interface’s methods, Helix Universal Server notifies the plug-in of additions, modifications, and deletions to the watched properties.

4. With the registry initialized, your plug-in can set watches through IHXPropWatch (see “Watching Properties” in Volume 1, on page 126) or IHXScheduler (see “Using the Scheduler” in Volume 1, on page 127). The plug-in methods are called when watches are triggered or, if a callback has been put into the scheduler, whenever the time slice expires.

Watching Properties

The Helix Universal Server implementation of monitoring enables monitor plug-ins to watch desired resources much as a debugger does. You can set watches on composites or any other type of property, but there is a subtle difference between the two. For composite properties, Helix Universal Server sends notification only when one of the following happens:

• The watched composite has been deleted.
• A property has been added immediately below the watched composite.
• A property has been deleted immediately below the watched composite.

For all other properties, Helix Universal Server sends notification on one of the following:

• The watched property has been modified.
• The watched property has been deleted.

Use IHXPropWatch::SetWatchOnRoot to watch for additions or deletions of properties at the registry’s root level. To set watches on specific properties, use either IHXPropWatch::SetWatchByName or IHXPropWatch::SetWatchById.

When Helix Universal Server adds, modifies, or deletes the watched property, it notifies the plug-in through the IHXPropWatchResponse::AddedProp, IHXPropWatchResponse::ModifiedProp, and IHXPropWatchResponse::DeletedProp methods. This notification includes the watched property’s ID value, its data type, and the ID of its parent property. The plug-in can then query the registry for the property’s new value.

**Note:** From the time the IHXPropWatch::SetWatchXXX method is called until the time the watch is actually set on the property, if any new properties are added under the composite property about to be watched, notification for these new properties is not sent back to the plug-in.

Keep in mind the following when setting watches:

• You cannot set watches on integer reference properties. To access these properties, use one of the IHXRegistry methods (such as IHXRegistry::GetIntByName or IHXRegistry::GetIntById) or use the IHXScheduler interface along with IHXCallback.

**For More Information:** See “Using the Scheduler” in Volume 1, on page 127.
• Just as IHXPropWatch::SetWatchxxx methods set watches on properties, IHXPropWatch::ClearWatchxxx methods are available to clear the watches when necessary.

Using the Scheduler

From the system context pointer it receives during initialization, a plug-in can use IUnknown::QueryInterface to obtain the IHXScheduler interface. The plug-in then creates an IHXCallback interface and enters it into the scheduler using either IHXScheduler::AbsoluteEnter, an absolute time defined by the HXTimeval structure, or IHXScheduler::RelativeEnter, the number of milliseconds from the current time.

The scheduler then calls IHXCallback::Func after the specified time slice has elapsed. See the monitor plug-in sample code for an example of one way to do this.

Modifying the Monitor Plug-in Sample Code

The Helix Client and Server Software Development Kit includes two monitor plug-in samples—an intermediate one and an advanced one—which are discussed in this section. The SDK includes source code for an intermediate-level sample monitor plug-in you can use as a template:

• source/samples/server/monitor/monitor.cpp

This sample plug-in monitors active Helix Universal Server connections and displays the information on the console of the Helix Universal Server computer.

➤ To modify the sample code:

1. Make copies of the sample files.

2. Specify the copyright and URL information for your plug-in in the CExampleMonitor::GetPluginInfo method.

3. Replace the following methods with the user interface for your monitor:
   CExampleMonitor::PrepareCounter
   CExampleMonitor::UpdateCounter

4. Modify the CExampleMonitor::SetWatches method to specify the properties you want to monitor.

5. Modify the following methods to process the notifications as the watched properties are added, modified, and deleted:
   CExampleMonitor::AddedProp
   CExampleMonitor::ModifiedProp
   CExampleMonitor::DeletedProp

   Tip: Limit the amount of processing within methods to prevent slowing down Helix Universal Server.

An advanced monitor plug-in sample is also included in the Helix Client and Server Software Development Kit:

• source/samples/server/livepacketsink/livepacketsink.cpp

This is an example of a live packet sink plug-in that uses monitoring. It sets a watch on the registry's LiveConnections composite key so that it will receive notifications about new live connections. It then
examines the headers and packets sent to the server from the live source. The plug-in can use this information for many purposes, including logging statistics about the live content or archiving the live streams to disk. See the source file for instructions on how the plug-in works.

Examples

Example 1: Adding a Simple Property to the Registry

The following example shows how to add an IntRef property. It demonstrates how the property creator can change its value using a simple “C”-style dereference:

```cpp
// creator
int* int_val = new int;
*int_val = 99;
...
UINT32 iref_id = m_registry->AddIntRef("mystats.foobar", int_val);
...
/*
modifying the variable pointed to by int_val automatically reflects
in the registry because it is the same address.
*/
(*int_val)++;
...
// accessor -- some other user
INT32 i_val = 0;
m_registry->GetIntByName("mystats.foobar", i_val);
...```

Example 2: Adding Statistics to the Registry

This next example adds registry statistics that show how many clients are connected per hour and how many packets are lost per hour:

```cpp
char *str[] = {"mystats",
   "mystats.num_player_per_hour",
   "mystats.packets_lost_per_hour"
};

// add the COMPOSITE under which our INTEGER Props will go
mystats_id = m_pRegistry->AddComp(str[0]);
npph_id = m_pRegistry->AddInt(str[1], num_plyrs_per_hr);
plph_id = m_pRegistry->AddInt(str[2], pckts_lost_per_hr);
...
if (m_pRegistry->SetIntById(npph_id, num_plyrs_per_hr) != HXR_OK)
{
   // error reporting here...
}
```

This code adds the composite property mystats to the registry. This property contains the two string properties mystats.num_player_per_hour and mystats.packets_lost_per_hour. To retrieve the values of the properties you can do either of the following:
```c
int int_val = 0;
if (m_pRegistry->GetIntByName(str[1], int_val) != HXR_OK)
{
    // do some error checking here
}

or:

int int_val = 0;
if (m_pRegistry->GetIntById(npph_id, int_val) != HXR_OK)
{
    // do some error checking here
}

For a string, retrieve the value as follows:
IHXBufffer* str_val = 0;
if (m_pRegistry->GetStrByName(prop_name, str_val) != HXR_OK)
{
    // error checking here
}
// after you are finished with the buffer RELEASE it
str_val->Release();

Note: When retrieving string values, make sure that you pass in a pointer to an IHXBufffer
that has not been created. Helix Universal Server returns a created buffer with the string
value in it. Be sure to use IUnknown::Release to release the buffer after using it.

To get the unique ID value of a property, pass the property name to the IHXRegistry::GetId method. If
the property does not exist, the method returns 0 (zero) as the ID.
CHAPTER 10

RESTRICTING ACCESS

Helix Universal Server provides various means of restricting access to content. The server can grant permission for a specific user to gain access to specific server content. This would include access to pay-per-view content, specific file systems, and so on. Authentication plug-ins can be used to verify whether a specific user is granted permission to access specific content on the server.

The other type of restriction is provided by allowance plug-ins, which the server uses to monitor the restrictions on continuing communication between the server and specific clients.

Server HTTP and RTSP Authentication

Note: The authentication information in this section is obsolete and is included for information only. Future versions of the Helix Client and Server Software Development Kit documentation will contain up-to-date material as the new authentication interfaces become available.

Helix Universal Server can store an encrypted password file for verifying user names and passwords supplied by users. It can thereby validate file requests or any other operation requiring authentication parameters. When communicating through RTSP, Helix can transmit passwords securely over a network. This section discusses supported authorization types and explains how to add HTTP and RTSP authentication functionality to a Helix plug-in.

For More Information: You also need to set Helix Universal Server’s authentication parameters as described in the authentication section in Helix Universal Server Configuration File Reference. For information on how the Helix client gathers a user name and password from the user, see “Handling Password Authentication” in Volume 1, on page 158.

Interfaces

Helix HTTP and RTSP authentication involves the following interfaces:

- IHXAuthenticator. Header file: hxauth.h.
  
  The authenticator object, which performs the password validation, implements this interface. The response interface is IHXAuthenticatorResponse.

- IHXAuthenticatorRequest. Header file: hxauth.h.
  
  The object that has access to the parameters to be authenticated implements this interface.

- IHXAuthenticatorResponse. Header file: hxauth.h.
  
  The object that receives notice whether the authentication succeeded or failed implements this interface.
Helix Client and Server Software Developer’s Guide, Volume 1

- **IHXAuthenticationManager**. Header file: `hxauth.h`
  The Helix top-level client implements this interface to receive notice from the client core that it should gather a user name and password.

- **IHXAuthenticationManagerResponse**. Header file: `hxauth.h`
  The Helix client core implements this interface to receive usernames and passwords from the top-level client.

- **IHXFileAuthenticator**. Header file: `hxfiles.h`
  File objects implement this interface to become associated with authenticator objects.

- **IHXPassword**. Header file: `hxauth.h`
  Helix implements this interface to provide a means for the client to transmit encrypted or unencrypted passwords.

### Implementing Authentication

Any plug-in that needs to perform password authorization can compare user name and password values sent in by a user against those stored in Helix Universal Server’s encrypted password file. As the following examples illustrate, Helix’s structure of authenticator object, authentication request object, and authentication response object provides a flexible means of implementing authentication under different circumstances.

**Example 1: A File System Plug-in**

The most common use of authentication is a file system plug-in that performs password validation for a file requested by a Helix client. This process works as follows:

1. Helix Universal Server accepts a client connection through HTTP or Real-Time Streaming Protocol (RTSP) and creates an authenticator object with `IHXCommonClassFactory`. It identifies itself as the authentication request object by calling `IHXAuthenticator::InitAuthenticator`.

2. The client requests a password-protected file.

3. To serve the request, Helix Universal Server instantiates a file system plug-in that in turn creates a file object with access to the file data.

   ![For More Information](See “Chapter 4: File Handling” beginning in Volume 1, on page 49.)

4. Helix Universal Server calls `IHXFileAuthenticator::SetAuthenticator` on the file object to associate it with the `IHXAuthenticator` interface.

5. During initialization (or whenever appropriate), the file object calls `IHXAuthenticator::Authenticate`, identifying itself as the response object that implements `IHXAuthenticatorResponse`. Optionally, the `IHXAuthenticator::Authenticate` call can include a pointer to an `IHXValues` interface that contains a value for `AuthType`. A value of 1 instructs Helix Universal Server to use basic authorization for an HTTP connection. A value of 2 tells it to use DIGEST authorization (which is always used on an RTSP connection). If this information is not included, Helix Universal Server determines the authorization type, as described in steps 6 and 7.
6. The authenticator object calls IHXAuthenticatorRequest::GetAuthValues to get from the authentication request object (Helix Universal Server) the user name and password to be authenticated. This method passes along the IHXValues interface that sets the authorization type.

7. The authentication request object (Helix Universal Server) retrieves the user name and password sent from the client by the appropriate means, such as an RTSP authorization header. If the authenticator object did not specify the authorization type, Helix Universal Server looks at its HTTPAuthType variable to determine whether to use basic or DIGEST authorization for an HTTP connection. It always uses DIGEST authorization for an RTSP connection.

   **For More Information:** See the authentication section in *Helix Universal Server Configuration File Reference.*

The authentication request object calls IHXAuthenticator::AuthValuesReady to pass the authenticator object a pointer to an IHXValues interface that contains the Username and Password values to be authenticated. The values interface also contains the AuthType value.

   **For More Information:** For information on how the client gathers passwords, see “Handling Password Authentication” in Volume 1, on page 158.

8. The authenticator object verifies the retrieved user name/password against the password file, calling IHXAuthenticatorResponse::AuthenticateDone to pass the file object a status code of HXR_OK or HXR_NOT_AUTHORIZED. The call includes a pointer to an IHXValues interface that contains Username and Password.

   **For More Information:** See “Status Codes” in Volume 1, on page 85.

9. The file object then takes the appropriate action. If authorization succeeds, it returns HXR_OK through IHXFileResponse::InitDone. If authorization fails, it passes along HXR_NOT_AUTHORIZED. Because it has access to the user name and password in the IHXValues interface it receives, the file object can perform its own authorization checking to narrow the scope of valid users.

   Suppose that users Blakes7 and RedDwarf are validated by Helix Universal Server’s password file, but only Blakes7 should have access to files served by this file system plug-in. The plug-in can implement its own user list that its file object accesses after a successful system validation. This prevents RedDwarf from accessing the file even though Helix Universal Server’s authenticator object returned HXR_OK.

**Example 2: A Monitor Plug-in**

Helix can perform password authorization on actions other than file requests. A monitor plug-in, for example, may validate passwords when a client connects to Helix Universal Server. In the following example, the monitor plug-in implements both IHXAuthenticatorRequest and IHXAuthenticatorResponse, supplying the values to be authenticated and receiving the authentication results. During this process, the following actions occur:

1. The monitor plug-in is instantiated on Helix Universal Server and uses Network Services to listen for connections on a specific port or ports.

2. A client connects to a monitored port and requests Helix Universal Server information.
3. The plug-in requests authorization identification.
4. The client sends a user name and password to the monitor plug-in.
5. The plug-in puts the values in an IHXValues interface under Username and Password.
6. The plug-in uses IHXCommonClassFactory to instantiate an authenticator object.
7. The plug-in calls IHXAuthenticator::InitAuthenticator to identify itself as the authentication request object.
8. The plug-in calls IHXAuthenticator::Authenticate, identifying itself as the authentication response object.
9. The authenticator object calls IHXAuthenticatorRequest::GetAuthValues to get from the plug-in the user name and password to be authenticated.
10. The plug-in calls IHXAuthenticator::AuthValuesReady to pass the authenticator object a pointer to the IHXValues interface that contains the Username and Password values.
11. The authenticator object verifies the user name and password against the password file, calling IHXAuthenticatorResponse::AuthenticateDone to pass the plug-in a status code of HXR_OK or HXR_NOT_AUTHORIZED.

For More Information: See “Status Codes” in Volume 1, on page 85.

Authorization Types

Helix supports two authorization types:

- **PN_AUTH_BASIC**
  HTTP 1.0, Netscape Navigator 3.0, and Internet Explorer 3.0 support basic authorization, which uses unencrypted user names and passwords. The client uses IHXPassword::Crypt to encode two strings for the user name and password in base-64 format for transmission to Helix Universal Server or an HTTP server.

- **PN_AUTH_DIGEST**
  RTSP and HTTP 1.1 support the DIGEST type of secure password authorization. (Note that not all browsers support this.) The authorization type contains strings for Nonce, URL, Realm, and Opaque, as described in RFC 2069. The client uses IHXPassword::Crypt to encrypt those four values along with the user name and password. Helix Universal Server stores the encrypted string in IHXValues for comparison with its password file.

RTSP communication between Helix Universal Server and the client uses DIGEST authorization. The Helix Universal Server configuration parameter HTTPAuthType determines whether HTTP communication between Helix Universal Server and a client uses basic or DIGEST authentication.
Creating an Allowance Plug-in

An allowance plug-in is a server component that accepts only certain types of requests or actions from a Helix client. Applications that can use allowance plug-ins include:

- **Pay-per-view**
  In this application, the plug-in checks a URL requested by the client. If the URL is secure, the plug-in forces the client to authenticate the URL.

- **IP address restriction**
  In this application, the plug-in checks the IP address of the connecting client and disconnects the client if the IP address is not within a particular range (domain restriction), is not found on a list of authorized IP addresses (membership restriction), or fails to meet any other IP address criteria.

- **Content rotation**
  In this application, the plug-in redirects incoming clients to different clips based on the time of day, player version, connection bandwidth, or any other available criteria.

Design Considerations

In addition to the general plug-in design considerations described in “Designing a Plug-in” in Volume 1, on page 35, keep the following points in mind as you develop your allowance plug-in:

- **You should optimize the allowance plug-in’s use of processor time and memory.** Helix Universal Server creates one instance of the allowance plug-in for each client stream—this means there will be at least one instance for each connected client. Plug-ins that require significant system resources can quickly degrade Helix Universal Server performance.

- **Allowance plug-ins can only be used to control clients using the Real-Time Streaming Protocol (RTSP) and Progressive Networks Audio (PNA) protocols.** If the application must also stream content using HTTP, an allowance plug-in will not work.

- **The allowance plug-in can be designed to receive notification of client and session property changes dynamically through the use of monitoring interfaces.** For details, see “Creating a Monitor Plug-in” in Volume 1, on page 124.

- **An allowance plug-in is instantiated every time a client requests a stream,** as mentioned previously. Keeping this in mind as you read the following sections will help you understand how allowance plug-ins work and how to design them. For the sake of simplicity, we describe the initialization and functioning of the plug-in as if it were a single instance; that is, it is loaded when Helix Universal Server starts, and its `IHXPlayerConnectionAdviseSink::OnConnection` method is called when the client connects. This explanation, while true, is somewhat simplified.

Because the allowance plug-in gets created for each stream requested by a client, its startup routines (`HXCreateInstance`, `IXPlugin::GetPluginInfo`, `IXPlugin::InitPlugin`) always get called before the server calls the plug-in’s `IHXPlayerConnectionAdviseSink::OnConnection` method. This detail is omitted from the
discussion that follows, but understanding it will prove beneficial when you are working with allowance plug-ins.

For a complete description of all the client and session properties that can be used as criteria for controlling connected clients, see “Chapter 9: Managing the Server Registry”.

**Interfaces**

Allowance plug-ins typically use the following interfaces:

- **IHXPlugin.** Header file: hxplugn.h.
  
  Every plug-in implements this interface, which Helix uses to determine the plug-in’s characteristics.

- **IHXPlayerConnectionAdviseSink.** Header file: hxallow.h.
  
  The allowance plug-in implements this interface, and its methods are called by the server as various client actions occur.

- **IHXPlayerConnectionResponse.** Header file: hxallow.h.
  
  The Helix Universal Server core implements this interface and passes it to the allowance plug-in as a parameter to IHXPlayerConnectionAdviseSink::OnConnection. The plug-in uses this interface to tell the server when the plug-in is finished with a specific IHXPlayerConnectionAdviseSink notification.

- **IHXPlayerController.** Header file: hxallow.h.
  
  Helix Universal Server implements this interface and passes it to the allowance plug-in as a parameter to IHXPlayerConnectionAdviseSink::SetPlayerController. Using this interface, the plug-in can control the client by telling Helix Universal Server to alert, redirect, or disconnect that client.

**Coding the Plug-in**

The following sections explain how Helix Universal Server and an allowance plug-in use the Helix interfaces to control client access. The sample files exallow.cpp and expvpvln.cpp, included with this software development kit (SDK), illustrate many of these features. You can use these samples as a starting point for building your own plug-in. Refer to the Helix Client and Server Software Development Kit header files for more information on function variables and return values.

**Note:** The order of function calls listed in the following sections provides a generalized explanation and is for illustrative purposes only. Because Helix is asynchronous, the plug-in must be able to handle any call made to it while it is processing data or waiting for a response from another object. Do not code the plug-in so that it expects a specific sequence of events to occur as it interacts with Helix Universal Server.

**Starting Up**

When Helix Universal Server is started, it loads the allowance plug-in. During this process, the following actions occur:

1. Helix Universal Server calls HXCreateInstance to create an instance of an IHXPlugin interface. See “Creating a Plug-in Instance” in Volume 1, on page 36 for more on this method.
2. Helix Universal Server calls `IHXPlugin::GetPluginInfo`, which returns descriptive information about the plug-in, including its copyright and “more information” URL. The `bLoadMultiple` property of `IHXPlugin::GetPluginInfo` must be set to `TRUE`.

3. Helix Universal Server calls `IHXPlugin::InitPlugin`, which initializes the plug-in for use. Specifically, this method does two things:
   - It stores a reference to the `IHXCommonClassFactory` interface. This interface creates commonly used interfaces such as `IHXPacket`, `IHXValues`, and `IHXBuffer`.
   - It stores a reference to the `IHXRegistry` interface, which retrieves various properties of the connected player and the session (see `IHXPlayerConnectionAdviseSink::SetRegistryID` in step 6).

4. When a player connects to the Helix Universal Server computer, the server calls the plug-in’s `IHXPlayerConnectionAdviseSink::OnConnection` method, and passes a pointer to the `IHXPlayerConnectionResponse` interface. The plug-in uses this response interface to respond to the various `IHXPlayerConnectionAdviseSink` notifications sent by the server.
   
   It is important to remember that until the appropriate `IHXPlayerConnectionResponse` method is called, nothing will happen with the connected player. For instance, the player will not be able to begin playback of a URL requested with `IHXPlayerConnectionAdviseSink::OnURL` until the allowance plug-in calls `IHXPlayerConnectionResponse::OnURLDone`.

5. Helix Universal Server will also call the plug-in’s `IHXPlayerConnectionAdviseSink::SetPlayerController` method to pass a reference to an `IHXPlayerController` interface, specific to this player’s connection. The plug-in uses this interface to control the player.

6. Helix Universal Server calls the plug-in’s `IHXPlayerConnectionAdviseSink::SetRegistryID` method to pass to the plug-in the ID for this player, stored in the Helix Universal Server Property Registry. This can be used to retrieve a number of the player's properties, such as its IP address, its globally unique identifier (GUID), and the protocol it is using. See “Chapter 9: Managing the Server Registry” beginning in Volume 1, on page 119 and “Creating a Monitor Plug-in” in Volume 1, on page 124 for more information.

7. When the connected player requests a URL, the server calls the plug-in’s `IHXPlayerConnectionAdviseSink::OnURL` method, passing an `IHXRequest` interface. The requested URL is stored in this `IHXRequest` interface, along with the request headers.
   
   You can have your allowance plug-in deal with this request in whatever way you decide. Very often, you will use one of the `IHXPlayerController` interface’s methods described in step 5 to disconnect or redirect the player—or you can take no action, in which case the server sends the player the requested content. Remember, however, that the server does not release any content until the plug-in calls `IHXPlayerConnectionResponse::OnURLDone`.

8. During the life of the player’s connection, various actions taken at the player will cause the Helix Universal Server to notify the plug-in by calling the appropriate `IHXPlayerConnectionAdviseSink` methods. These are:
   - `IHXPlayerConnectionAdviseSink::OnURL`
     
     As previously described, the player has requested a URL. After processing this notification, the plug-in should respond with `IHXPlayerConnectionResponse::OnURLDone`.
   - `IHXPlayerConnectionAdviseSink::OnBegin`
The player has begun or resumed playback. The response is 
IHXPlayerConnectionResponse::OnBeginDone.

- IHXPlayerConnectionAdviseSink::OnPause
  The player has paused the playback. The response is IHXPlayerConnectionResponse::OnPauseDone.

- IHXPlayerConnectionAdviseSink::OnStop
  The player has stopped the playback. The response is IHXPlayerConnectionResponse::OnStopDone.

- IHXPlayerConnectionAdviseSink::OnDone
  The player has disconnected from the server. There is no response to this notification.

Remember that the appropriate IHXPlayerConnectionResponse method should be called as soon as the plug-in finishes processing the notification.

Modifying the Allowance Plug-in Sample Code

The Helix Client and Server Software Development Kit includes a couple of sample allowance plug-ins—an introductory one and an intermediate one:

- /source/samples/server/allowance/allowance.cpp
  This sample plug-in monitors client connections to the server and makes decisions about what content each client can receive and play. It receives a notification for each significant event in a client’s connection lifetime, and can restrict the client’s rights by, for example, disconnecting, redirecting, authenticating, or granting limited permissions to the client for content playback.

- /source/samples/server/payperview/payperview.cpp
  This sample plug-in parallels the content of the introductory allowance plug-in, but is specifically written to handle pay-per-view authentication. It monitors all incoming client connections, and password protects all URLs that contain the word “protected” in them. When a client attempts to access this URL, this plug-in triggers the authentication process and, if the client is successfully authenticated, the client is enabled to play the content.

Follow the instructions in these files to test the samples, or use them as the basis for your own plug-in.

➤ To modify either sample file:

1. Copy the source code from the samples directory to a working directory.

2. Change the file names (exallow.cpp and exallow.h for the introductory allowance plug-in and expvpvln.cpp and expvpvln.h for the pay-per-view allowance plug-in) to something appropriate for your environment or application. Also change the class name (CExampleAllowance or CExamplePayPerView) to something suitable.

3. Change the plug-in description, copyright, and “more information” URL stored in zm_pDescription, zm_pCopyright, and zm_pMoreInfoURL.

4. Modify the implementation code for the various IHXPlayerConnectionAdviseSink methods as needed:
   - You might want to modify IHXPlayerConnectionAdviseSink::OnConnection to retrieve client information from the Helix Universal Server Property Registry. You might also want to use various monitoring interfaces to set watches on some of these properties. For more information, see “Creating a Monitor Plug-in” in Volume 1, on page 124 and “Chapter 9: Managing the Server Registry”.
• Modify the implementation code for the other IHXPlayerConnectionAdviseSink methods. 
  IHXPlayerConnectionAdviseSink::OnURL, IHXPlayerConnectionAdviseSink::OnBegin, 
  IHXPlayerConnectionAdviseSink::OnPause, IHXPlayerConnectionAdviseSink::OnStop, and 
  IHXPlayerConnectionAdviseSink::OnDone will probably all need to be modified. Remember that all 
  but the last of these must call its corresponding IHXPlayerConnectionResponse method with 
  HXR_OK. Also remember that the quicker your plug-in can process the notification and call the 
  response interface, the better.

5. Compile, debug, and test your plug-in.

Adaptive Stream Management (ASM) rules describe a streamed data type to Helix Universal Server, helping it make intelligent decisions about how to deliver that data type's packets efficiently and robustly. File format and broadcast plug-ins define the ASM rules, which, in their simplest form, assign predefined property sets such as “priority” and “average bandwidth” to groups of packets. If Helix Universal Server needs to resend packets, for example, the packet priorities tell it which packets are most important to redeliver.

In their advanced form, ASM rules enable plug-ins to modify the delivery of packets based on changing network conditions. Each ASM rule can have an expression that defines a condition. For example, one rule’s expression might define the condition in which available client bandwidth is from 5,000 to 15,000 bps and packet loss is less than 2.5 percent. If that condition describes the client’s current network connection, the client subscribes to the rule. The properties defined within the rule then help Helix Universal Server stream the packets effectively. If network conditions changes, the client can subscribe to a different rule.

To implement ASM, the file format plug-in defines one or more rules (a “rule book”), each of which contains a set of properties and, optionally, an expression. When a client requests a file, the file format plug-in sends the client a stream header that contains the rule book. The client evaluates the rules’ expressions against its own properties, such as its available bandwidth, and subscribes to one or more rules. It does not subscribe to a rule, for example, if the expression specifies a necessary bandwidth that is higher than the current network connection.

The file format plug-in begins streaming the file after the client transmits its subscription choices back to Helix Universal Server. The plug-in associates each packet with a rule, sending to Helix Universal Server only the packets for the subscribed rules. If there are three possible rules, for example, and the client subscribes only to the first two, the file format plug-in should not produce packets associated with the third rule.

Interfaces

The following interfaces are used with ASM:

• IHXASMSrc. Header file: hxasm.h.
  
  The file format plug-in implements this interface to receive rule subscription information from Helix Universal Server.

• IHXASMStream. Header file: hxasm.h.
  
  The Helix client core stream object implements this interface, which the client uses to inform Helix Universal Server of stream subscription choices. A rendering plug-in can also use this interface to create a stream sink object.
• IHXASMStreamSink. Header file: hxasm.h.

The rendering plug-in implements this interface to receive notification of stream rule subscription and unsubscription.

For More Information: See “Custom Properties” in Volume 1, on page 147.

Note: IHXBackChannel, which is also defined in hxasm.h, is not used with ASM. It simply provides a generic method for rendering and file format plug-ins to communicate. For more information, see “Sending Back-Channel Packets” in Volume 1, on page 166.

Example of ASM Interfaces in Use

The following sequence of steps explains how the ASM interfaces interact with one another. The scenario involves two data types, A and B, requested by a Helix client. The file format plug-in for each data type contains multiple ASM rules to which the client can subscribe.

1. The client requests from Helix Universal Server a presentation containing data type A and data type B.
2. The file format plug-ins for data type A and data type B send the stream headers to Helix Universal Server.
3. Helix Universal Server transmits the rules, which are defined by the ASMRuleBook variable in the stream headers, to the client.
4. By comparing the rules’ expressions to its own network connection, the client picks rules to subscribe to for each stream.
5. The client queries for IHXASMStreamSink on the rendering plug-ins. If the plug-ins implement this interface, they can receive notice of rule subscriptions and perform necessary actions when the client subscribes to a rule that contains a custom property not defined in Helix.

For More Information: See “Custom Properties” in Volume 1, on page 147.

6. If the ASM rules for data type B contain properties not defined in Helix, the rendering plug-in for data type B calls IHXASMStream::AddStreamSink on the client to set up a stream sink and receive notifications of rule subscriptions.
7. The client calls IHXASMStreamSink::OnSubscribe(<rule number>) for the data type B rendering plug-in to notify it of the client’s rule subscription.
8. The client calls IHXASMStream::Subscribe(<rule number>) for each stream to transmit the rule subscription information to Helix Universal Server.
9. Helix Universal Server calls IHXASMSource::Subscribe(<rule number>) to transmit the client’s rule subscription choice to each file format plug-in.
10. The file format plug-in begins to send Helix Universal Server packets associated with the subscribed rules.
Note: When a broadcast plug-in implements ASM, Helix Universal Server always subscribes to all rules. For each client, Helix Universal Server internally filters out packets associated with unsubscribed rules.

During the presentation, the client or the rendering plug-in (if it monitors rule subscriptions) can change the rule subscription because of changing conditions in the client connection, using the following sequence of actions:

1. The client determines that, due to changing conditions in the connection, it must drop a rule subscription for data type B.
2. The client calls IHXASMSink::OnUnsubscribe(<rule number>) to notify the rendering plug-in for data type B.
3. The client calls IHXASM::Unsubscribe(<rule number>) to relay the unsubscribe action to Helix Universal Server.
4. Helix Universal Server calls IHXASMS::Unsubscribe(<rule number>) to transmit the rule change to the file format plug-in for data type B.
5. When the client determines that it can reinstate the rule, it calls IHXASMS::Subscribe(<rule number>) and starts the subscription process previously described. Helix Universal Server then waits for a packet with the ASM_SWITCH_ON flag and starts resending the packets for the reinstated rule.

ASM Rules and Properties

A file format or broadcast plug-in defines a rule book, which is a set of rules for streaming packets of that format’s or plug-in’s data type. Each rule can optionally have an expression that the client evaluates when determining which rules to subscribe to. Each rule sets a relative priority for its packets and contains information that helps Helix Universal Server determine how best to stream the packets, as well as how to handle certain situations that may arise. Here are examples of two simple rules:

\[
\text{AverageBandwidth}=12000, \\
\text{Priority}=7; \\
#16000 <= $\text{Bandwidth}, \\
\text{AverageBandwidth}=4000, \\
\text{Priority}=6;
\]

The client always subscribes to Rule 1 and always receives the Rule 1 packets. It subscribes to Rule 2 and receives the Rule 2 packets only if its total available bandwidth is at least 16,000 bps. Note that this does not mean that Rule 2 packets stream at 16,000 bps. If Rule 1 packets currently consume 12,000 bps and total available bandwidth is 16,000 bps, for example, Rule 2 packets stream at 4,000 bps. The different priority values indicate that if packets need to be resent or dropped, Rule 1 packets take priority over Rule 2 packets.

Helix predefines properties such as “Priority.” Note, however, that the architecture also supports custom properties known only to the rendering and the file format plug-ins. The predefined rule
properties are listed and described in the following table and are then explained in the sections that follow.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
<th>Property type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bandwidth</td>
<td>INT32</td>
<td>Expression variable</td>
</tr>
<tr>
<td>PacketLoss</td>
<td>Float</td>
<td>Expression variable</td>
</tr>
<tr>
<td>Priority</td>
<td>1-10</td>
<td>Priority factor</td>
</tr>
<tr>
<td>AverageBandwidth</td>
<td>INT32</td>
<td>Informational</td>
</tr>
<tr>
<td>AverageBandwidthStd</td>
<td>Float</td>
<td>Informational</td>
</tr>
<tr>
<td>TimeStampDelivery</td>
<td>Boolean</td>
<td>Directive</td>
</tr>
<tr>
<td>WaitForSwitchOff</td>
<td>Boolean</td>
<td>Directive</td>
</tr>
<tr>
<td>Marker</td>
<td>Boolean</td>
<td>RTP marker bit</td>
</tr>
</tbody>
</table>

For More Information: See “Custom Properties” in Volume 1, on page 147. For rule syntax descriptions and examples, see “Creating a Rule Book” in Volume 1, on page 147.

Expression Variables

The Helix client can subscribe to ASM rules by evaluating each rule’s expression. If the rule has no expression, it is active at all times and the client always receives packets associated with that rule. The expression can contain the following variables, which are known to Helix. You cannot use other predefined ASM properties or custom properties defined in your file format and rendering plug-ins in an expression.

For More Information: See “Custom Properties” in Volume 1, on page 147.

Bandwidth

This variable triggers subscription to the rule based on the client’s available bandwidth. The rule book needs to cover the continuous range of possible bandwidths, from the minimum acceptable bandwidth to “infinite” bandwidth. For example, the bandwidth properties in the expressions for the following two rules leave a gap for client bandwidth between 16000 and 20000 bps:

#(12000 < $Bandwidth) && ($Bandwidth < 16000),
.
.
.
#(20000 < $Bandwidth) && ($Bandwidth < 24000),
.
.
.

But this set of rules is acceptable:

#12000 < $Bandwidth,
.
.
.
#16000 < $Bandwidth,
.
.
.

This set of rules is also acceptable:

#(12000 < $Bandwidth ) && ($Bandwidth < 16000),
.
.
.
#16000 < $Bandwidth,
.
.
.
Note that these two sets of acceptable rules cause different client actions. In the first set, the client subscribes to Rule 1 if bandwidth is greater than 12,000 bps. It subscribes to Rule 1 and Rule 2 if bandwidth is greater than 16,000 bps. With the second set of rules, the client subscribes to either Rule 1 or Rule 2, but not both, depending on its available bandwidth.

**PacketLoss**

This property is a percentage that specifies a range of packet loss. A rule book that uses this variable typically states it in an expression such as:

\[(12000 < \text{Bandwidth} < 16000) \&\& (20.0 < \text{PacketLoss})\]

**Priority**

Helix Universal Server uses the **Priority** property to determine how best to stream a presentation to a client. The property takes a value from 1 to 10 that sets the relative priority of the packets associated with the rule. The highest priority, 10, is reserved for packets that are essential to a presentation. Helix does everything necessary, including stalling a client presentation, to deliver these packets. Use this priority only if losing a packet in the stream invalidates a presentation. Packets that are critical but not essential, such as key frames, should use priority 9.

The remaining priorities, 8 to 1, are relative rankings that help Helix Universal Server decide what packets to resend first and which to drop if needed. Audio data would typically fall in the range of 6 to 8. Video p-frames would typically be ranked at priority 5 or 6. Lower rankings indicate data that can be dropped without damaging the presentation severely.

**Informational Properties**

Informational properties help Helix Universal Server determine how best to stream packets associated with the rule. The rule book expresses them with this syntax:

\[<\text{property}> = <\text{value}>\]

**AverageBandwidth**

*AverageBandwidth* is an estimate of the average bits per second of the packets streamed for this rule.

**Note:** All rules must provide a value for this property, unless *TimeStampDelivery* is set to \text{TRUE} in the rule.

**AverageBandwidthStd**

This optional parameter indicates an estimated standard deviation of the average bandwidth. Data with a stable bit rate, such as audio, should have a standard deviation of 0. Data with variable bit rates, such as video, should have a value appropriate to the data type.

**Directives**

Directives place conditions on serving packets for the rule. Most directives are Boolean values that tell Helix Universal Server to perform certain actions while the rule is active. The rule book expresses directives with this syntax:

\[<\text{property}> = \text{TRUE}|\text{FALSE}|<\text{value}>\]
**TimeStampDelivery**

When the Boolean property `TimeStampDelivery` is set to `TRUE`, Helix Universal Server delivers packets in step with the presentation timeline and does not attempt to send packets in advance of their rendering time. This is useful for data types such as RealText that consume small amounts of bandwidth and have highly variable bandwidth usage patterns. It is not to be used for audio, video, or other data types that consume high bandwidth and require initial buffering.

If `TimeStampDelivery` is set to `TRUE`, `AverageBandwidth` cannot be used in the rule. When `TimeStampDelivery` is set to its default value of `FALSE`, `AverageBandwidth` must be set in the rule. Note, however, that one rule could use `AverageBandwidth`, while another rule uses `TimeStampDelivery`.

**WaitForSwitchOff**

This Boolean property has a default value of `TRUE`. In this case, after Helix Universal Server notifies the file format plug-in that a client has unsubscribed from a rule, it waits for a packet with the flag `ASM_SWITCH_OFF` before it stops sending packets for that rule. If `WaitForSwitchOff` is set to `FALSE`, Helix Universal Server stops sending packets as soon as it receives the unsubscribe request from the client. Helix Universal Server discards any further packets for that rule that the file format plug-in may send.

**For More Information:** See “Creating Stream Packets” in Volume 1, on page 62.

**RTP Marker Bit Property**

If the file format supports Real-Time Protocol (RTP) payloads that make use of the marker bit in the RTP header, it must create the ASM rule book to identify which ASM rules map to RTP packets with the marker bit set. By convention, if a rule book is to contain only two entries, rule 0 corresponds to marker bit off and rule 1 to marker bit on. Because the ASM rule book requires the `AverageBandwidth` (or `TimeStampDelivery`) property to be set, the minimum rule book for an RTP payload making use of marker bits looks like this:

```
Marker=0, AverageBandwidth= <half of total bitrate>;
Marker=1, AverageBandwidth= <other half of total bitrate>;
```

where `<other half of total bitrate>` = `<total bitrate>` - (ULONG32) (`<total bitrate>` / 2).

After the mapping has been established, the file format plug-in assigns the ASM rule numbers to packets to match the desired RTP marker bit setting.

The renderer performs the reverse of this mapping. It receives the ASM rule book through its `IHXRenderer::OnHeader` method and parses the ASM rule book to determine which ASM rules correspond to RTP packets with the marker bit set. For data types that do not support ASM switching and therefore contains only two rules in the ASM rule book, by convention, rule 0 corresponds to marker bit off and rule 1 to marker bit on.

**Note:** When RealPlayer receives RTP packets from a server that is not taking advantage of ASM, it forms the ASM rule book always mapping the packets without a marker bit set to ASM rule number 0 and packets with a marker bit set to ASM rule number 1.

**For More Information:** See the table “RTP Payload Types” in Volume 1, on page 64.
Custom Properties

Rendering plug-ins can monitor rules that are not predefined in Helix. Helix ignores any property it does not know about, passing the property value to the rendering plug-in. If a file format plug-in defines custom properties, its corresponding rendering plug-in becomes a stream sink with IHXASMStream::AddStreamSink. The Helix client then uses IHXASMStreamSink to notify the rendering plug-in of rule subscriptions. The plug-in can thereby take the appropriate actions when a rule contains a custom property.

**Note:** You cannot use custom properties as expression variables.

Creating a Rule Book

You can define or generate a rule book within the file format plug-in code or within another source that the plug-in reads. The latter method is useful if the rules for delivering the data type are subject to change. The plug-in passes the rules to Helix Universal Server as a single, long string in the file stream header. The following are the syntax rules for the ASM rule book:

- Rule numbers start at 0 (zero) and depend solely on the order of the rules. The first defined rule is Rule 0, the second is Rule 1, and so on.
- A semicolon terminates a rule.
- Commas separate properties and expressions within a rule.
- A pound sign (#) indicates an expression that the client evaluates to determine whether or not to subscribe to the rule.
- A dollar sign ($) indicates a variable within an expression.
- An expression can contain any number of acceptable operators, which are listed in the table “ASM Rule Book Operators” in Volume 1, on page 148.
- You can use parentheses to specify the evaluation order of operators and their arguments.
- Double quotes must surround strings used as properties values.
- White space outside of quotation marks is ignored.

The following example shows a simple rule book defined within the file format plug-in code:

```c
const char* pRuleBook =
{
    /* Rule 0 */
    /*
        #$Bandwidth < 16000,
        AverageBandwidth=12000,
        AverageBandwidthStd=0,
        Priority=7;
    */
    /* Rule 1 */
    /*
        #16000 < $Bandwidth,
        AverageBandwidth=16000,
        AverageBandwidthStd=0;
    */
};
```
Rule Book Expressions

Rule expressions can employ any of the C-style operators listed and defined in the following table. The operators follow the standard C++ evaluation order, which you can modify with parentheses. Note that expressions can be arbitrarily complex.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>==</td>
<td>Equal to</td>
</tr>
<tr>
<td>!=</td>
<td>Not equal to</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>AND</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Rule Examples

Example 1: RealAudio Sample Rules

The following table shows several rules defined for RealAudio streams. The various rules are used by the client to switch between different Dolby codecs.

<table>
<thead>
<tr>
<th>Rule number</th>
<th>Low-bandwidth bps</th>
<th>High-bandwidth bps</th>
<th>Average bandwidth</th>
<th>Average bandwidth standard</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>16000</td>
<td>12000</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>16000</td>
<td>20000</td>
<td>16000</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>20000</td>
<td>40000</td>
<td>20000</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>40000</td>
<td>80000</td>
<td>40000</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>80000</td>
<td>0xffffffff</td>
<td>80000</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>(Used for events)</td>
<td>5</td>
<td>3000</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>
**Example 2: 28.8 Kbps RealVideo Sample Rules**

This table defines rules for delivery of RealVideo over a 28.8 Kbps connection.

<table>
<thead>
<tr>
<th>Stream number</th>
<th>Rule number</th>
<th>Low-bandwidth bps</th>
<th>High-bandwidth bps</th>
<th>Average bandwidth standard</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0 (audio)</td>
<td>0</td>
<td>0xxffffffff</td>
<td>8500</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0 (video keyframes)</td>
<td>0</td>
<td>0xxffffffff</td>
<td>4000</td>
<td>5000</td>
</tr>
<tr>
<td>2</td>
<td>1 (video P frames)</td>
<td>20000</td>
<td>0xxffffffff</td>
<td>7500</td>
<td>1000</td>
</tr>
</tbody>
</table>
The following chapters explain how to build and use plug-ins and features for Helix-based streaming media clients. Topics include clip rendering, audio services, and windowing. See also the chapters in Part I for information about features common to both Helix servers and clients.
TOP-LEVEL CLIENT

A Helix client, such as RealPlayer, consists of two major parts: the top-level client and the client core. The client core handles data transport with Helix Universal Server and provides Helix features such as Audio Services. The top-level client supplies the user interface, communicating to the core and other components, such as rendering plug-ins, through Helix interfaces.

The Helix Client and Server Software Development Kit provides the interfaces required to link to the client core. This chapter describes how to incorporate Helix client methods into an application to communicate with Helix Universal Server and render Helix files through the client’s rendering, file format, and file system plug-ins. The various plug-in sections describe how the plug-ins communicate to the client core response interfaces.

Note: This chapter does not explain how to use ActiveX or Javascript to embed RealPlayer functionality in Web pages. For more on this topic, see the RealOne Player Scripting Guide, available at http://service.real.com/help/library/encoders.html.

Interfaces

A top-level client typically implements the following interfaces:

- **IHXClientAdviseSink.** Header file: hxclsnk.h.
  
The top-level client implements this interface to receive notifications from the client core about changes in a presentation’s playback status.

- **IHXSiteSupplier.** Header file: hxwin.h.
  
  A top-level client implements this interface for use by the client core when displaying data in windows.

- **IHXAuthenticationManager.** Header file: hxauth.h.
  
The top-level client implements this interface to receive notification to gather a user name and password.

- **IHXPacketHook.** Header file: hxphook.h.
  
The top-level client implements this interface to receive presentation recording information from the client core.

- **IHXVolumeAdviseSink.** Header file: hxausvc.h.
  
  A client can implement this interface to receive notifications of changes in volume level or mute state.

- **IHXErrorSink.** Header file: hxerror.h. A top-level client can implement this interface to receive notice of system errors.
• **IHXPropWatchResponse. Header file: hxmon.h.** The top-level client can implement this interface to receive notifications when monitored resources in the Helix Universal Server registry change.

## Building a Client

You can build a top-level client with any development tool. At startup time, the top-level client loads the client core (`rmacore.dll`) and calls that library’s `CreateEngine` function to return a pointer to the client engine object. The top-level client can then use `IHXClientEngine` to utilize the client core and create a player object. On shutdown, the top-level client calls the C-style function `CloseEngine` to close the client engine.

## Using the Client Core

The top-level client receives a pointer to the client core when it creates the client engine at startup time. A rendering plug-in uses `IHXPlayer::GetClientEngine` to get a pointer to the client core object. The top-level client or plug-in can then use `IHXClientEngine` methods to set up new player objects:

- **IHXClientEngine::CreatePlayer**
  Creates a new `IHXPlayer` interface. The top-level client creates a new player object for each presentation. A rendering plug-in can create a new player object to, for example, start a new time and insert an instantaneous sound into the presentation.

  **For More Information:** See “Implementing Midstream Playback and Instant Sounds” in Volume 1, on page 177.

- **IHXClientEngine::ClosePlayer**
  Closes the player object.

- **IHXClientEngine::GetPlayerCount**
  Returns the number of player objects currently associated with the client core.

- **IHXClientEngine::GetPlayer**
  Returns a player object supported by the client core. The top-level client or plug-in uses this after getting the player object count.

- **IHXClientEngine::EventOccurred**
  Passes operating system events to all player objects. The header file `hxwintyp.h` defines the cross-platform event structure.

## Using the Select Method on UNIX

On UNIX, top-level clients should use the `IHXClientEngineSelector::Select` method in place of the UNIX `select`. To the top-level client, this method appears identical to the UNIX `select`, as it includes file descriptor sets and a timeout value. But using this method instead of the UNIX `select` method enables the client core to perform the actions necessary for it to select on the sets.

A top-level client that does not need to control the main loop can call `IHXClientEngineSelector::Select` with no timeout value and possibly nothing set in the file descriptor sets (if it does not have anything it
needs to read from or write to). If the top-level client needs to poll the client core, it can call this method with a timeout of 0 (zero). The core does a quick select operation to determine what needs to be done.

**Note:** IHXClientEngineSelector::Select returns whenever the inner select does. This enables the top-level client to pass nothing in the call, which for a normal select would return only when interrupted by a signal.

### Using a Player Object

For each presentation, there is at least one player object that top-level clients and rendering plug-ins can access through IHXPlayer and IHXPlayer2. A presentation can also have multiple objects. An audio renderer can, for example, create a new player object to start a new timeline and insert an instantaneous sound into the presentation.

**For More Information:** For information on creating new player objects, see “Implementing Midstream Playback and Instant Sounds” in Volume 1, on page 177.

The top-level client uses the following IHXPlayer methods to associate itself with the client core:

- **IHXPlayer::AddAdviseSink**
  Becomes a client advise sink and receives notifications of player status.

  **For More Information:** See “Getting Information About a Presentation” in Volume 1, on page 157.

- **IHXPlayer::GetClientContext**
  Returns the current top-level client.

- **IHXPlayer::SetClientContext**
  Sets itself as the top-level client for the client core.

The IHXPlayer interface includes several methods that enable the top-level client or rendering plug-in to control the presentation. The top-level client typically calls the following methods in response to user input:

- **IHXPlayer::Begin**
  Begins playback of all sources or restart playback after a pause.

- **IHXPlayer::OpenURL**
  Opens the specified URL. The top-level client calls this when the user specifies a location or file to open. Instead of using this method, rendering plug-ins can cause the client or browser to open a URL through hypernavigation.

  **For More Information:** See “Hypernavigating” in Volume 1, on page 166.

- **IHXPlayer::Pause**
  Pauses playback of all sources. If IHXPlayer::Pause is called multiple times, IHXPlayer::Begin must be called the same number of times to restart the presentation. If a renderer and the top-level client
both call IHXPlayer::Pause, for example, they must both call IHXPlayer::Begin before the presentation restarts. The opposite is not true, however. Calling IHXPlayer::Pause always pauses the presentation regardless of how many times IHXPlayer::Begin was called to start the presentation.

- IHXPlayer::Seek
  Seeks in the playback timeline to the designated time, expressed in milliseconds from the start of the presentation timeline.

- IHXPlayer::Stop
  Stops playback of all sources. If restarted, the presentation commences at the beginning of its timeline.

The following IHXPlayer methods enable the top-level client or rendering plug-in to gather information about the presentation:

- IHXPlayer::GetCurrentPlayTime
  Gets the current time on the presentation timeline. Rendering plug-ins do not need to call this method because they receive updates automatically from the client based on their requested synchronization parameters.

  **For More Information:** See “Timing and Synchronization” in Volume 1, on page 169.

- IHXPlayer::GetSource
  Returns a pointer to a designated stream source object. The top-level client or renderer can then use IHXStreamSource to access the stream source object. This object can provide access to the individual stream objects, which use IHXStream. The rendering plug-in section contains information on using these objects and interfaces.

- IHXPlayer::GetSourceCount
  Returns the number of stream sources in the player presentation.

- IHXPlayer::IsDone
  Asks the player object if the current presentation has finished.

- IHXPlayer::IsLive
  Asks the player object if the current presentation contains a live broadcast source.

The following IHXPlayer2 methods enable the rendering plug-in to get or set the player’s preroll properties:

- IHXPlayer2::GetMinimumPreroll
  Gets the minimum preroll for a clip.

- IHXPlayer2::SetMinimumPreroll
  Sets the minimum preroll for a clip.
Getting Information About a Presentation

The top-level client can use `IHXPlayer::AddAdviseSink` to receive notifications of presentation status from a player object. The client core then uses the following `IHXClientAdviseSink` methods to inform the top-level client about presentation changes:

- **`IHXClientAdviseSink::OnPosLength`**
  The presentation timeline’s current position or total length has changed. The method supplies the new position and length.

- **`IHXClientAdviseSink::OnPresentationOpened`**
  The presentation has been opened and is starting playback.

- **`IHXClientAdviseSink::OnPresentationClosed`**
  The presentation has been closed.

- **`IHXClientAdviseSink::OnStatisticsChanged`**
  The presentation statistics have changed.

- **`IHXClientAdviseSink::OnPreSeek`**
  A seek is about to occur. The method passes the last stream time value before the seek and the new time value for when the seek is completed. Renderers are passed this information through `IHXRenderer::OnPreSeek`.

- **`IHXClientAdviseSink::OnPostSeek`**
  The seek has finished. The call includes the point in the stream timeline where the seek occurred and the new time following the seek. Renderers are passed this information through `IHXRenderer::OnPostSeek`.

- **`IHXClientAdviseSink::OnStop`**
  The presentation has stopped.

- **`IHXClientAdviseSink::OnPause`**
  The presentation has paused. The call includes the last time in the presentation timeline before the pause.

- **`IHXClientAdviseSink::OnBegin`**
  Presentation has begun or restarted. The call includes the point in the timeline where the presentation began or restarted.

- **`IHXClientAdviseSink::OnBuffering`**
  The presentation is buffering. The call includes the reason for buffering and the percentage of buffering completed.

- **`IHXClientAdviseSink::OnContacting`**
  The client core is contacting the host. The call includes a pointer to a buffer containing the host address.
Manipulating Groups in a Presentation

The top-level client can manipulate groups in a Helix presentation by using the following interfaces:

- **IHXGroupManager**
  A top-level client or rendering plug-in uses this interface to add or remove sinks; create or add groups; get, set, or remove groups, or get the number of groups in a presentation.

- **IHXGroupSink**
  A top-level client can use this interface to receive notifications of tracks added to a group, tracks removed from a group, tracks started or stopped in a group, and groups added to or removed from a presentation.

- **IHXGroup**
  IHXGroupManager uses this interface to add a track to a group, remove a track from a group, start or get a track in a group, get the number of tracks in a group, and get or set properties of a group.

Adding Site Support to a Client

If you are building a top-level client, you should add site support as described in this section. The top-level client implements IHXSiteSupplier, which the client core uses to inform the top-level client of the need for sites. The top-level client then uses IHXSiteManager to communicate with the client core.

For More Information: See “Sites (Windowing)” in Volume 1, on page 183.

When the client core needs a site, it calls IHXSiteSupplier::SitesNeeded to pass the top-level client a request ID and a pointer to an IHXValues interface that contains the site properties. The top-level client then creates the window and calls IHXSiteManager::AddSite to pass the core a pointer to the IHXSite interface.

The client core calls IHXSiteSupplier::SitesNotNeeded when it is finished with a site. The top-level client then destroys the window and calls IHXSiteManager::RemoveSite to pass the core a pointer to the IHXSite interface.

If the presentation layout changes so that new windows are needed or existing windows are not, the client core calls IHXSiteSupplier::BeginChangeLayout to notify the top-level client. The top-level client can then expect to receive IHXSiteSupplier::SitesNeeded and IHXSiteSupplier::SitesNotNeeded calls. The core calls IHXSiteSupplier::DoneChangeLayout when the new presentation layout is complete.

Handling Password Authentication

When file authentication is required, the top-level client handles the password input from the user. For example, RealPlayer uses a pop-up dialog box to gather the user’s user name and password. If you are implementing Helix client functionality, you need to ensure that your application handles password input.

For More Information: See “Chapter 14: Client Authentication” beginning in Volume 1, on page 189.
Modifying the Top-level Client Sample Code

The Helix Client and Server Software Development Kit provides several sample versions of a top-level client. Each sample demonstrates one or more different nuances for using these top-level clients.

The TestPlay application is a sample top-level client that does not include a graphical user interface. It is provided as source code in the `/source/samples/clientapps/testplay` directory. You can use TestPlay as the basis for building your own Helix client. The TestPlay sample contains the following files:

- `/source/samples/clientapps/testplay/advise_sink.cpp`
  This sample file contains a class that implements the `IHXAdviseSink` interface. Methods in this class will be called by the client core to inform the top-level client when certain events have occurred.

- `/source/samples/clientapps/testplay/authentication_manager.cpp`
  This sample file contains the implementation of a class that handles authentication requests from the client core.

- `/source/samples/clientapps/testplay/context.cpp`
  This sample file contains the implementation of a class that handles the client context.

- `/source/samples/clientapps/testplay/error.cpp`
  This sample file contains the implementation of a class that reports the severity and type of error that occurred.

- `/source/samples/clientapps/testplay/site_supplier.cpp`
  This sample file contains the implementation of a class that creates and destroys the site.

- `/source/samples/clientapps/testplay/fivemmap.cpp`
  This sample file contains a "5-minute map," which is a generalized key/value map.

- `/source/samples/clientapps/testplay/main.cpp`
  This sample file builds an example top-level client that can be used to develop a custom top-level client or as a testing tool for plug-ins.


Another version of TestPlay is located in the `/source/samples/audio/audio_device` directory. This version of TestPlay demonstrates how to implement low-level audio support in the top-level client. The `FakeAudioDevice` interface used in the `audio_device.cpp` file is analogous to some operating system-specific audio services, such as DirectSound. The purpose of the sample is to show how to get audio packets from the core, then pass them to an abstracted and generic audio device. The sample, however, does not actually do anything with the audio packets.

Another TestPlay sample is located in the `/source/samples/video/capture_vframes` directory. For more information on this sample, see “Modifying the Site Sample Code” in Volume 1, on page 187.
CHAPTER 13

RENDERING PLUG-IN

After building a file format plug-in, you create a rendering plug-in that interprets the stream of Helix packets created by the file format plug-in. Using this rendering plug-in, the client can play back your data type whether it is streamed from Helix Universal Server, pseudo-streamed through HTTP, or accessed from a local file. Content tools that need to reassemble Helix streams may also use your rendering plug-in.

Developing Basic Rendering Plug-ins

The rendering plug-in takes the Helix packet stream and, for audio data, decodes and sends the data to the client’s Audio Services interface. For video, image, and text data, the plug-in decodes the data and writes to a client window. A rendering plug-in can also direct the Web browser on the client to display URLs (hyperlinks) at specified times in the stream. This is called hypernavigation.

The following illustration shows how a rendering plug-in receives RealMedia data packets and then sends the data to clients after decoding it.

**Rendering Plug-in**

- RealMedia packets
- Audio Services
- Client window
- Rendering plug-in
- Client

Design Considerations

In addition to the general plug-in design considerations discussed in “Designing a Plug-in” in Volume 1, on page 35, keep the following points in mind as you develop your rendering plug-in:

- Write your rendering plug-in to support the playing of several streams at the same time on the same computer.
- Carefully optimize the rendering plug-in for CPU performance. This helps reserve processing power for rendering other data types simultaneously, which is an important feature of Helix.
• As the client buffers the stream, the rendering window should display the data type’s logo. Design a logo suitable for rendering, but enable producers who create streaming content to turn off the logo display through the Synchronized Multimedia Integration Language (SMIL).

• Consider carefully whether producers will lay out content of your data type in a browser driven by Helix or in RealPlayer using SMIL. RealNetworks can consult with you on the renderer user interface design.

• Content producers are concerned with providing a pleasing user experience on 640x480 and 800x600 screens, given that users often have big browser buttons and multiple toolbars. They need the ability to alter the rendering window size through SMIL and the stream. The rendering window therefore needs to respond rationally to size change requests.

Interfaces

A rendering plug-in typically implements the following interfaces:

• IHXPlugin. Header file: hxplugn.h.
  Every plug-in implements this interface, which Helix uses to determine the plug-in’s characteristics.

• IHXRenderer. Header file: hxrendr.h.
  All rendering plug-ins must implement this interface, which handles header and packet reception, as well as stream status information.

• IHXSiteUser. Header file: hxwin.h.
  A display renderer typically implements this interface, which the client uses to associate the renderer with a site object and inform it of events.

• IHXSiteUserSupplier. Header file: hxwin.h.
  This interface is used to get instances of the objects that use sites. A rendering plug-in or the client can implement this interface.

• IHXPacketHookHelper. Header file: hxplugn.h.
  This interface instructs the plug-in to send packets used to record the presentation on the client computer.

In addition, a rendering plug-in typically uses the following interfaces:

• IHXSite2. Header file: hxsite2.h.
  This interface is implemented by the client core and is used by the renderer to obtain a reference to the client’s video surface.

• IHXVideoSurface. Header file: hxvsurf.h.
  The client core implements this interface, and the rendering plug-in uses it to draw on the client’s display. This interface offers device-independent methods for image rendering.
Coding the Plug-in

The following sections explain how a Helix client and a rendering plug-in use the Helix interfaces to render data. The sample files included with this SDK illustrate many of these features. You can use these sample files as a starting point for building your own plug-in. Refer to the Helix Client and Server Software Development Kit (SDK) header files for more information on function variables and return values.

**Note:** The order of function calls listed in the following sections provides a generalized explanation and is for illustrative purposes only. Because Helix is asynchronous, your plug-in must be able to handle any call made to it while it is processing data or waiting for a response from another object. Do not code your plug-in so that it expects a specific sequence of events to occur as it interacts with Helix.

Starting Up

When the Helix client is started, it loads each rendering plug-in. The Helix client performs a series of calls to your plug-in’s functions and methods in a specific order.

➤ To provide start-up support in your rendering plug-in:

1. Create a new instance of the renderer by implementing `HXCreateInstance`. The client calls this function at start-up and each time it receives a stream to be rendered by the plug-in.

   **For More Information:** See “Creating a Plug-in Instance” in Volume 1, on page 36.

2. Return descriptive information about the plug-in, including its copyright and “more information” URL, by implementing the `IHXPlugin::GetPluginInfo` method. The `bLoadMultiple` attribute should be set to `TRUE` to enable the client to open multiple instances of the plug-in in separate processes.

   **For More Information:** See “Defining Plug-in Attributes” in Volume 1, on page 37.

3. Return functional information about the renderer by implementing the `IHXRenderer::GetRendererInfo` method. The values for the `pStreamMimeType` parameter indicate which stream MIME types the renderer handles. The client uses this information when determining which renderer to use for a stream. The value for the `unInitialGranularity` parameter indicates how often the renderer wants to receive timeline synchronization information from the client. The minimum time is 20 milliseconds.

   **For More Information:** See “Timing and Synchronization” in Volume 1, on page 169.

Initializing

When the Helix client receives a stream, it identifies the appropriate rendering plug-in to use based on the stream’s Multipurpose Internet Mail Extensions (MIME) type and the `pStreamMimeType` values returned by the rendering plug-ins during startup. If two or more plug-ins handle the same MIME type, the client uses the first plug-in for that MIME type that it loaded during startup.

During rendering plug-in initialization, the client performs a series of calls to your plug-in’s methods in a specific order.
To provide initialization support in your rendering plug-in:

1. Perform any necessary plug-in initialization procedures by implementing the \texttt{IHXPlugin::InitPlugin} method. The client uses this method to pass a pointer to the system context. Minimally, this method can use the context pointer to store a reference to \texttt{IHXCommonClassFactory} so the plug-in can later create Helix objects used in rendering data.

2. Perform any necessary rendering initialization procedures by implementing the \texttt{IHXRenderer::StartStream} method. The client calls this method to give the plug-in access to the client through \texttt{IHXPlayer}, as well as to give it access to the stream to be rendered through \texttt{IHXStream}. Within \texttt{IHXRenderer::StartStream}, the renderer can perform any other initialization functions based on its supported features:

- Setting itself up as a site user if it is a display renderer.

  \textbf{For More Information:} See “Sites (Windowing)” in Volume 1, on page 183.

- Querying the stream object to see whether it supports \texttt{IHXBackChannel} communications to the file format plug-in.

- Querying the stream object for \texttt{IHXASMStream} to see whether ASM is supported. If so, the renderer can use \texttt{IHXASMStream::AddStreamSink} to set itself up to receive the client's rule subscription choices. If it receives Real-Time Protocol (RTP) payloads, the renderer must do this to find out if the RTP payload marker bit is on or off.

  \textbf{For More Information:} For information on Adaptive Stream Management (ASM), see “Chapter 11: Adaptive Stream Management”. The marker bit property is described in “RTP Marker Bit Property” in Volume 1, on page 146. For more on RTP payloads, see “Supporting Multiple Packet Formats” in Volume 1, on page 63.

3. Access the stream header data by implementing the \texttt{IHXRenderer::OnHeader} method. The client calls this method to pass the renderer a pointer to the stream header object created by the file format plug-in. This method uses \texttt{IHXValues} methods to retrieve the header data and then releases the object.

4. Return the renderer's preferred display type by implementing the \texttt{IHXRenderer::GetDisplayType} method. The plug-in returns \texttt{HX_DISPLAY_WINDOW} if it renders data on the screen or \texttt{HX_DISPLAY_NONE} if it does not use a screen because it is, for example, an audio renderer.

\textbf{Rendering Streams}

After the rendering plug-in has been initialized, the client and the rendering plug-in can then begin rendering a data stream.

To render a data stream:

1. Receive the call indicating the beginning of playback by implementing the \texttt{IHXRenderer::OnBegin} method. The client calls this method to inform the renderer that playback has begun or has resumed after a pause. The client passes the renderer the stream's timeline value, in milliseconds. This is 0 (zero) if the stream is just starting. When resuming after a pause, the client passes an integer value that indicates how many milliseconds into the stream timeline to begin the playback.
2. Receive the call indicating that a packet is ready to render by implementing the `IHXRenderer::OnPacket` method. The client calls this method each time a packet is ready (or should be ready but is lost), passing it a pointer to the `IHXPacket` object to be rendered. The client also passes in the packet’s time offset from the start of the stream. Although the client ensures that packets are delivered in order and are resent if necessary, it cannot guarantee that packets are not lost. Before attempting to process the packet, your rendering plug-in can call the `IHXPacket::IsLost` method to determine whether the packet has been lost. Your rendering plug-in is responsible for taking the appropriate action to handle the packet loss.

3. Receive the call indicating that data is buffering by implementing the `IHXRenderer::OnBuffering` method. The client passes in the reason for buffering (such as the stream starting, a seek has occurred, network congestion, and so on) and the percent complete.

4. Receive the call indicating a time sync has occurred by implementing the `IHXRenderer::OnTimeSync` method. The client calls this method periodically depending on the value of `unInitialGranularity` (`IHXRenderer::GetRendererInfo`) that was returned by your rendering plug-in during client startup. The client passes in the current playback time. The renderer uses this information to synchronize playback of its stream with the presentation.


5. Render the data as necessary. This can include the following:
   
   • Sending audio data to Audio Services.
   
   Audio Services provides a device-independent, cross-platform interface that enables multiple renderers to share the audio device. It also allows access to the data sent to the audio device. This enables the plug-in to add audio effects or perform other processing.

   For More Information: See “Audio Services” in Volume 1, on page 171.

   • Painting display data in a site window.
   
   Helix site interfaces enable the rendering plug-in to perform windowing actions on multiple platforms through generic code.

   For More Information: See “Sites (Windowing)” in Volume 1, on page 183.

The renderer can render data in `IHXRenderer::OnPacket` or store packets until `IHXRenderer::OnTimeSync` is called. (Note that, because of the automatic support for object lifetime that the Component Object Model (COM) provides, storing packets does not require copying the data.) Renderers that use Audio Services, for example, can render in `IHXRenderer::OnPacket` and leave the timeline synchronization to Audio Services. Video renderers typically render in `IHXRenderer::OnTimeSync`.

6. Call `IUnknown::Release` when finished with the data. The renderer provided with the SDK sample code, for example, keeps only the last packet.

Seeking

During a presentation, the user can use the client’s seek function to go to a different point in the presentation’s timeline.
To enable seeking in a presentation:

1. Implement `IHXRenderer::OnPreSeek`, which the client calls to pass the last stream time value before the seek and the new time value for when the seek completes.

2. Implement `IHXRenderer::OnPacket` and `IHXRenderer::OnBuffering`, which the client calls to pass the renderer any buffered packets that post-date the seek action. Although your renderer should not render this data, it has access to the packets for any purpose. It can simply release the packets if it has no need for them.

3. Implement `IHXRenderer::OnPostSeek`, which the client calls as soon as the seek is completed. The call includes the time in the stream timeline when the seek occurred, and the new timeline value following the seek.

4. Implement `IHXRenderer::OnPacket`, `IHXRenderer::OnBuffering`, and `IHXRenderer::OnTimeSync`, which the client calls to pass the renderer packets for data beginning at the new point in the presentation timeline, and to provide synchronization information.


Pausing

During a Helix presentation, the user can pause the presentation.

To enable pausing of a presentation:

1. Implement `IHXRenderer::OnPause`, which the client calls to provide the renderer with the stream’s time value in milliseconds just before pausing.

2. Implement `IHXRenderer::OnBegin`, which the client calls to inform the renderer that playback has resumed after the pause. It passes the renderer an integer value that indicates the stream’s time value in milliseconds after the pause.

Hypernavigating

Hypernavigation occurs when a rendering plug-in directs the client to display a URL at a specified time in the stream. When the plug-in issues a hypernavigation request, the default Web browser opens. If the browser is open already, the target URL displays in the current window. The plug-in can also specify that the URL display in a specific frame of the current browser window.

A rendering plug-in hypernavigates with `IHXHyperNavigate::GoToURL`. The function takes two parameters, a fully qualified URL and a frame target (NULL for no frame target). The following sample code shows a hypernavigation request that does not target a frame:

```cpp
m_pHyperNavigate->GoToURL("http://www.real.com", NULL);
```

Sending Back-Channel Packets

If the renderer’s corresponding file format plug-in implements `IHXBackChannel`, the renderer can send the file format plug-in data in an `IHXPacket`. This can be any feedback or control data that is opaque to the Helix architecture and is necessary for plug-in operation.

The rendering plug-in queries for `IHXBackChannel` on the `IHXStream` interface. It then calls the `IHXBackChannel::PacketReady` method to pass the file format plug-in a pointer to the packet. Note the following, however:
• The back channel is supported only by the RTSP protocol. The back channel is not available when client/server communication uses HTTP, for example.

• The back channel is not available if the stream source is a container data type that does not support the channel.

For More Information: For the basics of packet creation, see “Using IHXPacket to Create Stream Packets” in Volume 1, on page 33.

Terminating a Presentation

Once all of the packets have been delivered to the rendering plug-in, the client informs the plug-in that the presentation has terminated.

➤ To enable proper termination of a presentation:

1. Implement IHXRenderer::OnEndOfPackets, which the client calls to inform the renderer that all packets have been delivered to it. The user might still seek backwards through the stream, however, so the renderer should not deallocate resources at this point.

2. Implement IHXRenderer::EndStream, which the client calls to inform the renderer when the stream ends. The renderer then deallocates resources as necessary. At this point the renderer can no longer access the stream object, but it can still paint to a window and handle interaction such as mouse clicks.

3. The client destroys the renderer object when a new stream begins.

Accessing Stream and Player Objects

A rendering plug-in has access to several objects that it can use to gather stream information and affect the presentation. During initialization, the client passes the rendering plug-in pointers to the player and stream objects. The renderer can then use IHXPlayer to access client player functions (see “Chapter 12: Top-Level Client”) and use IHXStream to get stream information (see the following section).

Using the Stream Object

The IHXStream interface gives the rendering plug-in access to the stream object for a stream it is rendering. The rendering plug-in can then use this interface’s methods to get the stream number, the stream MIME type, and a pointer to the IHXValues interface that contains the stream header information (the plug-in also receives this pointer during initialization through IHXRenderer::OnHeader).

Because the renderer can best judge how problems such as packet loss are affecting the presentation, it can call IHXStream::ReportQualityOfService to report a change in service quality. The method passes an integer value denoting the relative level of service, with 0 as the worst possible. When the problem has been eliminated, the renderer reports service quality of 100. The client can report this information in the user interface.

The renderer uses IHXStream::ReportRebufferStatus to report that available data has dropped critically low. The method takes two values, the number of packets needed to render the presentation smoothly and the number currently available. For example, it calls the method with “5,0” if it needs five packets and none are available. In this case it continues to pass its status (“5,1”, “5,2” and so on) until it receives all the packets it needs and calls the function with “5,5”.

167
During startup, the plug-in returns a value for unInitialGranularity through IHXRenderer::GetRendererInfo. This value sets how frequently it wants to receive timeline synchronization information from the client. The plug-in can later use IHXStream::SetGranularity to change this timing granularity.

**For More Information:** See “Timing and Synchronization” in Volume 1, on page 169.

**Using the Stream Source Object**

A rendering plug-in can call IHXStream::GetSource to get a pointer to the stream source object, which is the file object from which the rendered stream or streams are generated. The renderer then has access to the IHXStreamSource methods that determine whether the stream is a live broadcast, that get a pointer to the player object (the plug-in also receives this pointer during initialization through IHXRenderer::StartStream), get the requested URL for the stream source, return the number of streams supported by the source, and return a pointer to a stream.

**Rendering Images on the Client Display**

For a Helix client to display images, a renderer must draw them on the client’s target display area. The renderer can do this using platform-specific functions (making it a “Windowed Renderer”), or it can use platform-independent Helix functions (making it a “Windowless Renderer”).

**For More Information:** See “Sites (Windowing)” in Volume 1, on page 183.

Helix functions are much more convenient than platform-specific functions. To use platform-specific functions, for example, the renderer must handle the PAINT message appropriate for each platform. To use Helix functions, however, the renderer only needs to handle the cross-platform HX_SURFACE_UPDATE message.

The Helix Architecture (HX) client core provides this device-independent windowing functionality to renderers. Each drawing area owned by the client is supplied as a “site.” To send display data to the site, the renderer registers as a “site user.”

The site’s drawing area, or “video surface,” is made available to the rendering plug-in through a IHXVideoSurface interface. This interface supports a variety of image formats and is uniform across all platforms.

The renderer, in turn, implements the IHXSiteUser interface so that it can receive HX event messages (HX_SURFACE_UPDATE and others) from the client, and so it can attach, monitor, and detach sites.

The client core also implements the IHXSite2 interface, and the renderer calls the IHXSite2::GetVideoSurface method to get an IHXVideoSurface interface for the site. This IHXVideoSurface interface, also implemented by the client core, is used by the renderer to:

- Specify the bitmap format to be used (see hxvsurf.h for the supported formats).
- Ascertaining the preferred format for this video surface.
- Send the bitmap images.
- Release the video surface when the renderer is finished sending image data.
In addition, the renderer must implement the IHXSiteUser interface, because its methods are called by the client whenever the renderer must respond to events, either the HX_SURFACE_UPDATE message (the surface must be redrawn) or mouse events. The client also uses the IHXSiteUser interface to attach sites to the renderer as needed for new image data streams, and to detach sites when rendering is complete.

After the rendering plug-in has obtained the IHXVideoSurface for the client site, it is ready to draw images. To do this, the renderer:

1. Calls IHXVideoSurface::BeginOptimizedBlt to set the drawing format, such as the size, compression type, and so on.
2. Calls IHXVideoSurface::OptimizedBlt on the video surface, passing the image data to be displayed. The Helix core draws the bitmap on the client’s screen.
3. Responds as needed to the HX_SURFACE_UPDATE event message, or any of the various HX_MOUSE_EVENTS (see hxevent.h)
4. Calls IHXVideoSurface::EndOptimizedBlt when the drawing is complete.

To get a better understanding of rendering to Helix’s HXVideoSurface, refer to the sample rendering plug-in located at /source/samples/datatype/videosurface_renderer/videosurface_renderer.cpp. This example clearly demonstrates the use of the video surface and its supporting interfaces. You will also want to refer to hxvsurf.h, hxwin.h, and hxsite2.h.

**Timing and Synchronization**

In Helix, the client sends time synchronization intervals to renderers on a per renderer rather than per stream basis. This is because Helix supports container data types in which multiple renderers read from the same source. At startup time, each renderer requests a time synchronization interval in milliseconds using the unInitialGranularity attribute. The client then calls IHXRenderer::OnTimeSync each interval.

*Note:* Audio Services, when used, sets the interval to the shortest interval requested by any rendering plug-in. These services are explained in “Audio Services” in Volume 1, on page 171.

The time stamps assigned to packets are relative to the start time of the stream. Packet times are adjusted by the stream start time and the preroll:

\[
\text{delivery time} = \text{packet time} - \text{stream preroll} + \text{stream start}
\]

For example, if stream 1 has a preroll of 3000 milliseconds and a start time of 180000 milliseconds (3 minutes), and the first packets are time stamped 0, 500, and 1000 milliseconds, these packets are delivered to your rendering plug-in at 177000, 177500, and 178000 milliseconds.

How the rendering plug-in renders the data is a detail of your data type. You can implement a scheme whereby you pass the render times down with the packets. Or, you can use some offset from the delivery time. Suppose that you have two distinct data items, item A and item B, in your presentation. Item A is to be displayed 5 seconds into the presentation and item B is to be displayed 10 seconds in. The information that Item A needs to be displayed at 5 seconds is something you should include in your data type packet payload just as you need to store information about “packetization,” such “this packet is part N of M of Item A.” You have two options; you can use the “delivery time” values and set a
preroll to give your plug-in computation time. Or, you can place rendering time information in your opaque data.

“Duration” for non-time-based object streams, such as still images, is also up to your data type. A timeline-oriented tool would enable the user to specify the image display time. A powerful tool would enable the user to specify that an image display after a second image has been downloaded. You could, for example support downloading several images while the first one is displayed, then quickly flip through the downloaded image. This is all specific to your data type implementation.

Modifying the Basic Rendering Plug-in Sample Code

The Helix SDK includes sample rendering plug-ins that you can use as a starting point for building your own plug-in:

- /source/samples/datatype/fileformat1_renderer/fileformat1_renderer.cpp
  A basic rendering plug-in that renders data streamed from its corresponding, basic file format plug-in. Build these sample files to learn basic concepts about rendering Helix packets.

- /source/samples/datatype/example_renderer/example_renderer.cpp
  This intermediate sample file renders data from the sample file format plug-in /source/samples/server/fileformat/fileformat.cpp to a window. It is designed to support multiwindow rendering.

- /source/samples/datatype/singlewindow_renderer/singlewindow_renderer.cpp
  This intermediate sample file is designed to support single-window rendering without the use of the IHXMultiInstanceSiteUserSupplier object. It also illustrates how to create and manage child window controls such as buttons.

- /source/samples/datatype/example_renderer2/example_renderer2.cpp
  This advanced sample file performs the same functions as the intermediate sample renderer. It also shows how to create a new player to start a new timeline and play continuous background music, or open a different, unrelated URL, such as a stream for advertisement.

  **Note:** The SDK also provides sample renderers that use the Audio Services interface. For these, see “Modifying the Audio Rendering Sample Code” in Volume 1, on page 181.

Perform the steps in the following procedure to change the intermediate or advanced sample renderer code. These steps assume that your company name is “Foo Bar, Inc.”, your file extension is .foo, and the MIME type of your data stream is application/x-foobar.

➤ To modify the plug-in sample code:

1. Copy the sample code from the Samples directory to a working directory. Change the file name and class names to match your file format name. For example, if you are implementing a Foo data type you might replace all occurrences of CExampleRenderer with CFooRenderer, and rename the files foorendr.cpp and foorendr.h.

2. Change the plug-in description, copyright information, and more info URL stored in zm_pDescription, zm_pCopyright, and zm_pMoreInfoURL. For the Foo example, you could change the values as follows:
char* CFooRenderer::zm_pDescription = "Foo Rendering Plug-in";
char* CFooRenderer::zm_pCopyright = "(c)1997 Foo Bar";
char* CFooRenderer::zm_pMoreInfoURL = "http://www.foobar.com";

3. Change the stream MIME types stored in zm_pStreamMimeTypes. For the Foo example, you could change the values as follows:
   char* CFooRenderer::zm_pStreamMimeTypes = "application/x-foobar";

4. Process the Helix packets, and render the data as necessary.

5. Compile, debug, and test your plug-in.


Audio Services

Helix’s Audio Services provides device-independent, cross-platform audio services to a rendering plug-in. The rendering plug-in can use Audio Services to render audio streams without concern for the specifics of the audio hardware. Audio Services provides several useful features for rendering audio in Helix:

- Audio mixing
  Audio Services converts streams in supported input formats to the same sampling rate and then mixes the streams to produce a single output.


- Post-processed audio data
  Through Audio Services, a plug-in has access to decoded streams of pre-mixed audio data and to the final, mixed audio data. Plug-ins can intercept this data to add special effects or perform other processing.


- Volume control
  With the volume interface, a plug-in can control the volume of individual streams, of the final mixed stream, and of the audio hardware.


- Midstream playback and instant sounds
  This feature enables the rendering plug-in to start an audio stream at any specified time in the current presentation or play an “instant sound” at the current time.

   For More Information: See “Implementing Midstream Playback and Instant Sounds” in Volume 1, on page 177.

- Audio synchronization
Audio Services provides playback synchronization to all rendering plug-ins. The synchronization values are based on the actual playback time in the audio hardware.

For More Information: See “Synchronizing to Audio” in Volume 1, on page 181.

As shown in the following illustration, a rendering plug-in uses the Player object to register with Audio Services and request audio data. The plug-in then writes each audio stream to a separate stream object.

Interfaces

A rendering plug-in implements the following interfaces, depending on which Audio Services features it needs to use:

- IHXAudioHook. Header file: hxausvc.h.
A rendering plug-in implements this interface to access pre- or post-mixed audio data, as well as to get post-processed audio buffers and their associated audio formats.

- **IHXAudioStreamInfoResponse**. Header file: hxausvc.h.
  
  The rendering plug-in implements this interface to receive notification of the total number of streams associated with an audio player.

- **IHXDryNotification**. Header file: hxausvc.h.
  
  A rendering plug-in implements this interface to receive notice of a dry audio stream.

- **IHXVolumeAdviseSink**. Header file: hxausvc.h.
  
  A rendering plug-in implements this interface to receive notifications of changes in volume level or mute state. The plug-in uses the **IHXVolume** interface to register to receive the notifications.

A rendering plug-in uses the following interface to access Audio Services functions:

- **IHXAudioPlayer**. Header file: hxausvc.h.
  
  This interface provides access to Audio Services. A rendering plug-in uses it to create audio streams, “hook” post-mixed audio data, and control volume levels. Its response interface, used solely by the Helix client to receive playback notifications, is **IHXAudioPlayerResponse**.

- **IHXAudioStream**. Header file: hxausvc.h.
  
  The renderer uses this interface to access a stream object to play audio, “hook” audio stream data, and get audio stream information.

**Supported Input Formats**

Audio Services accepts 8-bit and 16-bit pulse code modulation (PCM) data. The rendering plug-in must convert audio data in other formats, such as Musical Instrument Digital Interface (MIDI) or µ-law, to PCM before sending it to Audio Services. However, the plug-in can also write MIDI data directly to the MIDI hardware, bypassing Audio Services entirely.

The audio data can be either monaural or stereo at any of the following sampling rates:

- 8000 Hz
- 11025 Hz
- 16000 Hz
- 22050 Hz
- 32000 Hz
- 44100 Hz

**Mixing Audio Streams**

When multiple rendering plug-ins send audio data to Audio Services, Audio Services mixes the inputs as follows:

- With a 16-bit sound card, all mixed output is 16-bit, even if all inputs are 8-bit or a combination of 8-bit and 16-bit. However, if the **Player Sound card compatibility** preference is set to disable 16-bit sound, the mixed output is always 8-bit.
• With a stereo sound card, all output is stereo, even if all inputs are monaural or a combination of monaural and stereo.

• If the inputs are encoded at different sampling rates, Audio Services resamples the inputs to a single rate based on the values of two RealPlayer preferences:
  • If the **Sound card compatibility** is set to disable custom sampling rates, Audio Services converts all inputs to a 11,025-Hz sampling rate.
  • If the **Playback performance** slider is set to (or next to) **Best Audio Quality**, Audio Services upsamples to the highest sampling rate of any input. But if the **Playback performance** slider is set to (or next to) **Lowest CPU Usage**, Audio Services downsamples to the lowest sampling rate of any input.

Suppose that one plug-in renders 11,025 Hz stereo and another plug-in renders 22,050 Hz monaural at the same time. Playback performance is set to **Best Audio Quality**. In this case, Audio Services upsamples the 11,025 Hz input to 22,050 Hz and converts the monaural input to stereo before mixing the two signals. It then sends a 22,050 Hz stereo signal to the audio device.

**Quality of Mixed Audio Streams**

When multiple streams are mixed, the quality of the mixed audio output depends on the type and quality of the input streams. Some types of input mix better than others, and higher quality input always gives higher quality output.

To achieve the best possible quality, encode all mixed input streams at 44100 Hz. If the audio is digitized from an analog source, use the same audio hardware to digitize each input. Avoid mixing audio encoded at multiples of 8000 Hz with audio encoded at multiples of 11025 Hz.

**Coding a Rendering Plug-in for Audio**

Perform the steps in the following procedure to use Audio Services to render audio data with your rendering plug-in. These steps are based on the sample rendering plug-in `exaudio.cpp`. For more information about this sample file, see “Modifying the Audio Rendering Sample Code” in Volume 1, on page 181.

**For More Information**: Be sure to review “Chapter 13: Rendering Plug-In”.

- **To render audio data with a rendering plug-in:**
  1. In the `IHXRenderer::StartStream` method of the rendering plug-in, get the interface to the `AudioPlayer` object:
     ```
     QueryInterface(IID_IHXAudioPlayer, (void**) &m_pAudioPlayer)
     ```
  2. Create an `AudioStream` object with `IHXRenderer::OnHeader`:
     ```
     m_pAudioPlayer->CreateAudioStream(&m_pAudioStream);
     ```
     If the rendering plug-in needs to render more than one audio stream, create a stream object for each stream.
  3. Determine the format of the audio data sent to the rendering plug-in. This is often specified in parameters passed in `IHXRenderer::OnHeader`. The following data are required:
• The number of channels
• The number of bits per sample
• The number of samples per second
• The maximum block size of the data

4. Set the values of the audio stream parameters and then initialize the AudioStream object. The pValues parameter contains information to identify this stream uniquely.

\[
m\_pAudioStream->Init(&AudioFmt, pValues);
\]

5. In the rendering plug-in’s IHXRenderer::OnPacket method, perform renderer-specific decoding or processing of each audio packet sent by RealPlayer. Set the buffer size and the start time of the packet in milliseconds. Set the uAudioStreamType member of the HXAudioData structure to one of these values:

• TIMED_AUDIO
  Use this value for the first packet of the stream or, if packets were lost, the first received packet that follows the lost packets.

• STREAMING_AUDIO
  Use this value for all packets that follow the TIMED_AUDIO packet.

• INSTANTANEOUS_AUDIO
  Use this value to play the buffer immediately.

For More Information: See “Implementing Midstream Playback and Instant Sounds” in Volume 1, on page 177.

Note that within a single stream you cannot write packets that have start times earlier than packets already written. If you need to process these packets first, either buffer them for later writing within the current stream or create a separate audio stream for them.

6. Write the PCM audio data to the AudioStream object:

\[
m\_pAudioStream->Write(&audioData);
\]

After calling IHXAudioStream::Write for timed audio, the plug-in increments the ulAudioTime member of the HXAudioData structure by the length of the buffer just written to get the time of the next buffer.

7. In IHXRenderer::EndStream, which RealPlayer calls when the stream finishes, release the audio player and stream objects:

\[
if (m\_pAudioPlayer)
{
  m\_pAudioPlayer->Release();
  m\_pAudioPlayer = NULL;
}
if (m\_pAudioStream)
{
  m\_pAudioStream->Release();
  m\_pAudioStream = NULL;
}
\]
Controlling Volume

Audio Services provides the IHXVolume interface to query, set, and mute volume, as well as register a plug-in for notifications through the IHXVolumeAdviseSink interface. IHXVolumeAdviseSink then enables your plug-in to receive notices of changes to volume and mute settings. Audio Services enables plug-ins to control the volume of individual streams, of the final mixed stream, and of the physical audio device, as described in the following paragraphs:

- **Individual streams**
  
  Each input stream has an IHXVolume interface that maintains the volume and mute settings. You control the stream volume by multiplying each audio sample by a volume value. The maximum volume setting of 100 yields 100 percent of the input signal. Values less than 100 reduce the volume proportionally. Call IHXAudioStream::GetAudioVolume to return a pointer to the IHXVolume interface.

- **Final mixed stream**
  
  The final mixed stream for the player has its own IHXVolume interface. A volume setting of 100 yields 100 percent of the input signal. The maximum volume setting of 100 yields 100 percent of the input signal. Values less than 100 reduce the volume proportionally. Call IHXAudioPlayer::GetAudioVolume to return a pointer to the IHXVolume interface.

- **Audio device**
  
  An IHXVolume interface also controls the audio device volume. The audio device volume can range from 0 to 100 percent. A volume setting of 0 yields no sound. A volume setting of 100 means the maximum volume for the audio hardware. To return a pointer to the IHXVolume interface, call IHXAudioPlayer::GetDeviceVolume.

The following illustration shows the relationships among the plug-in, RealPlayer, the audio device, and the various volume objects:

![Volume Control Diagram](image-url)
Implementing Midstream Playback and Instant Sounds

Audio Services makes it possible for a rendering plug-in to begin playback of an audio stream at any specified time in the current presentation's timeline. It also provides a special case of midstream playback that starts a new stream at the current time. These “instant sounds” are typically linked to events such as keyboard or mouse input. The following illustration shows a stream (Stream 1) being played. At time \( T \), another audio stream starts. At the current time, an instant sound plays.

![Instant Sound Diagram](image)

The following procedure and illustration explain how to add midstream playback and instant sound capabilities to your presentations.

1. **To implement midstream playback and instant sounds:**
   1. Get the **AudioPlayer** objects, and initialize the stream as described in “Coding a Rendering Plug-in for Audio” in Volume 1, on page 174.
   2. Use the **ulAudioTime** member of the **HXAudioData** structure to set the packet time to begin playing the new stream. This attribute determines the number of milliseconds into the presentation timeline to start the new stream. To start the new stream at 15 seconds into the current stream, for example, set **ulAudioTime** to 15000. To find out what time **ulAudioTime** should be set to for playback to begin with the next audio block being written, call **IHXAudioStream::Write** with **pData** set to NULL. **ulAudioTime** will be set to the next audio time stamp.
   3. Call **IHXAudioStream::Write**.
   4. Send additional packets of the new stream using **IHXAudioStream::Write**. Increment **ulAudioTime** by the length (in milliseconds) of the last buffer sent.

Midstream Playback and Instant Sound

![Midstream Playback and Instant Sound Diagram](image)
Using Post-Processed Audio Data

An application that needs access to data sent to the audio device, such as an application that adds sound effects to a stream, can receive pre-mix audio data (individual decoded streams) or post-mix audio data (final mixed stream). The plug-in receives data as headerless buffers that it can modify and return to Audio Services, or even pass to another plug-in. Various Audio Services methods enable the plug-in to obtain the stream's sampling rate, number of channels, and audio format attributes. As explained in “Supported Input Formats” in Volume 1, on page 173, the audio output format depends on the inputs to Audio Services.

**Note:** Your plug-in must be able to receive and modify post-processed audio data synchronously in real-time. Ensure that your plug-in platform is capable of performing such real-time processing.

Getting Pre-Mix Audio Data

As shown in the following illustration, a plug-in can examine and modify pre-mix audio data, which is the decoded data from a single stream, before Audio Services mixes it with other streams.

**Pre-Mix Audio Data**

The following procedure explains how to get pre-mix audio data. For more information on using this code, see “Modifying the Audio Rendering Sample Code” in Volume 1, on page 181.

**To get pre-mix audio data:**

1. Implement an IHXAudioStreamInfoResponse class.
2. Implement an IHXAudioHook class.
3. Do the following in your IHXRenderer::StartStream method:
   a. Create an IHXAudioPlayer using IUnknown::QueryInterface.
   b. Create an IHXAudioHook interface.
   c. Create an IHXAudioStreamInfoResponse interface.
   d. Register your IHXAudioStreamInfoResponse interface with the AudioPlayer:
      ```cpp
      m_pAudioPlayer->SetStreamInfoResponse(m_pResp);
      ```
4. When the AudioPlayer object passes the stream (or streams) to your renderer with IHXAudioStreamInfoResponse::OnStream, test the appropriate IHXValues name/value pair to determine whether this is the desired stream. For example, the following sample code locates the stream with “MimeType” equal to audio/x-hx-wav:

```c
{ 
    IHXValues*         pValues = 0;
    IHXBuffer*         pMimeType = 0;

    pValues = pAudioStream->GetStreamInfo();
    pValues->GetPropertyCString("MimeType", pMimeType);

    char* pMime = (char*) pMimeType->GetBuffer();
    char* pStreamName = (char*) m_pHookStreamName->GetBuffer();

    /* In this example, let's hook all wav streams. */
    if (pMime && pStreamName && (!strcmp(pMime, pStreamName)))
    {
        /* Add pre mix hook on this stream. */
        pAudioStream->AddPreMixHook(m_pHook, FALSE);
    }

    return HXR_OK;
}
```

The call to `pAudioStream->AddPreMixHook(m_pHook, FALSE)` adds the pre-mix hook. The `m_pHook` parameter is the pointer to the IHXAudioHook interface. The `bDisableWrite` parameter is set to FALSE to send the stream to the audio mixer. Set `bDisableWrite` to TRUE to keep the stream out of audio mixing. Remove a hook with `pAudioStream->RemovePreMixHook(m_pHook)`.

5. After IHXAudioStream::AddPreMixHook is called, the AudioStream object calls IHXAudioHook::OnInit and passes the audio format of the audio stream. Within this method, initialize the plug-in as needed.

6. For each buffer of audio data in the stream, the AudioStream object calls IHXAudioHook::OnBuffer and passes the audio data from the stream. Copy the contents of the buffer and process as needed. Do the following if you need to modify the audio data:

   a. Create an IHXBuffer interface to store the modified audio data.
   b. Modify the data as needed.
   c. Return a pointer to the modified data as the second parameter of IHXAudioHook::OnBuffer.

Getting Post-Mix Audio Data

As shown in the following illustration, a plug-in can modify the post-mix audio data, which is the final audio stream after all audio streams are mixed.
Perform the steps in the following procedure to get post-mix audio data. For more information on using this code, see “Modifying the Audio Rendering Sample Code” in Volume 1, on page 181.

➤ To get post-mix audio data:

1. Implement an IHXAudioHook class.
2. Do the following in the IHXRenderer::StartStream method:
   a. Get and save an IHXAudioPlayer interface through IUnknown::QueryInterface.
   b. Create an IHXAudioHook interface.
3. With IHXRenderer::OnHeader, add the post-mix hook:
   // Add post process hook
   
   BOOL bDisableWrite = FALSE; //write data to the audio device
   
   BOOL bFinal = FALSE;

   m_pAudioPlayer->AddPostMixHook(m_pHook, bDisableWrite, bFinal);

   Specifying bDisableWrite as TRUE prevents Audio Services from sending audio data to the audio device. The plug-in then must write the data to the audio device itself. Even when the plug-in writes the data itself, Audio Services provides all renderers with time synchronization based on a real-time clock. Remove a hook with
   
   M_pAudioPlayer->RemovePostMixHook(m_pHook).

At this point, the following actions occur:

1. The AudioPlayer object calls IHXAudioHook::OnInit with the audio format of the hooked data.
2. The AudioSession object calls IHXAudioHook::OnBuffer with the post-mixed audio data.
3. IHXAudioHook::OnBuffer may change the data; but it must create its own IHXBuffer to do this (use IHXCommonClassFactory) and return the modified data in the pAudioOutData parameter.
Note: If there are multiple audio players due to multiple timelines, the post-mix hook gets data only for its audio player. This data may differ from what gets written to the audio device if there are other audio streams in other timelines as well.

Receiving Notification of a Dry Stream

Helix provides IHXStream::ReportRebufferStatus as a standard means for a plug-in to notify a client that the available data has dropped to a critically low level and rebuffering should occur. If your renderer does not send buffered data because, for example, the rendered data stems from interactive input, you can implement IHXDryNotification to receive notification of a stream running dry, which occurs when the player must write data to the audio device but it does not have enough data to write.


Set up a notification response object with IHXAudioStream::AddDryNotification. The player core then uses IHXDryNotification::OnDryNotification to notify your renderer of a stream running dry. This method passes the following two parameters:

- ulCurrentStreamTime
  This parameter specifies the point in the stream’s timeline when the next packet is expected.
- ulMinimumDurationRequired
  This parameter specifies the minimum amount (time length) of data that needs to be written to prevent silence from occurring.

The renderer must take action synchronously within the function call. It is acceptable for the renderer not to respond. It just means that silence occurs until the renderer delivers the next packets.

Synchronizing to Audio

Helix synchronizes playback to a presentation’s audio track. If there is no audio track, it synchronizes playback based on the system time.

During startup, rendering plug-ins request periodic time synchronization callbacks. The audio hardware generates the synchronization signals based on the actual playback of the audio track. The AudioDevice object passes these signals back to the client, which then issues callbacks to the rendering plug-in through IHXRenderer::OnTimeSync.

The rendering plug-in’s IHXRenderer::GetRendererInfo method specifies the granularity of the time synchronization that the plug-in needs. The player issues callbacks as closely as possible to the requested interval. The minimum granularity is 20 milliseconds.

Modifying the Audio Rendering Sample Code

The Helix Client and Server Software Development Kit (SDK) includes sample Audio Services plug-ins that you can use as a starting point for creating your own plug-in. The /source/samples directory contains code for several basic Audio Services plug-ins. The following list describes the location of plug-in samples that use more advanced Audio Services features:

- /source/samples/datatype/pcm_renderer/pcm_renderer.cpp
This is an intermediate sample file that shows how to render PCM audio data in streaming mode.

- `/source/samples/datatype/pcm_renderer2/pcm_renderer2.cpp`  
  This advanced sample shows how to get notifications from Audio Services when the audio stream is running dry. It is useful if and only if your data type writes minimal audio data in advance. Otherwise, refer to the `pcm_renderer.cpp` sample.

- `/source/samples/datatype/audio_renderer/audio_renderer.cpp`  
  This intermediate sample rendering plug-in uses midstream playback to render data sent by the sample file format plug-in, `/source/samples/server/fileformat`. Before you compile the sample code, change the value of the `pURL` global variable in the source file to the fully qualified URL of `frog.pcm`, located in the `/source/samples/datatype/audio_renderer/testdata` directory. The plug-in will not function correctly without this change.

- `/source/samples/datatype/audio_renderer2/audio_audiorenderer2.cpp`  
  This advanced sample file performs the same functions as the intermediate `exaudio.cpp` sample, but also shows how to create a new player object to start a new timeline and use it for instantaneous sound. Before you compile the sample code, change the value of the `pURL` global variable in the source file to the fully qualified URL of `frog.pcm`, located in the `/source/samples/datatype/audio_renderer/testdata` directory. The plug-in will not function correctly without this change.

- `/source/samples/datatype/pre_post_renderer/premixrd.cpp`  
  This sample uses the pre-mix audio interface to intercept a stream before it is mixed. It demonstrates how to change the audio data by reducing the volume of the stream. This plug-in renders data sent by the post-process sample file format plug-in, `/source/samples/datatype/pre_post_fileformat/pre_post_fileformat.cpp`.

- `/source/samples/datatype/pre_post_renderer/pstmixrd.cpp`  
  This sample uses the post-mix audio interface to intercept the final stream after all the inputs are mixed. It also demonstrates how to change the audio data by reducing the stream volume. This plug-in renders data sent by the post-process sample file format plug-in `/source/samples/datatype/pre_post_fileformat/pre_post_fileformat.cpp`.

➤ **To modify the code in any of the sample files:**

1. Back up the original sample files.
2. Modify the Multipurpose Internet Mail Extensions (MIME) type of data rendered or captured by the plug-in.
3. Write your application-specific code, as needed.
4. Build and test your plug-in.

**For More Information:** See “Building a Sample Plug-in” in Volume 1, on page 43.
Sites (Windowing)

Helix consolidates windowing functions in the client core and provides these functions to rendering plug-ins as a platform-independent service. This means that rendering plug-ins do not need to implement window subclassing. Instead, the client core supplies a “site” and the rendering plug-in registers as a “site user.” The rendering plug-in can then send data to the client core without providing platform-specific commands for displaying the data.

The Helix sites feature also supports “Wall of TVs,” which enables a renderer to display data in more than one site simultaneously. In this case the renderer simply sends rendered data to a special default object instantiated by the client. That object then handles displaying the data in the multiple sites without placing any additional burden on the renderer.

Interfaces

To use Helix sites, a rendering plug-in implements the following interfaces:

- **IHXSiteUser.** Header file: hxwin.h.

  A display renderer implements this interface, which the client uses to associate the renderer with a site object and inform the renderer of events.

- **IHXSiteUserSupplier.** Header file: hxwin.h.

  This interface is used with multiple sites to get instances of the objects that use the sites.

- **IHXSiteWatcher.** Header file: hxwin.h.

  The rendering plug-in can implement this interface if it needs to monitor a window’s size and position changes.

The Helix top-level client implements this interface:

- **IHXSiteSupplier.** Header file: hxwin.h.

  Interface implemented by the top-level client for use by the client core.

The Helix client core implements these interfaces:

- **IHXSiteFullScreen.** Header file: hxwin.h.

  Interface implemented by the client core for use by the top-level client.

- **IHXSiteManager.** Header file: hxwin.h.

  Interface implemented by the client core for use by the top-level client.

- **IHXMultiInstanceSiteUserSupplier.** Header file: hxwin.h.

  This interface is implemented by a special object that acts as a site user supplier for any site user that wants support for multiple instances. The site user must work as a windowless site for this to work. The default object also implements IHXSite to enable the site user object to control all the sites through a single interface instance.

The client core implements these interfaces for site objects:

- **IHXSite.** Header file: hxwin.h.

  All site objects implement this interface.
• IHXSiteWindowed. Header file: hxwin.h.

Site objects associated with platform-specific window objects on Microsoft Windows or X-Windows implement this interface.

• IHXSiteWindowless. Header file: hxwin.h.

Site objects that are not associated with platform-specific window objects, hence “windowless,” implement this interface. This is always used on the Macintosh, as well as on other platforms when data is rendered in a Web browser.

Coding a Rendering Plug-in for Sites

During rendering plug-in initialization, Helix queries the plug-in for its display type. If the rendering plug-in returns HX_DISPLAY_WINDOW to indicate that it is a display renderer, the system queries for the plug-in’s support of the site interfaces. The plug-in’s response to these queries determines whether it supports a single site (one instance of rendered data) or multiple sites (one or more instances of rendered data).

Supporting a Single Site

To support a single site, the rendering plug-in implements IHXSiteUser but not IHXSiteUserSupplier. When the client calls IHXSiteUser::NeedsWindowedSites, the renderer responds with TRUE to render in an operating system-specific window (Windows or UNIX) through IHXSite and IHXSiteWindowed (a windowed renderer). It responds with FALSE to render operating system-generically through IHXSite and IHXSiteWindowless (a windowless renderer). The following illustration shows the relationships between objects and main windowing interfaces in a single site that’s not specific to any one operating system, or platform.

Single Cross-Platform Site

Supporting Multiple Sites

RealNetworks recommends that a rendering plug-in supports the multisite feature (Wall of TVs), which enables the plug-in to render data to more than one site. To do this, the plug-in can implement IHXSiteUser and use its own methods of rendering to the various sites. However, Helix provides the MISUS object, a standard method of rendering to multiple sites with no extra overhead. To use this
feature, however, the plug-in must render data in a generic (cross-platform), “windowless” manner. The client’s MISUS object then manages the multiple sites.

To support the multisite feature, the plug-in implements IHXSiteUser. When the system queries for IHXSiteUserSupplier, the plug-in responds by querying for that same interface through IHXMultInstanceSiteUserSupplier. This latter interface is for the MISUS object, the special window management object instantiated by the client. That object will become the “site” where the renderer sends its data. It will also be the site user supplier for the actual sites (one or more) where the display events occur.

The following excerpt from a Helix Client and Server Software Development Kit (SDK) sample file shows how the plug-in responds to the interface query. Here, m_pMISUS has been defined as a pointer to IHXMultInstanceSiteUserSupplier:

```c++
else if (IsEqualIID(riid, IID_IHXSiteUserSupplier))
{
    if (m_pMISUS)
    {
        return m_pMISUS->QueryInterface(IID_IHXSiteUserSupplier, ppvObj);
    }
    else
    {
        *ppvObj = NULL;
        return HXR_UNEXPECTED;
    }
}
```

Because multisite support requires that the renderer function in a “windowless” state, the plug-in returns FALSE when the system calls IHXSiteUser::NeedsWindowedSites. Before it begins to render data, the plug-in must create an instance of the MISUS object and call the object’s IHXMultInstanceSiteUserSupplier::SetSingleSiteUser method to identify itself as the site user. This can happen during plug-in initialization in the IHXRenderer::StartStream method. The following excerpt from a Helix Client and Server Software Development Kit sample file illustrates this:

```c++
m_pCommonClassFactory->CreateInstance(CLSID_IHXMultInstanceSiteUserSupplier,
    (void**)&m_pMISUS);
```

m_pMISUS->SetSingleSiteUser((IUnknown*)(IHXSiteUser*)this);

As shown in the following illustration, the plug-in then sends rendered data to the MISUS object by using IHXSite and IHXSiteWindowless methods. The MISUS object handles displaying the data in the actual sites. The rendering plug-in does not need to have any knowledge of the number of sites being used.
Multiple Sites

Attaching a Site

The client calls the plug-in's IHXSiteUser::AttachSite method to pass it a pointer to the site object that receives the rendered data. When the multisite feature is used, the MISUS object becomes the site object. This is illustrated in the following example:

```c
CExampleRenderer::AttachSite(IHXSite* /*IN*/ pSite)
{
    if (m_pMISUSSite) return HXR_UNEXPECTED;
    m_pMISUSSite = pSite;
    m_pMISUSSite->AddRef();
}
```

Within the IHXSiteUser::AttachSite call, the renderer should use IHXSite::SetSize to set the site size from information received in the stream header.

**Note:** Typically, the system calls IHXRenderer::OnHeader before calling IHXSiteUser::AttachSite. Therefore, the renderer should not call the site object or the MISUS object in IHXRenderer::OnHeader.

Handling Events

To inform the renderer of an event, the client calls the renderer's IHXSiteUser::HandleEvent method, using the HXxEvent structure defined in header file hxwintyp.h. This is illustrated in the following example:

```c
IHXSiteUser::HandleEvent(HXxEvent*)
```

The plug-in can then use the methods of IHXSite, as well as IHXSiteWindowless or IHXSiteWindowed to handle the site display.

The HXxEvent structure contains the following public input members:

```
ULONG32 event;
void* window;
void* param1;
void* param2;
```
The following table describes how to use these members on different platforms.

<table>
<thead>
<tr>
<th>Member</th>
<th>Windows</th>
<th>UNIX (X Windows)</th>
<th>Macintosh</th>
</tr>
</thead>
<tbody>
<tr>
<td>event</td>
<td>Represents a message passed to the window’s WndProc. This message can be a standard message like WM_PAINT or an event defined in Helix.</td>
<td>Corresponds to an event such as Expose, ButtonPress, or ButtonRelease.</td>
<td>Corresponds to an event such as updateEvt, activateEvt, mouseDown, or mouseUp.</td>
</tr>
<tr>
<td>window</td>
<td>Represents an HWND.</td>
<td>Represents a widget object. For example: HWND hWnd = (HWND)pEvent-&gt;window</td>
<td>Represents a HXxWindow object. You can use this object to access the window for drawing and other window-related tasks. For example: HXxWindow* pHXxWindow = (HXxWindow*)PEvent-&gt;window</td>
</tr>
</tbody>
</table>

**Note:** On the Macintosh, the origin, port and clipping region are set before the renderer’s IHXXSiteUser::HandleEvent method is called. This way the renderer can always assume that its “window” is located at 0,0 and that its port is set up correctly.

**Watching a Site**

The rendering plug-in implements IHXXSiteWatcher if it needs to monitor and, if necessary, override a window’s size and position changes. It attaches to the site as a watcher with IHXSite::AttachWatcher. It can later detach itself from the site as a watcher with IHXSite::DetachWatcher.

The client core then notifies the renderer about changes in window size and placement with IHXXSiteWatcher::ChangingPosition or IHXXSiteWatcher::ChangingSize, passing the renderer the old and new position or size parameters, respectively.

**Detaching a Site**

When the presentation ends, the client calls the plug-in’s IHXXSiteUser::DetachSite method to release the site. If the renderer uses the MISUS object, it needs to call that object’s IHXXMultiInstanceSiteUserSupplier::ReleaseSingleSiteUser method and then release the object.

**Modifying the Site Sample Code**

The Helix Client and Server Software Development Kit provides sample code that demonstrates how to write a top-level client with which you can capture or intercept video frames. Possible reasons to do this include applying some visual effects to the frames of video, displaying frames using a nonstandard
windowing API (for example, on UNIX you might want to support GTK instead of Motif), or grabbing “thumbnail” images (that is, the first frame of a clip).

**Note:** Although this sample appears to show how to implement a windowless site, it actually demonstrates how to implement `IHXSiteWindowed`, and is only called windowless because it does not actually display any windows.

The sample code is implemented in the following files:

- `/source/samples/video/capture_vframes/advise_sink.cpp`
  This sample file contains a class that implements the `IHXAdviseSink` interface. Methods in this class will be called by the client core to inform the top-level client when certain events have occurred.

- `/source/samples/video/capture_vframes/authentication_manager.cpp`
  This sample file contains the implementation of a class that handles authentication requests from the client core.

- `/source/samples/video/capture_vframes/context.cpp`
  This sample file contains the implementation of a class that handles the client context.

- `/source/samples/video/capture_vframes/error.cpp`
  This sample file contains the implementation of a class that reports the severity and type of error that occurred.

- `/source/samples/video/capture_vframes/nowindow_site.cpp`
  This sample file contains the implementation of a class that manages the site by handling users, placing the window, managing child sites, and supervising events that occur on the site.

- `/source/samples/video/capture_vframes/site_supplier.cpp`
  This sample file contains the implementation of a class that creates and destroys the site.

- `/source/samples/video/capture_vframes/video_surface.cpp`
  This sample file contains the implementation of a class that manages the data that is drawn on the video surface.

- `/source/samples/video/capture_vframes/fivemlist.cpp`
  This sample file contains a generalized linked list implementation.

- `/source/samples/video/capture_vframes/fivemmap.cpp`
  This sample file contains a "five minute map", a generalized key/value map.

- `/source/samples/video/capture_vframes/main.cpp`
  This sample file builds an example top-level client that demonstrates how to capture or intercept video frames. This sample can be used to develop a custom top-level client or as a testing tool for plug-ins.
CHAPTER 14

CLIENT AUTHENTICATION

To display some content, a Helix client may be required to gather authentication information from the user. This information generally consists of a user name and password.

Client HTTP and RTSP Authentication

When file authentication is required, the top-level client handles the password input from the user. For example, RealPlayer uses a pop-up dialog box to gather the user’s user name and password. If you are implementing Helix client functionality, you need to ensure that your application handles password input.

For More Information: “Server HTTP and RTSP Authentication” in Volume 1, on page 131 describes how Helix Universal Server handles user name and password verification.

Interfaces

A top-level client typically implements the following interfaces:

• IHXAuthenticationManager Header File: hxauth.h.
  This interface is called by the client core which instructs the top-level client to retrieve a user name and password from the client’s user. Typically, a dialog box can be used to gather this information from the user. The response interface is IHXAuthenticationManagerResponse.

• IHXAuthenticationManager2 Header File: hxauth.h.
  This interface also retrieves a user name and password, but can also retrieve other user-defined authentication information. The response interface is also IHXAuthenticationManagerResponse.

Retrieving the User Name and Password

The client core calls the top-level client’s IHXAuthenticationManager::HandleAuthenticationRequest or IHXAuthenticationManager2::HandleAuthenticationRequest2 method to get the top-level client to query the user for a user name and password. In most cases, you can retrieve this information using a dialog box in which the user fills in their user name and password. You can then return the user name and password to the client core through the core’s IHXAuthenticationManagerResponse::AuthenticationRequestDone method.

Generally, the IHXAuthenticationManager::HandleAuthenticationRequest method is sufficient for handling the query for a user name and password. However, the IHXAuthenticationManager2::HandleAuthenticationRequest2 method contains a pointer to an IHXValues interface that can be used to pass additional authentication parameters to the top-level client. For
example, the IHXValues interface could be used to pass parameters for support of proxy authentication, which might include a “pseudonym” header or other information.

If the client is connected to Helix Universal Server, the client core handles the transmission of the user name and password to the server. If the server connection is through HTTP, the client uses the HTTP file system plug-in to send the user name and password back as a WWW-authorization header through IHXPassword::AsString. Note that the file system plug-in prepends “authorization” before sending back the string.

Modifying the Authentication Sample Code

The Helix Client and Server Software Development Kit includes a sample authentication plug-in in /source/samples/server/authbasic/authbasic.cpp. This plug-in implements a simple authentication protocol as described in the HTTP/1.1 standard (rfc2068). Follow the instructions in the source file to test the sample or use it as the basis for your own plug-in.

The TestPlay application is a sample top-level client that does not include a graphical user interface. It is provided as source code in the /source/samples/clientapps/testplay directory. The /source/samples/clientapps/testplay/authentication_manager.cpp sample file contains an implementation of a class that handles authentication requests from the client core.
GLOSSARY

A  Adaptive Stream Management (ASM)
   Rules that describe a streamed data type to Helix, helping it make intelligent decisions about how
to deliver that data type's packets efficiently and robustly.

asynchronous
   The ability to send or receive calls independently and in any order.

Audio Services
   Device-independent, cross-platform features used by audio rendering plug-ins. The plug-in can use
Audio Services without concern for the specifics of the audio hardware.

authentication
   The process by which a plug-in performs password authorization by comparing user name and
password values sent in by a user against those stored in Helix Universal Server's encrypted
password file.

authentication manager
   The object that gathers a user name and password from the user. This is typically the Helix top-
level client.

authentication request object
   The object with access to the user name and password values to be authenticated. It provides these
values to the authenticator object on request.

authentication response object
   The object that receives notice that the authentication succeeded or failed. When the validation
succeeds, the response object carries out the action, such as file access, that was originally
requested.

authenticator object
   An object instantiated by Helix Universal Server that performs authentication by comparing a user
name/password combination against Helix Universal Server's encrypted password file.

B  Backchannel
   A communications channel from a client to Helix Universal Server. To stream data between server
and client, Helix components use IHXPacket objects. A file format plug-in, for example, prepares
packets that Helix Universal Server streams to the client. As well, the file format’s corresponding
rendering plug-in can use the back channel to send packets of information back to the file format
plug-in.

bandwidth
   The amount of data that can be sent over a client’s connection.

Basic authorization
   A form of password authorization that uses unencrypted user names and passwords.
bit rate
The rate at which a presentation is streamed, usually expressed in kilobits per second (Kbps).

broadcasting
Streaming the same data source, whether live or prerecorded, to multiple clients. A broadcast application passes buffers of encoded data from a live source to the Remote Broadcast Library, which then connects to a Helix broadcast plug-in and delivers the stream to Helix Universal Server.

buffering
Receiving and storing data before playing it back. The initial buffering time is called preroll. After preroll, excessive buffering may stall the presentation.

C
client
An application, such as RealPlayer, that receives Helix presentations from Helix Universal Server.

client core
The part of a Helix client that handles data transport with Helix Universal Server and provides Helix features such as Audio Services.

clip
A media file within a presentation. Clips typically have an internal timeline, as with RealAudio and RealVideo.

codec
Coder/decoder. An algorithm for compressing files for use with Helix Universal Server. A variety of codecs exist for different streaming bit-rates and content types. For example, a 32 Kbps music codec is for high-quality music streaming, while a 6.5 Kbps voice codec works for a simple voice file.

COM
Component Object Model, a technology used by Helix for describing interfaces and exporting objects that implement those interfaces.

Common Class Factory
The IHXCommonClassFactory interface used by a component to create objects passed to other Helix components.

container data type
A data type, such as RealMedia File Format (RMFF) or ASF, that can contain other data types. Each container data type is identified by a unique MIME type.

D
data type
A single data type—such as RealAudio, Macromedia Flash, or MIDI—that can be rendered by a client. Each data type is identified by a unique MIME type.

DIGEST authorization
A form of password authorization that uses encrypted user names and passwords. Communication between Helix Universal Server and RealPlayer uses DIGEST authorization.

dry stream
A stream containing too little data for the RealPlayer to write to an audio device.
**E** encoder
An application that converts various types of media files into Helix clips ready to be streamed by a Helix Universal Server. Helix Producer is a standard Helix encoder used to create RealVideo and RealAudio files or live streams.

**expression**
An optional part of a rule that defines a condition the client evaluates. The client subscribes to the rule if the condition fits its current network connection.

**H** header
A chunk of data, delivered from a source to a rendering plug-in when first connecting to a stream, usually used to initialize the stream.

**Helix**
A general system for streaming media clips over a network. It consists of Helix Universal Server, RealPlayer, and production tools.

**Helix client**
An application such as RealPlayer that consists of a top-level client and the client core. The top-level client supplies the user interface, communicating to the core and rendering plug-ins. The core handles data transport and provides features such as Audio Services.

**Helix Universal Server**
Server software developed by RealNetworks and based on the Helix architecture. It provides robust streaming of many data types to Helix clients.

**HTTP playback**
A reasonable method for sending short clips from a Web server using the HTTP protocol. HTTP streaming does not support all the data types of and is not as robust as RTSP streaming, however.

**hypernavigation**
Occurs when a rendering plug-in directs the client to display a URL at a specified time in a stream. When the plug-in issues a hypernavigation request, the default Web browser opens.

**I**

**Interface**
A collection of related functions exposed by an object, and accessed through a unique interface ID.

**IHX**
Interface Helix architecture. Helix, formerly called RealMedia Architecture (RMA), is based on the Component Object Model (COM) jointly developed by Microsoft Corporation and Digital Equipment Corporation.

**M** MIME type
Multipurpose Internet Mail Extension. A data type specification originally used for mail messages and now generalized to identify data types delivered over the Internet.

**MISUS object**
The Multi-Instance Site User Supplier object, a default object instantiated by the client. When a renderer supports the “Wall of TVs” feature, it sends rendered data to this object as if it were the actual site. This object then passes the data to one or more sites.
Network Services
Services that provide cross-platform methods for managing network communications. Any server-side or client-side Helix component can use Network Services to create TCP or UDP connections for reading and writing data. Network Services also provides interfaces that enable components to resolve DNS host names and listen for TCP connections on specified ports.

Object
A unique instance of a data structure defined according to the template provided by its class. Each object has its own values for the variables belonging to its class, and can respond to the methods defined by its class.

Packet
A chunk of data delivered from a source to a rendering plug-in at a particular time during the playback of a stream.

Packet Loss
The percentage of packets dropped during a Helix streaming presentation.

Plug-in
An executable file that extends or enhances the operation of Helix. Helix server-side or client-side plug-ins use the same architecture.

PNA
A proprietary control protocol used by RealServer versions 3.0 through 5.0 to communicate with RealPlayer. RealServer G2 and later and RealPlayer G2 and later use the Real Time Streaming Protocol (RTSP) instead of PNA but support PNA for backward compatibility.

Preroll
The amount of data in milliseconds a rendering plug-in requests before it receives its first time synchronization notification. The actual preroll is always equal to or greater than the value requested by the plug-in.

Presentation
One or more Helix clips that are streamed from a server to a client.

Profile
A clip or group that can be rendered differently depending on the preferences of the client. For example, for the same video one client could choose a Japanese audio stream and another could choose a Spanish audio stream. In the SMIL file, <switch> statements list profile options.

Property
Information that helps Helix Universal Server stream packets associated with the rule intelligently. For example, the mandatory informational property “AverageBandwidth” tells Helix Universal Server the average bandwidth needed to stream packets of this rule effectively.

RealAudio
A RealNetworks data type for streaming highly compressed audio over a network.

RealMedia
A blanket term used to refer to the various RealNetworks data types that it can stream.
RealMedia File Format (RMFF)
A standard tagged file format that uses four-character codes to identify file elements. These codes are 32-bit, represented by a sequence of one to four ASCII alphanumeric characters, padded on the right with space characters. The data type for four-character codes is FOURCC.

RealPix
A Helix clip type (file extension .rp) for streaming still images over a network. It uses a markup language for creating special effects such as fades and zooms.

RealPlayer
Helix client designed to play multimedia presentations streamed by Helix Universal Server or a Web server.

RealText
A Helix clip type (file extension .rt) for streaming text over a network. It uses a markup language for formatting text.

RealVideo
A RealNetworks data type for streaming video.

registry
The Helix Universal Server property registry. This is separate from the Windows registry.

Remote Broadcast Services
A library of methods that ties an encoding application to Helix Universal Server’s broadcast plug-in for live broadcast. The encoding application passes encoded data to the library, which then connects to a Helix broadcast plug-in. The plug-in delivers the stream to Helix Universal Server, which broadcasts it.

RTSP
Real-Time Streaming Protocol. A protocol jointly developed by RealNetworks and Netscape Communications for streaming multimedia over IP networks. RTSP works with established protocols such as RTP and HTTP and can be implemented on client and server across multiple operating system platforms, including the Macintosh, Windows, and UNIX.

rule
A set of properties and, optionally, an expression that helps Helix Universal Server transmit a group of packets intelligently. Each packet is associated with one rule. Rules are part of Adaptive Stream Management (ASM).

rule book
Used in Adaptive Stream Management (ASM), a set of rules that describes a media stream. Each stream has a rule book defined by its file format plug-in. The plug-in uses the stream header to send rules to the client, which subscribes to one or more of them based on properties such as its available bandwidth.

site
An object that receives rendered data for display. The client core supplies a site, and the rendering plug-in registers as a site user. The plug-in can then send data without providing platform-specific commands for data display.
site user
The object, typically the renderer, that “uses” a site by sending it display data.

site user supplier
An object, either the renderer or the “client core, that creates and destroys site user objects.

slider
A control found on RealPlayer that enables the user to scan forward or backward through a presentation.

SMIL
Synchronized Multimedia Integration Language. A markup language for specifying how and when each clip plays. SMIL files use the extension .smil or .smi.

source
A file or live source of data, represented by a single URL, consisting of one or more streams of data.

status code
A value returned by Helix in response to a Helix function call. The header file hxresult.h defines the status codes used in many Helix function calls, as well as system error messages.

stream
v. To send a media clip over a network so that it begins playing back as quickly as possible.

n. A flow of a single type of data, measured in kilobits per second (Kbps). A RealVideo clip’s sound track is one stream, for example.

subscribe
To pick a rule from a rule book and receive packets associated with that rule. The client can subscribe to any number of rules at any time.

SureStream
A technology that makes it possible to switch to a lower-bandwidth encoding in a RealAudio or RealVideo clip to compensate for network congestion.

TestPlay
The core of RealPlayer with a command line interface. Use Testplay to test streaming data types in a client environment other than RealPlayer. For example, use TestPlay to mimic the delivery of streaming data directly within a Web browser.

top-level client
The part of a Helix client that supplies the user interface, communicating to the core and other components such as rendering plug-ins through Helix interfaces.

unsubscribe
To stop receiving packets for a rule.

windowed renderer
A renderer that writes data to an operating system-specific site window through Helix. It can pass operating system-specific commands to the site.
**windowless renderer**

A renderer that uses Helix to write operating system-generic data to a site. The site could be an operating system window or a “windowless” environment such as a browser. The renderer does not have specific knowledge of the display environment and does not pass operating system-specific commands to the site.

**XML**

Extensible Markup Language. XML is simplified SGML (Standard Generalized Markup Language), the international standard for markup languages. XML enables you to create your own markup tags and is designed for use on the World Wide Web.
### INDEX FOR VOLUME 1

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>asynchronous operation, 33</td>
<td>back channel packets, 166</td>
<td>CloseEngine, 154</td>
<td>Data conversion, 4</td>
<td>Encryption, 4</td>
<td>file handling, 49</td>
<td>Helix object, 39</td>
</tr>
<tr>
<td>audio</td>
<td>Bandwidth variable, 144</td>
<td>configuration</td>
<td>Data reversion, 4</td>
<td>error severity, 86</td>
<td>file object, 67</td>
<td>Helix Universal Server, 24</td>
</tr>
<tr>
<td>instant sounds, 177</td>
<td>buffers, 31</td>
<td>Web server, 15</td>
<td>Decryption, 4</td>
<td>ErrorLogPath, 85</td>
<td>creating, 53</td>
<td>HTTP, 29</td>
</tr>
<tr>
<td>post-processed data, 178</td>
<td>Conversion of data streams, 4</td>
<td>Conversion of data streams, 4</td>
<td>DNS host name, resolving, 83</td>
<td>expressions, rule book, 148</td>
<td>HTTP streaming, 46</td>
<td>HTTP streaming, 46</td>
</tr>
<tr>
<td>rendering, 174</td>
<td></td>
<td>CreateEngine, 154</td>
<td>dry streams, 181</td>
<td>i/o callback on UNIX, 84</td>
<td></td>
<td></td>
</tr>
<tr>
<td>synchronizing playback, 181</td>
<td></td>
<td></td>
<td></td>
<td>IHXASMSouce, 141</td>
<td></td>
<td></td>
</tr>
<tr>
<td>audio streams, mixed, 174</td>
<td></td>
<td></td>
<td></td>
<td>IHXASMSouce, 58</td>
<td></td>
<td></td>
</tr>
<tr>
<td>authentication</td>
<td></td>
<td></td>
<td></td>
<td>IHXASMSream, 141</td>
<td></td>
<td></td>
</tr>
<tr>
<td>file system plug-in, 132</td>
<td></td>
<td></td>
<td></td>
<td>IHXASMSreamSink, 142</td>
<td></td>
<td></td>
</tr>
<tr>
<td>monitor plug-in, 133</td>
<td></td>
<td></td>
<td></td>
<td>IHXAsyncIOSelection, 81</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AverageBandwidth property, 145</td>
<td></td>
<td></td>
<td></td>
<td>IHXAudioHook, 172</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AverageBandwidthStd property, 145</td>
<td></td>
<td></td>
<td></td>
<td>IHXAudioPlayer, 173</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HXAudioData, 177</td>
<td></td>
<td></td>
<td></td>
<td>IHXAudioPlayerResponse, 173</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HXCreateInstance, 36</td>
<td></td>
<td></td>
<td></td>
<td>IHXAudioStream, 173</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HXxEvent, 186</td>
<td></td>
<td></td>
<td></td>
<td>IHXAudioStreamInfoResponse, 173</td>
<td></td>
<td></td>
</tr>
<tr>
<td>hyперnavigation, 166</td>
<td></td>
<td></td>
<td></td>
<td>IHXAuthenticationManager, 132, 153</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td>IHXAuthenticationManagerResponse, 132</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IHXAsyncIOSelection, 81</td>
<td></td>
<td></td>
<td></td>
<td>IHXAuthenticator, 131</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IHXAuthenticationManager, 132</td>
<td></td>
<td></td>
<td></td>
<td>IHXAuthenticationManagerRequest, 131</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IHXBackChannel, 58, 142</td>
<td></td>
<td></td>
<td></td>
<td>IHXAuthenticationResponse, 131</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IHXBroadcastFormatObject, 92</td>
<td></td>
<td></td>
<td></td>
<td>IHXBackChannel, 58, 142</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IHXBroadcastMapper, 92</td>
<td></td>
<td></td>
<td></td>
<td>IHXClientAdviseSink, 153, 157</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IHXBuffer, 31</td>
<td></td>
<td></td>
<td></td>
<td>IHXClientEngine, 154</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IHXCallback, 92, 124</td>
<td></td>
<td></td>
<td></td>
<td>IHXCommonClassFactory, 39</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IHXClientAdviseSink, 153, 157</td>
<td></td>
<td></td>
<td></td>
<td>IHXDirHandlerResponse, 58</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IHXClientEngine, 154</td>
<td></td>
<td></td>
<td></td>
<td>IHXDryNotification, 173</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IHXErrorMessages, 85</td>
<td></td>
<td></td>
<td></td>
<td>IHXErrorSink, 85, 153</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IHXErrorSink, 85, 153</td>
<td></td>
<td></td>
<td></td>
<td>IHXErrorSinkControl, 85</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IHXFileAuthentication, 132</td>
<td></td>
<td></td>
<td></td>
<td>IHXFileAuthenticator, 132</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IHXFileExists, 51, 92</td>
<td></td>
<td></td>
<td></td>
<td>IHXFileExists, 51, 92</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IHXFileNotFoundException, 57</td>
<td></td>
<td></td>
<td></td>
<td>IHXFileFormatObject, 57</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IHXFileMimeMapper, 51</td>
<td></td>
<td></td>
<td></td>
<td>IHXFileMimeMapper, 51</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IHXFileObject, 51</td>
<td></td>
<td></td>
<td></td>
<td>IHXFileObject, 51</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IHXFileResponse, 58</td>
<td></td>
<td></td>
<td></td>
<td>IHXFileResponse, 58</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IHXFileStat, 51</td>
<td></td>
<td></td>
<td></td>
<td>IHXFileStat, 51</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IHXFileStatResponse, 58</td>
<td></td>
<td></td>
<td></td>
<td>IHXFileStatResponse, 58</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
IHXFileSystemManagerResponse, 58
IHXFileSystemObject, 51, 92
IHXGetFileFromSamePool, 51
IHXGroup, 158
IHXGroupManager, 158
IHXGroupSink, 158
IHLListenResponse, 81
IHLiveFileFormatInfo, 103
IHLiveRealPix, 111
IHLiveRealPixResponse, 111
IHLiveText, 106
IHMMultiInstanceSiteUserSupplier, 183
IHNNetworkServices, 81
IHPacket, 33
IHPacketFormat, 58
IHPacketHook, 153
IHPacketHookHelper, 162
IHPacketTimeOffsetHandler, 102
IHPacketTimeOffsetHandlerResponse, 103
IHPassword, 132
IHPendingStatus, 51, 58
IHPlayer, 155
IHPlayer2, 155
IHPlayerConnectionAdviseSink, 136
IHPlayerConnectionResponse, 136
IHPlayerController, 136
IHPlugin, 37
IHPropWatch, 125
IHPropWatchResponse, 124, 154
IHRegistry, 124
IHRenderer, 162
IHRequestHandler, 51, 92
IHResolver, 83
IHResolverResponse, 81
IHXScheduler, 125
IHXSite, 183
IHXSite2, 162, 168
IHXSiteFullScreen, 183
IHXSiteManager, 183
IHXSiteSupplier, 153, 183
IHXSiteUser, 162, 169, 183
IHXSiteUserSupplier, 162, 183
IHXSiteWatcher, 183
IHXSiteWindowed, 184
IHXSiteWindowless, 184
IHXStream, 167
IHXStreamSource, 168
IHXTCPResponse, 81
IHXTCPSocket, 82

IHXUDPResponse, 81
IHXUDPsocket, 83
IHXValues, 32
IHXVideoSurface, 162, 169
IHXVolume, 173, 176
IHXVolumeAdviseSink, 153, 173, 176
indexed lists, 32

L
listen socket, creating, 82
Local files, playing, 30

M
Marker property, 146
midstream playback, 177
MIME type, 163
MIME types, 15
mixed audio streams, 174
multiple packet formats, 63
multiple relative files, 69
multiple sites, 184
Multiple streams, 29

P
PacketLoss property, 145
packets, 33
passwords, 158
pause, 166
pending status, 66
player objects, 167
Plug-in, 26
plug-ins
  compiling, 43
  creating, 36
  designing, 35
  enumerating, 42
testing, 44
PN_AUTH_BASIC, 134
PN_AUTH_DIGEST, 134
Priority property, 145
properties, custom, 147
property registry, 119
Protocol support, 25
pulse code modulation (PCM), 173

R
ramgen, 28
RealOnePlayer, 16, 45
RealPix
  initializing, 111
  sending images, 113
  stopping, 115
RealText
<table>
<thead>
<tr>
<th>Index for Volume 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>communicating with, 108</td>
</tr>
<tr>
<td>defining the window, 107</td>
</tr>
<tr>
<td>initializing, 106</td>
</tr>
<tr>
<td>sending text, 107</td>
</tr>
<tr>
<td>reference counting, 30</td>
</tr>
<tr>
<td>registry, examples, 128</td>
</tr>
<tr>
<td>registry, Helix Universal Server property, 119</td>
</tr>
<tr>
<td>relative file access, 54</td>
</tr>
<tr>
<td>relative file objects, 69</td>
</tr>
<tr>
<td>rendering plug-in</td>
</tr>
<tr>
<td>coding for sites, 184</td>
</tr>
<tr>
<td>rendering streams, 164</td>
</tr>
<tr>
<td>response headers, 66</td>
</tr>
<tr>
<td>Reversion of data streams, 4</td>
</tr>
<tr>
<td>RTP stream header, 64</td>
</tr>
<tr>
<td>rule book, 143</td>
</tr>
<tr>
<td>rule book, creating, 147</td>
</tr>
<tr>
<td>sample code</td>
</tr>
<tr>
<td>allowance plug-in, 138</td>
</tr>
<tr>
<td>audio rendering, 181</td>
</tr>
<tr>
<td>authentication, 190</td>
</tr>
<tr>
<td>broadcast plug-in, 96</td>
</tr>
<tr>
<td>file format plug-in, 70</td>
</tr>
<tr>
<td>file system plug-in, 56</td>
</tr>
<tr>
<td>monitor plug-in, 127</td>
</tr>
<tr>
<td>RealPix broadcast, 115</td>
</tr>
<tr>
<td>RealText broadcast, 108</td>
</tr>
<tr>
<td>rendering plug-in, 79, 170</td>
</tr>
<tr>
<td>sites, 187</td>
</tr>
<tr>
<td>top-level client, 159</td>
</tr>
<tr>
<td>scheduler, 127</td>
</tr>
<tr>
<td>seek, 63, 165</td>
</tr>
<tr>
<td>select(), 154</td>
</tr>
<tr>
<td>single sites, 184</td>
</tr>
<tr>
<td>sites</td>
</tr>
<tr>
<td>attaching, 186</td>
</tr>
<tr>
<td>detaching, 187</td>
</tr>
<tr>
<td>handling events, 186</td>
</tr>
<tr>
<td>watching, 187</td>
</tr>
<tr>
<td>SMIL, 26</td>
</tr>
<tr>
<td>stream headers</td>
</tr>
<tr>
<td>creating, 95</td>
</tr>
<tr>
<td>stream headers, creating, 61</td>
</tr>
<tr>
<td>stream objects, 167</td>
</tr>
<tr>
<td>stream packets, 33</td>
</tr>
<tr>
<td>creating, 62, 96</td>
</tr>
<tr>
<td>stream source object, 168</td>
</tr>
<tr>
<td>streaming, 29</td>
</tr>
<tr>
<td>Streaming files, 27</td>
</tr>
<tr>
<td>synchronization, 169</td>
</tr>
<tr>
<td>T</td>
</tr>
<tr>
<td>TCP socket, using, 82</td>
</tr>
<tr>
<td>Technical support, 12</td>
</tr>
<tr>
<td>terminating a presentation, 167</td>
</tr>
<tr>
<td>TestPlay, 16, 45, 159</td>
</tr>
<tr>
<td>TimeStampDelivery property, 146</td>
</tr>
<tr>
<td>top-level client</td>
</tr>
<tr>
<td>adding site support, 158</td>
</tr>
<tr>
<td>U</td>
</tr>
<tr>
<td>UDP socket, using, 83</td>
</tr>
<tr>
<td>V</td>
</tr>
<tr>
<td>volume</td>
</tr>
<tr>
<td>controlling, 176</td>
</tr>
<tr>
<td>W</td>
</tr>
<tr>
<td>WaitForSwitchOff property, 146</td>
</tr>
<tr>
<td>watching properties, 126</td>
</tr>
<tr>
<td>Web server configuration, 15</td>
</tr>
</tbody>
</table>