HELIX™ DNA PRODUCER SDK DEVELOPER'S GUIDE

Release 11.0

Revision Date: 15 February 2005

RealNetworks, Inc.
2601 Elliott Avenue, Suite 1000
Seattle, WA 98121
U.S.A.

http://www.real.com
http://www.realnetworks.com

Printed in the United States of America.

CONTENTS

INTRODUCTION

Welcome to the Helix™ DNA Producer 11.0 Software Development Kit (SDK), which RealNetworks has created for developers working with media production. This developer's guide will help you use the SDK to configure and control encoding sessions, as well as create custom media plug-ins used by the Helix DNA Producer media engine.

> **For More Information:** Be sure to read the SDK license agreement in full. The license agreement is located in the license.txt file provided with the Helix DNA Producer SDK.

Using the SDK

Because Helix DNA Producer is based on the Component Object Model (COM) binary standard, you can develop producer 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 producer components. Note, however, that Helix DNA Producer diverges from the COM standard to simplify cross-platform development.

Header Files

The Helix DNA Producer header files define the Helix DNA Producer interfaces. When you are ready to begin developing producer components, refer to the header files along with this documentation. The header files 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 DNA Producer components. Using the sample code requires a

knowledge of C or C++. All sample code supplied with this SDK is platform-independent.

> **Note:** RealNetworks recommends using specific compilers for compiling code based on the sample files. For more information, see "Platforms" on page 17.

SDK Notes

Before you begin using the Helix DNA Producer SDK, you should be aware of the following:

- The Helix DNA Producer SDK APIs documented in this guide are supported only in versions 9.0 and later of the producer product line, and are not compatible with earlier versions of RealProducer.

How this Book Is Organized

Chapter 1: Helix DNA Producer SDK
This chapter contains installation and system requirements, general information about using the SDK, and basic concepts of the Helix DNA Producer SDK.

Chapter 2: SDK Organization
This chapter contains organizational information and a list of the files supplied with the Helix DNA Producer SDK.

Chapter 3: Encoding Overview
This chapter describes how to use the Helix DNA Producer SDK interfaces to configure and control encoding sessions.

Chapter 4: Logging System
This chapter discusses how to implement a system that produces log files for your application.

Chapter 5: Helix DNA Producer Plug-In API
This chapter provides an in-depth explanation of the methods and policies that go into writing low-level producer plug-ins.

Chapter 6: RealMedia Edit API
This chapter describes how to edit `.rm` files using the RealMedia Edit API.

Appendix A: Interface List

This appendix contains reference material that describes all of the Helix DNA Producer interfaces and methods supplied with the Helix DNA Producer SDK.

Appendix B: RealMedia Edit Interface List

This appendix contains reference material that describes all of the RealMedia Edit interfaces and methods supplied with the Helix DNA Producer SDK.

Appendix C: Function List

This appendix contains reference material that describes all of the functions supplied with the Helix DNA Producer SDK.

Additional Resources

In addition to this guide, you may need the following RealNetworks documentation resources:

- *RealProducer User's Guide*

  This user's guide gives you the step-by-step instructions for running Helix DNA Producer, which turns audio and video files into streaming media clips. An online version of this guide is available through the Helix DNA Producer **Help** menu. You can obtain a copy of the Helix DNA Producer User's Guide at **http://service.real.com/help/library/encoders.html**.

- *Helix Client and Server Software Developer's Guide*

  This SDK contains the architecture upon which the latest RealPlayer and Helix Universal Server are built. Developers wanting to build client or server applications for Helix should use this SDK. It contains documentation, samples, and header files that demonstrate every interface in the system, including documentation of the "IHX" interfaces for the client and server. You can obtain a copy of the *Helix Client and Server Software Developer's Guide* at **https://helix-client.helixcommunity.org** in the documentation section under Software Development Kits.

Conventions Used in this Book

The following table explains the typographical conventions used in this book.

Notational Conventions

Convention	Meaning
emphasis	Bold text is used for in-line headings, user-interface elements, URLs, and e-mail addresses.
terminology	Italic text is used for technical terms being introduced in a given manual or other document, and to lend emphasis to generic English words or phrases.
`syntax`	This font is used for fragments or complete lines of programming syntax (code or markup)—whether within text or set off—and for command-line instructions.
`syntax emphasis`	Bold syntax character formatting is used for program names and to emphasize specific syntax elements.
`variables`	Italic syntax character formatting denotes variables within fragments or complete lines of syntax.
`[options]`	Square brackets indicate values you may or may not need to use. As a rule, when you use these optional values, you do not include the brackets themselves.
`choice 1|choice 2`	Vertical lines, or "pipes," separate values you can choose between.
…	Ellipses indicate nonessential information omitted from code or markup examples.
" "	Curly ("smart") quotation marks are used for direct quotes, to call out words or phrases that are being used to mean something other than what they mean in everyday English, and to enclose chapter titles and section headings in cross-references.

Technical Support

To reach RealNetworks' Technical Support, please fill out the form at:

- **http://customerrelations.real.com/scripts/rnforms/contact_tech_service.asp**

The information you provide in this form will help Technical Support personnel respond promptly. For general information about RealNetworks' Technical Support, visit this Web page:

- **http://service.real.com/help/call.html**

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/**

To learn more about Helix DNA Producer, visit this Web page:

- **https//helix-producer.helixcommunity.org/**

RealForum

RealNetworks also encourages you to join RealForum, an e-mail discussion group about RealNetworks products where developers and content producers post tips and ask for assistance. RealNetworks employees monitor the postings and offer suggestions as appropriate. You can sign up for RealForum by connecting to **http://realforum.real.com/** and clicking on **New user**.

QUICK START

The Helix DNA Producer SDK provides support for a number of different user scenarios. To accomplish this, the SDK is divided into distinct areas of functionality and interfaces. The following information is provided to help you identify which areas of the SDK documentation to focus on.

Scenario 1

The information in this section identifies the parts of the guide you should read first if you:

- Have an application that manipulates raw audio, video, and event data and you want to export to a RealMedia file (.rm).
- Used RealProducer SDK 8.5 in the past and want to continue working with the SDK in the same way.

Sections

The following sections of the guide contain relevant information:

- New Features in Helix DNA Producer
- Chapter 3: Encoding Overview
- Appendix A: Interface List

Interfaces

Primary interfaces you will use include:

- `IHXTEncodingJob`
- `IHXTMediaInputPin`
- `IHXTAudioPinFormat`
- `IHXTVideoPinFormat`

- IHXTMediaSample
- IHXTEventSample
- IHXTAudienceEnumerator
- IHXTMediaProfile
- IHXTOutputProfile
- IHXTDestination

Sample Application

The following sample demonstrates a simple media sink application that can be used as a starting point for your own application:

- \producersdk\samples\mediasinkencoder

Scenario 2

The information in this section identifies the parts of the guide you should read first if you:

- Want to automate your encoding production process or provide remote control and configuration using a custom-built application.

Sections

The following sections of the guide contain relevant information:

- Helix DNA Producer Encoding System section in Chapter 1: Helix DNA Producer SDK.
- Chapter 3: Encoding Overview

Interfaces

Primary interfaces you will use:

- IHXTEncodingJob
- IHXTInput
- IHXTAudience
- IHXTAudienceEnumerator
- IHXTMediaProfile
- IHXTOutputProfile
- IHXTDestination

- IHXTPropertyBag

Sample Applications

The following samples demonstrate applications that can be used as a starting point for your own application:

- \producersdk\samples\encoder
- \producersdk\samples\advencoder

Scenario 3

The information in this section identifies the parts of the guide you should read first if:

- You want to write a custom application that reads or edits existing RealMedia (.rm) files.
- Your custom application needs to adds image map or RealMedia event streams to existing RealMedia files.

Sections

The following sections of the guide contain relevant information:

- Chapter 6: RealMedia Edit API
- Appendix B: RealMedia Edit Interface List

Interfaces

Primary interfaces you will use:

- IHXRMEdit
- IHXRMEdit2
- IHXRMFFDump
- IHXPacket
- IHXBuffer
- IHXValues
- IHXRMFileSink
- IHXRMEvents

Sample Applications

The following samples demonstrate applications that can be used as a starting point for your own application:

- \producersdk\samples\rmeditor
- \producersdk\samples\rmevents

Scenario 4

The information in this section identifies the parts of the guide you should read first if you:

- Want to extend the functionality of the encoder by providing custom audio or video filters, support new file formats or capture devices, provide custom encryption, or broadcast over proprietary networks.

Sections

The following sections of the guide contain relevant information:

- Chapter 5: Helix DNA Producer Plug-In API
- Plug-in Categories
- Creating Custom Media Plug-ins

Interfaces

Primary interfaces you will use:

- IHXTConfigurationAgent
- IHXTConnectionAgent
- IHXTFilter
- IHXTInputFilter
- IHXTTransformFilter
- IHXTOutputFilter

Sample Applications

The following samples demonstrate applications that can be used as a starting point for your own application:

- \producersdk\samples\inputplugin
- \producersdk\samples\prefilterplugin

NEW FEATURES IN HELIX DNA PRODUCER

Many new features have been added to the Helix DNA Producer SDK to build a robust, reliable, and fault-tolerant encoding tool that creates high-quality media and provides increased encoding efficiency and flexibility. If you are familiar with previous versions of the producer SDK, this chapter gives you a quick look at changes in the latest release of the Helix DNA Producer SDK.

What's New in Helix DNA Producer 11.0 SDK

The following list describes the new features included in the Helix DNA Producer 11.0 SDK Developer's Guide:

- The Helix DNA Producer 11.0 SDK comes with the addition of a low latency mode. Helix DNA Producer latency is defined as the time between when the producer receives an audio sample or video frame and the time that the packet containing that sample or frame is passed to the network layer of the operating system. Note that latency is based on actual elapsed time, not the timestamps in a media packet. See " Media Profile Properties" on page 55 for more information on low latency mode.
- Packet sizes can be configured to specific sizes. Audio packetization is the process of grouping audio frames into physical entities known as blocks (packets) and grouping blocks into logical entities known as superblocks. Video packetization is the process of either grouping small video B and P-frames into single packets, or splitting larger video key frames into many packets. See " Media Profile Properties" on page 55 for more information on configurable packet sizes.
- This version of the SDK drops support for some legacy broadcast support. This includes the removal of several libraries, the removal of the -sg option from the command line, and the removal of the legacy broadcast option from the drop down list and related advanced options from the

dialog in the Helix DNA Producer UI. The following libraries have been removed:

- encn3260.dll
- auth3260.dll
- basc3260.dll
- rn5a3260.dll
- sdpp3260.dll

- This version of the SDK provides support for the Microsoft Visual C++ version 7 compiler on Windows.
- This version of Helix DNA Producer SDK supports IPv6 broadcasting. With this support, Helix DNA Producer can broadcast to Helix Universal Server 11.0 and onwards using IPv6 addresses. Helix DNA Producer continues to support IPv4 for backwards compatibility.
- Applications using this version of the Helix DNA Producer SDK can preview audio samples in audio level format. In previous versions of the Helix DNA Producer SDK, application would receive raw audio sample PCM data. If you needed to display audio data in the form of audio level and volume bars, your application was required to do all the processing. With this version of the SDK, it is possible to get audio data in audio level format, which makes drawing audio levels much simpler. For more infomation, see the IHXTAudioLevelChannel and IHXTAudioLevelChannels interfaces in "Appendix A: Interface List" beginning on page 161. You can also see how these interfaces are used by examining the advanced encoder sample in producersdk/samples/advencoder.

What's New in Helix DNA Producer 10.0 SDK

The following list describes the new features included in the Helix DNA Producer 10.0 SDK Developer's Guide:

- The Helix DNA Producer 10.0 SDK supports parallel inputs, which provide a means of encoding more than one input at a time. This is especially useful if your audio and video sources are separate and you have to encode them into a single file. For more information, see "Creating Parallel Inputs" on page 38.
- In previous versions of Helix DNA Producer SDK, only one output profile could be created at a time. The Helix DNA Producer 10.0 SDK supports

multiple output profiles. For more information, see "Multiple Output Profiles" on page 43.

- Two new prefilters were added to the Helix DNA Producer 10.0 SDK: the audio delay compensation prefilter and the video resize prefilter. For more information, see "Prefilters" on page 39.
- RealMedia events can now be added to an on-demand RealMedia file or inserted into a live broadcast stream during the encoding process. For more information, see "Encoding RealMedia Events" on page 63.
- A new complexity mode property was added to set the complexity of audio and video encoding. For more information on setting audio complexity, see the Audio Stream Properties table under "Streams" on page 57. For more information on setting video complexity, see the Video Stream Properties table under "Streams" on page 57.
- Audio and video preview has been updated to include new features. You can now adjust the video width, height, and color format of a video preview, and the sample rate, sample size, and number of channels of an audio preview. In addition, you can also start and stop the previews during an encoding session. For more information, see "Audio and Video Preview" on page 71.
- Automatic selection of input (reader) file plug-in type and destination (writer) file plug-ins are now automatically selected by Helix DNA Producer. For more information on automatic selection of input file plug-ins, see "Audio/Video Files" on page 34. For more information on automatic selection of destination file plug-ins, see "File Writers" on page 49.
- Helix DNA Producer's method of automatically renaming RealMedia files has changed. For more information, see "Automatic File Renaming" on page 49.
- Helix DNA Producer can now report its system resource availability to codecs and plug-ins. With this information, your codec or plug-in can modify its behavior to reduce or increase its use of system resources. For more information, see "Load Management" on page 88.
- The Helix DNA Producer SDK now provides a mechanism to enable applications to easily support new codecs without significant changes to the application. Automatic codec selection gives your application the ability to distinguish between codecs by using additional plug-in

properties. For more information, see "Automatic Codec Selection" on page 82.

- The following interface information was added to "Appendix A: Interface List" beginning on page 161:
  - `IHXTAudienceEnumerator2`
  - `IHXTCodecUpdater`
  - `IHXTDestinationEnumerator`
  - `IHXTInput2`
  - `IHXTLoadAdjustment`
  - `IHXTOutputProfile2`
  - `IHXTPreviewSinkControl3`
  - `IHXTPropertyUtility`

What's New in Helix DNA Producer 9.1 SDK

The following list describes the main differences between the Helix DNA Producer 9.0 SDK and the Helix DNA Producer 9.1 SDK:

- Added support for Mac OS X. SDK users can create Mach-O applications with the Helix DNA Producer SDK. In addition, CFM support for Mac OS X has been dropped—the Helix DNA Producer SDK does not support CFM applications. Information related to Mac OS X can be found throughout the text of this guide. For information on using Mac OS X to compile the samples included in this SDK, see "Using the Samples" on page 27.
- To make all the APIs in the Helix DNA Platform consistent, header files and interface names used in Helix DNA Producer were changed to IHXT (IHX in some cases). For more information on updating existing applications from the Helix Producer SDK to the Helix DNA Producer SDK, see "Helix DNA Producer Interface Name Change" on page 18.

What's New in Helix DNA Producer 9.0 SDK

The following list describes the main differences between the RealProducer 8.5 SDK and the Helix DNA Producer 9.0 SDK:

- **Main Helix DNA Producer SDK object.** The primary RealProducer 8.5 SDK configuration and control interface was the `IRMABuildEngine`

interface. The Helix DNA Producer 9.0 SDK equivalent is IRTAEncodingJob. The IRMABuildClassFactory interface has been replaced with IRTAClassFactory. The semantics for the two class factories are similar. The CMediaSinkEncoderApp::CreateJob and CMediaSinkEncoderApp::SetupJob methods in the mediasinkencoder sample application show how to instantiate and configure an IRTAEncodingJob.

- **Configuring encoding sessions.** The RealProducer 8.5 SDK encoding configuration was set through dedicated PNCOM interfaces and methods such as IRMAVideoFilters::SetNoiseFilter or IRMABuildEngine::SetDoMultiRateEncoding. The Helix DNA Producer 9.0 SDK uses the IRTAPropertyBag interface and its set property methods, such as IRTAPropertyBag::SetBool or IRTAPropertyBag::SetUint with property name/value pairs.
- **Inputs**. The RealProducer 8.5 SDK did not have any support for inputs. The SDK application was required to pass in uncompressed audio/video samples. The Helix DNA Producer 9.0 SDK now has broad support for inputs—it ships with file reader and capture plug-ins. You can use the Helix DNA Producer 9.0 SDK without having to write any code that reads files or deals with the operating system's capture subsystem. SDK applications can continue to provide custom input to the SDK by either implementing an input plug-in or by passing media samples to the media sink input (a pre-existing input plug-in of type kValuePluginTypeInputMediaSink). The CMediaSinkEncoderApp::SetupInput method in the mediasinkencoder sample application shows how to use the media sink input.
- **Prefilters.** The majority of RealProducer 8.5 SDK prefilter settings were specified using the IRMAVideoFilters interface. Prefilters are now full-blown plug-ins, for example, kValuePluginNamePrefilterCropping or kValuePluginTypePrefilterDeinterlace. The CMediaSinkEncoderApp::SetupPrefilters method in the mediasinkencoder sample application shows an example of how to configure prefilters. Several new prefilters, such as the audio gain prefilter and the black level prefilter, have been added in Helix DNA Producer 9.0.
- **Destinations.** The RealProducer 8.5 SDK allowed an encoding session to have a single file output and a single broadcast output. The Helix DNA Producer 9.0 SDK supports outputs as plug-ins. There is no limit to the number of outputs. Outputs in the Helix DNA Producer 9.0 SDK are independently viable destinations, that is, while encoding, a failure in one

destination will not cause the others to fail—the RealProducer 8.5 SDK would stop an encoding session if any output failed. The Helix DNA Producer 9.0 SDK only stops an encode if all outputs have failed or have finished. There is a new broadcast plug-in that utilizes the Real broadcast protocol. See the `CMediaSinkEncoderApp::SetupDestination` method in the `mediasinkencoder` sample application.

- **Audiences/Streams.** The RealProducer 8.5 SDK allowed audiences to be selected through `IRMABasicTargetSettings`. Audiences are now set on the `IRTAMediaProfile`. The RealProducer 8.5 SDK allowed enumeration of pre-built audiences through the `IRMATargetAudienceManager` interface. This has been replaced with `IRTAAudienceEnumerator`. One major change from the RealProcuer 8.5 SDK is that SureStream substreams are no longer automatically generated—if you picked the 28k audience and set up a SureStream encode, the RealProducer 8.5 SDK would automatically generate a 15k and 12k substream. With the Helix DNA Producer 9.0 SDK, it is now necessary to manually select the "12k Substream for 28k Dial-up" and "16k Substream for 28k Dial-up". See the `CMediaSinkEncoderApp::SetupAudiences` method in the `mediasinkencoder` sample application.
- **Log system.** The Helix DNA Producer 9.0 SDK includes a logging system that allows SDK issues to be much more readily diagnosed. See the `CMediaSinkEncoderApp::InitializeLogSystem` method in the `mediasinkencoder` sample application for an example.
- **Serialization.** The Helix DNA Producer 9.0 SDK allows `IRTAEncodingJob` and `IRTAAudience` to be serialized or deserialized. This may be useful for saving the application state and restoring it at a later time. See the `CMediaSinkEncoderApp::SerializeJob` method in the `mediasinkencoder` sample application.

CHAPTER

1

HELIX DNA PRODUCER SDK

Welcome to the Helix DNA Producer software development kit (SDK), which RealNetworks has created for developers working with digital media production for broadcast streaming and download. This developer's guide will help you use the SDK to create and modify various components and interfaces of Helix DNA Producer.

Helix DNA Producer SDK consists of three major areas:

- Helix DNA Producer Encoding API
- Helix DNA Producer Plug-in API
- RealMedia Edit API

Platforms

The Helix DNA Producer SDK 11.0 is supported on the following platforms:

- Windows 98 Second Edition, Windows ME, Windows NT 4.0, Windows 2000,and Windows XP.
- Mac OS X v10.2.
- Linux 2.2 kernel, libc6.

To compile the sample applications on Windows, Microsoft Visual C++ 6.0 SP3 or later is required. To compile the sample applications on a MacIntosh, Project Builder 2.1 and gcc 3.1 are required. To compile the sample applications on Linux, the gcc compiler, version 2.95.2 or later, is required (gcc version 3.x is currently not supported).

Installation

Install the Helix DNA Producer SDK by copying the appropriate compressed file from the Helix community site on to your system. Then uncompress the file using an appropriate application for your operating system.

Helix DNA Producer Interface Name Change

To make all the APIs in the Helix DNA platform consistent, header files and interface names used in Helix Producer SDK from RealNetworks were changed to IHXT (IHX in some cases). To help in updating existing applications from the Helix Producer SDK from RealNetworks to the Helix DNA Producer 11.0 SDK, a Python script is provided that works in conjunction with the Helix DNA Producer SDK name change dictionary file.

> **Note:** Because support for Mac OS X was not implemented in the Helix DNA Producer SDK until after the header file and interface name change, you do not need to run this script on Mac OS X.

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**. Python 2.2 is recommended to run the scripts. The Python script will search your source code for Helix Producer SDK from RealNetworks interface names and update them to Helix DNA Producer SDK interface names.

To run the name change script, you need:

- the producersdk interface name change dictionary file (`producersdkdict.txt`)
- the Python script (`namereplace.py`)

You can obtain both of these from the Helix community web site in **https://producersdk.helixcommunity.org/source/browse/producersdk/installer/resource/convert/files/**. These files need to be downloaded to your hard drive.

The name change can be executed on a Windows-based machine with the following syntax:

```
python namereplace.py --dict=producersdkdict.txt --word-chars=[\w.] --do-edits
mysdkapp *.cpp *.h
```

The name change can be executed on a UNIX-based machine with the following syntax:

```
python namereplace.py --dict=producersdkdict.txt --word-chars='[\w.]' --do-edits
mysdkapp '*.cpp' '*.h'
```

> **Note:** If Python is not in your path, you will need to type the full path to the location of the Python executable.

The `namereplace.py` file is the Python script that changes the header files and interface names. If the name change script is not in the working directory, then specify the entire path.

The name of the dictionary file is declared under the `--dict` argument. For Helix DNA Producer, this file is `producersdkdict.txt`.

The target directory is the directory containing the source code that is to be updated with the new interface names. In the examples shown here, `mysdkapp` is the target directory. Files in this directory matching the file extensions specified in this command line that contain references to the Helix Producer 9.0 and 9.0.1 SDK header and interfaces will be updated to reference the new Helix DNA Producer 11.0 SDK header and interfaces.

The file extension indicates the files for which search and replace will be performed. Any files matching the file extension pattern will be searched for the entries in the `producersdk` dictionary and be modified if matches are found. Wildcards should be included. This parameter must be provided at least once, but can be included multiple times for each file type you want to search and replace. Generally you are going to want to specify "`*.cpp`" and "`*.h`" file extensions. On UNIX machines, make sure to quote these file extensions to avoid having the shell interpret the wildcards before they get to the Python script.

> **Warning!** The "`--word-chars=[\w.]`" argument defines a regular expression that identifies the start of words. There should be no reason to change this for C++ source code, but the parameter must be included in the command line. It is advised that you just leave this argument as-is. If you are going to modify it, you do so at your own risk. Unless you know what you are doing, regular expressions can be dangerous.

Directory Structure

Previous versions of the Helix DNA Producer SDK required that you install all the SDK DLLs in a fixed directory structure. However, now you can copy all of the directories and files found in `\producersdk\bin` to a new directory you specifically create for your own application. When you change directories, you must specify the location of the Helix DNA Producer DLLs using `SetDLLAccessPath` in your application for your application to execute properly.

For example, the Windows-based directory structure for your application could look like this:

```
\MyApp
   \audiences
   \codecs
   \common
   \plugins
   \tools
```

Encoding System SDK Areas

The following areas are included as part of the Helix DNA Producer SDK:

- Helix DNA Producer Encoding API
- Helix DNA Producer Plug-in API
- RealMedia Edit API

Helix DNA Producer Encoding API

The Helix DNA Producer encoding API is a set of interfaces that configure and control encoding sessions. This encoding API provides a means for you to specify input, output, and encoding settings to create RealMedia files or broadcast RealMedia streams to a Helix Universal Server.

Helix DNA Producer Plug-in API

The Helix DNA Producer Plug-in API lets you build custom media plug-ins used by the Helix DNA Producer filter graph. The filter graph is then used by an encoding job to execute an encoding session. Helix DNA Producer is built on the concept of arranging media input filters, transform filters, and output filters in a filter graph that can encode, manipulate, or broadcast media data. The system is extensible in such a way that you can write plug-ins that are

either dynamically discovered and used at runtime, or can be programmatically inserted in a filter graph.

RealMedia Edit API

The RealMedia Edit API consists of a set of interfaces that you use to edit existing RealMedia (.rm) files. It also lets you to add RealMedia events and image maps to RealMedia files. The RealMedia Edit API is divided into two main interfaces which are referred to as the RealMedia edit and RealMedia events interfaces.

The RealMedia edit interfaces allow you to perform the following operations on a .rm file:

- Edit title, author, copyright, and comment fields.
- Modify Allow Recording and Allow Download settings.
- Trim the start and end times of a .rm file. This is referred to as a cut operation.
- Paste two or more .rm files together. This is referred to as a paste operation.
- Dump the contents of a .rm file to a text file. This is known as a dump operation.
- Add meta information to the file that is specific to your application.
- Obtain information on the types of streams contained in the .rm file.
- Obtain the size of the video image.
- Determine whether a .rm file is single rate or SureStream.

The RealMedia events interfaces allow you to perform the following operations on a .rm file:

- Add events and image maps to a .rm file.
- Dump events and image maps from a .rm file to a text file.

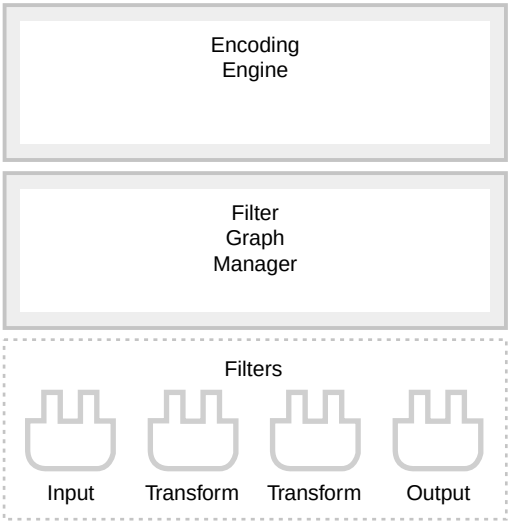
Helix DNA Producer Encoding System

The Helix DNA Producer encoding system provides the ability to configure and run an encoding job for the purpose of creating media files or broadcasting media to a Helix Universal Server. It also provides plug-in

interfaces that allow developers to extend the capabilities of the encoding system. The main components of the encoding system are:

- Encoding engine.
- Filter graph manager.
- Plug-ins.

The following figure shows the layers of the encoding system provided by Helix DNA Producer.



Encoding Engine

The Helix DNA Producer encoding engine consists of a number of SDK objects and interfaces used to configure and run an encoding job. Examples of these objects include Encoding Job, Input, Prefilter, Output, and Stream. With these objects, an application can specify a file or capture device for input, configure a set streams and their encoding settings, and broadcast to Helix Universal Server or generate a RealMedia file.

Filter Graph Manager

The Helix DNA Producer filter graph manager is used by the encoding engine to manage the connection of various media filters, and manage data flow through the filters once they are connected. There are no public interfaces for

controlling the filter graph manager since simpler encoding-specific methods are provided in the Helix DNA Producer SDK through the encoding engine objects. This component is described here to give filter plug-in writers a better understanding of the layer that is loading and pushing data through their media filters.

Plug-ins

Helix DNA Producer plug-ins are media filters that can be dynamically discovered and used in the Helix DNA Producer encoding system. These plug-in filters fit into the filter graph at the lowest level of the architecture. There are three different types of plug-in filters: input, transform and output.

There are distinct categories of each plug-in filter type. Inputs can be either file reader or capture filters. Prefilters and postfilters are examples of transforms. Output can be either broadcast or file output filters. These categories are described in more detail in later chapters.

Plug-in filters either generate or receive media samples. These media samples can be manipulated in the filter, then are passed on to the next filter in the filter graph.

Using the Helix DNA Producer SDK

This section includes information about the basic components of the Helix DNA Producer SDK. In addition, it includes material about new concepts introduced for the first time in the Helix DNA Producer SDK.

COM

The Helix DNA Producer SDK 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.

Most interfaces in this SDK begin with the prefix “IHXT”. There are also interfaces with the prefix “IHX”; these are interfaces shared between the Helix SDK and the Helix DNA Producer SDK.

Helix DNA Producer 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. The Helix DNA Producer implementation of COM eliminates the need for heavyweight Windows components like the registry and the COM and OLE runtime libraries.

Property Bags

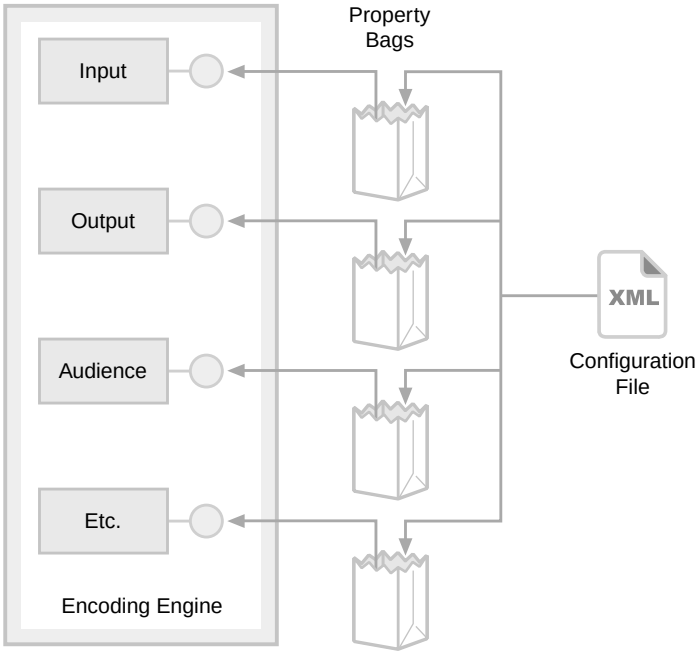
Property bags are used throughout the Helix DNA Producer SDK. A property bag is an object that stores a collection of properties. Each property has a name and a value. The property name is a string. The property value is a distinct data type, such as a string, an unsigned integer, a double, and so on.

Property bags are useful because they provide a single interface for configuring most components in the system. It is also flexible and doesn't require constantly updating an interface to support new properties of an object.

XML Configuration Files

The Helix DNA Producer SDK supports the notion of serializing and deserializing objects in the system using XML configuration files. The SDK supports job files, audience template files, and server template files. The SDK installs some sample audience files in the `\producersdk\bin\audiences` directory. To understand how job files work, you can run the Helix DNA Producer 11.0 GUI application, set your encoding settings, then choose the **File->Save Job** menu item. You can open up the resulting file in a text editor to examine how your settings were saved.

The following figure demonstrates the relationship between XML configuration files, property bags, and the encoding engine objects that use the properties.


Property
Bags
Input
Output
Audience
Etc.
Encoding Engine
XML
Configuration
File

CHAPTER

2

SDK ORGANIZATION

This chapter lists the header files and sample files you can use to create Helix DNA Producer components. The chapters that follow describe in detail how to use these files.

The include Directory

The include directory of the SDK download contains the header files that describe Helix DNA Producer's public interfaces. For descriptions of the interfaces defined in these header files, see "Appendix A: Interface List" beginning on page 161.

Header Files

File	Defines
hxbastsd.h	Definitions used by hxtypes.h that correctly define the basic size types.
hxccf.h	RealMedia Architecture common class factory interface.
hxcom.h	Definitions for items required for Component Object Model (COM) interfaces in Helix.
hxcomm.h	Common utility interfaces.
hxengin.h	Callback, networking, and scheduling interfaces.
hxevtype.h	Media sample event enumerators.
hxiids.h	All interface IDs (IIDs) used in Helix interfaces.
hxplgns.h	Additional plug-in interfaces.
hxplugn.h	Plug-in inspector interface.
hxplugncompat.h	Provides backward compatibility for previous versions of the producer CreateInstance function.
hxresult.h	Definitions for Helix status codes.
hxtypes.h	Definitions for several types used in Helix.
hxvalue.h	Option and key value interfaces.
ihxpckts.h	Packet, buffer, and stream interfaces.

(Table Page 1 of 2)

Header Files (continued)

File	Defines
ihxtaudioformat.h	Enumerators and macros for the audio format field.
ihxtbase.h	The basic public classes for the Helix DNA Producer SDK components.
ixhtconfigagenthelper.h	A configuration agent template for implementing the property bag and configuration classes for external plug-in agent classes.
ihxtconstants.h	Constants used by the Helix DNA Producer SDK components.
ihxtedit.h	Main interfaces and functions used for the RealMedia file editing.
ihxtedit2.h	Basic information about a RealMedia file.
ihxtencodingjob.h	Encoding job interfaces.
ihxteventcodes.h	Event codes for the Helix DNA Producer SDK components.
ihxtevnts.h	Definitions for classes of events.
ixhtfdump.h	Interface for dumping a RealMedia file to text file format.
ihxtfileobserver.h	File observer interface to the logging system.
ihxtinputformathelper.h	Helper class designed to consistently manage handling and display of input properties for all input sources—file and capture based.
ihxtlogsystem.h	Main interfaces for the logging system.
ihxtplugininfobase.h	A template for implementing the IHXPlugin and IHXPluginProperties interfaces that define an external plug-in for the Helix DNA Producer SDK components.
ihxtpreviewsink.h	Interfaces used to receive preview media samples.
ihxtpropertybag.h	Defines a property bag for the Helix DNA Producer SDK components.
progsink.h	Progress indication interfaces used for the RealMedia Edit API.
rmflsnk.h	Definitions for the RealMedia file sink interface.
rmmetain.h	Definitions for the "well-known" or automatically added metadata property names.
setdllac.h	Function pointer definition for SetDLLAccessPath exported by the encoding engine.

(Table Page 2 of 2)

The samples Directory

The samples directory contains sample C++ files, as well as header files and make files, or project files, for making Helix DNA Producer components. Use

the sample files to learn about the RealSystem Transform Architecture and to build your own encoders and plug-ins.

Sample Files

Directory	Samples
advencoder	An example encoder application that includes advanced features, such as logging messages.
encoder	An example encoder application that demonstrates basic encoding.
inputplugin	An example input plug-in.
mediasinkencoder	A command-line test application that demonstrates how to encode using a media sink, which provides custom input without having to write a plug-in.
prefilterplugin	An example generic prefilter transform plug-in.
rmeditor	An example RealMedia editor application.
rmevents	An example application that processes RealMedia events.

Using the Samples

The Helix DNA Producer SDK contains several samples that demonstrate how to develop customized Helix DNA Producer components. Each sample contains three make files for compiling the components in Windows, Linux, or Mac OS X.

To compile the samples, go to the directory of the sample you want to compile and use the following command:

- For Windows

  ```
  nmake /f win_vc6.mak
  ```

- For Mac OS X

  ```
  make -f mach-o.mak && make -f mach-o.mak copy
  ```

- For Linux

  ```
  make -f linux.mak && make -f linux.mak copy
  ```

To run the sample applications in Windows, Linux, and Mac OS X once they are compiled, you will need to copy the executables over to the `\producersdk\bin` directory. The sample applications are coded to load the main Helix DNA Producer SDK DLL from this location.

To test Helix DNA Producer plug-in DLLs in Windows or Linux, copy the plug-in DLLs to the `\producersdk\bin\tools` directory.

To test a plug-in in Mac OS X, you must insert the plug-in bundle (directory) in the `/producersdk/bin/tools` folder.

When you start creating your own applications, you can copy all of the directories and files found in `\producersdk\bin` to a new directory you specifically create for your own application. When you change directories, you must specify the location of the Helix DNA Producer DLLs using `SetDLLAccessPath` in your application for your application to execute properly.

CHAPTER

3

ENCODING OVERVIEW

The RealSystem Transform Architecture encoding API is a set of interfaces that configures and controls encoding sessions. This encoding system provides a means for you to specify input, output, and encoding settings to create RealMedia files or broadcast RealMedia streams to a Helix Universal Server.

Interfaces

An encoding system typically implements the following interfaces:

- `IHXTClassFactory`. Header file: `ixhtencodingjob.h`.

  This interface provides a means of creating Helix DNA Producer objects, such as `IHXTEncodingJob`, `IHXTInput`, and so on. The class factory can either just create the object, or create and initialize the object.

- `IHXTConfigurationAgent`. Header file: `ixhtbase.h`.

  This interface initializes all of the encoding interfaces and configures all of their encoding properties.

- `IHXTEncodingJob`. Header file: `ihxtencodingjob.h`.

  This interface specifies the basic inputs, metafiles, and profiles, as well as supplies the primary encoding session controls and access to the event system manager. All other encoding session interfaces are essentially child processes of the encoding job.

- `IHXTInput`. Header file: `ihxtencodingjob.h`.

  This interface configures the input property bag for the encoding job using the methods inherited from the `IHXTConfigurationAgent` interface. In addition, you use this interface to add and manipulate prefilters used to modify the data that is being input.

- IHXTPrefilter. Header file: ihxtencodingjob.h.

  This interface configures the prefilter property bag for the encoding job using the methods inherited from the IHXTConfigurationAgent interface.

- IHXTPostfilter. Header file: ihxtencodingjob.h.

  This interface configures the postfilter property bag for the encoding job using the methods inherited from the IHXTConfigurationAgent interface.

- IHXTOutputProfile. Header file: ihxtencodingjob.h.

  This interface associates a list of destinations (file writers or broadcast transmitters) with a single media profile (how something will be encoded). In addition, this interface inherits its configuration methods from the IHXTConfigurationAgent interface.

- IHXTMediaProfile. Header file: ihxtencodingjob.h.

  This interface configures the media profile property bag with the current encoding mode (video and audio) using the methods inherited from the IHXTConfigurationAgent interface. In addition, this interface adds, removes, and manages audiences.

- IHXTDestination. Header file: ihxtencodingjob.h.

  This interface designates the destination of the stream (either a file writer or several broadcast destination types) with the destination property bag created using the methods inherited from the IHXTConfigurationAgent interface. In addition, this interface adds, deletes, and manages any required postfilters.

- IHXTAudience. Header file: ihxtencodingjob.h.

  This interface configures the audience property bag using the methods inherited from the IHXTConfigurationAgent interface. In addition, this interface adds, removes, and manages a list of stream configurations (typically video and audio codecs).

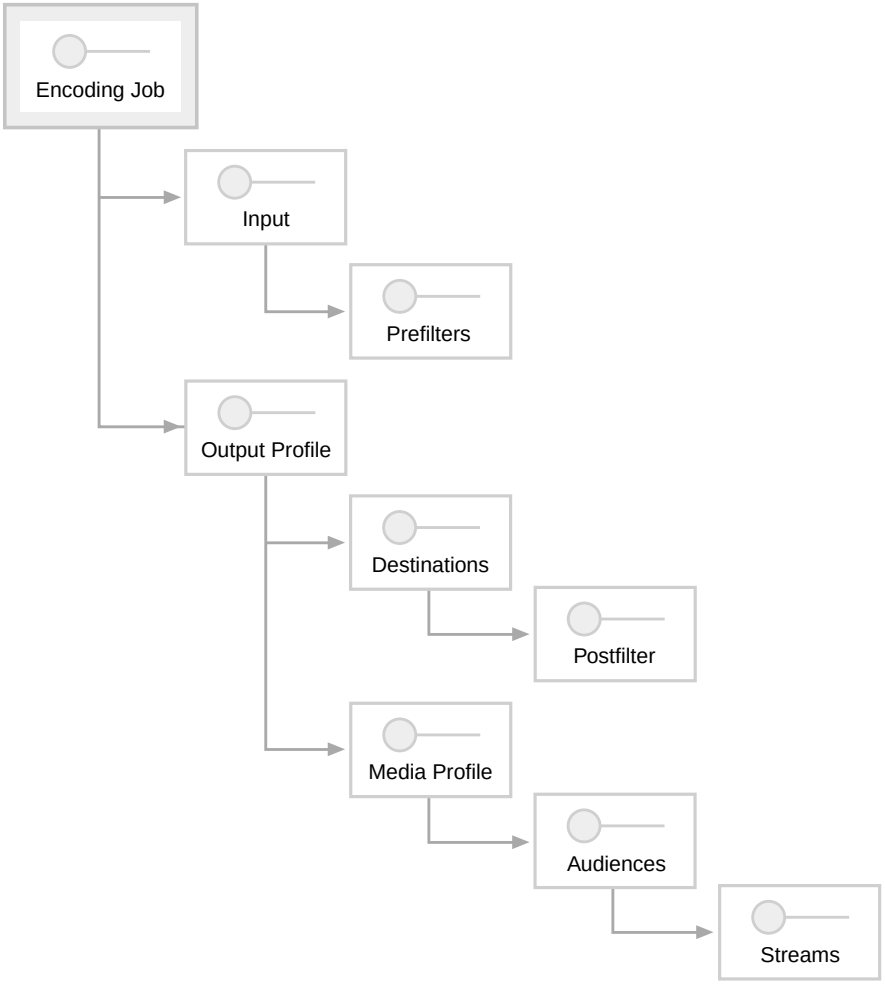
- IHXTAudienceEnumerator. Header file: ihxtencodingjob.h.

  This interface searches a specific directory path for audience template files, such as 56k Dial-up.rpad and 150k LAN.rpad, and gets the audience information from the specified files.

- `IHXTStreamConfig`. Header file: `ihxtencodingjob.h`.

  This interface configures the stream property bag for the specific stream configurations required. This interface inherits all of its methods from the `IHXTConfigurationAgent` interface.

The following figure demonstrates the hierarchical associations between the basic interfaces for an encoding job.



Getting Started

The starting point for all operations in the Helix DNA Producer encoding API is the Helix DNA Producer DLL (encsession.dll on Windows, encsession.so on Linux, and encsession.bundle on Mac OS X). To access the main class factory object, load the DLL and get a function pointer to the exported function HXTCreateJobFactory. Use this function to create the class factory for the SDK. The class factory is used to create all objects needed to set up an encoding job.

Class Factory

The class factory creates all encoding objects. The IHXTClassFactory interface offers various methods for creating objects. The most basic method is the create instance method (IHXTClassFactory::CreateInstance). This method creates objects specified by an interface GUID, but does not initialize the newly-created objects. The following sample demonstrates how this method is used:

```
IHXTClassFactory* pFactory; // Created earlier with HXTCreateJobFactory
IHXTPropertyBag* pInitParams; // Created earlier IHXTClassFactory::CreateInstance
pInitParams->SetBool(kPropEnableTwoPass, FALSE);
IHXTEncodingJob* pJob = NULL;
HX_RESULT res = pFactory->CreateInstance(IID_IHXTEncodingJob, &pJob);
if (SUCCEEDED(res))
   res = pJob->Initialize(pInitParams);
```

Like IHXTClassFactory::CreateInstance, the build instance methods (IHXTClassFactory::BuildInstance, IHXTClassFactory::BuildInstanceFromBuffer, IHXTClassFactory::BuildInstanceFromFile, and IHXTClassFactory::BuildInstanceFromObject) create objects specified by interface GUID. However, these methods also attempt to initialize the newly-created object using a caller-specified property bag. The following sample demonstrates how these methods are used:

```
IHXTClassFactory* pFactory; // Created earlier with HXTCreateJobFactory
IHXTPropertyBag* pInitParams; // Created earlier IHXTClassFactory::CreateInstance
pInitParams->SetBool(kPropEnableTwoPass, FALSE);
IHXTEncodingJob* pJob = NULL;
HX_RESULT res = pFactory->BuildInstance(IID_IHXTEncodingJob, &pJob,
pInitParams);
```

In this sample, IHXTClassFactory::BuildInstance automatically initializes the newly-created object with pInitParams.

Setting Up an Encoding Job

The properties and collective child objects of an encoding job represent the complete makeup of an encoding session. The encoding job object specifies input, metadata, and ouptut profiles for an encoding session. It also provides the primary encoding session controls, that is, controls to start encoding (`IHXTEncodingJob::StartEncoding`), stop encoding (`IHXTEncodingJob::StopEncoding`), and cancel encoding (`IHXTEncodingJob::CancelEncoding`). Access to the event system manager is also provided from the job object using the `IHXTEncodingJob::GetEventManager` method.

To create an encoding job, call `IHXTClassFactory::CreateInstance` using the `IID_IHXTEncodingJob` reference identifier.

All of the encoding interfaces (`IHXTEncodingJob`, `IHXTInput`, `IHXTAudience`, and so on) inherit methods from `IHXTConfigurationAgent`. They must be initialized by calling the `IHXTConfigurationAgent::Initialize` method prior to calling any other method. Once `IHXTConfigurationAgent::Initialize` has been successfully called, their encoding properties can be configured using the type-specific get and set property methods (such as `Get/SetUint` and `Get/SetString` with property name/value pairs). The following example shows this initalization process:

```
IHXTPropertyBag* pInitParams; // Created earlier IHXTClassFactory::CreateInstance
pInitParams->SetBool(kPropEnableTwoPass, FALSE);
IHXTEncodingJob* pJob; // Created earlier with IHXTClassFactory::CreateInstance
pJob->Initialize(pInitParams);
pJob->SetBool(kPropEnableTwoPass, FALSE);
```

> **Note:** For reasons of brevity, return codes are not shown in this sample.

The following table describes the encoding job properties contained in the `ihxtconstants.h` header file.

Encoding Job Properties

Property	Type	Description
`kPropObjectName`	string	Descriptive name of the object instance.
`kPropEnableTwoPass`	BOOL	Enables two-pass encoding.

Input

Input objects are created with the class factory interface using a property bag to specify what type of input source you want. Three types of input are supported: file readers, capture devices, and custom media inputs. In addition to a single input, you can also create multiple, parallel inputs.

Once you have populated a property bag, call `IHXTClassFactory::BuildInstance` using the `IID_IHXTInput` reference identifier and pass in the property bag.

The following table contains the properties supported by all input types.

All Input Properties

Property	Type	Description
`kPropObjectName`	string	Descriptive name of object instance.
`kPropHasAudio`	BOOL (read-only)	Determines if input contains audio.
`kPropHasVideo`	BOOL (read-only)	Determines if input contains video.
`kPropInputWidth`	UINT32 (read-only)	If the input contains video, determines input video width.
`kPropInputHeight`	UINT32 (read-only)	If the input contains video, determines input video height.

Audio/Video Files

File inputs, such as audio/video readers, are input plug-ins that can read and decompress input sources such as .AVI files, QuickTime files, .WAV files, and so on. They require the plug-in type property and the input pathname property be specified when the object is initialized. The following table contains the properties supported by audio/video readers.

Audio/video Readers Input Properties

Property	Type	Description
`kPropPluginType`	string (required, initialization only)	Plug-in type. Audio/video readers must have a plug-in type of `kValuePluginTypeInputAVFile`.
`kPropPluginName`	string (initialization only)	Name of the plug-in to load. Not setting this property allows the Helix DNA Producer SDK to pick the optimal plug-in based on `kPropInputPathname`.
`kPropInputPathname`	string (required, initialization only)	Pathname of the source file.

(Table Page 1 of 2)

Audio/video Readers Input Properties (continued)

Property	Type	Description
kPropDuration	IUnknown / IHXTTime (read-only)	Duration of the source file.
kPropNumTracks	UINT32 (read-only)	Number of tracks available in the source file.

(Table Page 2 of 2)

The encoder sample in the \producersdk\samples\encoder directory demonstrates how these properties are used.

Helix DNA Producer provides a mechanism that enables applications to easily support new input file formats without significant changes to the application. You can add support for additional file formats by including file extension support for the file format in your input plug-in's plug-in property table.

When you add a new file format to the input plug-in's property table, you must also specify a ranking that determines which plug-in to use when two or more plug-ins support the same file extension. The plug-in with the highest priority is used to read the input. Priority ranking is set from 0 to 100, with 0 being the lowest priority and 100 being the highest, with 80 being the normal value.

All file readers can also use a wildcard (*) and a priority for the wildcard to indicate they will attempt to handle any file extension. A file reader that cannot read a given file should return the proper error result to Helix DNA Producer so that the producer can continue trying file readers until a suitable reader for the input is found. Priorities for wildcard should be such that these are tried as a last resort to minimize the number of failed attempts at reading an input file.

The following table documents the priorities that are given the file reader plug-ins that ship with Helix DNA Producer.

Input Plug-in Priority Mapping

Plug-in	Extensions	Priority (factor 0-100)
rn-avfile-avireader	*	60
rn-avfile-dsreader	.AVI	70
	*	80
	.AVI	80
	.MPG	80

* matches any file extension

Input Plug-in Priority Mapping

Plug-in	Extensions	Priority (factor 0-100)
	.MP3	70
	.WAV	50
	.MPA	80
	.WMA	80
	.WMV	80
	.MOV	20
	.MPEG	80
	.ASF	80
rn-avfile-movreader	*	50
	.MOV	80
rn-avfile-qtreader	*	70
	.AVI	60
	.MPG	80
	.MP3	40
	.WAV	40
	.MOV	60
	.DV	80
	.AIFF	80
rn-avfile-wavreader	*	40
	.WAV	80

* matches any file extension

For example, you might want to create a plug-in that supports input in MPEG file format. You would create an input plug-in for reading MPEG files (as well as writing a codec plug-in for uncompressing media in the MPEG video codec). Your MPEG plug-in would specify what file extensions that plug-in is capable of reading and what priority to give that input plug-in.

The following sample shows the file extension and priority section of a plug-in property table in an input plug-in designed for an MPEG file:

```
//  STEP 4) fill in the use preference field
 { eStringType, kPropFileExtensions, (void*)
                  ".MPEG:80,.MPG:80,.M1V:70,.M1A:70",NULL },
```

Once compiled, the new MPEG input plug-in can be added to the `plugins` directory of an application built on the Helix DNA Producer architecture. If

that application is designed to be extensible with respect to inputs, the user can then specify the file extension from which they would like to input data. The application only needs to identify the input filename (with extension) and Helix DNA Producer will then take care of identifying the input plug-ins that are capable of handling that file extension (hence that file type) and which one has the highest priority.

Capture

Capture devices are plug-ins that wrap operating-specific capture subsystems, such as DirectShow or Video4Linux. They require the plug-in type property (`kPropPluginType`) to be specified when the object is initialized. The `kPropAudioDeviceID` and `kPropVideoDeviceID` properties should be set to the string name of the devices. A wildcard (*) value can also be used to specify the first available device for both audio and video device identifiers. Optionally, you can also provide audio and video ports in this property bag using `kPropAudioDevicePort` and `kPropVideoDevicePort` string properties. The following table contains the properties supported by capture devices.

Capture Device Properties

Property	Type	Description
`kPropPluginType`	string (required, initialization only)	Plug-in type. Capture devices must have a plug-in type of `kValuePluginTypeInputCapture`.
`kPropPluginName`	string (initialization only)	Name of the plug-in to load. Not setting this property allows the Helix DNA Producer SDK to pick the optimal plug-in.
`kPropDuration`	`IUnknown` / `IHXTime`	Length of time to capture the audio/video.
`kPropAudioDeviceID`	string	Name of the audio device (that is, which sound card) from which to capture.
`kPropAudioDevicePort`	string	Name of the audio port (such as microphone, line in, and so on) from which to capture.
`kPropVideoDeviceID`	string	Name of the video device (that is, which video capture card) from which to capture.
`kPropVideoDevicePort`	string	Name of the video port (such as, composite, svideo, and so on) from which to capture.
`kPropVideoFrameWidth`	UINT32	Video frame width to capture.
`kPropVideoFrameHeight`	UINT32	Video frame height to capture.

Creating Parallel Inputs

Parallel inputs provide a means of encoding more than one input at a time. This is especially useful if your audio and video sources are separate and you have to encode them into a single file.

For example, some capture cards support capture of audio and video to separate files. To encode these separate files, you need to use parallel inputs. Alternatively, you might be encoding an audio/video file in multiple languages in which you have created a video-only file and audio for each language. You can then use parallel inputs to encode the audio and video for each language.

➤ To create multiple parallel inputs:

1. Use IHXTClassFactory::CreateInstance to create an IHXTPropertyBag interface. Fill the created property bag with the parallel input initialization parameters (kPropPluginType, kValuePluginTypeInputParGroup).

```
res = m_pFactory->CreateInstance(IID_IHXTPropertyBag,
(IUnknown**)ppInitParams);
if(SUCCEEDED(res))
{
   res = (*ppInitParams)->SetString(kPropPluginType,
kValuePluginTypeInputParGroup);
}
```

2. Use IHXTClassFactory::BuildInstance to create an instance of the first input source (IHXTInput).

```
IHXTInput* pInput = NULL;
res = m_pFactory->BuildInstance(IID_IHXTInput, pInitParams,
(IUnknown**)&pInput);
```

3. Use IHXTEncodingJob::SetInput to set this IHXTInput instance as the input source for the encoding job.

```
res = m_pJob->SetInput(pInput);
```

4. Query (IUnknown::QueryInterface) for the IHXTInput2 interface from the IHXTInput interface.

```
res = pInput->QueryInterface(IID_IHXTInput2, (void**)&pInputGroup);
```

5. Release the IHXTInput interface.

For each additional input, perform the following:

6. Set up either a capture device or a file for the input source.

7. Use `IHXTClassFactory::BuildInstance` to create an instance of the `IHXTInput` interface for the input source.
8. Use `IHXTInput2::AddInput` to add the input source to the list of inputs.

   ```
   res = pInputGroup->AddInput(pInput);
   ```

9. If the input source is a file, set up audio, video, events, and image maps as required.
10. Release the `IHXTInput` interface.
11. Repeat steps 6 through 10 for each additional parallel input.
12. Release `IHXTInput2`.

The `\samples\advencoder\encoder.cpp` file included with the Helix DNA Producer SDK provides an example implementation of parallel inputs.

Prefilters

Several types of audio and video prefilters can be set on the encoding job. Some filters merely have a name and require no configuration.

The Helix DNA Producer SDK allows you to add and manipulate prefilters as if they were in a container similar to a list or array. The prefilters are then added to the underlying filter graph in that same order. Therefore, it is important to load your prefilters in the order in which you want them used. For example, you probably would not want to resize something before you deinterlace it, since you would probably never end up doing any deinterlacing.

The order of the built-in Helix DNA Producer video prefilters is:

- Cropping
- Inverse telecine
- Deinterlace
- Video noise reduction
- Black level
- Resize

> **Note:** If a resize filter is specified as a prefilter, resizing will be performed at that point in the prefilter chain. In general, a resize prefilter should not be used because it is more efficient to have the video codec perform resizing. However, if the

encoding job has multiple outputs, there may be a performance advantage to using a resize filter (since it is resized at one point, as opposed to at each video codec).

The following tables contain the properties supported by various types of prefilters.

All Prefilter Properties

Property	Type	Description
kPropObjectName	string	Descriptive name of object instance.
kPropIsEnabled	BOOL	Enables or disables the prefilter.

Video Cropping Filter Properties

Property	Type	Description
kPropPluginType	string (required, initialization only)	Plug-in type. Possible values: kValuePluginTypePrefilterCropping.
kPropPluginName	string (initialization only)	Plug-in name. Possible values: kValuePluginNamePrefilterCropping.
kPropCropLeft	UINT32	Leftmost value of the crop region.
kPropCropTop	UINT32	Top value of the crop region.
kPropCropWidth	UINT32	Width value of the crop region.
kPropCropHeight	UINT32	Height value of the crop region.

Video Black Level Filter Properties

Property	Type	Description
kPropPluginType	string (required, initialization only)	Plug-in type. Possible values: kValuePluginTypePrefilterBlackLevel.
kPropPluginName	string (initialization only)	Plug-in name. Possible values: kValuePluginNamePrefilterBlackLevel.

Video Noise Reduction Filter Properties

Property	Type	Description
kPropPluginType	string (required, initialization only)	Plug-in type. Possible values: kValuePluginTypePrefilterVideoNoiseReduction.
kPropPluginName	string (initialization only)	Plug-in name. Possible values: kValuePluginNamePrefilterVideoNoiseReduction.
kPropNRLevel	string	Indicates the filter strength. One of the following: kValueNROff kValueNRLow kValueNRHigh

Video Deinterlace/Inverse Telecine Filter Properties

Property	Type	Description
kPropPluginType	string (required, initialization only)	Plug-in type. Possible values: kValuePluginTypePrefilterDeinterlace.
kPropPluginName	string (initialization only)	Plug-in name. Possible values: kValuePluginNamePrefilterDeinterlace.
kPropDITManual	BOOL	Determines whether the prefilter will attempt to automatically sense if the deinterlace and inverse telecine filters are needed.
kPropDITDeinterlace	BOOL	If the kPropDITManual property is TRUE, turns on the deinterlace filter. Otherwise this value is ignored.
kPropDITInvTelecine	BOOL	If the kPropDITManual property is TRUE, turns on the inverse telecine filter. Otherwise this value is ignored.

Video Resize Filter Properties

Property	Type	Description
kPropPluginType	string (required, initialization only)	Plug-in type. Possible values: kValuePluginTypePrefilterResizer.
kPropPluginName	string (initialization only)	Plug-in name. Possible values: kValuePluginNamePrefilterResizer.

(Table Page 1 of 2)

Video Resize Filter Properties (continued)

Property	Type	Description
kPropResizeQuality	string	Affects the resulting quality of an output video when resize is applied. Choosing high quality results in a better quality resize but uses considerably more CPU cycles. Possible values: kPropResizeQualityFast kPropResizeQualityHigh
kPropOutputWidth	UINT32	Sets the output video width in pixels. If this property is set to 0, then the width is computed from the height property and the input aspect ratio. If both the width and height properties are set to 0 or the width and height properties are omitted, no resize occurs. This value should be set in an increment supported by the codec; if it is not, the video is cropped down to the nearest multiple that is supported.
kPropOutputHeight	UINT32	Sets the output video height in pixels. If this property is set to 0, then the height is computed from the width property and the input aspect ratio.

(Table Page 2 of 2)

Audio Gain Filter Properties

Property	Type	Description
kPropPluginType	string (required, initialization only)	Plug-in type. Possible values: kValuePluginTypePrefilterAudioGain.
kPropPluginName	string (initialization only)	Plug-in name. Possible values: kValuePluginNamePrefilterAudioGain.
kPropAudioLimiterGain	UINT32	The audio gain (in dB) with a range from +12 dB gain (amplification) to -12 dB gain (attenuation).

Audio "Watchdog" Filter Properties

Property	Type	Description
kPropPluginType	string (required, initialization only)	Plug-in type. Possible values: kValuePluginTypePrefilterLevelMeter.
kPropPluginName	string (initialization only)	Plug-in name. Possible values: kValuePluginNamePrefilterLevelMeter.

Audio Delay Compensation Filter Properties

Property	Type	Description
kPropPluginType	string (required, initialization only)	Plug-in type. Possible values: kValuePluginTypePrefilterAudioDelayComp.
kPropPluginName	string (initialization only)	Plug-in name. Possible values: kValuePluginNamePrefilterAudioDelayComp.
kPropAudioDelay	IUnknown / IHXTTime	The value, in seconds, by which timestamps are delayed. If this property is set, the audio signal is delayed by the amount of time specified by this property. This is done by padding the specified number of seconds of silence at the beginning of the audio stream and adjusting the timestamp on all other audio samples so they occur the specified number of seconds earlier. If both the delay and advance properties are set, then they are summed such that the two are essentially subtracted from one another and the remainder is applied as delay or advance, whichever is applicable.
kPropAudioAdvance	IUnknown / IHXTTime	The value, in seconds, by which timestamps are advanced. If this property is set, the audio signal is advanced by that amount of time by removing the specified number of seconds of audio samples at the beginning of the audio stream and adjusting the timestamp on all other audio samples so they occur the specified number of seconds later in time.

Output Profile

Output profiles are container objects that specify how media will be encoded through the media profile, as well as where the encoded data is sent through destinations. An output profile holds a single media profile (which contains audience and stream settings) and multiple output destinations. It is important to note that every destination in an output profile gets the same set of streams with the same encoding settings as all other destinations in the output profile.

To create an output profile, call `IHXTClassFactory::CreateInstance` using the `IID_IHXTOutputProfile` reference identifier. The following table contains the properties supported by output profiles.

Output Profile Properties

Property	Type	Description
`kPropObjectName`	string	Descriptive name of object instance.

Multiple Output Profiles

Multiple output profiles provide you with the ability to encode more than one output at a time from a single input, each of which can address different uses. That is, one input source file will be compressed to multiple output formats in a single pass. For example, you can use multiple output profiles to:

- Encode to multiple bit rates.
- Broadcast at one bit rate and archiving at another.
- Provide audio only and audio-video output.

Encoding to Multiple Bit Rates

SureStream encodes one file that spans the range of a few kilobits to several megabits. However, there may be times when you require additional capabilities not supported by SureStream. In this case, multiple output profiles allow you to simultaneously encode more than one file with a wide range of capabilities.

Mulitple output profiles can provide you with the following capabilities:

- Different video frame sizes

  Multiple output profiles allow you to independently set frame size for different bit rates. For example,you might want to encode dial-up speeds at a smaller frame size of about 176 by 144 pixels to maintain a reasonable quality, and encode larger bit rates of 256 kilobits per secont (Kbps) and up at higher frame sizes of 320 by 240 pixels. Download media in the 1 megabits per second (Mbps) range and up could even be encoded at even larger frame sizes of 640 by 480 pixels, and possibly HDTV resolution encoding.

- Large bit rate gaps

  With multiple output profiles, you can create independent files with large gaps in bit rate. Because of its original design, SureStream may not

upshift over gaps in bit rate of 50 Kbps or greater. This sometimes occurs at low bit rates because SureStream's algorithm for upshifting is based on increasing the bit rate as a percentage of the current bit rate. In practice, this means the client might not upshift from 56 Kbps streams to 128 Kbps or higher streams once a downshift occurs. Therefore if you are encoding content with large gaps in bit rate, you should use mulitiple output profiles to create two files,one for low bit rate users and a second for high bit rate users.

By using multiple output profiles you can, for instance, target two distinct audiences: those on low bit rate connections and those on high bit rate connections. To do this, you can use multiple output profiles to target dial-up users in the range of 28 Kbps to 56 Kbps and then a second group of users at DSL speeds in the range of 256 Kbps to 512 Kbps. For example:

- Low bit rate: 12 Kbps, 16 Kbps, 28 Kbps, 26 Kbps, and 56 Kbps at 176 by 144 pixels
- High bit rate: 256 Kbps at 320 by 240 pixels

Broadcasting at One Bit Rate and Archiving at Another

You can also use multiple output profiles to broadcast content at a relatively conservative bit rate and offer an on-demand copy of the media for download or on-demand streaming at a later time. Often, a live event results in much larger viewership at a single time than on-demand content due to the fact that on-demand content can be viewed at any time by users all around the globe, while live content must be viewed at a single time by everyone. Thus, while it might be feasible to allow users to download a very high bit rate media file, it would not be feasible to offer the same bit rates during the live broadcast. Therefore, you could configure one output profile with a single server destination and a second output profile with a high bit rate file destination.

For example, a company might want to broadcast their CEO All-Employees Briefing over an intranet. They do not have a multicast-enabled network and so decide to offer the webcast at a relatively low bit rate during the broadcast but archive a copy at a much higher bit rate for on-demand viewing. To achieve this goal, multiple output profiles with the following bit rates are used:

- 80 kbps audio/video stream for the live broadcast.
- 225 kbps audio/video archive file uploaded to a server after the webcast.

Providing Audio-only and Audio/Video Output

Multiple output profiles can also be used to simultaneously create an audio-only version of an audio/video clip. Without multiple output profiles, you would have to encode the media file once as audio and video, and then a second time as audio-only.

Using multiple output profiles, you can set up one encoding session with one audio-only and one audio/video output. In most cases, both output profiles could use the same target audience, although this is not required. For example, if this was a live broadcast each output profile might have both an archive file destination as well as a server broadcast destination. For on-demand content creation, both output profiles would likely have only one destination.

Output Filename Redundancy

It is possible for two destinations, either in the same output profile or in different output profiles to use the same filename and path. This situation will not result in a loss of data because Helix DNA Producer uses a filename collision avoidance algorithm to prevent overwriting previous filenames.

During encoding, all files are written to a temporary filename. At the end of encoding, Helix DNA Producer renames the output file to the name specified by the user. If at that time the filename is in use by an existing file, Helix DNA Producer renames the existing file by appending the string "_arch*NNN*", and the new file is written to the requested name. Thus, Helix Producer simply renames any existing file and does not overwrite it.

In addition, when an existing file is renamed the following warning category log message is generated:

```
File %s already exists. Archiving existing file to %s. Writing new file to %s.
```

For More Information: See "Output Error Handling" on page 47.

Independent Metadata

When encoding content to different output profiles, you might want to add different metadata for each output independently. For example, high bit rate content might be encoded and targeted towards DSL connection users, whereas low bit rate content would be targeted towards dial-up users. In this case, it might be desirable to label each of these media outputs separately. In another example, high bit rate archive outputs might be labeled differently than lower bit rate live broadcast outputs.

Using multiple output profiles, clip information can be defined in each output profile independently. Any clip information values that could be added in previous versions of Helix DNA Producer to the entire job can now be added individually to a single output profile. Include the metadata you require in the individual output profile using the `IHXTOutputProfile2::SetMetadata` method. The clip information defined in a given output profile is written to each of the destinations within that output profile.

To maintain consistency with the previous model, clip information can still be defined globally for an entire job. The clip information defined in an output profile is in addition to the clip information defined globally in an encoding job. Clip information in the output profile overrides clip information defined in the encoding job on a field-by-field basis. In the event that a clip information field is defined in both the encoding job and the output profile, the clip information field in the output profile overrides the corresponding clip information field in the job.

For example, assume a job has two output profiles. For the entire job the following clip information fields are defined:

- `Title="General Title"`
- `Copyright = "My company (c) 2003"`

The first of the two output profiles also has the following clip information defined:

- `Title="Specific Title"`
- `Description="Some text string"`

In this example, the first output profile contains the title and description defined in the output (metadata defined in the output always overrides the job metadata). The first output profile will also contain the copyright since it was defined at the global job level only and not in that specific output profile. The second output profile will contain just the title and copyright that were defined globally since it had no clip information of its own defined.

Audience Restrictions

Any single output profile can have only one variable bit rate (VBR) audience, or it can have one or more constant bit rate (CBR) audiences. With multiple outputs, some output profiles with VBR audiences and others with CBR audiences is allowed.

Output Error Handling

When an error occurs during output, Helix DNA Producer simply raises an error to the application and keeps going. Your code can then either choose to stop the encoding and allow the user to make necessary modifications before restarting encoding, or ignore the failed destination or output without stopping the entire encode.

The following information outlines the error handling behavior for the Helix DNA Producer 11.0 SDK:

- If one or more output profiles or destinations fail at start up (when encoding starts), all other output profiles continue. An event is raised and an error is logged.
- If one or more output profiles or destinations fails during steady-state operation, all other output profiles continue. An event is raised and an error is logged.

Creating Multiple Output Profiles

Use the following steps to create basic multiple output profiles:

➤ To create multiple outputs:

1. Use `IHXTClassFactory::BuildInstance` to create an instance of the IHXTOutputProfile interface.

   ```
   res = m_pFactory->BuildInstance(IID_IHXTOutputProfile, NULL,
   (IUnknown**)&pOutputProfile);
   ```

2. Use `IHXTEncodingJob::AddOutputProfile` to add the new output file to the encoding job.

   ```
   res = m_pJob->AddOutputProfile(pOutputProfile);
   ```

3. Use `IHXTEncodingJob::GetOutputProfileCount` to get the number of available output profiles.

   ```
   *pnOutputIndex = m_pJob->GetOutputProfileCount() - 1;
   ```

4. Repeat steps 1 through 3 for each required output profile.

The `\samples\advencoder\encoder.cpp` file included with the Helix DNA Producer SDK provides an example implementation of multiple outputs.

Destination

Destinations specify where encoded packets are sent. One file writer and several broadcast destination types are supported. To create a destination object, call `IHXTClassFactory::CreateInstance` with `IID_IHXTDestination`. The following table contains the properties supported by all destinations.

All Destinations Properties

Property	Type	Description
`kPropObjectName`	string	Descriptive name of object instance.

File Writers

File writers are file output destination plug-ins that can receive encoded data packets. They require the plug-in type property to be specified when the object is initialized. The following table contains the properties supported by file writers.

File Writer Properties

Property	Type	Description
`kPropPluginType`	string (required, initialization only)	Plug-in type. Possible values: `kValuePluginTypeDestinationFile`.
`kPropPluginName`	string (initialization only)	Plug-in name. Possible values: `kValuePluginNameFileDestRealMedia`.
`kPropOutputPathname`	string	Path name of the destination file.
`kPropTempDirPath`	string	Path name of the temporary directory used for scratch files.
`kPropFileRollSize`	UINT32	Files size limit (in megabytes) at which rollover should occur. File rolling is only supported for RealMedia files. The maximum allowable value for RealMedia files is 4 gigabytes (GB), the limit of the RealMedia file format. Setting destination roll size larger than this value will result in an unreadable file being created. The new file is appended with a number (for example, movie1.rm). On Linux the destination file limit is 2 GB. On Macintosh OS X the file limit is 1.45 GB.

(Table Page 1 of 2)

File Writer Properties (continued)

Property	Type	Description
kPropFileRollTime	IUnknown (see IHXTTime)	File time limit at which rollover should occur. A duration with the syntax dd:hh:mm:ss where seconds are ignored. Days and hours may be omitted if they are 0. File rolling is only supported for RealMedia files. The maximum allowable value for RealMedia files is 4 GB (limit of RealMedia file format). Setting destination roll time such that the corresponding size is larger than this value results in an unreadable file being created. The new file is appended with a number (for example, movie1.rm). On Linux the destination file limit is 2 GB. On Macintosh OS X the file limit is 1.45 GB.
kPropMergeWriteSize	UINT32	
kPropMergeWriteInterval	UINT32	

(Table Page 2 of 2)

Automatic File Renaming

Earlier versions of Helix DNA Producer automatically renamed RealMedia files to either RM or RMVB. Any files passed to the RealMedia file writer would be renamed to ".rm" if they were constant bit rate (CBR) files and ".rmvb" if they were variable bit rate (VBR) files.

This behavior has changed as follows:

- Helix DNA Producer renames file extensions only if the file extension is ".rm" (case insensitive) and the codec is VBR (for example, RealVideo version 9 with VBR enabled or the lossless audio codec).
- If you specify the file extension as ".rmj" or any other file extension, Helix DNA Producer does not rename the file.
- If no file extension is specified, then no automatic renaming occurs.

Helix DNA Producer (without additional plug-ins) also defaults to RealMedia file format when the file extension is not recognized as a result of giving the RealMedia file format writer the highest priority (80 out of a scale of 1-100). However, new plug-ins added to the Helix DNA Producer SDK might assign a higher priority for unrecognized file extensions and thus take priority for files with no extension or unrecognized extensions.

Adding New File Writer Formats

Helix DNA Producer provides a mechanism that enables applications to easily support new output file formats without significant changes to the application. You can add support for additional file formats by including file extension support for the file format in your output destination plug-in's plug-in property table. When you add a new file format to the output destination plug-in's property table, you must also specify a ranking that determines which plug-in to use when two or more plug-ins support the same file extension. The plug-in with the highest priority is used to write the output. Priority ranking is set from 0 to 100, with 0 being the lowest priority and 100 being the highest, with 80 being the normal value.

Although applications can still explicitly request a specific file writer, by including file extension support in your output destination plug-in, applications can locate the "best" plug-in by enumerating all output destination plug-ins.

For example, you might want to create a plug-in that supports output in MPEG file format. You would create an output file destination plug-in for writing MPEG files (as well as writing a codec plug-in for compressing media in the MPEG video codec). Your MPEG plug-in would specify what file extensions that plug-in is capable of writing and what priority to give that output destination plug-in.

The following sample shows the file extension and priority section of a plug-in property table in an output file destination plug-in designed for an MPEG file:

```
//  STEP 4) fill in the use preference field
 { eStringType, kPropFileExtensions, (void*)
                  ".MPEG:80,.MPG:80,.M1V:70,.M1A:70",NULL },
```

Once compiled, the new MPEG output destination plug-in can be added to the `plugins` directory of an application built on the Helix DNA Producer architecture. If that application is designed to be extensible with respect to outputs, the user can then specify the file extension to which they would like to output data. The application only needs to set the output filename (with extension) and Helix DNA Producer will then take care of identifying the output destination plug-ins that are capable of handling that file extension (hence that file type) and which one has the highest priority.

Broadcast Destination Types

The following table contains properties supported by G2 Push Broadcast.

G2 Push Broadcast Properties

Property	Type	Description
kPropPluginType	string (required, initialization only)	Plug-in type. Possible values: kValuePluginTypeDestinationG2PushServer.
kPropPluginName	string (initialization only)	Plug-in name. Possible values: kValuePluginNameServerg2Push.
kPropBroadcastAddress	string	Helix Universal Server address (IP address or domain name).
kPropBroadcastPort	UINT32	Helix Universal Server's encoder port that will be receiving the broadcast.
kPropBroadcastStreamname	string	Name of the live stream.
kPropBroadcastUsername	string	Helix Universal Server encoder account username.
kPropBroadcastPassword	string	Helix Universal Server encoder account password.
kPropBroadcastTransport	string	Broadcast transport type between the encoder and Helix Universal Server.

The following table contains the properties supported by RBS push (account-based) broadcast

Account-based RBS Push Broadcast Properties

Property	Type	Description
kPropPluginType	string (required, initialization only)	Plug-in type. Possible values: kValuePluginTypeDestinationPushServer.
kPropPluginName	string (initialization only)	Plug-in name. Possible values: kValuePluginNameServerRBS.
kPropBroadcastAuthType	string	Authentication type. Possible values: kValueAccountBased.
kPropBroadcastAddress	string	Helix Universal Server address (IP address or domain name).
kPropBroadcastPort	UINT32	Helix Universal Server's encoder port that will be receiving the broadcast.
kPropBroadcastStreamname	string	Name of the live stream.

(Table Page 1 of 2)

Account-based RBS Push Broadcast Properties (continued)

Property	Type	Description
kPropBroadcastPath	string	Broadcast path (appended to the stream name).
kPropBroadcastUsername	string	Helix Universal Server encoder account user name.
kPropBroadcastPassword	string	Helix Universal Server encoder account password.
kPropBroadcastTransport	string	Broadcast transport type between the encoder and Helix Universal Server.
kPropBroadcastListenAddress	string	Encoder IP address.
kPropBroadcastMulticastAddress	string	Multicast server address used when the transport type is udp/multicast.
kPropBroadcastAllowResend	BOOL	Set to TRUE to allow resend of packets from the encoder.
kPropBroadcastFecPercent	UINT32	Percentage of error correction data.
kPropBroadcastFecOffset	UINT32	Number of seconds to offset redundant packets when FEC is 100%.
kPropBroadcastMulticastTTL	UINT32	Multicast Time to Live.
kPropBroadcastMetadataResendInterval	UINT32	Number of seconds between resending of header packets.
kPropBroadcastEnableTCPReconnect	BOOL	If TRUE, then the encoder would keep on reconnecting on TCP connection failure.
kPropBroadcastTCPReconnectInterval	UINT32	Number of seconds the encoder waits before attempting a reconnection after losing a connection with a Helix Universal Server.

(Table Page 2 of 2)

The following table contains the properties supported by RBS push (password-based) broadcast.

Password-based RBS Push Broadcast Properties

Property	Type	Description
kPropPluginType	string (required, initialization only)	Plug-in type. Possible values: kValuePluginTypeDestinationPushServer.
kPropPluginName	string (initialization only)	Plug-in name. Possible values: kValuePluginNameServerRBS.
kPropBroadcastAuthType	string	Authentication type. Possible values: kValueSinglePassword.
kPropBroadcastAddress	string	Helix Universal Server address (IP address or domain name).
kPropBroadcastPort	UINT32	Start port of port range (in the receiver configuration on Helix Universal Server).
kPropBroadcastPortEnd	UINT32	End port of port range (in the receiver configuration on Helix Universal Server).
kPropBroadcastStreamname	string	Name of the live stream.
kPropBroadcastPath	string	Broadcast path (prepended to the stream name).
kPropBroadcastPassword	string	Helix Universal Server encoder account password.
kPropBroadcastTransport	string	Broadcast transport type between the encoder and Helix Universal Server.
kPropBroadcastListenAddress	string	Encoder IP address.
kPropBroadcastMulticastAddress	string	Multicast server address used when the transport type is udp/multicast.
kPropBroadcastAllowResend	BOOL	Set to TRUE to allow resending of packets from the encoder.
kPropBroadcastFecPercent	UINT32	Percentage of error correction data.
kPropBroadcastFecOffset	UINT32	Number of seconds to offset redundant packets when FEC is 100%.
kPropBroadcastMulticastTTL	UINT32	Multicast Time to Live.
kPropBroadcastMetadataResendInterval	UINT32	Number of seconds between resending of header packets.

(Table Page 1 of 2)

Password-based RBS Push Broadcast Properties (continued)

Property	Type	Description
kPropBroadcastEnableTCPReconnect	BOOL	If TRUE, then the encoder would keep on reconnecting on TCP connection failure.
kPropBroadcastTCPReconnectInterval	UINT32	Number of seconds the encoder waits before attempting a reconnection after losing a connection with a Helix Universal Server.

(Table Page 2 of 2)

The following table contains the properties supported by RBS pull broadcast.

RBS Pull Broadcast Properties

Property	Type	Description
kPropPluginType	string (required, initialization only)	Plug-in type. Possible values: kValuePluginTypeDestinationPullServer.
kPropPluginName	string (initialization only)	Plug-in name. Possible values: kValuePluginNameServerRBS.
kPropBroadcastStreamname	string	Name of the live stream.
kPropBroadcastPath	string	Broadcast path (prepended to the stream name).
kPropBroadcastListenAddress	string	Encoder IP address.
kPropListenPort	UINT32	Port on which the encoder listens for the pull request.
kPropBroadcastServerTimeout	UINT32	Number of seconds encoder will wait for a ping from the Helix Universal Server before assuming no clients are connected to the stream, and closing the connection.
kPropBroadcastEnableTCPReconnect	BOOL	If TRUE, the encoder would keep on reconnecting on TCP connection failure.
kPropBroadcastTCPReconnectInterval	UINT32	Number of seconds the encoder waits before attempting a reconnect after losing a connection with a Helix Universal Server.

Postfilters

Postfilters operate on audio and video data that has already been encoded. This is in contrast to prefilters which operate on uncompressed audio and video data. An example of a postfilter plug-in is a digital rights management

(DRM) postfilter that encrypts all packets. The Helix DNA Producer 11.0 SDK does not ship with any prebuilt postfilters.

Media Profile

The *media profile* specifies encoding settings, and also acts as a container for one or more audiences in the encoding job. If the media profile contains only one audience, then a single-rate file is generated. If it contains multiple audiences, then a SureStream file is created.

It is important to note that, unlike previous versions of the Helix DNA Producer SDK, no streams are automatically added to SureStream files. For example, in previous versions of the SDK selecting the 28.8k audience with SureStream enabled would also cause 12k and 16k substreams to be added to the encoding job. To achieve the same effect with the current SDK, the "12k Substream for 28k Dial-up" and "16k Substream for 28k Dial-up" audiences must be explicitly added to the media profile. If a SureStream presentation is created without substreams, the resulting .rm file presentation may play back in a sub-optimal manner when network conditions are not pristine. Good substream values are 80% and 60% of the actual audience.

To create a media profile, call `IHXTClassFactory::CreateInstance` using the `IID_IHXTMediaProfile` reference identifier. The following table contains the properties supported by media profiles.

Media Profile Properties

Property	Type	Description
`kPropObjectName`	string	Descriptive name of object instance.

`kPropVideoMode`	string	Video encoding mode. Possible values are:
		`kValueVideoModeNormal`
		`kValueVideoModeSharp`
		`kValueVideoModeSmooth`
		`kValueVideoModeSlideshow`

`kPropDisableAudio`	BOOL	If `TRUE`, disables the encoding of audio.
`kPropDisableVideo`	BOOL	If `TRUE`, disables the encoding of video.
`kPropDisableEvents`	BOOL	If `TRUE`, disables the encoding of events.
`kPropDisableImageMaps`	BOOL	If `TRUE`, disables the encoding of image maps.
`kPropMaxPacketSize`	UINT32	The maximum audio or video packet size for all streams in the audience. For more information about this property, see the *Helix DNA Producer User's Guide*.

(Table Page 1 of 2)

Media Profile Properties (continued)

Property	Type	Description
kPropMaxPacketDuration	UINT32	The maximum duration between starting and finishing an audio or video packet. For more information about this property, see the *RealProducer User's Guide*.
kPropMaxPacketInterleavingDuration	UINT32	The maximum amount of time, in seconds, that packets are interleaved. For more information about this property, see the *RealProducer User's Guide*.
kPropLatencyMode	UINT32	Latency mode. Possible values are: kValueNormalLatencyMode kValueModerateLatencyMode kValueLowLatencyMode Note that setting this property to kValueModerateLatencyMode or kValueLowLatencyMode makes the stream incompatible with Helix Universal Server versions 10 and earlier. A setting of 0 (kValueNormalLatencyMode) maintains backward compatibility. For more information about this property, see the *RealProducer User's Guide*.
kPropAudioMode	string	Type of audio content being encoded.
kPropAudioResamplingQuality	string	Resampler quality setting. Possible values are: kValueAudioResamplingQualityFast kValueAudioResamplingQualityHigh
kPropOutputWidth	UINT32	Playback width of the encoded video.
kPropOutputHeight	UINT32	Playback height of the encoded video.
kPropResizeQuality	string	Video resize quality setting. Possible values are: kValueResizeQualityFast kValueResizeQualityHigh

(Table Page 2 of 2)

Audiences

An audience holds a set of streams for a particular target bit rate. The audience defines a combination of audio, video, events, and image map streams. Depending upon the type of input and the media profile settings, different streams in the audience will be used. For example, if the input is audio-only and audio format in the media profile specifies "music", then only an audio stream is generated using a music codec that takes up nearly all the target bit rate

To create an audience, call IHXTClassFactory::CreateInstance using the IID_IHXTAudience reference identifier. The following table contains the properties supported by audiences.

Audience Properties

Property	Type	Description
kPropObjectName	string	Descriptive name of object instance.
kPropAvgBitrate	UINT32	Average bitrate for CBR and VBR target bitrate encodes.
kPropMaxBitrate	UINT32	Maximum bitrate for VBR target bitrate and VBR quality mode encodes.

An audience enumerator is provided to look through a directory of audience template files and instantiate an audience object based on each settings file. To create an audience enumerator, call IHXTClassFactory::CreateInstance with IID_IHXTAudienceEnumerator.

Streams

Stream objects describe a single media stream for a particular datatype. There are four different types of streams supported in this SDK: audio, video, events, and image map. A single-rate audio/video presentation consists of one audio stream and one video stream. A SureStream audio/video presentation is made up of multiple audio and video stream pairs. An audience designates which streams are grouped together for a target bit rate and presentation type.

To create a stream property, call IHXTClassFactory::CreateInstance using the IID_IHXTStreamConfig reference identifier. The following tables contain the properties supported by audiences.

All Stream Properties

Property	Type	Description
kPropObjectName	string	Descriptive name of object instance. Possible values are: kValueAudioCodec.
kPropEncodingType	string	Type of encoding. Possible values are: kValueEncodingTypeCBR kValueEncodingTypeVBRBitrate kValueEncodingTypeVBRQuality

Audio Stream Properties

Property	Type	Description
kPropObjectName	string	Descriptive name of object instance.
kPropPluginType	string (required, initialization only)	Plug-in type. Possible values are: kValuePluginTypeAudioStream
kPropPluginName	string (initialization only)	Name of the plug-in to load. Possible values are: kValuePluginNameCodecRealAudio.
kPropCodecName	string	Codec name. Possible values are cook, atrc, sipr.
kPropCodecFlavor	UINT32	Codec flavor.
kPropEncodingComplexity	string	Controls the resulting size of the audio file by varying the complexity of the encoding algorithm. One of the following (from largest file size to smallest file size): kValueEncodingComplexityVeryFast kValueEncodingComplexityFast kValueEncodingComplexityNormal kValueEncodingComplexityHigh kValueEncodingComplexityVeryHigh
kPropAvgBitrate	UINT32 (read-only)	Average bitrate.
kPropMaxBitrate	UINT32 (read-only)	Maximum bitrate.
kPropStreamContext		Property bag used by the Helix DNA Producer SDK to select the audio stream if the audience contains multiple audio streams (the contents of which are shown in the following table).

Audio Stream Context Property Bag

Property	Type	Description
kPropAudioMode	string	Audio mode. Possible values are: kValueAudioModeMusic kValueAudioModeVoice
kPropPresentationType	string	Presentation type. Possible values are: kValuePresentationAudioOnly kValuePresentationAudioVideo

Video Stream Properties

Property	Type	Description
kPropPluginType	string (required, initialization only)	Plug-in type. Possible values are: kValuePluginTypeVideoStream
kPropPluginName	string (initialization only)	Name of plug-in to load. Possible values are: kValuePluginNameCodecRealVideo.
kPropCodecName	UINT32	Codec name. Possible values are: kValueCodecNameRV9 (RealVideo 9) kValueCodecNameRV8 (RealVideo 8) kValueCodecNameRV6 (RealVideo G2 with SVT)
kPropEncodingComplexity	string	Controls the resulting encoding quality by varying the complexity of the encoding algorithm. One of the following (from lowest quality to highest quality): kValueEncodingComplexityVeryFast kValueEncodingComplexityFast kValueEncodingComplexityNormal kValueEncodingComplexityHigh kValueEncodingComplexityVeryHigh
kPropAvgBitrate	UINT32	Average bitrate. Must be zero. Zero means the Helix DNA Producer SDK will calculate the bitrate based on the audience average bitrate and other active streams within an audience.
kPropMaxBitrate	UINT32	Maximum bitrate. Must be zero. Zero means the Helix DNA Producer SDK will calculate the bitrate based on the audience maximum bitrate and other active streams in an audience.
kPropEncodingQuality	double	Variable bit rate (VBR)-quality mode setting.
kPropMaxStartupLatency	UINT32	Maximum video preroll (initial buffering on playback).
kPropLossProtection	BOOL	Adds error correction codes to the stream.
kPropMaxOutputFrameRate	double	Maximum encoded frame rate.
kPropMaxTimeBetweenKeyFrames	double	Maximum amount of time between key frames.

Setting Up Optional Encoding Job Features

A basic encoding job can be enhanced by setting up additional encoding job features.

Metadata

Metadata (also known as clip info) is set on the encoding job object to describe the resulting output. RealPlayer then displays the title, author, and copyright metadata strings. You can also add your own custom metadata that you can extract and utilize in your own custom applications. This custom metadata is stored in the file header.

Metadata is stored in the encoding job object as a property bag. It can be accessed using the `IHXTEncodingJob::GetMetadata` and `IHXTEncodingJob::SetMetadata` methods. The following example demonstrates how to use metadata:

```
IHXTPropertyBag* pMetadata = NULL;
HX_RESULT res = m_pJob->GetMetadata(&pMetadata);

if (SUCCEEDED(res))
{
   pMetadata->SetBool(kPropAllowRecording, "FALSE");
   pMetadata->SetString(kPropTitle, szTitle);
   pMetadata->SetString(kPropKeywords, szKeywords);
   pMetadata->SetString(kPropDescription, szDescription);
   pMetadata->SetString(kPropAuthor, szAuthor);
   pMetadata->SetString(kPropCopyright, szCopyright);
   pMetadata->SetString("My Custom Data", szCustomData);
}
```

> **Note:** Although metadata is supplied in a property bag, only strings and UINTs are supported as metadata.

The sample demonstrates how to set values to specific properties supported as metadata. In addition, you can include any custom metadata (shown as "My Custom Data" in the sample). Although this custom metadata is not currently interpreted or displayed by the player, you can extract and use this metadata in your code.

Serialization

The serialization interfaces provided with the Helix DNA Producer SDK save the configuration settings of an object to a file or to a buffer in memory. These serialized settings can then be used later to recreate an object with exactly the same settings. The primary use for these interfaces is to serialize

job and audience files. These files are used to save job or audience settings beyond the lifetime of the application.

Two serialization interfaces are provided with the Helix DNA Producer SDK. The first, IHXTSerializeBuffer, serializes information to a buffer that can later be deserialized to create an exact copy of a specific object. The second, IHXTUserConfigFile, saves specific object information to a file. This file can then persist beyond the lifetime of the application using the object. Once the application is restarted, the specific object information can be deserialized from the file, recreating the settings from the previously saved information.

During serialization, you can also examine an object's individual property bag content, and modify or reject the serialization of the particular object you are examining. Both IHXTSerializeBuffer::WriteToBuffer and IHXTUserConfigFile::WriteToFile contain a pSerialCallback parameter that points to an IHXTSerializationCallback interface that lets you control what is being serialized. For example, each time an object with a property bag (such as inputs, outputs, prefilters, and so on) is serialized, the IHXTSerializationCallback::OnSerializeObject method is called, allowing you the chance to examine the contents of the property bag. You can then either modify the contents of the property bag being serialized using the pObject parameter, or prevent its serialization by setting the pbIsOkToSerialize parameter to FALSE. If the pSerialCallback parameter in either IHXTSerializeBuffer::WriteToBuffer or IHXTUserConfigFile::WriteToFile is set to NULL (the default), no callbacks occur.

Both the \producersdk\samples\encoder\encoder.cpp and the \producersdk\samples\mediasinkencoder\mediasinkencoder.cpp sample files demonstrate how to save the setup of an encoding job to a file using serialization.

Statistics

You can retrieve statistics from various objects in the encoding job using the IHXTStatistics interface. The statistics returned from the IHXTStatistics::GetCurrentStatistics method indicate the current status of the object. The statistics returned from the IHXTStatistics::GetLifeTimeStatistics method indicate the status of the object from the beginning of the encoding job.

> **Note:** Currently, only video and audio streams give out statistics.

The statistics are returned as a set of properties in a property bag. For example, statistics returned from a video stream include the average bit rate in bits per second, the average frames per second, the minimum frames per second, the average quality, the minimum quality, preroll, duration, and time. Statistics returned from an audio stream include the average bit rate in bits per second, preroll, duration, and time.

The duration statistic returned in the property bag refers to the duration of time for which the statistics apply.

The time statistic refers to the time stamp of the last sample for which statistics apply.

See the `\producersdk\samples\encoder\encoder.cpp` file for a demonstration of setting up the statistics interface, and how to update statistics.

Logging

Numerous log messages are provided throughout the Helix DNA Producer SDK. These log messages can be used to see what is happening inside the producer, and to determine the probable causes of any errors that occur.

To access these log messages, you must create an instance of the logging system, and use the logging system interfaces to observe the messages being sent from the producer.

For more information on using the logging system, see "Chapter 4: Logging System" beginning on page 95.

See the `\producersdk\samples\encoder\encoder.cpp` file for a demonstration of setting up the logging system.

Progress Events and Asynchronous Errors

Asynchronous events and errors are handled by the event system. To receive events, you must implement the `IHXTEventSink` interface and subscribe to the events through the event manager (`IHXTEventManager`). You can access the event manager by calling the `IHXTEncodingJob::GetEventManager` method on the job object. A pointer to the `IHXTEventSink` interface is passed to the `IHXTEventManager::Subscribe` method to set up your event observer.

The types of events you can receive are documented in the `ihxteventcodes.h` file located in the SDK's `include` directory. The encoding job sends out a progress event when encoding starts and stops, and during encoding with percent complete updates. The event that handles percent complete progress is

eEventEncodeProgress. In the case of the eEventEncodeProgress event, the puValue parameter passed to the IHXTEventSink::HandleEvent method is a pointer to a unsigned integer from 1 to 100 representing percent complete. The same is true for the eEventAnalysisProgress event, which gives you progress updates during the analysis phase of two-pass encoding mode.

Encoding RealMedia Events

RealMedia events can be added to an on-demand RealMedia file or inserted into a live broadcast stream during the encoding process using a media sink plug-in. An event consists of two components: the event type and the event value. All standard event types are supported, that is, URL, title, author, and copyright events. In addition, you can also create a custom event that is made up of any name/value pair.

> **Note:** To edit or add events to an existing RealMedia file, use the RealMedia edit APIs as described in "Chapter 6: RealMedia Edit API" beginning on page 147.

The following actions can be triggered by events:

- RealMedia URL events

  Causes the client to send URLs to the user's browser at a specified time when the stream is played.

- Title, Author, or Copyright events

  Changes the title, author or copyright information that can be displayed in a client.

- Custom events

  Initiates no pre-defined action in the player. Any action taken from a custom event must be programmed into the client using one of the client APIs, such as C++, Visual Basic, VBScript or JavaScript.

  > **For More Information:** See the *Helix Client and Server SDK Developer's Guide* for information on programming the client C++ APIs. See the *RealPlayer Scripting Guide* for information on programming RealPlayer in VBScript or JavaScript.

Event Components

The event value consists of a string that specifies the appropriate value for the event. The strings have the following syntax.

- URL - <FRAME>:<URL>

  <FRAME> is the name of a frame in which the event should be directed. <URL> is the URL to be passed to the browser.

  The format of the URL should not be assumed to follow any specific URL syntax and in fact can be any legal string accepted by a browser including "`JavaScript:`" syntax. This string can also contain multi-byte characters. The length is limited to 65 kilobytes (K) - 1 bytes.

- Title, Author and Copyright

  Strings which can contain any data up to 65K - 1 bytes in length. This information should generally be characters but should not be assumed to be single-byte characters.

- Custom

  Any string up to 65K - 1 characters. This information should generally be characters but should not be assumed to be single-byte characters.

Event Duration

Every event, in both on-demand and live streams, has a start and end time. With linear playback where there is no seeking on the timeline, such as during live broadcasting, events are triggered at the start time only. If a user seeks to a given point in the media with on-demand media, the event whose start time and end time include that event is triggered.

The event's end time can be specified either explicitly by setting a duration, or implicitly by leaving the duration as its default value of infinite. If a subsequent event is defined subsequently, the new event takes the place of the last event as opposed to adding to the last event. That is, events never overlap. This means that it is perfectly safe, and preferable, to always use infinite as the duration of an event if the desired behavior is for an event to last up until the next event is triggered.

The minimum event duration for URL and custom events is 200 milliseconds. If a URL or custom event contains a duration lower than 200 milliseconds, the duration is automatically modified by Helix DNA Producer to 200 milliseconds.

TAC events implicitly have an infinite duration, that is, the `pTimeEnd` parameter of the `IHXTMediaSample::SetTime` method is set to `0xFFFFFFFF`.

File to File Encoding Rules

The following rules describe how to insert events for file to file (on demand) encoding:

- Events should be added prior to the start of encoding. Inserting an event in a file to file encode after encoding has started is allowed but not advised. Because the encoding is not being performed in real-time, the timing of the event will therefore be unpredictable.
- Events must be added in order of increasing start time. Adding an event with a start time less than a previously-added event results in an error.
- Start time is specified explicitly. Start time is required if set prior to encode time. Not setting the start time before encoding is started results in an error.
- End time is computed from the duration. If the duration is not set, duration automatically defaults to infinity (duration can also be explicitly set to infinity by setting the `pTimeEnd` parameter of the `IHXTMediaSample::SetTime` method to `0xFFFFFFFF`). Subsequent events overwrite existing events, that is, if the end time of one event overlaps the start time of a subsequent event, the new event replaces the first event. For example given the following events:
  - Event1: StartTime=0, EndTime=60
  - Event2: StartTime=30, EndTime=60

  In this case, if the client seeks to time equals 45 seconds, only Event2 is triggered.

Live Encoding

The following rules describe how to insert events during capture input or broadcast output encoding:

- Events can be added prior to starting encoding or after encoding has started.
- Events must be added in order of increasing start time. Adding an event with a start time less than a previously added event results in an error.

- Start time must be specified explicitly in the call to add a RealMedia event. A method is provided to get the current timestamp. If the time for the event has passed, the method fails with an appropriate error.
- The end time must be defined explicitly in the call to add a RealMedia event. The end time can be set to either a timestamp after the start time or infinity.
- For URL and custom events, the minimum allowable value of duration is 200 milliseconds. If a URL or custom event contains a duration lower than 200 milliseconds, the duration is automatically modified by the SDK to 200 milliseconds.
- For TAC events, the minimum allowable duration is infinity. The duration for TAC events are always set to infinity.

URL Events

URL events provide a mechanism to trigger a web page to be loaded on playback. URL events can either be defined without a target frame, in which case the event loads into a new browser window, or it can be targeted to a specific frame in a browser or a specific pane in RealPlayer.

> **For More Information:** More information on RealMedia events can be found in Chapter 4 (Clip-Encoded URLs) of the *Introduction to Streaming Media* guide at **http://service.real.com/help/library/encoders.html**.

URL events are specified by setting the `nType` parameter of the `IHXTEventSample::SetAction` method to `RSEventMediaSample_URL`. The `pString` parameter of this method is then set to the URL. This URL is encoded into the RealMedia stream, delivered to the client, and is subsequently passed to the browser at the appropriate time.

URL events can be targeted either to a new browser window or to a specific frame of a multi-frame HTML page within the browser. In addition, URL events can be targeted to one of the three panes in the three-pane RealPlayer. By default, URL events open a new browser window. However, a frame can be specified to target the URL event to a specific frame of a multi-frame HTML document or to the related info or media browser panes of the RealPlayer.

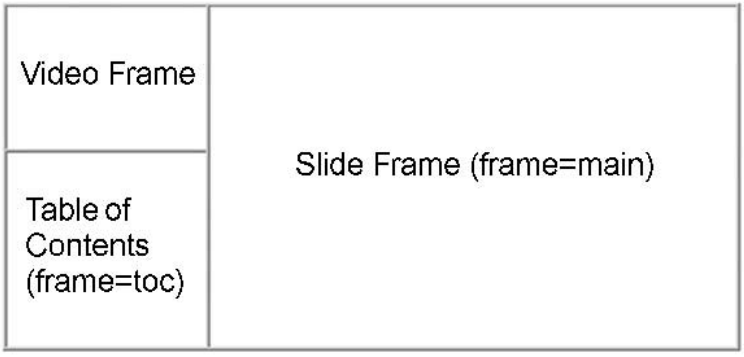
The following syntax targets a specific frame:

`&&`*`FRAME`*`&&`*`URL`*

In this example, *FRAME* is optional and identifies an HTML frame to be used as the target and *URL* is the URL that is to be loaded.

URL events can be used with an embedded client or in a stand-alone client.

When URL events are used with a presentation embedded within the browser, any frame, other than the current HTML frame, can be specified into which to load the URL, as shown in the following figure.



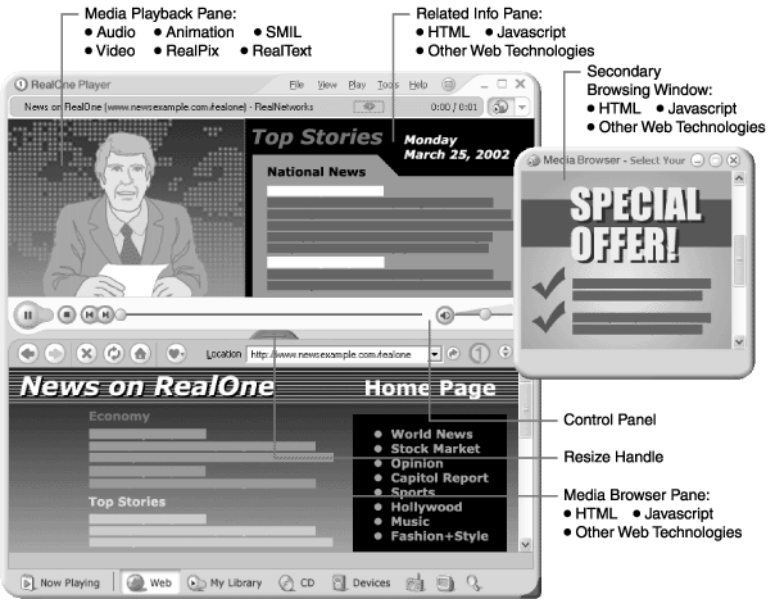
In this example, URL events containing URLs to slides to be synchronized with a presentation would be targeted to the Slide Frame (main). The `pString` value in `IHXTEventSample::SetAction` for this would be `&&main&&http://youserver.yourcompany.com/slides/slide1.htm`, where the string `main` between the double ampersands (&&) indicate that the URL should be loaded into the Slide Frame.

When URL events are used with a presentation playing in RealPlayer, any pane of the RealPlayer other than the media playback pane can be specified into which to load the URL.

RealPlayer is split into the following display regions:

- Media playback pane (playback of audio, video and other multimedia data types)
- Related info pane (display of related information)
- Media browser pane (display of related web pages)
- Secondary browser windows (other web pages displayed in a separate window)

The following figure shows the different panes of RealPlayer.



The related info pane, media browser pane, and the secondary browser windows can display HTML pages and other web technologies like Flash or Images. Anything that can be displayed in a browser can be loaded in either of these panes.

You can control which pane of the RealPlayer an event is loaded in by using the appropriate frame name for that pane. The following table lists the frame name for each of the relevant panes:

RealPlayer Frame Names

Pane	Frame Name
Media playback pane	NA (See below)
Related info pane	_rpcontextwin
Media browser pane	_rpbrowser
Secondary browsing window	_rpexternal

For example, to load a URL into the media browser pane, the `nType` parameter of the `IHXTEventSample::SetAction` method is set to `RSEventMediaSample_URL` and the `pString` value would be `&&_rpbrowser&&http://youserver.yourcompany.com/slides/slide1.htm`. The string

_rpbrowser between the double ampersands (&&) indicate that the URL should be loaded into the media browser pane.

You can also use the rpcontextheight and rpcontextwidth parameters in the URL. For more information on valid URL parameters, see the "Clip-Encoded URLs" chapter of *Introduction to Streaming Media with RealPlayer* at **http://service.real.com/help/library/encoders.html**.

For launching media, such as an audio or video clip, no frame name is required because media clips always play back in the media playback pane and there is no need to target that pane specifically. In some cases, audio or video content that would otherwise play back in the media playback pane can be loaded into the media browser pane by embedding that media in a separate HTML page. The separate HTML page can then be loaded in the media browser pane using the frame name identified in the RealPlayer Frame Names table.

Setting a RealMedia Event

The following steps demonstrate how to set up a media sink plug-in to provide a basic RealMedia event in your presentation.

1. Create a set of parallel inputs as described under "Creating Parallel Inputs" on page 38. One of these inputs can provide input for an audio/video source while the other will provide an input for the RealMedia event.

2. Set up the media sink plug-in by creating the property bag that contains the initialization parameters, setting the plug-in type, creating the input, and setting the input on the encoding job.

```
// Create the property bag used to initialize the input
IHXTPropertyBag* pInitParams = NULL;
if (SUCCEEDED(res))
   res = m_pFactory->CreateInstance(IID_IHXTPropertyBag,
   (IUnknown**)&pInitParams);

// Set the plugin type
if (SUCCEEDED(res))
   res = pInitParams->SetString(kPropPluginType,
   kValuePluginTypeInputMediaSink);

// Media samples generated in StartEncoding are not strictly real-time
if (SUCCEEDED(res))
   res = pInitParams->SetBool(kPropIsRealTime, FALSE);
```

```
// Create the input
IHXTInput* pInput = NULL;
if (SUCCEEDED(res))
   res = m_pFactory->BuildInstance(IID_IHXTInput, pInitParams,
   (IUnknown**)&pInput);
// Set the input on the encoding job
if (SUCCEEDED(res))
   res = pInputGroup->AddInput(pInput);
```

3. Get the event pin using the IHXTPropertyBag::GetUnknown method with the kPropEventInputPin property.

```
// Get the event pin
IUnknown* pUnk = NULL;
res = pInput->GetUnknown(kPropEventInputPin, &pUnk);
```

4. Query (IUnknown::QueryInterface) for the IHXTMediaInputPin interface.

```
res = pUnk->QueryInterface(IID_IHXTMediaInputPin, (void**)&m_pEventPin);
```

5. Enable the event pin by setting the bEnable parameter of the IHXTMediaInputPin::SetPinEnabled method to TRUE.

Once you have set up the media sink plug-in and have enabled the event pin, use the following steps to set a RealMedia event.

1. Create an instance of the IHXTEventSample interface using IHXTClassFactory::CreateInstance.

2. Set the event you want to occur using IHXTEventSample::SetAction. For example, if the action type in the nType parameter is set to RSEventMediaSample_URL, a web browser will be launched. You would also need to set the URL address as the pString parameter, as shown below:

```
res = pEventSample->SetAction(HXEventMediaSample_URL,
"http://www.real.com", NULL);
```

3. Optionally, you can set a start and ending time for the event by calling IHXTMediaSample::SetTime. The start and end times are given in milliseconds, for example:

```
res = pEventSample->SetTime(0, 6000);
```

4. Encode the event using the IHXTMediaInputPin::EncodeSample method.

```
res = m_pEventPin->EncodeSample( pEventSample );
```

5. Release the IHXTEventSample interface.

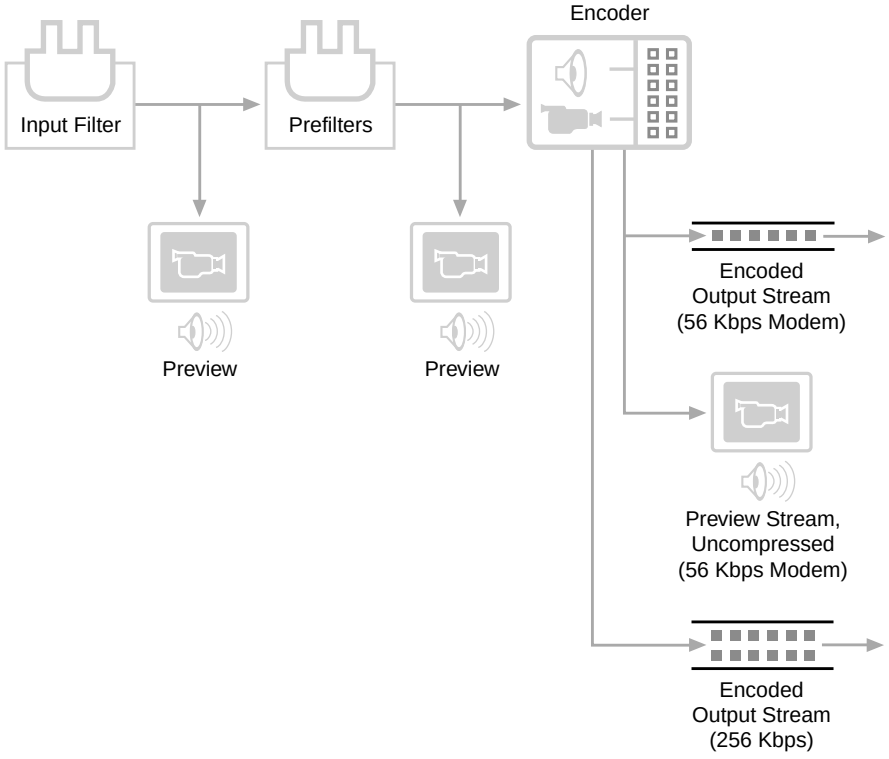
The \samples\mediasinkencoder\mediasinkencoder.cpp file included with the Helix DNA Producer SDK provides an example implementation of RealMedia events. This sample includes setting up a basic URL event, and also demonstrates how to create a custom event.

Audio and Video Preview

You can register to receive preview audio or video samples from an encoding filter graph at three different positions in the graph:

- Immediately after the input source.
- Immediately before the encoder.
- Immediately after the encoder.

The following figure demonstrates the positions where callers to audio and video preview can plug into the filter graph.



After Input Source

To register to receive preview samples after the input source, callers must perform the following steps:

1. Implement the IHXTPreviewSink interface on the object that receives callbacks with the preview samples.
2. Find out the optimal preview settings, that is, the least CPU-intensive format settings to request. You can choose to use the optimal settings or modify the settings (for example, to resize the video to a larger window). Use the following steps to get the optimal preview settings after the input source:
   a. Query the IHXTInput interface for the IHXTPreviewSinkControl interface.
   b. Create an IHXTPropertyBag.
   c. Call the IHXTPreviewSinkControl::GetOptimalSinkProperties method with a pointer to the IHXTPropertyBag interface you created in the previous step. This property bag is populated with all the optimal settings. Refer to the ihxtpreviewsink.h header file for a description of these returned properties.
3. Set the position of the preview in the filter graph. Call the IHXTPropertyBag::SetUint method with the kPropPreviewSinkPosition property and kValueBeforeAllPrefilters value.
4. Set how often you want to receive media samples. Currently you can only receive all media samples (kValueAllSamples) or the first sample (kValueFirstSample). Call the IHXTPropertyBag::SetInt method with the kPropSinkUpdateInterval property, and either the kValueFirstSample or kValueAllSamples value.
5. In addition, for video preview you can set the video width (kPropVideoFrameWidth), height (kPropVideoFrameHeight), and color format (kPropVideoColorFormat) you would like to preview:
   a. Call the IHXTPropertyBag::SetUint method with the kPropVideoFrameWidth property and the width you would like to preview.
   b. Call the IHXTPropertyBag::SetUint method with the kPropVideoFrameHeight property and the height you would like to preview.

c. Call the IHXTPropertyBag::SetUint method with the kPropVideoColorFormat property and the color format you would like to preview. See ihxtconstants.h for a list of possible color formats.

For audio preview you can set the sample rate (kPropAudioSampleRate), sample size (kPropAudioSampleFormat), and number of channels (kPropAudioChannelFormat) you would like to preview:

a. Call the IHXTPropertyBag::SetUint method with the kPropAudioSampleRate property and the sample rate you would like to preview.

b. Call the IHXTPropertyBag::SetUint method with the kPropAudioSampleFormat property and the sample format you would like to preview.

c. Call the IHXTPropertyBag::SetUint method with the kPropAudioChannelFormat property and the channel format you would like to preview.

6. Register as a preview sink. Call the IHXTPreviewSinkControl::AddSink with a pointer to the IHXTPreviewSink interface used to receive the preview media samples, and a pointer to the IHXTPropertyBag interface that manages the settings for the preview.

7. If you're not encoding yet and want to preview the input, call the IHXTInputPreviewControl::Open method. If you don't want to preview before encoding, you'll begin receiving frames once you start encoding.

You can also enable and disable preview sampling after the input source during an encoding process. To enable preview during run time, first query (IUnknown::QueryInterface) IHXTInput for the IHXTPreviewSinkControl3 interface. Call IHXTPreviewSinkControl3::EnableSink, passing in the pointer to the preview sink to which you want to send the preview samples. To disable preview during run time, first query (IUnknown::QueryInterface) IHXTInput for the IHXTPreviewSinkControl3 interface. Next, call IHXTPreviewSinkControl3::DisableSink, passing in the pointer to the preview sink to which you want to stop sending the preview samples.

Before the Encoder

To register to receive preview samples after all prefilters, but before the encoder, callers must perform the following steps:

1. Implement the IHXTPreviewSink interface on the object that receives callbacks with the preview samples.
2. Find out the optimal preview settings, that is, the least CPU-intensive format settings to request. You can choose to use the optimal settings or modify the settings (for example, to resize the video to a larger window). Use the following steps to get the optimal preview settings after the last prefilter:
   a. Query the IHXTInput interface for the IHXTPreviewSinkControl interface.
   b. Create an IHXTPropertyBag.
   c. Call the IHXTPreviewSinkControl::GetOptimalSinkProperties method with a pointer to the IHXTPropertyBag interface you created in the previous step. This property bag is populated with all the optimal settings. Refer to the ihxtpreviewsink.h header file for a description of these returned properties.
3. Set the position of the preview in the filter graph. Call the IHXTPropertyBag::SetUint method with the kPropPreviewSinkPosition property and kValueAfterAllPrefilters value.
4. Set how often you want to receive media samples. Currently you can only receive all media samples (kValueAllSamples) or the first sample (kValueFirstSample). Call the IHXTPropertyBag::SetInt method with the kPropSinkUpdateInterval property, and either the kValueFirstSample or kValueAllSamples value.
5. In addition, for video preview you can set the video width (kPropVideoFrameWidth), height (kPropVideoFrameHeight), and color format (kPropVideoColorFormat) you would like to preview:
   a. Call the IHXTPropertyBag::SetUint method with the kPropVideoFrameWidth property and the width you would like to preview.
   b. Call the IHXTPropertyBag::SetUint method with the kPropVideoFrameHeight property and the height you would like to preview.
   c. Call the IHXTPropertyBag::SetUint method with the kPropVideoColorFormat property and the color format you would like to preview. See ihxtconstants.h for a list of possible color formats.

For audio preview you can set the sample rate (kPropAudioSampleRate), sample size (kPropAudioSampleFormat), and number of channels (kPropAudioChannelFormat) you would like to preview:

a. Call the IHXTPropertyBag::SetUint method with the kPropAudioSampleRate property and the sample rate you would like to preview.

b. Call the IHXTPropertyBag::SetUint method with the kPropAudioSampleFormat property and the sample format you would like to preview.

c. Call the IHXTPropertyBag::SetUint method with the kPropAudioChannelFormat property and the channel format you would like to preview.

6. Register as a preview sink. Call the IHXTPreviewSinkControl::AddSink with a pointer to the IHXTPreviewSink interface used to receive the preview media samples, and a pointer to the IHXTPropertyBag interface that manages the settings for the preview.

7. If you're not encoding yet and want to preview the prefiltered frames, call the IHXTInputPreviewControl::Open method. If you don't want to preview before encoding, you'll begin receiving frames once you start encoding.

You can also enable and disable preview sampling before the encoder during an encoding process. To enable preview during run time, first query (IUnknown::QueryInterface) IHXTInput for the IHXTPreviewSinkControl3 interface. Call IHXTPreviewSinkControl3::EnableSink, passing in the pointer to the preview sink to which you want to send the preview samples. To disable preview during run time, first query (IUnknown::QueryInterface) IHXTInput for the IHXTPreviewSinkControl3 interface. Next, call IHXTPreviewSinkControl3::DisableSink, passing in the pointer to the preview sink to which you want to stop sending the preview samples.

Just before the encoding filter graph starts, Helix DNA Producer calls IHXTPreviewSink::OnFormatChanged with a property bag that contains the properties of the media samples about to be encoded. This information is useful because the format could have been changed, depending on what prefilters are present and their settings. Because all the properties are known about an input source before encoding begins, this is only relevant for preview sinks placed just before or just after the encoder.

If for some reason you need to be sure of the media dimensions before encoding begins, you can inspect the prefilters to determine what the settings would be at various points in the graph. Practically speaking, the only prefilter that would affect these preview properties is the cropping filter.

To look for the cropping filter and determine its output dimensions:

1. Call `IHXTInput::GetPrefilterCount` to find out the number of prefilters.
2. Iterate though all the prefilters by calling `IHXTInput::GetPrefilter` with the `ulIndex` parameter set from zero (0) to `GetPrefilterCount-1`.
3. On each prefilter, call `IHXTPrefilter::GetString` with the property `kPropPluginName` to retrieve the prefilter's name.
4. If you find a prefilter with the plug-in name of `kValuePluginNamePrefilterCropping`, call `IHXTPrefilter::GetBool` with the property `kPropIsEnabled`.
5. If the result of calling `IHXTPrefilter::GetBool` is true, then call `IHXTPrefilter::GetUint` with the property `kPropCropWidth`, and again with `kPropCropHeight` to get the cropped dimensions. The dimensions returned by these calls will most likely be the dimensions of the "just before the encoder" preview frames once encoding has begun.

After the Encoder

To register to receive preview samples after passing through the encoder, callers must perform the following steps:

1. Call `IHXTEncodingJob::GetOutputProfile` to get a pointer to the `IHXTOutputProfile` interface associated with the current job.
2. Call `IHXTOutputProfile::GetMediaProfile` to get a pointer to the `IHXTMediaProfile` interface that manages the media profile.
3. Call `IHXTMediaProfile::GetAudience` with the index to the audience you want to preview.
4. Query the retrieved audience for `IHXTPreviewSinkControl`.
5. Implement the `IHXTPreviewSink` interface on the object that receives callbacks with the preview samples.
6. Find out the optimal preview settings, that is, the least CPU-intensive format settings to request. You can choose to use the optimal settings or modify the settings (for example, to resize the video to a larger window).

Use the following steps to get the optimal preview settings after the encoder:

a. Query the IHXTInput interface for the IHXTPreviewSinkControl interface.

b. Create an IHXTPropertyBag.

c. Call the IHXTPreviewSinkControl::GetOptimalSinkProperties method with a pointer to the IHXTPropertyBag interface you created in the previous step. This property bag is populated with all the optimal settings. Refer to the ihxtpreviewsink.h header file for a description of these returned properties.

7. Set the position of the preview in the filter graph. Call the IHXTPropertyBag::SetUint method with the kPropPreviewSinkPosition property and kValueAfterCodec value.

8. Set how often you want to receive media samples. Currently you can only receive all media samples (kValueAllSamples) or the first sample (kValueFirstSample). Call the IHXTPropertyBag::SetInt method with the kPropSinkUpdateInterval property, and either the kValueFirstSample or kValueAllSamples value.

9. In addition, for video preview you can set the video width (kPropVideoFrameWidth), height (kPropVideoFrameHeight), and color format (kPropVideoColorFormat) you would like to preview:

   a. Call the IHXTPropertyBag::SetUint method with the kPropVideoFrameWidth property and the width you would like to preview.

   b. Call the IHXTPropertyBag::SetUint method with the kPropVideoFrameHeight property and the height you would like to preview.

   c. Call the IHXTPropertyBag::SetUint method with the kPropVideoColorFormat property and the color format you would like to preview. See ihxtconstants.h for a list of possible color formats.

   For audio preview you can set the sample rate (kPropAudioSampleRate), sample size (kPropAudioSampleFormat), and number of channels (kPropAudioChannelFormat) you would like to preview:

   a. Call the IHXTPropertyBag::SetUint method with the kPropAudioSampleRate property and the sample rate you would like to preview.

   b. Call the IHXTPropertyBag::SetUint method with the kPropAudioSampleFormat property and the sample format you would like to preview.

   c. Call the IHXTPropertyBag::SetUint method with the kPropAudioChannelFormat property and the channel format you would like to preview.

10. Register as a preview sink. Call the IHXTPreviewSinkControl::AddSink with a pointer to the IHXTPreviewSink interface used to receive the preview media samples, and a pointer to the IHXTPropertyBag interface that manages the settings for the preview.

You can also enable and disable preview sampling after the encoder during an encoding process. To enable preview during run time, first query (IUnknown::QueryInterface) IHXTAudience for the IHXTPreviewSinkControl3 interface. Call IHXTPreviewSinkControl3::EnableSink, passing in the pointer to the preview sink to which you want to send the preview samples. To disable preview during run time, first query (IUnknown::QueryInterface) IHXTAudience for the IHXTPreviewSinkControl3 interface. Next, call IHXTPreviewSinkControl3::DisableSink, passing in the pointer to the preview sink to which you want to stop sending the preview samples.

If for some reason you need to be sure of the media dimensions before encoding begins, you can inspect the prefilters and the media profile to determine what the settings would be at various points in the graph. Practically speaking, the only prefilter that would affect these preview properties is the cropping filter.

To look for the cropping filter and determine its output dimensions:

1. Call IHXTInput::GetPrefilterCount to find out the number of prefilters.
2. Iterate though all the prefilters by calling IHXTInput::GetPrefilter with the ulIndex parameter set from zero (0) to GetPrefilterCount-1.
3. On each prefilter, call IHXTPrefilter::GetString with the property kPropPluginName to retrieve the prefilter's name.
4. If you find a prefilter with the plug-in name of kValuePluginNamePrefilterCropping, call IHXTPrefilter::GetBool with the property kPropIsEnabled.
5. If the result of calling IHXTPrefilter::GetBool is true, then call IHXTPrefilter::GetUint with the property kPropCropWidth, and again with

kPropCropHeight to get the cropped dimensions. The dimensions returned by these calls will most likely be the dimensions of the "just before the encoder" preview frames once encoding has begun.

In addition to looking for prefilters that might affect the media dimensions, you also have to check the media profile to determine if the encoded output has been resized. Call IHXTMediaProfile::GetUint with the properties kPropOutputWidth and kPropOutputHeight to get the resize dimensions. If both the width and height returned are zero, then only the prefilters could affect the size. If both the width and height are non-zero, then these values are the dimensions of the encoded output.

If only one of the values for the width or height dimensions is zero but the other is non-zero, the output will be automatically resized to maintain the input source's or cropped sources' aspect ratio.

Use the following code sample to calculate the size of an unknown height dimension (that is, if the value for the height dimension was zero):

```
UINT32 calculateHeight(UINT32 ulOutputWidth, UINT32 ulInputWidth, UINT32
ulInputHeight)
{
     float fOutputHeight = (((float) ulInputHeight / (float) ulInputWidth) * (float)
ulOutputWidth );

      // Round height up to nearest whole integer
      UINT32 ulOutputHeight = (UINT32) fOutputHeight +0.5;

      // Round height to nearest multiple of 4
      ulOutputHeight = nearestModulus4(ulOutputHeight);
}
```

Use the following code sample to calculate the size of an unknown width dimension (that is, if the value for the width dimension was zero):

```
UINT32 calculateWidth(UINT32 ulOutputHeight, UINT32 ulInputWidth, UINT32
ulInputHeight)
{
     float fOutputWidth = (((float) ulInputWidth / (float) ulInputHeight) * (float)
ulOutputHeight );

      // Round width up to nearest whole integer
      UINT32 ulOutputWidth = (UINT32) fOutputWidth +0.5;

      // Round width to nearest multiple of 4
```

```
    ulOutputWidth = nearestModulus4(ulOutputWidth);

    return ulOutputWidth;
}
```

Capture Device Manager and Capture Device Enumeration

This section describes how to get information about capture devices using the Helix DNA Producer SDK.

1. Create an instance of the `CaptureDeviceInfoManager` class using the `IHXTClassFactory::CreateInstance` interface, as shown in the following example:

   ```
   IUnknown*pICaptureDeviceInfoUnknown;
   res = m_pFactory->CreateInstance(CLSID_IHXTCaptureDeviceInfoManager,
       (IUnknown**)&pICaptureDeviceInfoUnknown);
   ```

2. Query `CaptureDeviceInfoManager` for its `IHXTPluginInfoManager` interface, as shown in the following example:

   ```
   IHXTPluginInfoManager* pICaptureDeviceInfoManger = NULL;
   res = pICaptureDeviceInfoUnknown->
      QueryInterface(IID_IHXTPluginInfoManager, &pICaptureDeviceInfoManger);
   ```

3. Use the `IHXTPluginInfoManager::GetPluginInfoEnum` method to obtain a pointer to an `IHXTPluginInfoEnum` interface that manages the capture device information you are requesting. Use this method's `pIQueryPropertyBag` parameter to narrow the list of capture devices that will be enumerated by the `IHXTPluginInfoEnum` interface. Passing `NULL` to `pIQueryPropertyBag` will return information about all the capture devices.

4. Use the `IHXTPluginInfoEnum::GetCount` method to get the number of available capture devices matching your query, then use the `IHXTPluginInfoEnum::GetPluginInfoAt` method to retrieve the specific capture device information for each capture device that matches your query.

The following example demonstrates how to retrieve all available capture devices installed in Helix DNA Producer:

```
IHXTPluginInfoEnum *pIPluginInfoEnum;
pICaptureDeviceInfoManger->GetPluginInfoEnum(NULL, &pIPluginInfoEnum);

IHXTPropertyBag *pCaptureDeviceInfoBag;
```

```
for (UINT32 i=0; i<pIPluginInfoEnum->GetCount(); i++)
{
   pIPluginInfoEnum->GetPluginInfoAt(i,&pCaptureDeviceInfoBag);
}
```

The next example demonstrates how to retrieve all available audio capture devices installed in Helix DNA Producer:

```
pIQueryBag->
   SetString( kPropCaptureMediaType, kValueCaptureMediaTypeAudioCapture );

pICaptureDeviceInfoManger->GetPluginInfoEnum(pIQueryBag, &pIPluginInfoEnum);

IHXTPropertyBag *pCaptureDeviceInfoBag;
for (UINT32 i=0; i<pIPluginInfoEnum->GetCount(); i++)
{
   pIPluginInfoEnum->GetPluginInfoAt(i,&pCaptureDeviceInfoBag);
}
```

The next example demonstrates how to retrieve all available RealNetworks video capture devices installed in Helix DNA Producer:

```
pIQueryBag->
   SetString( kPropCaptureMediaType, kValueCaptureMediaTypeVideoCapture);

pICaptureDeviceInfoManger->GetPluginInfoEnum(pIQueryBag, &pIPluginInfoEnum);

IHXTPropertyBag *pCaptureDeviceInfoBag;
for (UINT32 i=0; i<pIPluginInfoEnum->GetCount(); i++)
{
   pIPluginInfoEnum->GetPluginInfoAt(i,&pCaptureDeviceInfoBag);
}
```

> **Note:** Be sure to release (`IUnknown::Release`) the `pCaptureDeviceInfoBag` that you get from `IHXTPluginInfoEnum::GetPluginInfoAt` to avoid memory leaks.

The following table contains the properties returned by capture devices in the ppIPluginInfo parameter of the IHXTPluginInfoEnum::GetPluginInfoAt method.

Capture Device Properties

Property	Type	Description
kPropCaptureType	string	Capture device type. Possible values are: kValuePluginNameCaptureAV
kPropVideoDeviceID	string	Video capture device identification.
kPropAudioDeviceID	string	Audio capture device identification.
kPropCaptureMediaType	string	Capture media type. Possible values are: kValueCaptureMediaTypeVideoCapture kValueCaptureMediaTypeAudioCapture
kPropCapturePorts	property bag	Contains a series of index numbers used as keys for each port, and a string value indicating the port name for each key. For example, the index could be "0" and the port name "Line-In". This property lists all the devices and ports.

Automatic Codec Selection

The Helix DNA Producer SDK, beginning with version 10.0, provides a mechanism for automatically selecting codecs to be loaded and used in your application. In previous versions, a codec could only be identified if either a plug-in name and plug-in type were given, in which case there was no ambiguity about which codec to load, or by plug-in type only. If only the plug-in type was supplied, Helix DNA Producer would search for a plug-in that supports that specified plug-in type. This was a convenience for users or developers since previous versions of Helix DNA Producer contained only RealAudio and RealVideo codecs.

However, as additional codecs are added to Helix DNA Producer, applications require a mechanism to easily support new codecs without requiring significant changes to the application. Automatic codec selection gives your application the ability to distinguish between various audio and video codecs by using additional plug-in properties, and without the specific knowledge of the plug-in name.

Codec Requirements

Before Helix DNA Producer can automatically select your codecs, you must make a minor modification to the codec properties and place your codec plug-

in in the proper Helix DNA Producer directory. Optionally if you are updating a previously-existing codec, you need to modify the codec mapping file to map your original codec to the new codec.

For Helix DNA Producer to automatically select your codecs, your codec must provide a codec name (`kPropCodecName` property) and a plug-in type (`kPropPluginType` property). Although explicitly setting the codec plug-in name (`kPropPluginName` property) is still supported by Helix DNA Producer, providing the codec name offers a more generic means of specifying the codec to use independent of any single codec plug-in.

Each codec plug-in must define what codec names it can handle and the plug-in type. The codec plug-in can also specify a priority. Priority determines which plug-in to use when two or more plug-ins support the same codec name. Priority ranking is set from 0 to 100, with 0 being the lowest priority and 100 being the highest, with 80 being the normal value. The plug-in with the highest priority is used to write the output. If no priority is specified, a value of 0 is used.

All codecs can also use a wildcard (*) and a priority for the wildcard to indicate they will attempt to handle any codec extension. Priorities for a wildcard should be such that these are tried as a last resort to minimize the number of failed attempts at loading a codec.

The following table documents the priorities given to the codec plug-ins that ship with Helix DNA Producer.

Codec Plug-in Priorities

Plug-in	Extensions	Priority (0-100)
rn-audiocodec-realaudio	*	80
	cook	80
	sipr	80
	raac	80
	racp	80
rn-videocodec-realvideo	*	80
	rv8	80
	rv9	80
	rv10	80
rn-file-ogg	vorbis	80

The wildcard (*) matches any file extension.

Modifying the Codec Mapping Table

If your new codec is replacing a previously-developed codec, you can map the new codec to the old codec by modifying the codec mapping table contained in the codec mapping file. The codec mapping file is a text file named `codecmapping.txt` located in the/tools directory of the Helix DNA Producer SDK. The file is comma delimited and consists of five columns, as shown in the follow table:

Column	Contents
1	Stream type
2	Old codec name
3	Old codec flavor
4	New codec name
5	New codec flavor

For video, the codec flavor column is left blank.

Blank lines or lines starting with the # character are ignored. The file must contain the correct number of strings separated by commas on each line or it will be rejected. If any codec ID is not identified as a valid codec, the line is rejected and ignored.

The following example shows the syntax of the codec mapping file:

```
# Video codec mappings (old,,new,)
videostream,rvg2svt,,rv8

# Audio Codec mappings
# old-name,old-flavor,new-name,new-flavor
audiostream,atrc,0,cook,24
audiostream,atrc,1,cook,25
audiostream,atrc,2,cook,25
audiostream,atrc,3,raac,2
audiostream,atrc,4,raac,2
audiostream,atrc,5,raac,3
audiostream,atrc,6,raac,5
audiostream,atrc,7,raac,6
audiostream,atrc,8,raac,7
audiostream,atrc,9,raac,7
audiostream,atrc,10,raac,8
audiostream,atrc,11,raac,10
audiostream,atrc,12,raac,11
```

Directory Placement

Once you have completed your new codec, the resulting plug-in must be placed in the producer's /tools directory. The information Helix DNA Producer obtains from querying the codecs in the /tools directory determines which codec should be selected for a given audio or video stream.

> **Note:** Do not place codecs intended to be loaded by the automatic codec selection mechanism in the producer's /codecs directory. Codecs located in the codecs directory are intended to be loaded by the RealAudio or RealVideo codec wrappers (which themselves advertise a number of codec names they support).

Codec Manager and Codec Enumeration

The codec manager provides enumeration capabilities that gather information about the codecs installed in Helix DNA Producer. (For information on automatically loading and using codecs in the producer, see "Automatic Codec Selection" on page 82.) The following steps describe how to obtain the codec information:

1. Create an instance of the CodecInfoManager class using the IHXTClassFactory::CreateInstance interface, as shown in the following example:

   ```
   IUnknown*pICodecInfoUnknown;
   res = m_pFactory->CreateInstance(CLSID_IHXTCodecInfoManager,
       (IUnknown**)&pICodecInfoUnknown);
   ```

2. Query CodecInfoManager for its IHXTPluginInfoManager interface, as shown in the following example:

   ```
   IHXTPluginInfoManager* pICodecInfoManger = NULL;
   res = pICodecInfoUnknown->QueryInterface(IID_IHXTPluginInfoManager,
       &pICodecInfoManger);
   ```

3. Use the IHXTPluginInfoManager::GetPluginInfoEnum method to obtain a pointer to an IHXTPluginInfoEnum interface that manages the codec information you are requesting. Use this method's pIQueryPropertyBag parameter to narrow the list of codecs that will be enumerated by the IHXTPluginInfoEnum interface. Passing NULL to pIQueryPropertyBag will return information about all the codecs.

4. Use the IHXTPluginInfoEnum::GetCount method to get the number of available codecs matching your query, then use the IHXTPluginInfoEnum::GetPluginInfoAt method to retrieve the specific codec information for each codec that matches your query.

The following example demonstrates how to retrieve all available codecs installed in Helix DNA Producer:

```
IHXTPluginInfoEnum *pIPluginInfoEnum;
pICodecInfoManger->GetPluginInfoEnum(NULL, &pIPluginInfoEnum);

IHXTPropertyBag *pCodecInfoBag;
for (UINT32 i=0; i<pIPluginInfoEnum->GetCount(); i++)
{
   pIPluginInfoEnum->GetPluginInfoAt(i,&pCodecInfoBag);
}
```

The next example demonstrates how to retrieve all available audio codecs installed in Helix DNA Producer:

```
pIQueryBag->SetString( kPropPluginType, kValuePluginTypeAudioStream );

pICodecInfoManger->GetPluginInfoEnum(pIQueryBag, &pIPluginInfoEnum);

IHXTPropertyBag *pCodecInfoBag;
for (UINT32 i=0; i<pIPluginInfoEnum->GetCount(); i++)
{
   pIPluginInfoEnum->GetPluginInfoAt(i,&pCodecInfoBag);
}
```

The next example demonstrates how to retrieve all available RealNetworks video codecs installed in Helix DNA Producer:

```
pIQueryBag->SetString( kPropPluginType, kValuePluginTypeVideoStream);
pIQueryBag->SetString( kPropPluginName, kValuePluginNameCodecRealVideo);

pICodecInfoManger->GetPluginInfoEnum(pIQueryBag, &pIPluginInfoEnum);

IHXTPropertyBag *pCodecInfoBag;
for (UINT32 i=0; i<pIPluginInfoEnum->GetCount(); i++)
{
   pIPluginInfoEnum->GetPluginInfoAt(i,&pCodecInfoBag);
}
```

The following table contains the properties returned by all codecs in the ppIPluginInfo parameter of the IHXTPluginInfoEnum::GetPluginInfoAt method.

All Codec Properties

Property	Type	Description
kPropPluginType	string (required, initialization only)	Plug-in type. Possible values are: kValuePluginTypeAudioStream kValuePluginTypeVideoStream
kPropPluginName	string (initialization only)	Name of the plug-in to load. Possible values are: kValuePluginNameCodecRealVideo kValuePluginNameCodecRealAudio third-party codec name
kPropCodecName	string	Codec name.

The following table contains the properties returned by audio codecs.

Audio Codec Properties

Property	Type	Description
kPropPluginType	string (required, initialization only)	Plug-in type. Possible values are: kValuePluginTypeAudioStream
kPropPluginName	string (initialization only)	Name of the plug-in to load. Possible values are: kValuePluginNameCodecRealAudio third-party codec name
kPropCodecName	string	Codec name.
kPropCodecFlavor	UINT32	Codec flavor.
kPropAudioSampleRate	UINT32	Audio sample rate.
kPropAudioSampleFormat	UINT32	Audio sample format. One of the sample formats specified by the EHXTAudioSampleFormat enumerator in ihxtaudioformat.h.
kPropAudioChannelFormat	UINT32	Audio channel format. One of the channel formats specified by the EHXTAudioChannelFormat enumerator in ihxtaudioformat.h.
kPropAvgBitrate	UINT32	Average bit rate in bits per second.
kPropMaxBitrate	UINT32	Maximum bit rate in bits per second.
kPropCodecPreferedType	string	The preferred type of audio. Possible values are: kValueAudioFormatMusic kValueAudioFormatVoice
kPropCodecLongName	string	The long name of the codec.

The following table contains the properties returned by video codecs.

Video Codec Properties

Property	Type	Description
kPropPluginType	string (required, initialization only)	Plug-in type. Possible values are: kValuePluginTypeVideoStream
kPropPluginName	string (initialization only)	Name of the plug-in to load. Possible values are: kValuePluginNameCodecRealVideo third-party codec name
kPropCodecName	string	Codec name.
kPropCodecLongName	string	The long name of the codec.

Load Management

Load management is the process of enabling plug-ins to scale back resources when Helix DNA Producer is working under overloaded conditions during real-time encoding scenarios (capture input or broadcast output). Under non-real-time encoding conditions (file to file) the load management system is disabled and your plug-in will receive no load management updates.. System resource availability is determined by Helix DNA Producer, and is reported to codecs and plug-ins that require intensive processing.

Plug-ins implementing the IHXTLoadAdjustment interface will receive information about the current load level from Helix DNA Producer. Your plug-in should then modify its behavior to reduce its use of system resources as the load level gets lower, or increase its system resources as the load level gets higher. The load level is reported to your plug-in as a number from 0 to 100.

If your plug-in supports and exposes the IHXTLoadAdjustment interface, it will automatically be connected to the Helix DNA Producer's load management system. As such, no calls need to be made to subscribe to receive the current load level. This also means that the plug-in is never unsubscribed and continues to receive load level updates for as long as data is being processed.

The IHXTLoadAdjustment::SetLoadLevel method is called by the producer's load management system to notify the plug-in that the current load level has changed. Upon receiving this call, the plug-in should attempt to adjust its processing to the percentage specified in the method's uLoadLevel parameter. For example, if the uLoadLevel value is 100, the plug-in should do as much processing as it normally would normally do under ideal circumstances. By

contrast, if uLoadLevel is 75, the plug-in should attempt to do 75 percent of the work it would normally do. Plug-ins should apply all performance reduction techniques before reaching a load level of 40. At a load level of 40, the load management system will begin to drop video frames in a last-ditch effort to reduce perofrmance to a sustainable level. Therefore, plug-ins should not themselves ever drop video frames. Also, a plug-in should not drop audio samples in an attempt to scale back performance since doing so impairs the player's ability to play the content. This method must be implemented in plug-ins that require load management reports from Helix DNA Producer. The producer's load management system does not take any additional action if this method fails, and will continue to call this method.

If you are implementing load management in your plug-in, the IHXTLoadAdjustment::GetLoadLevel method is optional. If your plug-in implements this method, it reports the location of the current load level to the caller. If this method is not implemented, it must return HXR_NTOIMPL.

Starting and Shutting Down the Encoding Job

Once you have set up the encoding job and all of its child processes, you start the encoding job by calling the IHXTEncodingJob::StartEncoding method. To shut down the encoding job, release (IUnknown::Release) all of the interfaces you used for your encoding job, and close all DLL connections.

You can also cancel or stop the encoding job at any time. If you cancel the encoding job using the IHXTEncodingJob::CancelEncoding method, no output files are created and encoding that occurred up to the time the job was cancelled is lost. If you stop the encoding job using the IHXTEncodingJob::StopEncoding method during a single-pass encoding job, all of the encoded data up to the point the encoding job was stopped is written to the output file. However, if you use this method to stop a two-pass encoding job during the first pass, no data is saved (since none has been encoded).

SDK Threading Model

In general, the Helix DNA Producer SDK interfaces and their methods are not thread-safe. The caller is responsible for serializing all calls into the SDK. For example, one thread cannot use the IHXTEncodingJob interface while another thread simultaneously calls the same encoding job object, or another object

associated with the encoding job (such as a media profile, a destination, and so on).

The follow methods are thread-safe and can be called simultaneously:

- `IHXTEncodingJob::StartEncoding`
- `IHXTEncodingJob::StopEncoding`
- `IHXTEncodingJob::CancelEncoding`

For example, it is possible to call the blocking version of `IHXTEncodingJob::StartEncoding`, and call `IHXTEncodingJob::StopEncoding` with a different thread.

It is acceptable to use multiple threads to make calls to objects that are not associated with each other. For example, if two encoding jobs are running, two different threads can be used to simultaneously call the SDK (that is, one thread for each encoding job).

Any callback interface (events, logging, preview, and so on) can call back on separate threads. For example, all events for a job can call back to the event sink using one thread while all log messages for a job can call back using a different thread. One important consequence of this is that if an SDK user's callback handler code ends up calling the Helix DNA Producer SDK interfaces and their methods, those calls must be serialized with all other calls into the SDK. Calls to callback interfaces will be serialized for a particular job. For example, an event sink will never receive more than one event callback from a particular job at a time. Of course, if a callback interface is registered for callbacks spanning multiple jobs, it is possible to receive simultaneous callbacks.

Helix DNA Producer SDK users must not release their last reference count on a job in their event handlers. Doing so will cause the job to shut down and wait for the event thread to exit, which will cause a deadlock.

Blocking a callback thread while waiting for an SDK method to complete can result in a deadlock situation. For example calls to `IHXTEncodingJob::StopEncoding` or `IHXTEncodingJob::CancelEncoding` does not return until the event queue has been flushed. If the event callback thread is blocked by application code that is waiting for `IHXTEncodingJob::StopEncoding` or `IHXTEncodingJob::CancelEncoding` to complete, neither thread will be able to execute any additional code.

SDK users should stop running jobs prior to releasing their last reference count on the job. If a job is running when its last reference count is released, the job will be cancelled.

Encoding Samples

The Helix DNA Producer SDK provides three samples containing code that demonstrate how to use the SDK's encoding capabilities. These samples are located in the following directories:

- \producersdk\samples\advencoder
- \producersdk\samples\encoder
- \producersdk\samples\mediasinkencoder

CHAPTER

4

LOGGING SYSTEM

Helix DNA Producer now includes a globally-available message logging system. This system allows individual components to send messages to interested observers regardless of where they are in the application, or how they were created. This creates a wealth of information available to both developers and end users of the products, useful both in understanding what is happening and diagnosing problems.

Interfaces

A standard logging system typically implements the following interfaces:

- `IHXTLogSystem`. Header file: `ihxtlogsystem.h`.

  This interface provides access to various areas of the log system, including the functional area, log observer, and the log writer. An instance of this interface is returned from the logging system shared library.

- `IHXTFileObserver`. Header file: `ihxtfileobserver.h`.

  This interface provides all of the capabilities required to log messages from the logging system to a log file.

A logging system that requires customization (for example, to send log messages to a network server) typically implements the `IHXTLogSystem` interface along with the following interfaces.

- `IHXTLogObserver`. Header file: `ihxtlogsystem.h`.

  This interface receives messages from the logging system then passes them on to your custom logging application.

- `IHXTLogObserverManager`. Header file: `ihxtlogsystem.h`.

This interface subscribes and unsubscribes the IHXTLogObserver interface to the logging system. Once subscribed, the log observer can then receive messages from the logging system.

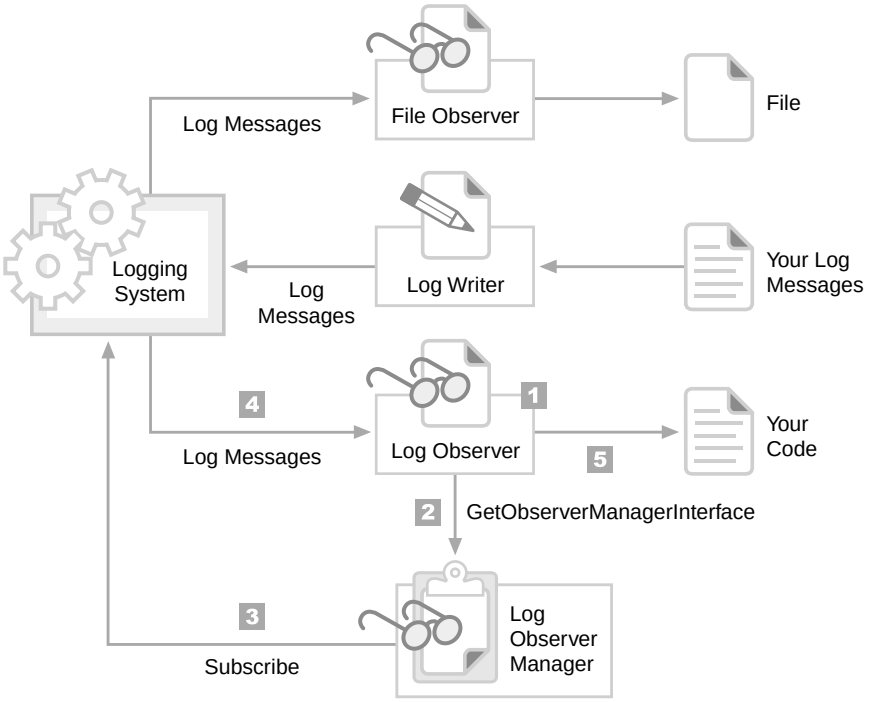
You can also send log messages from your application or plug-in back to the producer's logging system using the following interface:

- IHXTLogWriter. Header file: ihxtlogsystem.h.

  This interface provides access to the producer's logging system. Use the IHXTLogWriter::LogMessage method to send all of your log messages back to the logging system.

Using the Logging System

The following figure demonstrates the characteristics of the logging system provided by the Helix DNA Producer SDK. You can manage log messages either by using the file observer provided by the producer, or create your own system of managing the log messages using the log observer (the numbers indicate the order in which events occur using the log observer). In addition, your application can send its own messages to the logging system.



The Helix DNA Producer SDK has been created with numerous logging messages issued throughout the code to give users (both end-users and SDK developers) an idea of what is happening (and what possibly went wrong in the case of an error) inside the producer. For an application to make use of these log messages, the application must explicitly instantiate (and shutdown) the logging system.

Instantiating the Logging System

The logging system shared library (`log.dll` on Windows, `log.so` on Linux, or `log.bundle` on Mac OS X) exports two entry point functions, `RMACreateLogSystem` and `RMAGetLogSystemInterface`. `RMACreateLogSystem` is the entry point that instantiates the logging system, and returns an interface pointer to the instantiated logging system. `RMAGetLogSystemInterface` only succeeds if `RMACreateLogSystem` has already been called once by the application's process. Both of these entry points access a per-process singleton logging system, so they always return interface pointers to the same logging system.

To instantiate the logging system, load the logging system shared library and call the RMACreateLogSystem entry point, as shown in the following sample (Windows-specific code):

```
HINSTANCE  dllLogSystem = LoadLibrary("log.dll");
if ( dllLogSystem )
{
   // get pointer to RMACreateLogSystem function
   FPRMAGETLOGSYSTEMINTERFACE fpCreate =
           (FPRMAGETLOGSYSTEMINTERFACE)GetProcAddress ( m_dllLogSystem,
           "RMACreateLogSystem" );
   if ( fpCreate )
   {
      IHXTLogSystem pLogSystem = NULL;
      fpCreate( &pLogSystem );
```

The logging system is now instantiated, and any SDK component that contains logging code will log a message to that logging system.

The sample code provided with this SDK demonstrates how to use the logging system from the class factory. Using the class factory simplifies instantiating the logging system.

Shutting Down the Logging System

To work properly, the logging system must be explicitly shut down prior to the application's main thread finishing. Behavior is undefined if the logging system is created but not explicitly shut down. The logging system can be shut down using the IHXTLogSystem::Shutdown method, as shown in the following sample:

```
pLogSystem->Shutdown()
```

The IHXTLogSystem::Shutdown call might cause IHXTLogObserver::ReceiveMsg to be called on any subscribed observers, so a mutex must not be held between the caller of IHXTLogSystem::Shutdown and the observer during the IHXTLogSystem::Shutdown call.

Receiving Log Messages

There are two ways you can receive the log message sent from the logging system. You can either use the RealNetworks file observer that comes with the Helix DNA Producer SDK, or build your own observer class.

Using the RealNetworks File Observer

The RealNetworks file observer has been provided as a fully-functional observer that receives log messages and writes them to the specified file. The file observer is designed to facilitate ease of use in the simple cases, but also provides some advanced functionality.

Basic Use of the File Observer

There are two steps to instantiate a file observer:

1. Load the file observer DLL and create the file observer.

   This is much like instantiating the logging system. The DLL must be loaded, the address for the `CreateFileObserver` entry point must be obtained, then the `CreateFileObserver` function can be called to create a file observer. There is one major difference, for the file observer to discover the log system, you must call the `SetDLLAccessPath` entry point on the DLL to set the `DT_EncSDK dll` path to the location of the logging DLL used above to create the log system. The following sample code shows how to load the file observer DLL and create the file observer (Windows-specific):

```
// Create an instance of the RealNetworks File Observer
IHXTFileObserver* pFileLogObserver;
FPCREATEFILEOBSERVER pCreateObserverFunc;

HANDLE FileObserverDLL = LoadLibrary(".\\tools\\logobserver.dll");
if( m_FileObserverDLL )
{
   pCreateObserverFunc =
(FPCREATEFILEOBSERVER)(GetProcAddress(m_FileObserverDLL,
"CreateFileObserver"));

   // Must set dll access paths on the file observer dll in order for it
   // to properly locate the log dll

   // Get the SetDLLAccessPath entry point
FPRMBUILDSETDLLACCESSPATH fpSetDllAccessPath =
(FPRMBUILDSETDLLACCESSPATH)(::GetProcAddress(m_FileObserverDLL,
"SetDLLAccessPath"));
   // Set the DLL access paths
   HX_RESULT res = HXR_FAIL;
   if (fpSetDllAccessPath)
   {
       res = (*fpSetDllAccessPath)("DT_EncSDK=.\\tools");
```

```
}

if( SUCCEEDED(res) )
{
   res = (*pCreateObserverFunc)(&pFileLogObserver);
```

2. Initialize the observer with basic settings and subscribe to the log system.

   Once an instance of the file observer has been created from the file observer DLL, it must be initialized with the basic required settings. The only required setting is the file name to which the file observer will write all the log messages it receives. The file name is the only parameter to the `IHXTFileObserver::Init` method, and calling that method will subscribe the observer to the logging system, and start receiving messages. Because of this, if you want any settings to be applied to all messages the observer might receive, you must set those settings before calling the `IHXTFileObserver::Init` method.

Basic Settings

There are five settings on the file observer that most users should be aware of: category filtering, functional area filtering, format, separator, and SDK messages.

Category Filtering

You may only want to write log messages of a certain category, for example, only errors. To set the category mask, you simply call `IHXTFileObserver::SetCategoryFilter`, passing one or more bitmasks from the `EHXTLogCodeFilterMask` enumerator found in `ihxtlogsystem.h`, logically ORed together. For example, if you only wanted error level messages, you would call:

```
SetCategoryFilter(ERROR_MESSAGE_MASK)
```

Functional Area Filtering

Much like category filtering, you might only want to receive messages that originate from a certain area in the Helix DNA Producer SDK. To do this, you would pass a comma-separated list of the desired functional areas to the `IHXTFileObserver::SetFuncAreaFilter` method. See the next section for more details about filtering functional areas.

Format

The file observer supports the ability to write out log messages in two different formats: short and detailed. Messages written in short format include only the job name of the job that sent the message, and the message

text. The detailed formation includes the short format, plus the message category, functional area, and the time the message was logged. To change the format, call `IHXTFileObserver::SetFormat`, passing one of the enumerated values of `enumLogFormat`.

Separator

Between each element of a log message, the file observer inserts a separator character, which defaults to a comma, but may be changed to any character by calling the `IHXTFileObserver::SetSeperator` method.

SDK Messages

By default, the file observer ignores an Helix DNA Producer SDK-category messages it receives. To prevent this behavior, call the `IHXTFileObserver::EnableSDKMessages` method with the parameter set to `TRUE`.

Advanced Settings

In addition to the basic settings on the file observer, there are advanced settings the user should be aware of: file name, file rolling, and file appending.

File Name

The file observer supports variable replacement within the specified file name, allowing the user to add certain data to the file name. This is useful in the context of file rolling (explained in the next section) where the observer uses the original file name multiple times, and will simply append an increasing integer at the end of the file name (preceding the extension) to create a unique file name. The file observer substitutes the following character combinations with the following values:

Character sequence	Replaced with
%%	%
%h	hour of file creation
%m	minute of file creation
%s	second of file creation
%D	day (1-31) of file creation
%d	day(1-365) of file creation
%M	month of file creation
%Y	4 digit year of file creation
%y	2 digit year of file creation

For example, if you pass the string "LogFile %h:%m:%s.log" as the parameter to the IHXTFileObserver::Init method, the file observer will write to a log file that will look like "LogFile 4:32:27.log".

File Rolling

The file observer supports the ability to roll the file it is currently writing to based on either file size or system time (or both). To set a roll limit, use either the IHXTFileObserver::SetSizeRoll or the IHXTFileObserver::SetTimeRoll method.

When a log file is rolled by the file observer, the observer uses the original file name passed to the IHXTFileObserver::Init method and re-substitutes any variables present in the file name. Using the example file name in the previous section, if the user had set the file to roll at the 24-hour mark, a new file would be generated, possibly named "LogFile 00:01:13.log". This is due to the fact that a log file doesn't roll until a log message is actually received, and in our case, a log message wasn't received between midnight and 12:01:12 am.

File Appending

The file log observer is also designed to append whenever it is told to write to a preexisting file. This way, you can configure your application to always write to a log file called output.log, and you won't end up with output1.log, output2.log, output3.log, and so on (one for each time you run the application). This behavior can also be seen when the file observer rolls a log file. For example, assume you specified the name output.log as the target file, and set a limit of 2 MB per file, and ran an encode that logged 3 MB worth of messages. If you then run your application again, setting the same parameters to the file observer, it will examine output.log, see that it is already past the file size limit, then examine and start appending message to output2.log.

To have the same behavior when you use variable substitution in your file name, you must make use of the IHXTFileObserver::SetPreviousFilename method. The previous file name parameter tells the file observer that this is the last file to which it was writing. For example, you run your application, setting the observer file name to "Log %h:%m.log", and tell it to roll the file every day. It is currently 3:45pm on day one. The application runs for 6 hours. At the end of that run, you will have a log file called "Log 3:45.log". Now you immediately start your application again. If all you did was pass the same parameter (file name and roll limit), the file observer would generate a file name of "Log 9:45.log" and start writing to that new file. However, that's not what you want because you only want to roll the log file at the 24-hour mark of each day. To prevent generating a new file, you must set the previous log file name to "Log

3:45.log". The observer will examine this file, and determine from the file's creation date that it shouldn't yet roll, and start appending to that file, until the system time reaches midnight, at which point the observer will roll the file, and use the file name passed to the IHXTFileObserver::Init method ("Log %h:%m.log") to create a new file name.

Building Your Own Observer Class

There are two steps to creating an observer class that will receive log message notification from the Helix DNA Producer SDK: implementing the required interface and subscribing the observer to the logging system.

Required Interface

To receive log messages from the logging system, an observer class must implement the IHXTLogObserver interface. This interface contains two methods: IHXTLogObserver::OnEndService and IHXTLogObserver::ReceiveMsg.

The IHXTLogObserver::OnEndService method is called on the observer as notification that the logging system is about to release its reference count on that observer. This method tells the observer it will receive no IHXTLogObserver::ReceiveMsg calls from the logging system, and that it can release any resources it has reserved for that call.

The IHXTLogObserver::ReceiveMsg method is called by the logging system each time the logging system receives a message that the observer wants to receive. The logging system uses the filter the observer set using the IHXTLogObserverManager::SetFilter method to determine whether the observer wants to receive the message or not. See "Filtering" on page 104 for information on per observer filters. The parameters for the call contain the data for the log message (see "Appendix A: Interface List" beginning on page 161 for an explanation of each IHXTLogObserverManager::SetFilter parameter).

Subscribing to the Logging System

An observer must subscribe to the logging system to receive notification of log messages. When an observer subscribes to the logging system, the logging system queries (IUnknown::QueryInterface) for the IHXTLogObserver interface and, if that succeeds, maintains a reference count on the observer until either the observer explicitly unsubscribes from the logging system, or the logging system shutdowns, calling the IHXTLogObserver::OnEndService method on each observer before releasing the reference count it holds.

There are two steps to subscribe an observer to the logging system. First, the observer must acquire a pointer to the logging system. This is done by loading the logging system DLL, obtaining the address of the RMAGetLogSystemInterface function, and calling that function. An example of this can be found in the CSampleLogObserver::Initialize method (found in the RTASampleLogObserver.cpp files in the encoding samples supplied with this SDK). If the RMAGetLogSystemInterface function succeeds, the parameter will be initialized to point to the logging system. For that function call to succeed, the logging system must have already been initialized (see "Instantiating the Logging System" on page 97).

The next step is to subscribe. To do this, you must first obtain the IHXTLogObserverManager interface from the logging system using the IHXTLogSystem::GetObserverManagerInterface method on the IHXTLogSystem interface you just obtained from the RMAGetLogSystemInterface function call.

```
IHXTLogObserverManager* pIObserverManager;
m_pLogSystem->GetObserverManagerInterface(&pIObserverManager);
```

Once you have the IHXTLogObserverManager interface, you can call the IHXTLogObserverManager::Subscribe method, passing a pointer to your observer class as the first parameter. The other three parameters are explained in Appendix A beginning on page 161.

```
pIObserverManager->Subscribe((IUnknown*)this, NULL, NULL, TRUE);
```

The observer is now subscribed to the logging system, and will receive a call to IHXTLogObserver::ReceiveMsg for each log message sent to the logging system.

Advanced Observer Operations

Missed Messages

When you start up your application, and connect your observer to the logging system, you might discover that some log messages have already been sent by the SDK or your own objects. As a result, your observer might not receive certain messages it was interested in. The logging system provides a way to avoid missing messages.

If you examine the fourth parameter in the IHXTLogObserverManager::Subscribe call, you will see it is a boolean parameter named bCatchUp. This parameter tells the logging system that the observer wants the logging system to send any log messages that might have been processed before the observer subscribed. The logging system will only store up to 1000 previously-received

messages while running, and that is the maximum number of messages the observer can catch up by setting the `bCatchUp` parameter.

Filtering

You might have an observer that is only interested a specific category of log messages (errors, for example), or perhaps only those messages that come from a certain area within the product. While you could do this filtering on your own, the logging system provides a per observer filtering system that does this work for you.

An observer can set a filter for itself in the logging system in two different ways. The observer can call the `IHXTLogObserverManager::SetFilter` method, passing the filter XML string (described below) as the `szFileterStr` argument, or it could pass that same string as the second parameter to the `IHXTLogObserverManager::Subscribe` method call.

There are two important things to note about setting a filter. First, the `IHXTLogObserverManager::SetFilter` method will only succeed after the `IHXTLogObserverManager::Subscribe` method has been called for that observer. The `IHXTLogObserverManager::SetFilter` method is mainly used to alter an observer's filter while the observer is connected to the logging system. Second, most observers will want to filter all messages they receive, including those delivered by calling the `IHXTLogObserverManager::Subscribe` method with `bCatchUp` set to `true`. To do this, the observer must send the filter string as the second argument to the `IHXTLogObserverManager::Subscribe` call. That filter will be applied to any missed messages that are sent, as well as all new messages logged.

The format for an observer filter string is an XML string of the form:

```
<?xml version="1.0" encoding="UTF-8"?>
<Filter [LOGCODE="category bit mask"]
   [FUNCAREA="namespace:functional area index,..."]
   [MSGNUM="message numbers"]>
```

The `LOGCODE` attribute sets the filter for which values of the `nCode` parameter to `IHXTLogObserver::ReceiveMsg` are allowed through. This attribute should contain a bit mask that is applied to the `nCode` of all incoming messages, and if the result of the bit mask is non-zero, the message is passed on to the observer. For example, to only receive informational and error-level messages, you would set the `LOGCODE` attribute equal to `LC_SDK_ERROR|LC_SDK_INFO`. The full list of logging categories can be found in `ihxtlogsystem.h`.

The FUNCAREA attribute sets the filter for which values of the unFuncArea parameter to IHXTLogObserver::ReceiveMsg are allowed through. Note that functional areas are tied to the szNamespace parameter in the logging system, and therefore to set a filter on one or more functional areas, you must specify the namespace along with the functional area number. For example, to only receive messages from file readers and file writers within the SDK, you would set the FUNCAREA attribute equal to "RealNetworks:8,RealNetworks:7". If you look in ixhxlogsystem.h, you will see that 8 is the functional area index for FILEREAD, and 7 is the functional area index for FILEOUT. This attribute can also be set to a special value of "all", which tells the logging system to send all functional areas to the observer.

The MSGNUM attribute sets the filter for which values of the nMsg parameter to IHXTLogObserver::ReceiveMsg are allowed through. Only messages logged with one of the listed numbers will be send to the observer. For example, to only receive messages 45 and 883, you would set the MSGNUM attribute equal to "45,883".

One final important point on filters. All filter settings are applied using a logical AND operation. This means that if you set the LOGCODE filter to LC_SDK_ERROR and the FUNCAREA filter to RealNetworks:8, your observer will only ever receive error-level log messages from file readers within the SDK. The observer will not receive error-level messages from the file writer, nor will it receive warning-level messages from file readers.

Sending Messages to the Logging System

The IHXTLogWriter interface sends log messages from any point in the code to the logging system. There are two ways to obtain a reference to the IHXTLogWriter interface:

- Use the IHXCommonClassFactory provided in IHXTFilter::SetFactory.

  Whenever a Helix DNA Producer filter is instantiated, it's IHXTFilter::SetFactory method is called to provide it a class factory interface that it can use to create Helix DNA Producer SDK objects. This factory also provides access to the logging system. The following code shows how to ask that class factory for the IHXTLogWriter interface.

```
STDMETHODIMP CRSInPCMFilter::SetFactory( IHXCommonClassFactory* pCCF )
{
IHXTLogWriter* pILogWriter = NULL;
pCCF->CreateInstance( IID_IHXTLogWriter, (void**)&pILogWriter );
```

- Obtain the interface directly from the logging system.

  When discussing how to create the logging system (see "Instantiating the Logging System" on page 97), two entry points to the logging system shared library were described, with `RMACreateLogSystem` used to initialize the logging system. The other entrypoint, `RMAGetLogSystemInterface`, is used to obtain a reference to the logging system once it has been initialized. `RMAGetLogSystemInterface`, unlike `RMACreateLogSystem`, can be called multiple times; however it will not succeed unless `RMACreateLogSystem` has been called. Once you have a reference to the `IHXTLogSystem` interface, you can call the `IHXTLogSystem::GetWriterInterface` method to obtain the desired interface, as shown in the following sample:

```
FPGETLOGSYSTEMINTERFACE GetLogSystem = NULL;
IHXTLogSystem* pLogSystem = NULL;
IHXTLogWriter* pLogWriter = NULL;

HINSTANCE dllLogSystem = LoadLibrary("log.dll");

if ( dllLogSystem )
{
   GetLogSystem =
(FPGETLOGSYSTEMINTERFACE)(::GetProcAddress(dllLogSystem ,
"RMAGetLogSystemInterface");
   if(GetLogSystem)
   {
      (*GetLogSystem)( &pLogSystem );
   }
}

if(pLogSystem )
{
   pLogSystem ->GetWriterInterface(&pLogWriter);
```

Once you have obtained the `IHXTLogWriter` interface, sending a message to the logging system is accomplished by calling the `IHXTLogWriter::LogMessage` method. The parameters of the `IHXTLogWriter::LogMessage` call are explained in "Appendix A: Interface List" beginning on page 161.

Logging Samples

The Helix DNA Producer SDK provides three samples containing code that demonstrate how to use the SDK's logging capabilities:

- \producersdk\samples\encoder\encoder.cpp
- \producersdk\samples\mediasinkencoder\mediasinkencoder.cpp
- \producersdk\samples\advencoder\logobserver.cpp

CHAPTER

5

HELIX DNA PRODUCER PLUG-IN API

The plug-in portion of the Helix DNA Producer SDK provides interfaces and helper classes with which you can build custom media plug-ins used by the Helix DNA Producer filter graph. Helix DNA Producer is built on the concept of arranging media input filters, transforms filters, and output filters in a pipeline that can encode, manipulate, or broadcast media data. The system is extensible in such a way that you can write plug-ins that are either dynamically discovered and used at runtime, or can be programmatically inserted in a filter graph.

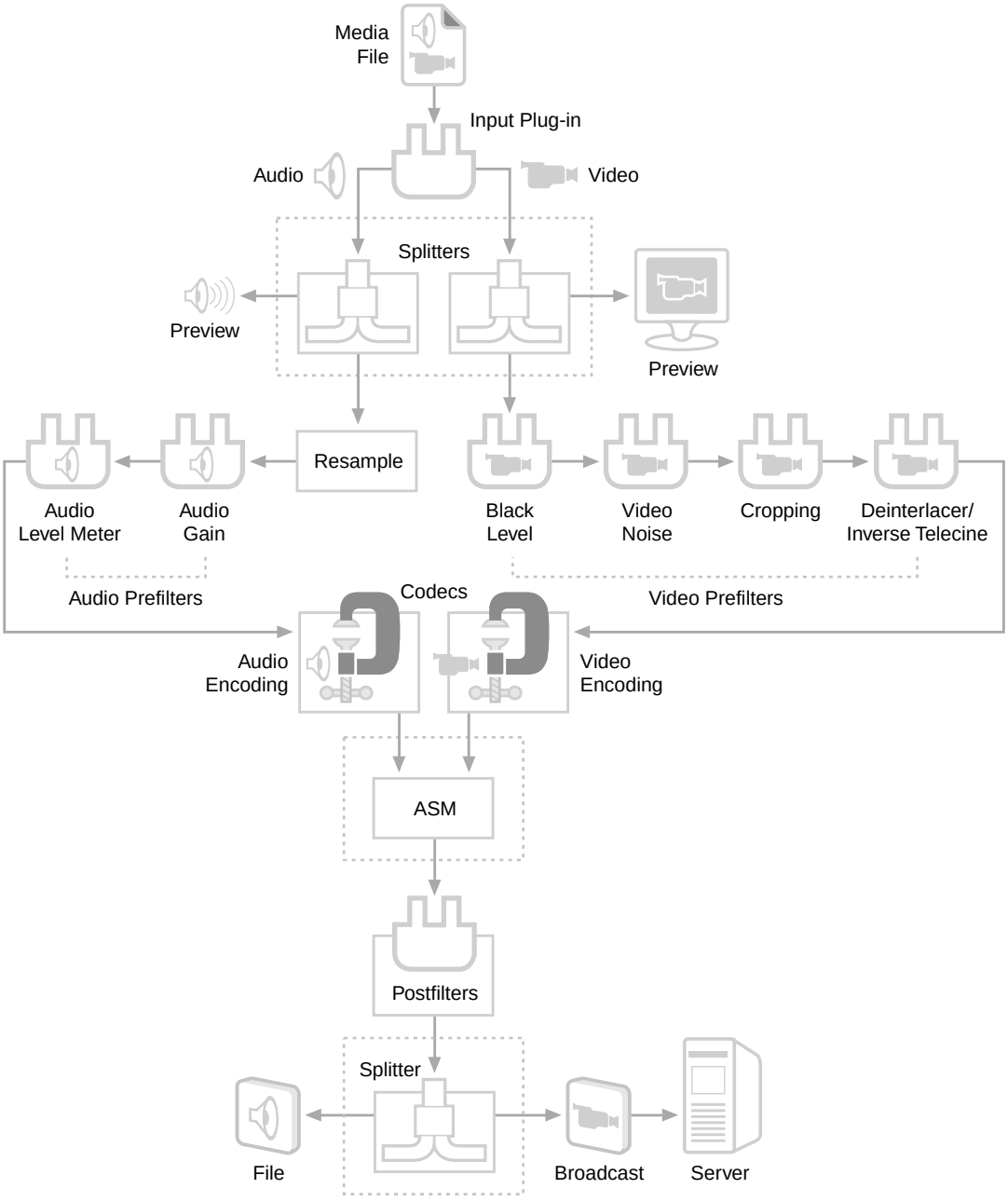
This chapter provides an in-depth explanation of the methods and policies that go into writing the low-level Helix DNA Producer SDK plug-ins. Together with the sample code, this chapter provides both an overview and enough details to address the issues you will face when writing Helix DNA Producer SDK plug-ins.

Plug-in Categories

The following categories of plug-ins are supported by the plug-in model of the Helix DNA Producer SDK:

- Input plug-ins
- Transform plug-ins
- Output plug-ins.

Applications using the Helix DNA Producer SDK will configure input plug-ins using the `IHXTInput` interface, transform plug-ins using the `IHXTPrefilter`, `IHXTPostfilter`, and `IHXTStreamConfig` interfaces, and output plug-ins using the `IHXTDestination` interface.



This figure demonstrates the layout of a typical encoding session in Helix DNA Producer. The figure shows a standard media file containing audio and

video passing though an input plug-in that splits the audio and video into separate channels. Each of these channels are processed by specific audio or video transform plug-ins (prefilters), encoded, interleaved back into a single audio/video channel, processed by a postfilter plug-in, and finally sent either to a file, broadcast, or both.

Input Plug-ins

The following types of input plug-ins can be used to input data into Helix DNA Producer:

- File reader plug-ins
- Capture device plug-ins

File reader plug-ins decode files of various formats into raw video and audio data that can be then be manipulated and encoded in the Helix DNA Producer media engine. File reader plug-ins are dynamically discovered and utilized by the Helix DNA Producer media engine when a file input is specified in a job.

File reader plug-ins specify which file extensions they handle. When the Helix DNA Producer media engine is given an input file, it first checks for any file reader plug-in that handles the given file extension. If all the plug-ins advertising support for the particular file extension fail to read the file, all other reader plug-ins will then be tried. See the input plug-in sample (`inputplugin`) for an example of this.

A capture device plug-in is a specific kind of input plug-in designed for obtaining audio and video samples from a capture device. Capture device plug-ins support an enumeration interface for obtaining information about capture devices, and also support a duration property for capture operations.

Transform Plug-ins

The following types of transform plug-ins can be used to transform data in Helix DNA Producer:

- Pre-encode transform filters
- Post-encode transform filters

Pre-encode transform filters (or prefilters for short) manipulate raw audio and video data prior to encoding. For example, you may want to write your own audio reverb filter.

A number of audio and video prefilters ship with Helix DNA Producer including video black-level, video deinterlace, and audio gain filters. Not all prefilter plug-ins appear in the Helix DNA Producer GUI application unless they are selected. To use a custom prefilter, you must use the Helix DNA Producer SDK encoding system to add your custom prefilter explicitly to the encoding job.

Output Plug-ins

Output plug-ins can be used as a sink for media streams generated upstream in the filter graph. Data received by an output plug-in is in the form of stream headers and encoded packets. Output plug-ins then output the data in the form appropriate for a designated destination, which could be a server, file, or other device. Output plug-ins can be used to support proprietary network protocols between encoder and server.

Helper Classes

The Helix DNA Producer SDK provides a set of helper classes that incorporate many of the low-level interfaces, such as `IHXTPropertyBag`, `IHXTUintList`, and so on, without requiring you to write low-level code to populate these interfaces in your plug-in.

All of these helper classes are specifically designed for plug-in developers. They include a configuration agent helper class for simplifying the plug-in configuration (layer 2), an external plug-in helper class that provides a template of your plug-in for the Helix DNA Producer SDK (layer 1), and an input format helper class for consistently managing the handling and display of input properties for all input sources—either file- or capture-based.

For more information on the configuration agent helper class, see "Configuration and Connection Agent Interfaces (Layer 2)" on page 135. In addition, examine the `ihxtconfigagenthelper.h` file and both the `inputplugin` and `prefilterplugin` samples for an example implementation of the configuration agent helper class.

For more information on the external plug-in helper class, see the `ihxtplugininfobase.h` file and both the `inputplugin` and `prefilterplugin` samples for an example implementation of this class.

For more information on the input format helper class, see the `ihxtinputformathelper.h` file and the `inputplugin` sample for an example implementation of this class.

Creating Custom Media Plug-ins

At its core, the Helix DNA Producer SDK follows a filter graph model, where input, transform, and output plug-ins are snapped together in filter graphs, and data packages are passed downstream from component to component until the desired transformation is accomplished. The Helix DNA Producer SDK is cross platform, lightweight, and layered such that the developer can build low level plug-ins—inputs, prefilters, postfilters, or outputs—to extend existing encoding functionality or write higher-level client code to control encoding operations through a dedicated Helix DNA Producer SDK encoding engine.

With the Helix DNA Producer SDK, a developer does not directly interact with the graph manager layer which sets up and manages component connection, and data flow control of the data packages through the filter graph layer.

Another vital component of the Helix DNA Producer architecture is property bags. Property bags—containers of varied types of property data—deliver properties to the plug-ins during the connection and configuration process. Property bags are also used with the Helix DNA Producer SDK encoding system to programmatically pass property data to and from the plug-ins.

Before you get started there are a few things you need to know:

- There are three basic types of plug-ins: input, transform, and output. When coupled together properly, these three types of plug-ins can accomplish almost any transformation or encoding task. Transform plug-ins further split into three categories: prefilters (which come before the codec), codecs, and postfilters (which come after codecs).
- Helix DNA Producer plug-ins are typically built using a layered approach and implement the following sets of interfaces:
    - level 1—plug-in (describes the plug-in to the system)
    - level 2—connection and configuration
    - level 3—filter (interacts with media samples and other plug-ins)
    - level 4—at the lowest working level, your actual operational code

Organization of a Typical Helix DNA Producer Plug-in

At the lowest level (layer 4), in its own class, is the operational code that does something, such as reads, writes, or transforms data. Ideally this code is isolated from the other higher layers of the plug-in except for certain points of contact with the other layers.

Above the operational code is the filter layer (layer 3) which generates or accepts media samples, operates on them, and passes them on to the next component in the data pipeline. The operational code interacts with this layer in the `IHXTInputFilter::ReadSample`, `IHXTTransformFilter::ReceiveSample`, or `IHXTOutputFilter::ReceiveSample` methods. The filter layer (layer 3) usually derives from layer 4—the operational class described in the previous paragraph.

Above the filter layer is the agent layer (layer 2). This is where the plug-in's property initialization and media format information is handled. Typically, this level interacts with the lowest-level operational classes (layer 4) just enough to obtain or pass on properties required for basic operations, such as open or close, to occur. This agent class (layer 2) typically derives from the lower filter-level class (layer 3) described above.

The highest layer is the plug-in layer (layer 1), which describes the functionality of the plug-in to the system. It consists primarily of a table of descriptive strings and numbers, and typically derives from the agent class (layer 2).

The following table shows the four layers of a typical plug-in.

Plug-in Layers

Layer No.	Layer 4	Layer 3	Layer 2	Layer 1
Description:	Performs some actual data operation.	Processes media samples at various stages.	Handles property and connection data.	Describes plug-in to the system.
Interface Names:	Operational classes	`IHXTFilter`*XXX* interfaces	Configuration and connection agent interfaces	Plug-in information table
File Name:	`MyOperation.cpp`	`MyFilter.cpp`	`MyAgent.cpp`	`MyPlugin.cpp`
Interface Methods:	(examples include:)	**`IHXTFilter`** (all plug-ins)	**`IHXTConnectionAgent`**	(contains fields:)
		`SetFactory`	`GetInputStreamCount`	`PluginType`

(Table Page 1 of 2)

Plug-in Layers (continued)

Layer No.	Layer 4	Layer 3	Layer 2	Layer 1
	Input file reading	SetGraphServices	GetSupportedInputFormat	ComponentName
	Input file capture	Prime	GetNegotiatedInputFormat	PluginLongName
		Teardown	SetNegotiatedInputFormat	PluginShortName
	Transform prefilters	DiscardCachedSamples	GetOutputStreamCount	ComponentCLSID
	Transform codecs		GetSupportedOutputFormat	Description
	Transform postfilters	**IHXTInputFilter**	GetNegotiatedOutputFormat	Copyright
		SetAllocator	SetNegotiatedOutputFormat	
	Output file writing	ReadSample		
	Output broadcasting		**CHXConfigurationAgentHelper**	
		IHXTTransformFilter	OnInitialize	
		SetAllocator	OnSetString	
		SetSampleSink	OnSetUint	
		ReceiveSample	OnSet*XXX*List	
			OnSet*XXX*Range	
		IHXTOutputFilter	Other overrides if needed	
		ReceiveSample		

(Table Page 2 of 2)

Fitting New Code in the Plug-in Layers (Layer 4)

If you already have code that performs an operation you want to design into a plug-in, try to package it neatly into its own class with minimal or no dependencies on the rest of the classes (layer 4). If you examine the code in the sample plug-ins, such as /samples/inputplugin, you can see how the file reading code is isolated from the rest of the plug-in. Follow the examples and do this with your own operational code.

Next, study the IHXTInputFilter::ReadSample, IHXTTransformFilter::ReceiveSample, or IHXTOutputFilter::ReceiveSample methods in the filter class (layer 3), and observe how data is obtained and passed on to the operational class (layer 4) defined in the previous paragraph. You might need to add some helper functions to actually access the operational code, but try to make the contact between the filter and the actual operational code as clean as possible. Additional points of contact between the layers might involve adding initialization code in the IHXTFilter::Prime call, and closing or buffer releasing code in the IHXTFilter::Teardown functions. The rest of the filter methods you can probably leave untouched unless some special need dictates otherwise.

At the next highest level up (layer 2)—known as the agent level—you receive initialization values and strings appropriate to your plug-in, which you must translate and pass on (or gather) from the lower-level operational code. Try to make explicit what the interaction between the layers is. You need ensure the property bags describing your data format are appropriate for the specific media type. The rest of the agent methods in the example plug-ins you can probably leave as is unless some special need dictates otherwise.

Finally, fill in the tables of the plug-in class with data appropriate for your specific type of plug-in. This way the plug-in handling methods can find and instantiate your plug-in appropriately (layer 1).

All Filter Interfaces (Layer 3)

All Filter Interfaces

IHXTFilter (all plug-ins)	IHXTInputFilter (input plug-ins only)	IHXTTransformFilter (transform plug-ins only)	IHXTOutputFilter (output plug-ins only)
SetFactory	SetAllocator	SetAllocator	ReceiveSample
SetGraphServices	ReadSample	SetSampleSink	
Prime		ReceiveSample	
Teardown			
DiscardCachedSamples			

Filter Set Calls

IHXTFilter::SetFactory, IHXTFilter::SetGraphServices, IHXTInputFilter::SetAllocator, IHXTTransformFilter::SetAllocator, and IHXTTransformFilter::SetSampleSink are initialization methods that provide some essential service to the filter layer,

depending on where it is in the filter graph pipeline. IHXTFilter::SetFactory provides an object factory the plug-in can use to create property bags and related types. IHXTFilter::SetGraphServices supplies the filter layer with services, such as an event sink that sends events (such as filter done) to higher layers. IHXTInputFilter::SetAllocator and IHXTTransformFilter::SetAllocator provide a memory allocator with which to obtain media samples. This method is only required for input and transform filters. IHXTTransformFilter::SetSampleSink is called only on transform plug-ins to provide a destination for their media samples once data starts flowing.

Filter Data Setup and Teardown

IHXTFilter::Prime is called immediately before data starts flowing to allow final initialization of objects. It is the opposite of IHXTFilter::Teardown, which is called at the end to inform the plug-in to tear down and release any objects or memory, as they are no longer needed. IHXTFilter::DiscardCachedSamples is provided for any plug-in that caches data samples.

Filter Data Flow

Data flows through repeated calls to IHXTInputFilter::ReadSample, IHXTTransformFilter::ReceiveSample, and IHXTOutputFilter::ReceiveSample, which carry media samples from plug-in to plug-in through the encoding pipeline.

IHXTFilter Interface

All plug-ins must implement the following methods:

```
DECLARE_INTERFACE_( IHXTFilter, IHXTFilter )
{
   STDMETHOD(SetFactory) (THIS_ IHXCommonClassFactory* pCCF) PURE;
   STDMETHOD(SetGraphServices) (THIS_ IHXTServiceBroker* pGraphServices) PURE;
   STDMETHOD(Prime) (THIS_ UINT32 uStream) PURE;
   STDMETHOD(Teardown) (THIS_ UINT32 uStream) PURE;
   STDMETHOD(DiscardCachedSamples) (THIS_ UINT32 uStream) PURE;
}
```

IHXTFilter::SetFactory Method

Provides an object factory the plug-in can use to create property bags and other objects.

Application:

This method is typically overridden by an agent class (to obtain access to the factory in the agent class), if one exists.

Parameters:

`IHXCommonClassFactory* pFactory`—call `CreateInstance` on factory to build objects. For example, the factory could be used to create a log writer interface for the plug-in. The log writer interface can be used by all layers to log various types of messages. See the `inputplugin` sample for an example plug-in log writer.

Requirements:

- Optionally, Addref `pFactory` object when received, Release `pFactory` when plug-in is destroyed.

Code Example: (external)

Shows how to pass the factory to the base class configuration helper for creation of property bags and media formats.

```
STDMETHODIMP CRSInPCMAgent::SetFactory( IHXCommonClassFactory* pFactory )
{
   if (!pFactory)
      return HXR_POINTER;

  HX_RESULT res = CHXConfigurationAgentHelper< IHXTConfigurationAgent >::Init(
pFactory );
   if (SUCCEEDED(res))
      FillInPropertyBags();
   return res;
}
```

Although `Setfactory` is a method on the `IHXTFilter` interface, it is typically handled in two places in a plug-in: in layer 3 on the filter interface and on the agent interface (layer 2) which derives from layer 3.

In the agent layer `SetFactory` first calls the filter base class which performs the functions described earlier in this section. It then calls `CHXConfigurationAgentHelper::init( pFactory )` which passes the factory to the helper class and creates a default property bag for the plug-in to use. Once the factory is made available, it is used to create additional property bags (as shown in the `CRSInWAVAgent::FillInPropertyBags` method in the `\samples\inputplugin\InWAVAgent.cpp` file).

```
STDMETHODIMP CRSInWAVAgent::SetFactory( IHXCommonClassFactory* pFactory )
{
      if (!pFactory)
      {
            return HXR_POINTER;
      }

      HX_RESULT res = CRSInWAVFilter::SetFactory( pFactory );
      if (SUCCEEDED(res))
      {
            res = CHXConfigurationAgentHelper::init( pFactory );
            if (SUCCEEDED(res))
                  res = FillInPropertyBags();
      }
      return res;
}
```

IHXTFilter::SetGraphServices Method

Provides service broker for retrieving event sink for message dispatching.

Application:
An event sink is only needed by asynchronous plug-ins to send events, such as errors or stream done, to higher layers.

Parameters:
`IHXTServiceBroker *pServiceBroker`—provides access to the event sink.

Requirements:

- Return `HXR_OK` if the plug-in does nothing in this method.
- Addref any service, such as `m_pEventSink`, that you need to use. Make sure to release `m_pEventSink` in destructor.
- Valid error codes are `HXR_POINTER`.

Code Example:
Shows how to use service broker to obtain event sink interface.

```
STDMETHODIMP CRSOutPCMFilter::SetGraphServices( IHXTServiceBroker
*pServiceBroker )
{
   HX_RESULT res = HXR_OK;
   IUnknown* pUnk = NULL;
   if ( NULL == m_pEventSink )
   {
```

```
        res = pServiceBroker->GetService( IID_IHXTEventSink, &pUnk );
        if (SUCCEEDED(res))
            res = pUnk->QueryInterface( IID_IHXTEventSink,
                            (void**) &m_pEventSink );
        HX_RELEASE( pUnk );

    return res;
}
```

IHXTFilter::Prime Method

Notification that IHXTInputFilter::ReadSample, IHXTTransformFilter::ReceiveSample, or IHXTOutputFilter::ReceiveSample will be called shortly.

Application:

Called to notify the filter that it will be receiving (or asked to provide) media samples shortly on the given stream. This call always precedes calls to IHXTInputFilter::ReadSample, IHXTTransformFilter::ReceiveSample, or IHXTOutputFilter::ReceiveSample. Typically, various initialization procedures are performed at this time. IHXTFilter::Prime is called once for each stream on a plug-in, so make sure to account for the fact it will typically be called a couple of times.

Parameters:

The stream identifier refers to output streams on input plug-ins, and input streams on transform and output plug-ins. Note that input plug-ins only have output streams, and output plug-ins only have input streams.

Threading Behavior:

This method should only return when it is prepared to process data on the associated stream. While responsive behavior is desired, it may need to block until this condition is met.

Requirements:

Upon successful completion of this method, the plug-in must be prepared to process data for the corresponding stream.

- Return HXR_OK if the plug-in does nothing in this method.
- Returning a failure code will prevent the plug-in from receiving data processing calls.
- Check for valid stream—ignore if stream ID is not used.

- Check for flags showing connection calls have already occurred. Otherwise, return HXR_ENC_IMPROPER_STATE.
- Check for required objects such as stream/fileheaders or required settings. Otherwise return HXR_POINTER.
- Check interrelated settings for validity. If any are invalid, return HXR_INVALID_PARAMETER.
- Create any required objects and optionally assert that important ones exist.
- Allocate buffers.
- Initialize and start any child or primary objects.
- Create any stream tracking variables that depend on knowing actual number of streams.
- If successful, set initialized flag, prevent on-the-fly changes flag, and reset variables used in read/receivesample.
- If failed, clean up any allocated buffers, release and/or close or delete objects.

Code Example:

Shows almost every kind of initialization that can occur on a Prime call.

```
STDMETHODIMP CRMGenericPlugin::Prime( UINT32 ulInputStreamID )
{
   // check valid stream id if you are going to use it
   if (ulInputStreamID != 0)
      return HXR_INVALID_PARAMETER;

   // optional - check negotiation objects or required settings or required objects
   if ( !m_bFormatReady || !m_bIsInputPropertiesSet || !m_pFileHeader)
      return HXR_ENC_IMPROPER_STATE;

   // optional -- assert that certain objects exist
   HX_ASSERT( m_spAllocator );

   // check settings against each other now that all have been set
   HX_RESULT res = GetUint( kPropCropLeft, &m_ulCroppedLeftTopX );
   if ( SUCCEEDED(res) && m_bIsOutputCropped == TRUE &&  m_ulCroppedLeftTopX
<  m_ulMinCropping)
      return HXR_INVALID_PARAMETER;

   // optional -- create any required objects
```

```
HX_RELEASE( m_pRBSActor );
m_pRBSActor = new CRBSActor;
m_pRBSActor->AddRef();

// optional - initialize objects used at prime
UINT32 uiBytesPerFrame;
res = m_pVitalObject->Init( &uiBytesPerFrame );

// optional - allocate any necessary buffers
if (SUCCEEDED(res))
   m_pInputBinBuffer = new BYTE[uiBytesPerFrame];

// optional -- handle stream related vars -- now that number of streams is known
if (SUCCEEDED(res) && !m_abStreamDone)
{
   m_nNumOutputStreams = GetNumTracks();
   m_abStreamDone = new BOOL[ m_nNumOutputStreams ];
   for ( UINT32 i=0; i< m_nNumOutputStreams; i++)
      m_abStreamDone[i] = FALSE;
}

// optional -- pass on initializations to child objects
if (SUCCEEDED(res))
   res = m_spAudioInput->SetRealTimeMode(m_bRealTimeInput);

// optional -- Start child objects
if (SUCCEEDED(res))
   res = StartReader();

// optional -- Start objects
if (SUCCEEDED(res))
   res = m_pRBSActor->Start();

// if successful set initialization and other variables as necessary
if (SUCCEEDED(res))
{
   // optional -- set flag to prevent changes after prime call
   m_bAllowPropChanges = FALSE;

   // optional -- set initialized flags
   m_bIsCodecInitialized = TRUE;

   // optional -- reset variables that are used in read/ receivesample
   m_bStreamEnded = FALSE;
```

```
    }
    // if prime failed -- cleanup
    else
    {
        // any allocated buffers
        HX_VECTOR_DELETE( m_pInputBinBuffer );

        // any objects
        HX_RELEASE( m_pRBSActor );

    }
    // optional -- make sure everything was okay
    assert( SUCCEEDED( res ) );
    return res;
}
```

Although Prime is a method on the IHXTFilter interface, it is typically handled in two places in a plug-in: in layer 3 on the filter interface and on the agent interface (layer 2) which derives from layer 3.

In the agent layer it first calls the base class, then calls CHXConfigurationAgentHelper::setAllowPropertyChanges with FALSE to prevent any property changes during encoding.

> **Note:** In the Teardown call, CHXConfigurationAgentHelper::setAllowPropertyChanges is called with TRUE to allow property changes to the plug-in again.

```
STDMETHODIMP CRSInWAVAgent::Prime( UINT32 ulOutputStreamID )
{
    HX_RESULT res = CRSInWAVFilter::Prime( ulOutputStreamID );

    // once prime is called -- prevent any changes to properties
    if (SUCCEEDED(res))
        CHXConfigurationAgentHelper::setAllowPropertyChanges( FALSE );

    return res;
}
```

IHXTFilter::DiscardCachedSamples Method

Method to optionally release any cached media samples.

Application:

Called to signal the plug-in to release any cached samples it may have. This method can be called at any time between IHXTFilter::Prime and IHXTFilter::Teardown. If IHXTTransformFilter::ReceiveSample or IHXTOutputFilter::ReceiveSample is called after IHXTFilter::DiscardCachedSamples, the sample will generally be marked with end of stream.

Parameters:

The stream identifier refers to output streams on input plug-ins, and input streams on transform and output plug-ins. Note that input plug-ins only have output streams, and output plug-ins only have input streams.

Requirements:

This method should return HXR_OK if there were no cached samples or if there were cached samples and they were released. It should return HXR_FAIL if there were cached samples and they could not be released.

- Return HXR_OK if the plug-in does nothing in this method.

Code Example:

Shows release of a cached sample.

```
STDMETHODIMP CRSExample::DiscardCachedSamples( UINT32 ulOutputStreamID )
{
   if (ulOutputStreamID != 0)
      return HXR_INVALID_PARAMETER;

   m_pLastSample->Release();
   return HXR_OK;
}
```

IHXTFIlter::Teardown Method

Method to stop operations, shut down and release objects, free buffers and reset flags.

Application:

This method is called by the filter graph when the graph is to be stopped. Teardown is equivalent to an abort or cancel call, so any devices or running processes should be stopped here in case they were not shut down elsewhere.

Parameters:

The stream identifier refers to output streams on input plug-ins, and input streams on transform and output plug-ins. Note that input plug-ins only have output streams, and output plug-ins only have input streams.

Threading Behavior:

This method should not block, so as to avoid unresponsive behavior in the graph.

Requirements:

The plug-in must guarantee that no samples are sent after `IHXTFilter::Teardown` returns. The plug-in must be written to safely handle being released after `IHXTFilter::Teardown` is called. This means that if the plug-in uses threads internally, and since `IHXTFilter::Teardown` should not block, the thread should gracefully exit asynchronously. It is recommended that a thread `AddRef` the plug-in layer to help achieve this.

- Return `HXR_OK` if the plug-in does nothing in this method.
- Return `HXR_INVALID_PARAMETER` for an invalid stream ID.
- After `IHXTFilter::Teardown`, neither `IHXTInputFilter::ReadSample`, `IHXTTransformFilter::ReceiveSample`, nor `IHXTOutputFilter::ReceiveSample` will be called again without first receiving another `IHXTFilter::Prime` call.
- Stop all operations.
- Close down child objects.
- Free outstanding buffers.
- Release or destroy remaining objects, including stream tracking variables.
- Reset on-the-fly protection flags.
- Optional—reset initialization flags.

Code Example:

Shows examples of various operations that can occur on a `IHXTFilter::Teardown` call:

```
STDMETHODIMP CRMGenericFilter::Teardown( UINT32 ulInputStreamID )
{
   // check valid stream id
   if (ulInputStreamID != 0)
      return HXR_INVALID_PARAMETER;
```

```
    // optional -- stop vital operations
    res = m_pRBSActor->Stop();

    // optional -- close down child objects
    TearDownReader();

    // optional -- example of special threadsafe shutdown
    if (m_pRMWriter)
    {
        // Block until writer finishes teardown
        m_pRMWriter->WaitForCompletion();

        // shutdown context thread and block until thread has exited
        m_pRMWriter->Shutdown();
        HX_RELEASE( m_pRMWriter );
    }

    // optional -- destroy buffers
    HX_VECTOR_DELETE( m_pInputBuffer );

    // optional -- release or destroy any existing objects
    HX_RELEASE( m_pRBSActor );

    // optional -- free any stream vars
    HX_VECTOR_DELETE( m_abStreamDone );

    // optional - reset protection flags
    m_bAllowPropChanges = TRUE;
    m_bFormatReady = FALSE;

    // optional -- reset initialization flags
    m_bIsCodecInitialized = TRUE;

    return res;
}
```

Although `Teardown` is a method on the `IHXTFilter` interface, it is typically handled in two places in a plug-in: in layer 3 on the filter interface and on the agent interface (layer 2) which derives from layer 3.

In the agent layer it first calls the base class, then calls `CHXConfigurationAgentHelper::setAllowPropertyChanges` with `TRUE` to allow property changes to the plug-in again.

```
STDMETHODIMP CRSInWAVAgent::Teardown( UINT32 ulOutputStreamID )
{
      HX_RESULT res = CRSInWAVFilter::Teardown( ulOutputStreamID );

      // reset the ability to set changes to properties
      CHXConfigurationAgentHelper::setAllowPropertyChanges( TRUE );

      // notify the Filter layer that a negotiated format must be set again
      SetFormatReady( FALSE );

      return res;
}
```

IHXTInputFilter Interface

```
DECLARE_INTERFACE_( IHXTInputFilter, IHXTFilter )
{
   STDMETHOD( SetAllocator ) ( UINT32 uStreamID, IHXTSampleAllocator
*pAllocator ) PURE;
   STDMETHOD( ReadSample )( UINT32 uStreamID, IHXTMediaSample** /*Out*/
ppSample ) PURE;
};
```

IHXTInputFilter::SetAllocator Method

Suggested allocator to use for a particular stream provided by a higher-level entity.

Application:

Provides an allocator for each of a plug-in's output streams for creation of media samples (only for the `IHXTInputFilter` and `IHXTTransformFilter` interfaces). The allocator can be reset at any time. The plug-in might not need an allocator (an example of this would be a plug-in that supports in-place transforms). A plug-in can provide its own allocator.

Requirements:

- Return `HXR_OK` if the plug-in does nothing with the allocator provided to it.
- A valid error code is `HXR_INVALID_PARAMETER`.
- Addref the allocator if you use it, and make sure it is released when the plug-in is destroyed.

Code Example:

Shows an allocator being AddReffed for later use in the creation of media samples.

```
STDMETHODIMP CRSGeneral::SetAllocator ( UINT32 ulOutputStreamID,
IHXTSampleAllocator* pAllocator )
{
   if ( ulOutputStreamID != 0 || pAllocator == NULL )
      return HXR_INVALID_PARAMETER;

   HX_RELEASE( m_pAllocator );
   m_pAllocator = pAllocator;
   m_pAllocator->AddRef();
   return HXR_OK;
}
```

IHXTIntputFilter::ReadSample Method

Outside entity pulls a media sample from the plug-in.

Application:

Native push models, such as QuickTime or DirectShow readers should internally buffer samples to convert push to pull when this method is called. To do this push/pull conversion, you may need to create your own media sample. In most pull cases, however, you obtain an `IHXTMediaSample` interface for the data using the allocator's `IHXTSampleAllocator::GetMediaSampleOfSize` method. Optionally call `IHXTMediaSample::SetDataSize` on the sample if the actual size is different than that what you requested in `IHXTSampleAllocator::GetMediaSampleOfSize`.

Requirements:

- Return `HXR_OK` if a valid sample is available.
- Return `HXR_S_NOT_HANDLED` if no sample is ready but more are expected. `IHXTInputFilter::ReadSample` will continue to be called.
- Return `HXR_S_END_OF_STREAM` when no more data is available or the end of the stream has been reached. The plug-in does not need to produce a sample marked with end of stream, as one will be manufactured by the system. After returning `HXR_S_END_OF_STREAM`, a plug-in can expect that `IHXTInputFilter::ReadSample` will not be called again.

- Return HXR_FAIL if the plug-in cannot produce a sample and is not able to recover. All calls to IHXTInputFilter::ReadSample on that stream will stop. If the plug-in can recover, it should return HXR_S_NOT_HANDLED instead.
- Set proper time stamps on the media sample. TimeStart (in the sample below) is the number of milliseconds from the start-of-buffer to start-of-stream.
- Set the TimeEnd field (if you don't know the sample duration, use the TimeStart value).
- Set the appropriate flags on the sample (such as HXT_SAMPLE_ENDOFSTREAM). See "IHXTMediaSample::SetSampleFlags" on page 271.
- The IHXTMediaSample::SetSampleFlag call clears previous flags—you must call IHXTMediaSample::GetSampleFlags prior to this call to preserve the previous flags.
- Set the appropriate fields on the sample (such as HXT_FIELD_STREAM_ID). See "IHXTMediaSample::SetSampleField" on page 270.

Code Example:

Illustrates typical things that occur on the IHXTInputFilter::ReadSample call.

```
STDMETHODIMP CRSInputPCM::ReadSample ( UINT32 uStreamID, IHXTMediaSample**
/*Out*/ ppSample )
{
   if (ulOutputStreamID != 0)
      return HXR_INVALID_PARAMETER;

   IHXTMediaSample* pOutSample;
   HX_RESULT res = m_spAllocator->GetMediaSampleofSize( m_uOnePCMFrameSize,
&pOutSample );

   // example reader
   if (SUCCEEDED(res))
   {
      UINT32 uBytesRead = m_pIBS->Read( pOutSample->GetDataStartForWriting(),
m_uOnePCMFrameSize );
      pOutSample->SetDataSize( uBytesRead );

      HXT_TIME TimeStart = (m_uTotalReadBytes /  m_uBytesPerSecond) * 1000.;
      uBytesRead += m_uTotalReadBytes;
```

```
        HXT_TIME TimeEnd = m_uTotalReadBytes /  m_uBytesPerSecond) * 1000.;

        pOutSample->SetSampleField( HXT_FIELD_STREAM_ID, uStreamID );

        pOutSample->SetTime( &TimeStart, &TimeEnd );

        if (uBytesRead != m_uOnePCMFrameSize)
        {
           pOutSample->SetSampleFlags( pOutSample->GetSampleFlags() |
HXT_SAMPLE_ENDOFSTREAM );
           res = HXR_S_END_OF_STREAM;
        }
        *ppSample = pOutSample;
     }
     return res;
}
```

IHXTTransformFilter Interface

```
DECLARE_INTERFACE_( IHXTTransformFilter, IHXTFilter )
{
  STDMETHOD( SetSampleSink ) ( UINT32 uStreamID, IHXTSampleSink* pOutputSink
) PURE;
   STDMETHOD( SetAllocator ) ( UINT32 uStreamID, IHXTSampleAllocator*
pAllocator ) PURE;
   STDMETHOD( ReceiveSample )( UINT32 uStreamID, IHXTMediaSample* pSample )
PURE;
};
```

IHXTTransformFilter::SetAllocator Method

This method is used the same way as the IHXTInputFilter::SetAllocator method.

IHXTTransformFilter::SetSampleSink

Destination to send processed media samples.

Application:

A sample sink will be provided to the plug-in for sending media samples to once they are ready. This call is guaranteed to come before IHXTTransformFilter::ReceiveSample or IHXTFilter::Prime is called. The plug-in can only send samples to the sink during the lifetime of a IHXTTransformFilter::ReceiveSample call. This is only ever an issue for a threaded transform plug-in.

Requirements:

- Valid error codes are HXR_INVALID_PARAMETER.
- Addref the output sink and make sure it is released by the time the plug-in is destroyed.

Code Example:

Illustrates the typical things that happen during a IHXTTransformFilter::SetSampleSink call.

```
STDMETHODIMP CRSExample::SetSampleSink ( UINT32 ulOutputStreamID,
IHXTSampleSink* pOutputSink )
{
   // check valid stream id and a valid Output sink
   if (ulOutputStreamID != 0 || pOutputSink == NULL)
      return HXR_INVALID_PARAMETER;

   HX_RELEASE( m_ pOutputSink );
   m_pOutputSink = pAllocator;
   m_pOutputSink ->AddRef();
   return HXR_OK;
}
```

IHXTTransformfilter::ReceiveSample Method

Handles the incoming media sample.

Application:

Typically in this method an outgoing media sample is obtained from the allocator using IHXTSampleAllocator::GetMediaSampleOfSize and the incoming media sample's data is extracted, transformed, and sent on to the next data sink with the outgoing media sample. In an "in place" transformation the data is processed "in place" in the data buffer of the incoming media sample.

Requirements:

- Return HXR_INVALID_PARAMETER for bad parameters or HXR_ENC_IMPROPER_STATE for an invalid state.
- Plug-ins must return a failure code if they generate an error during processing.
- Once an error is returned on a given stream ID, the plug-in will no longer receive data on that stream.

- Once a plug-in has received an error code from a sample sink, it must not send additional data to that sink.
- Plug-ins with a single output should always return the error from their `m_pOutputSink->ReceiveSample` call.
- Plug-ins with multiple outputs can choose to not return an error error code from a failed `m_pOutputSink->ReceiveSample` call so they can continue to send data to their remaining outputs.
- It is not necessary to call data processing operations on a zero-sized media sample.
- If the media sample's size is zero, you still must send the sample on to downstream plug-ins.
- Always check for the `HXT_SAMPLE_ENDOFSTREAM` flag and take appropriate actions when it is found.
- Set the proper time stamps on the outgoing sample (or copy them from the incoming sample using `IHXTMediaSample::CopyProperties`).
- Set any appropriate flags on the outgoing sample (such as `HXT_SAMPLE_ENDOFSTREAM`). See "IHXTMediaSample::SetSampleFlags" on page 271.
- Set any appropriate fields on the outgoing sample (see "IHXTMediaSample::SetSampleField" on page 270).
- If the plug-in obtained an outgoing sample from their assigned allocator (whether it contains data or not), they must release the `IHXTMediaSample` interface immediately following the call to the `m_pOutputSink->ReceiveSample`.
- An "in place" plug-in that operates on the incoming sample's buffer and merely passes the sample on to `m_pOutputSink->ReceiveSample` should not release the sample.
- Use the `CopyProperties` helper method `pOutSample->CopyProperties( pInSample )` to transfer sample properties such as timestamps, sample flags and fields from an incoming to an outgoing sample.

Code Example:

Illustrates typical things that happen during an input plug-in's `IHXTTransformFilter::ReceiveSample` call.

```
STDMETHODIMP CRSAnyTransFilter::ReceiveSample( UINT32 uInputStreamID,
IHXTMediaSample* pInSample )
{
   if (uInputStreamID != 0 || pInSample == NULL)
      return HXR_INVALID_PARAMETER;

   if ( !m_bReadyToOutputSamples )
      return HXR_ENC_IMPROPER_STATE;

   IHXTMediaSample* pOutSample = NULL;
   m_pAllocator->GetMediaSampleOfSize( pInSample->GetDataSize(), &pOutSample
);

   // Copied properties will pick up the end of stream and pass it on
   pOutSample->CopyProperties( pInSample );
   UINT32 uLen = pInSample->GetDataSize();
   UCHAR* pOut = pOutSample->GetBufferStartForWriting();
   UCHAR* pIn = pInSample->GetBufferStartForReading();

   // simple transform operation
   memcpy( pOut, pIn, uLen  );

   HX_RESULT res = m_spOutputSink->ReceiveSample( pOutSample );
   pOutSample->Release();
   return res;
}
```

IHXTOutputFilter Interface

```
DECLARE_INTERFACE_( IHXTOutputFilter, IHXTFilter )
{
   STDMETHOD( ReceiveSample )( UINT32 uStreamID, IHXTMediaSample* pSample )
PURE;
};
```

IHXTOutputFilter::ReceiveSample Method

Handles the incoming media sample.

Application:

This method is only similar to the IHXTTransformFilter::ReceiveSample method. The data in the media sample is still received and some kind of final processing is performed with it. However, no allocators or output sinks are used as this is the final destination for the data. Nor are any "in place" operations performed.

Requirements:

- Return HXR_INVALID_PARAMETER for bad parameters or HXR_ENC_IMPROPER_STATE for an invalid state.
- This method should track the HXT_SAMPLE_ENDOFSTREAM flags on different streams as they come through.
- When HXT_SAMPLE_ENDOFSTREAM is found on the final input stream, send the eEventStreamDone on the m_pEventSink->HandleEvent call.
- Optionally, when HXT_SAMPLE_ENDOFSTREAM is found on the final input stream, return HXR_S_END_OF_STREAM and an eEventStreamDone will be automatically generated for the plug-in.
- In all other cases (even when the media sample has a data size of zero), return HXR_OK.

Code Example:

Illustrates some typical things that happen during an output plug-in's IHXTOutputFilter::ReceiveSample call.

```
STDMETHODIMP CRSOutPCMFilter::ReceiveSample ( UINT32 ulInputStreamID,
IHXTMediaSample* pInSample )
{
   if (uInputStreamID != 0 || pInSample == NULL)
      return HXR_INVALID_PARAMETER;

   if ( !IsHeaderWritten() )
      WriteHeader();

   HX_RESULT res = HXR_OK;

   INT32 nLen = pInSample->GetDataSize();
   WriteFile( pInSample->GetBufferStartForReading(), nLen );
   AddToCumulative( nLen );

   if ( pInSample->GetSampleFlags() & HXT_SAMPLE_ENDOFSTREAM )
   {
      SeekToBeginning( );
      WritePCMTailerWaveBuffer( GetHeader(), GetCumulative() );
      WriteFile( m_pPCMHeader, GetHeaderSize() );
      DoClose();
         // return either this
         m_pEventSink-
>HandleEvent(eEventStreamDone,&ulInputStreamID,NULL,(IHXTConfigurationAgent
* )this);
```

```
        // or this
        res = HXR_OUTPUTFILTER_STREAM_DONE;
    }
    return res;
}
```

Configuration and Connection Agent Interfaces (Layer 2)

The agent layer is the next to the highest layer of the plug-in. It typically derives from the filter layer methods described in the previous sections. The agent layer negotiates connections between plug-ins, and handles all properties sent to and from the plug-in. It is also responsible for configuring plug-ins with the various strings, numbers, and types they need to carry out their operations.

The `IHXTConfigurationAgent` interface derives from the `IHXTPropertyBag` interface, and consequently must implement all of its methods. `IHXTConfigurationAgent` also adds an `IHXTConfigurationAgent::Initialize` method, which is called only once per plug-in. See "IHXTConfigurationAgent" on page 183 and "IHXTPropertyBag" on page 293 for information about the methods included with these interfaces.

Also included at this level is a helper class called `CHXConfigurationAgentHelper` which does most of the implementation work for you. Most plug-ins only need to override a few methods—`CHXConfigurationAgentHelper::OnInitialize`, `CHXConfigurationAgentHelper::OnSetUint`, and `CHXConfigurationAgentHelper::OnSetString`—to handle almost all property configuration situations. To support other data types, override the appropriate `CHXConfigurationAgentHelper::OnSet`*XXX* calls that apply to your plug-in. The `CHXConfigurationAgentHelper` class provided with the SDK includes the following methods:

- `CHXConfigurationAgentHelper::OnInitialize`
- `CHXConfigurationAgentHelper::OnSetString`
- `CHXConfigurationAgentHelper::OnSetUint`
- `CHXConfigurationAgentHelper::OnSetInt64`
- `CHXConfigurationAgentHelper::OnSetDouble`
- `CHXConfigurationAgentHelper::OnSetIntList`
- `CHXConfigurationAgentHelper::OnSetUintList`
- `CHXConfigurationAgentHelper::OnSetInt64List`
- `CHXConfigurationAgentHelper::OnSetDoubleList`

- CHXConfigurationAgentHelper::OnSetIntRange
- CHXConfigurationAgentHelper::OnSetUintRange
- CHXConfigurationAgentHelper::OnSetInt64Range
- CHXConfigurationAgentHelper::OnSetDoubleRange
- CHXConfigurationAgentHelper::OnSetInt
- CHXConfigurationAgentHelper::OnSetUnknown
- CHXConfigurationAgentHelper::OnSetProperty
- CHXConfigurationAgentHelper::OnRemove

Overriding the OnInitialize Method in Your Agent Class

A one time startup call to verify properties in the incoming property bag.

```
STDMETHOD(OnInitialize)( IHXTPropertyBag* pPropBag,
   IHXTPropertyBag* pConsumedPropBag,
   IHXTPropertyBag* pErrorBag
) {
   return HXR_OK;
}
```

Application:

Your configuration agent layer derives from the CHXConfigurationAgentHelper class that provides a default property bag implementation and override methods such as the default CHXConfigurationAgentHelper::OnInitialize method supplied in ihxtconfigagenthelper.h. In your configuration agent layer, OnInitialize checks that the required properties on a plug-in are set. For example, input plug-ins require an input file name to properly acquire properties of a file. Therefore, the OnInitialize call for input plug-ins validates that the input file name property has been set—as well as any other required settings. If it does not exist, the plug-in logs a warning and fails. Each plug-in can have a unique set of required properties that should be verified.

Requirements:

- OnInitialize should be called no more than once during the lifetime of the plug-in.
- Every plug-in should set GetActualPropertyBag().SetString(kPropPluginName, kValuePluginNameYourPlugin);.
- Prefilters should set GetActualPropertyBag().SetString(kPropPluginType, kValuePluginTypeUniquePluginType);.

- Following initialization, the plug-in's property bag will be populated with the set of properties accepted on the plug-in.
- Properties that can only be set at initialization time must use `OnInitialize` properties that can be set on-the-fly use the agent's `OnSet`*`XXX`* calls.
- If `OnInitialize` is given a property bag with an unsupported property or invalid value, the call fails and an error is logged.
- Use any of the `IHXTPropertyBag::Get`*`XXX`* calls to obtain the expected property fields—these can be strings, ints, property bags, and so on.
- Implement additional verification checks, such as whether a file loads and whether it contains the desired data types.
- Depending on what occurs during the initialization call, it might be necessary to update properties in other property bags.
- Once all properties are verified, call `pConsumedBag->Set`*`XXX`*`()` to place property fields in the plug-in's main property bag.
- If no property verification or plug-in initialization is required, it is not necessary to implement this method.
- Valid error codes are `HXR_POINTER`, `HXR_INVALID_PARAMETER`, `HXR_INVALID_FILE`, `HXR_BAD_FORMAT`, or other relevant specific initialization errors from `hxresult.h`.
- When it makes sense, the plug-in can implement a `PNCOM` configuration interface for which the client can `IUnknown::QueryInterface`. This is in the case where there is a general set of properties that make sense for a certain category of plug-ins and it is unlikely that the set of properties will change from one release to another.

Code Example:

An example of a configuration agent's `OnInitialize` call for an audio-only file reader.

```
STDMETHODIMP CRSGeneric::OnInitialize(  IHXTPropertyBag*
pInitBag,IHXTPropertyBag* pConsumed, IHXTPropertyBag* pError )
{
   if ( ! pInitBag || ! pConsumed )
      return HXR_POINTER;

   GetActualPropertyBag().SetString( kPropPluginName,
```

```
kValuePluginNameMyInput);
   GetActualPropertyBag().SetString( kPropPluginType, kValuePluginTypeMyInput);

   const char* pszFilename = NULL;
   HX_RESULT res = pInitBag ->GetString( kPropInputPathname, &pszFilename );
   if (SUCCEEDED(res))
   {
      if ( !LoadFile( pszFilename ) )
         res = HXR_INVALID_FILE;

      if (SUCCEEDED(res))
      {
        m_pOutputFormat->SetUint( kPropAudioChannelFormat, m_uChannelFormat
);
         m_pOutputFormat->SetUint( kPropAudioBitsPerSample, m_uSampleRate );
         m_pOutputFormat->SetUint( kPropAudioSampleFormat,
         HXT_MAKE_SAMPLE_FORMAT( HXT_ENDIANNESS_LITTLE, m_uBitDepth,
m_uBitDepth / 8 ));

         res = pConsumedPropBag->SetString( kPropInputPathname, pszFilename );
      }
   }
   return res;
}
```

Overriding the OnSetXXX Methods in Your Agent Class

```
STDMETHOD(OnSetUint)( const CHAR* pName, UINT32 uValue);
STDMETHOD(OnSetString)( const CHAR* pName, const CHAR *cszValue );
STDMETHOD(OnSetPropertyBag)( const CHAR* pName, IHXTPropertyBag *pValue);
```

> **Note:** OnSet*XXX* refers to the optional override methods in rtaconfigagenthelper.h.

Application:

You write your configuration agent's OnSetXXX methods to verify that only known properties are being set by the clients when they make one of their OnSet*XXX* calls on the plug-in. These calls might arrive at any time—even during data processing.

Requirements:

- If a user of the configuration agent attempts to set a property that the agent doesn't recognize, the OnSet*XXX*Type call should fail and an error should be logged.

- If an attempt is made to set a read-only property, the OnSet*XXX* call should fail (it is up to the plug-in to enforce this).
- If a set property implies an action, the plug-in can choose to invoke it in this method or defer it to a later call.
- Valid error codes are HXR_POINTER, HXR_INVALID_PARAMETER, HXR_ENC_IMPROPER_STATE, or other relevant specific initialization errors from hxresult.h.

Code Example:

Illustrates a configuration agent's OnSetString call that only recognizes kPropInputPathname as an accepted string.

```
STDMETHODIMP CRSInputPCM::OnSetString( const CHAR* szName, const CHAR*
szValue )
{
   HX_RESULT res = HXR_OK;
   // Validate params
   if (!szName || !szValue)
      return HXR_POINTER;

   if (!m_bAllowPropChanges)
      return HXR_ENC_IMPROPER_STATE;

   CPNString strName = szName;
   if ( strcmp( kPropInputPathname, szName ) == 0 )
      SetReaderFileName( szValue );
   else
      res = HXR_INVALID_PARAMETER;
   return res;
}
```

IHXTConnectionAgent Interface

The following table lists the IHXTConnectionAgent methods used with various types of plug-ins.

IHXTConnectionAgent Interface Methods (Layer3)

Input Plug-ins	Transform Plug-ins	Output Plug-ins
GetInputStreamCount	GetInputStreamCount	GetOutputStreamCount
GetSupportedInputFormat	GetSupportedInputFormat	GetSupportedOutputFormat
GetNegotiatedInputFormat	GetNegotiatedInputFormat	GetNegotiatedOutputFormat

(Table Page 1 of 2)

IHXTConnectionAgent Interface Methods (Layer3)

Input Plug-ins	Transform Plug-ins	Output Plug-ins
SetNegotiatedInputFormat	SetNegotiatedInputFormat	SetNegotiatedOutputFormat
	GetOutputStreamCount	
	GetSupportedOutputFormat	
	GetNegotiatedOutputFormat	
	SetNegotiatedOutputFormat	
	GetPreferredInputFormat (for future use)	
	GetPreferredOutputFormat (for future use)	

(Table Page 2 of 2)

IHXTConnectionAgent::GetInputStreamCount, IHXTConnectionAgent::GetSupportedInputFormat, IHXTConnectionAgent::GetOutputStreamCount, and IHXTConnectionAgent::GetSupportedOutputFormat identify which output streams from one plug-in will connect to which input streams of the next plug-in. These methods are called during the connection process in which media properties (input and output formats) associated with each stream are compared to determine the proper match between plug-ins. The final step of the connection process is when IHXTConnectionAgent::SetNegotiatedInputFormat and IHXTConnectionAgent::SetNegotiatedOutputFormat are called on a plug-in, identifying the exact media format properties supplied on each stream of the plug-in. These connection methods can be called many times until the various streams and plug-ins for a given operation are properly connected.

IHXTConnectionAgent::GetInputStreamCount Method

IHXTConnectionAgent::GetOutputStreamCount Method

The IHXTConnectionAgent::GetInputStreamCount method gets the number of possible input streams on a plug-in. The IHXTConnectionAgent::GetOutputStreamCount method gets the number of possible output streams on a plug-in.

Requirements:

- The number of streams may be variable rather than fixed depending on configuration or connection.

- Input plug-ins have only output streams, output plug-ins have only input streams, and transforms have both.

Code Examples:

Here is an implementation from a transform mux which supports multiple inputs, but only one output stream.

```
STDMETHODIMP_(UINT32) CRSExampleMux::GetInputStreamCount ()
{
   return m_nVideo + m_nAudio + m_nOther;
}

STDMETHODIMP_(UINT32) CRSExampleMux::GetOutputStreamCount ()
{
   return 1;
}
```

IHXTConnectionAgent::GetSupportedInputFormat Method

IHXTConnectionAgent::GetSupportedOutputFormat Method

The `IHXTConnectionAgent::GetSupportedInputFormat` method gets the supported format(s) for a particular input stream expressed using a property bag. The `IHXTConnectionAgent::GetSupportedOutputFormat` method gets the supported format(s) for a particular output stream expressed using a property bag.

Requirements:

- Other valid error codes are `HXR_POINTER` and `HXR_INVALID_PARAMETER`.

Code Examples:

Here is an implementation from a plug-in that supports multiple input formats—one audio and one video.

```
STDMETHODIMP CRSGeneric::GetSupportedInputFormat ( UINT32 ulInputStreamID,
                        IHXTPropertyBag** ppSupportedFormats )
{
   // Validate params
   if (!ppSupportedFormats)
      return HXR_POINTER;

   // This will greatly increase
   if (ulInputStreamID >= 2)
      return HXR_INVALID_PARAMETER;

   // Stream 0 is audio, Stream 1 is video
```

```
    if (ulInputStreamID == 0)
        *ppSupportedFormats = m_pInputFormatAudio;
    else
        *ppSupportedFormats = m_pInputFormatVideo;

    (*ppSupportedFormats)->AddRef();
    return HXR_OK;
}
```

IHXTConnectionAgent::SetNegotiatedInputFormat Method

IHXTConnectionAgent::SetNegotiatedOutputFormat Method

The `IHXTConnectionAgent::SetNegotiatedInputFormat` method provides notification to input stream data in a particular format. The `IHXTConnectionAgent::SetNegotiatedOutputFormat` method provides notification that an output stream will be receiving data in a particular format.

Requirements:

- Upon instantiation, no inputs or outputs are active. `IHXTConnectionAgent::SetNegotiatedInputFormat` or `IHXTConnectionAgent::SetNegotiatedOutputFormat` must be called to enable the stream.
- `IHXTConnectionAgent::SetNegotiatedInputFormat` and `IHXTConnectionAgent::SetNegotiatedOutputFormat` can be called multiple times. When possible, the plug-in should support this.
- Return a failure code if the format property bag contains properties it can't support or doesn't know about.
- If a plug-in requires certain output settings be set before it can receive data, it can set a flag when this call is made.
- Return `HXR_OK` if the plug-in accepts the assigned media format.
- `AddRef` any incoming media format. Make sure to release it later in the destructor.
- Valid error codes are `HXR_INVALID_PARAMETER` or `HXR_FAIL`.

Code Examples:

An example showing how the accepted connected format is AddReffed, and the state of the plug-in is set to accept data.

```
STDMETHODIMP CGeneric::SetNegotiatedOutputFormat ( UINT32 ulOutputStreamID,
                        IHXTPropertyBag* pOutputFormat )
{
   // Validate params -- this plugin will only ever have a single output
   if (ulOutputStreamID != 0 || pOutputFormat == NULL)
      return HXR_INVALID_PARAMETER;

   if ( SUCCEEDED ( res ))
   {
      HX_RELEASE( m_pConnectedOutputFormat );
      m_pConnectedOutputFormat = pOutputFormat;
      m_pConnectedOutputFormat->AddRef();

      // Validate state by setting a flag that will be checked on data flow calls
      if ( m_pConnectedOutputFormat && m_pAllocator && m_pOutputSink )
         m_bReadyToOutputSamples = TRUE;
   }
   return res;
}
```

Audio and Video Media Formats

The following fields make up the uncompressed audio media format and the uncompressed video media format. See the ihxtconstants.h file for more information on media formats.

Uncompressed audio will support the following properties:

```
kPropAudioChannelFormat[ ] = "audioChannelFormat";    // UINT32
kPropAudioSampleRate[ ] =    "audioSampleRate";       // UINT32
kPropAudioSampleFormat [ ] = " audioSampleFormat ";   // UINT32
```

> **Note:** Use the macros and enumerators supplied in the ihxtaudioformat.h file to condense audio channel and audio sample information.

Uncompressed video will support the following properties:

```
kPropVideoColorFormat[ ] =    "videoColorFormat";      // UINT32
kPropVideoFrameWidth[ ] =     "videoFrameWidth";       // UINT32
kPropVideoFrameHeight[ ] =    "videoFrameHeight";      // UINT32
kPropVideoFrameRate[ ] =      "videoFrameRate";        // double
```

Producer SDK Error Result Codes and Policies

The policy for choosing a result code is:

1. First check to see if one of the COM common errors is appropriate:
   - Return `HXR_INVALID_PARAMETER` if a parameter does not exist or has an incorrect value.
   - Return `HXR_POINTER` if an incoming pointer is null.
   - Return `HXR_OUTOFMEMORY` for any allocation or new failure.
   - Return `HXR_NOTIMPL` when a COM method on an interface isn't implemented.
   - Return `HXR_NOINTERFACE` when `IUnknown::QueryInterface` fails to find one.
2. Look next at these special case uses:
   - Checking the state of a plug-in:
     - Return `HXR_ENC_IMPROPER_STATE` if the filter is not in a state to accept the call.
   - Input readers:
     - Return `HXR_INVALID_STREAM` if the plug-in can handle at least one stream in a plug-in, but can't handle one or more others.
   - File handling:
     - Return `HXR_INVALID_FILE` if the file does not exist.
     - Return `HXR_BAD_FORMAT` if the file contains a format that cannot be handled.
3. Look through `hxresult.h` for other relevant error codes.
4. Only as a last resort, return `HXR_FAIL` when no other code seems appropriate.

Plug-in Samples

The Helix DNA Producer SDK provides two samples containing code that demonstrate how to use the SDK's plug-in capabilities. These samples are located in the following directories:

- `\producersdk\samples\inputplugin`
- `\producersdk\samples\prefilterplugin`

CHAPTER

6

REALMEDIA EDIT API

The RealMedia Edit API consists of a set of interfaces that allow you to edit existing RealMedia (.rm) files. The RealMedia Edit API supports both single rate and SureStream versions of .rm files. The RealMedia Edit API is divided into two main interfaces, which are referred to as the IHXRMEdit and IHXRMEvents interfaces.

With the IHXRMEdit interface, you can perform the following operations on a .rm file:

- Edit title, author, copyright, and comment fields.
- Modify Allow Recording and Allow Download settings.
- Trim the start and end times of a .rm file. This is referred to as a cut operation.
- Paste two or more .rm files together. This is referred to as a paste operation.
- Dump the contents of a .rm file to a text file. This is known as a dump operation.
- Add meta information to the file that is specific to your application.
- Obtain information on the types of streams contained in the .rm file.
- Obtain the size of the video image.
- Determine whether a .rm file is single rate or SureStream.

With the IHXRMEvents interface, you can perform the following operations on a .rm file:

- Add events and image maps to a .rm file.
- Dump events and image maps from a .rm file to a text file.

Editing RealMedia Files

This section describes how to use the RealMedia Edit API to create an application that can edit a .rm file. A sample application demonstrating the IHXRMEdit interface is located in the \samples\rmeditor directory.

Using the IHXRMEdit Interface

The following steps describe how to run an editing session with the IHXRMEdit interface.

1. Create the IHXRMEdit interface directly from the RealMedia tools DLL (rmto3260.dll on Windows and rmtools.so.6.0 on Linux) using the RMACreateRMEdit function. Query the returned IUnKnown pointer for the IHXRMEdit interface. For example:

```
IUnknown* pUnk = NULL;
IHXRMEdit* pEdit = NULL;
HX_RESULT  res = RMACreateRMEdit(&pUnk);
// Get the IHXRMEdit interface
if (SUCCEEDED(res))
{
   res = pUnk->QueryInterface(IID_IHXRMEdit,(void**)&pEdit);
}
HX_RELEASE(pUnk);   //no longer need the Iunknown pointer.
HX_RELEASE(pEdit);   // release the IHXRMEdit pointer when you are
                      // done using it.
```

2. Specify the path to the .rm file you want to edit using the IHXRMEdit::SetInputFile method. You must specify the full path to the file. An example of the Windows path would be:

   c:\\RealMedia\files\foo.rm.

   An example of the Linux path would be:

   /usr/local/RealMedia/files/foo.rm

   If you are pasting a number of .rm files together, call IHXRMEdit::SetInputFile with the path to the first .rm file and IHXRMEdit::AddInputFile for each of the remaining files. Note that start and end times are ignored during a paste operation; only entire files are pasted. Each of the files being pasted together must have been created with exactly the same encoding settings. Otherwise, the paste operation returns an error.

3. Specify the path to the output .rm file you want to contain the edited input file using the IHXRMEdit::SetOutputFile method. This file must be different than the input file. You must specify the full path to the file. If the file does not exit, the RealMedia Edit API will create a new file. If the file already exists, it will be overwritten.

4. Specify the Title, Author, Copyright, and Comment strings using the appropriate IHXRMEdit::Set*XXX* method. For example, to set the Title field of the .rm file, use the IHXRMEdit::SetTitle method.

5. If you want to trim the beginning of the input .rm file, specify a start time using the IHXRMEdit::SetStartTime method. You can specify the start time in milliseconds or using a string containing the start time in the format Days:Hours:Minutes:Seconds:Milliseconds (0:0:0:0:0).

6. If you want to trim the end of the input .rm file, specify an end time using the IHXRMEdit::SetEndTime method. You can specify the end time in milliseconds or using a string containing the start time in the format Days:Hours:Minutes:Seconds:Milliseconds (0:0:0:0:0). You cannot specify an end time earlier than the start time. Specifying an end time greater than the duration of the input .rm file results in the end time occurring at the end of the input file (EOF). You can also specify an endtime of 0 to indicate that you want the duration of the input file to be used. If you do not call IHXRMEdit::SetEndTime, the duration of the input file will be used.

7. Use the appropriate IHXRMEdit::Set*XXX* method if you want to specify the settings for the Allow Recording or Allow Download features of the output .rm file. For example, use the IHXRMEdit::SetSelectiveRecord method to modify the Allow Download setting.

8. To process the edit, you must call the IHXRMEdit::Process method. This method opens the input file and copies it to the output file according to the settings you specified in the previous steps.

9. If any of the methods return an error, you can convert the error number into an error string using the IHXRMEdit::GetErrorString method.

10. When you have finished editing, call HX_RELEASE on the IHXRMEdit interface you created in Step 1.

Sample code demonstrating the use of the IHXRMEdit interface is in the file \samples\rmeditor\crmedap.cpp.

Using the IHXRMEdit2 Interface

The IHXRMEdit2 interface was added to the RealMedia Edit API to give you access to more information about a .rm file. This interface is intended to be used in conjunction with the IHXRMEdit interface. For example, you must specify an input file with the IHXRMEdit::SetInputFile method before you can use any of the methods in the IHXRMEdit2 interface.

1. Create the IHXRMEdit interface directly from the RealMedia tools DLL (rmto3260.dll on Windows and rmtools.so.6.0 on Linux) using the RMACreateRMEdit function. Query the returned IUnKnown pointer for the IHXRMEdit2 interface. For example:

```
IUnknown* pUnk = NULL;
IHXRMEdit* pEdit = NULL;
IHXRMEdit2* pEdit2 = NULL;
HX_RESULT  res = RMACreateRMEdit(&pUnk);
// Get the IHXRMEdit interface
if (SUCCEEDED(res))
{
   res = pUnk->QueryInterface(IID_IHXRMEdit,(void**)&pEdit);
}
// Get the IHXRMEdit2 interface
if (SUCCEEDED(res))
{
   res = pUnk->QueryInterface(IID_IHXRMEdit2,(void**)&pEdit2);
}
HX_RELEASE(pUnk);   // no longer need the Iunknown pointer.
HX_RELEASE(pEdit);   // release the IHXRMEdit pointer when you are
                     //  done using it.
HX_RELEASE(pEdit2); // release the IHXRMEdit2 pointer when you are
                     // done using it.
```

2. Set the path to the input .rm file using the IHXRMEdit::SetInputFile method.
3. Use one of the IHXRMEdit2::Has*XXX* methods to determine if the .rm file contains audio, video, events, or image maps. For example, to determine if the file contains audio, use the IHXRMEdit2::HasAudio method.
4. If the input .rm file contains video, you can determine the size of the video image using the IHXRMEdit2::GetVideoSize method.
5. Newer .rm files have a meta information section that contains information about when the file was created and modified, the application used to create the file, and the audio and video target setting used to create the

file. This information is managed by an IHXValue interface, where each piece of information is stored as a name/value pair. For more information about the IHXValue interface, refer to the *Helix SDK Developer's Guide*, which can be found at **http://www.realnetworks.com/resource/sdk/index.html**. The IHXRMEdit2::GetMetaInformation method enables you to access the IHXValues object containing the meta information.

This version of the RealMedia Edit API allows you to set the following name/value pairs in the IHXValues object.

Tag	Name	Value Type	Description
RM_PROPERTY_GENERATOR	"Generated By"	string	The name of the application that created the file. This property defaults to a value of "Helix DNA Producer SDK [version] [platform]" when using the Helix DNA Producer SDK. Your application can override this property with your own application name and version information string.

All tags starting with RM_ are reserved by RealNetworks. You can add your own name/value pairs specific to your application.

The following properties are automatically added by the encoding engine.

Tag	Name	Value Type	Description
RM_PROPERTY_CREATION_DATE	"Creation Date"	string	The date the .rm file was first created.
RM_PROPERTY_MODIFICATION_DATE	"Modification Date"	string	The date the .rm file was last modified.
RM_PROPERTY_TARGET_AUDIENCES	"Audiences"	string	The Target Audience settings used to create this file.
RM_PROPERTY_AUDIO_FORMAT	"audioMode"	string	The audio format setting used to create this file, that is music or voice.
RM_PROPERTY_VIDEO_QUALITY	"videoMode"	string	The video quality setting used to create this file, that is normal, smooth, sharp, or slideshow.

An example of how read the name/value pairs using the IHXRMEdit2::GetMetaInformation method is located in the CRMEditApp::DisplayMetaInformation method in the file \samples\rmeditor\crmedap.cpp. An example of how to set name/value pairs is located in the CRMEditApp::SetMetaInformationString method in the same file.

Using the IHXRMEdit3 Interface

The RealMedia Edit API also includes the IHXRMEdit3 interface. This interface provides access to callbacks that furnish status information during the editing process.

The status information is provided through the IHXProgressSink interface. The status information provided by this interface includes notification that the process has started or stopped, and includes continuous callbacks that indicate how far the process has progressed (in percent). See "Using the IHXProgressSink Interface" on page 157 for more information.

Using the IHXRMFileSink Interface

The IHXRMEdit interface supports the use of a IHXRMFileSink interface. The IHXRMFileSink interface enables your application to register itself to receive callbacks containing the .rm file headers and the data packets before the headers and data packets are written to the output .rm file. These callbacks enable your application to modify or encrypt the headers and data packets before they are written to the file. An example of using the IHXRMFileSink interface is located in the file \samples\rmeditor\crmedap.cpp

To enable the sample code, make sure you modify the following line:

```
#define USE_RMFILESINK 0    // set USE_RMFILESINK to 1 in order to activate
                              // the RMFileSink code
```

1. Your application must implement the IHXRMFileSink interface.
2. Register your IHXRMFileSink interface with the IHXRMEdit interface using the IHXRMEdit::SetRMFileSink method.
3. After setting the input and output .rm files, call the IHXRMEdit::Process method to begin the copying process.
4. During the copying of headers from the input file to the output file, the IHXRMFileSink::OnMediaPropertyHeader method will be called for each

MediaProperties header being written to the output file. You will receive an IHXValues pointer you can use to read or modify the following fields:

Tag	Name	Value Type	Description
RM_MEDIA_PROP_STREAM_NUMBER	"RM_MEDIA_PROP_STREAM_NUMBER"	UINT32	The stream number of the MediaProperties header. Note: Do not modify this field; this is a read only field.
RM_MEDIA_PROP_MIMETYPE	"RM_MEDIA_PROP_MIMETYPE"	C String	The mime type for the stream. If you alter the data packets or any of the fields in the MediaProperties header, you should set a new mime type for the stream.
RM_MEDIA_PROP_TYPE_SPECIFIC_DATA	"RM_MEDIA_PROP_TYPE_SPECIFIC_DATA"	Buffer	The type specific data for the media stream. This data is used by the renderer in the RealPlayer to decode the stream. If you modify this data through encryption or add any additional information to this buffer, you must ensure that the original type specific data can be restored, otherwise playback of this stream will fail.

5. During the copying of packets from the input file to the output file, the IHXRMFileSink::OnPacket method will be called for each data packet being written to the output file. The data packet information is managed by an IHXPacket interface. You will need to retrieve the data packet from the IHXPacket, modify it, then set the data back into the IHXPacket. The following sample code from the CRMFileSink::OnPacket method in the sample application demonstrates how to access and modify the data packet.

```
STDMETHODIMP CRMFileSink::OnPacket(IHXPacket* pMediaPacket, BOOL
bIsKeyFrame)
{
   HX_RESULT res = HXR_OK;
   // get pointer to RMEditor SDK
   IHXRMEdit* pEdit =m_pOwnerApp->GetEditSDK();
   // Note: See /include/ihxpkts.h for the methods available in the
   // IHXPacket interface.
   // get the IHXBuffer containing the packet data
   IHXBuffer* pBuffer = pMediaPacket->GetBuffer();
```

```
if(pBuffer)
{
   // allocate a buffer to hold the packet data
   UINT32 ulDataSize = pBuffer->GetSize();
   UINT8* pTempBuf = new UINT8[ulDataSize];

   // copy the packet data into the temporary buffer
   memcpy(pTempBuf,pBuffer->GetBuffer(),ulDataSize);

   IHXBuffer* pDataBuffer = NULL;

   // create a new IHXBuffer to hold your modified type specific
   // data
   res = pEdit->CreateIHXBuffer(&pDataBuffer);
   if(SUCCEEDED(res))
   {
      pDataBuffer->AddRef();
      // modify the packet data here. In this example we will
      // just return the same packet data packet to the caller.
      // We first need to set the packet data into the new
      // IHXBuffer.
      res = pDataBuffer->Set(pTempBuf,ulDataSize);
   }
   // We now need to set the IHXBuffer into the IHXPacket.
   if(SUCCEEDED(res))
   {
      pMediaPacket->Set(pDataBuffer,
      pMediaPacket->GetTime(),
      pMediaPacket->GetStreamNumber(),
      pMediaPacket->GetASMFlags(),
      pMediaPacket->GetASMRuleNumber());
   }
   if(pTempBuf)
   {
      delete [] pTempBuf;
   }

   // create a new IHXBuffer to hold your modified type specific
   // data
   res = pEdit->CreateIHXBuffer(&pDataBuffer);
   if(SUCCEEDED(res))
   {
      pDataBuffer->AddRef();
      // modify the packet data here. In this example we will
      // just return the same packet data packet to the caller.
      // We first need to set the packet data into the new
      // IHXBuffer.
```

```
            res = pDataBuffer->Set(pTempBuf,ulDataSize);
        }
        // We now need to set the IHXBuffer into the IHXPacket.
        if(SUCCEEDED(res))
        {
            pMediaPacket->Set(pDataBuffer,
            pMediaPacket->GetTime(),
            pMediaPacket->GetStreamNumber(),
            pMediaPacket->GetASMFlags(),
            pMediaPacket->GetASMRuleNumber());
        }
        if(pTempBuf)
        {
            delete [] pTempBuf;
        }
        HX_RELEASE(pDataBuffer);
        HX_RELEASE(pBuffer);
    }
    return res;
}
```

6. After all of the data packets have been copied to the output file, your `IHXRMFileSink::OnMediaPropertyHeader` method will be called again for each MediaProperties header being written to the output file. You should repeat the actions you took in Step 4 to ensure the proper updating of the headers to the output file.

Processing Events

This section describes how to use the RealMedia Edit API to create an application that can modify events and image maps in a `.rm` file. A sample application demonstrating the `IHXRMEvents` interface is located in the `\samples\rmevents` directory.

Using the IHXRMEvents Interface

The following steps describe how to modify the events and image maps in a `.rm` file with the `IHXRMEvents` interface.

1. Create the `IHXRMEvents` interface directly from the RealMedia tools DLL (`rmto3260.dll` on Windows and `rmtools.so.6.0` on Linux) using the

RMACreateRMEvents function. Query the returned IUnKnown pointer for the IHXRMEvents interface. For example:

```
IUnknown* pUnk = NULL;
IHXRMEvents* pEvents = NULL;
HX_RESULT  res = RMACreateRMEvents(&pUnk);
// Get the IHXRMEvents interface
if (SUCCEEDED(res))
{
   res = pUnk->QueryInterface(IID_IHXRMEvents,(void**)&pEvents);
}
HX_RELEASE(pUnk);       // no longer need the Iunknown pointer.
HX_RELEASE(pEvents);    // release the IHXRMEvents pointer when you are
                         // done using it.
```

2. Specify the path to the .rm file you want to add events or image maps to using the IHXRMEvents::SetInputFile method. You must specify the full path to the file. An example of the Windows path would be:

   c:\\RealMedia\files\foo.rm.

   An example of the Linux path would be:

   /usr/local/RealMedia/files/foo.rm

3. Optionally specify the path to the input text file that contains the events that you want to merge into the output .rm file.

4. Optionally specify the path to the input text file that contains the image maps that you want to merge into the output .rm file.

5. Specify the path to the output .rm file you want to contain the merged input files using the IHXRMEvents::SetOutputFile method. This file must be different than the input file. You must specify the full path to the file. If the file does not exit, the RealMedia Edit API will create a new file. If the file already exists, it will be overwritten.

6. To merge the input files, you must call the IHXRMEvents::Process method. This method will open the input file, convert the events and image maps into the correct format, and merge them to the output file according to the settings you specified in the previous steps.

7. If any of the methods return an error, you can convert the error number into an error string using the IHXRMEvents::GetErrorString method.

8. When you are done with the interface, call HX_RELEASE on the IHXRMEvents interface you created in Step 1.

Using the IHXRMEvents2 Interface

The RealMedia Edit API also includes the IHXRMEvents2 interface. This interface provides access to callbacks that furnish status information during processing of events.

The status information is provided through the IHXProgressSink interface. The status information provided by this interface includes notification that the process has started or stopped, and includes continuous callbacks that indicate how far the process has progressed (in percent). See "Using the IHXProgressSink Interface" on page 157 for more information.

Dumping Events and Image Maps from a .rm File

The IHXRMEvents interface enables you to dump events and image maps contained in a .rm file to text files. You can then edit events and image maps contained in a .rm file.

1. Create the IHXRMEvents interface directly from the RealMedia tools DLL (rmto3260.dll on Windows and rmtools.so.6.0 on Linux) using the RMACreateRMEvents function. Query the returned IUnKnown pointer for the IHXRMEvents interface.
2. Specify the path to the .rm file containing the events or image maps using the IHXRMEvents::SetInputFile method.
3. Specify the path to the root file name that will contain the events and image maps using the IHXRMEvents::SetDumpFile method. For example, if you set the dump root path to c:\RealMedia\file\foo all events in the input file will be dumped into the file c:\RealMedia\file\foo_evt.txt and all image maps will be dumped to the file c:\RealMedia\file\foo_imap.txt.
4. Call IHXRMEvents::Process. All events in the input file will be dumped in text format to the output xxx_evt.txt file. All image maps in the input file will be dumped in text format to the output xxx_imap.txt file.
5. When you are done with the interface, call HX_RELEASE on the IHXRMEvents interface you created in Step 1.

Using the IHXProgressSink Interface

The IHXProgressSink interface provides state and progress callbacks from the object that performs editing of .rm files. In addition, you can also get state and

progress callbacks for the object that encodes events and image maps. These callbacks are useful because the calling application can provide feedback to the user during potentially long editing and encoding processes.

To set up callbacks from an editing session:

1. Create the IHXRMEdit interface directly from the RealMedia tools DLL (rmto3260.dll on Windows and rmtools.so.6.0 on Linux) using the RMACreateRMEdit function. Query the returned IUnKnown pointer for the IHXRMEdit3 interface. For example:

```
IUnknown* pUnk = NULL;
IHXRMEdit* pEdit = NULL;
IHXRMEdit3* pEdit3 = NULL;
HX_RESULT  res = RMACreateRMEdit(&pUnk);
// Get the IHXRMEdit interface
if (SUCCEEDED(res))
{
   res = pUnk->QueryInterface(IID_IHXRMEdit,(void**)&pEdit);
}
// Get the IHXRMEdit3 interface
if (SUCCEEDED(res))
{
   res = pUnk->QueryInterface(IID_IHXRMEdit3,(void**)&pEdit3);
}
HX_RELEASE(pUnk);   // no longer need the Iunknown pointer.
HX_RELEASE(pEdit);   // release the IHXRMEdit pointer when you are
                     //  done using it.
HX_RELEASE(pEdit3); // release the IHXRMEdit3 pointer when you are
                     // done using it.
```

2. Register the IHXRMProgressSink interface with the IHXRMEdit3 interface using the IHXRMEdit3::AddSaveProgressSink method.

During processing, you will receive a call to IHXProgressSink::NotifyStart, a number of IHXProgressSink::SetProgress calls, then a call to IHXProgressSink::NotifyFinish. After the return from the IHXRMEdit::Process call, you should call the IHXRMEdit3::RemoveSaveProgressSink method if you have no more processing to do on other files.

To set up callbacks when processing events:

1. Create the IHXRMEvents interface directly from the RMTOOLS DLL (rmto3260.dll on Windows and rmtools.so.6.0 on Linux) using the RMACreateRMEvents function. Query the returned IUnKnown pointer for the IHXRMEvents2 interface. For example:

```
IUnknown* pUnk = NULL;
IHXRMEvents* pEvents = NULL;
IHXRMEvents2* pEvents2 = NULL;
HX_RESULT  res = RMACreateRMEvents(&pUnk);
// Get the IHXRMEvents interface
if (SUCCEEDED(res))
{
   res = pUnk->QueryInterface(IID_IHXRMEvents,(void**)&pEvents);
}
// Get the IHXRMEvents2 interface
if (SUCCEEDED(res))
{
   res = pUnk->QueryInterface(IID_IHXRMEvents2,(void**)&pEvents2);
}
HX_RELEASE(pUnk);       // no longer need the Iunknown pointer.
HX_RELEASE(pEvents);    // release the IHXRMEvents pointer when you are
                        //  done using it.
HX_RELEASE(pEvents2);  // release the IHXRMEvents2 pointer when you are
                         // done using it.
```

2. Register the IHXRMProgressSink interface with the IHXRMEvents2 interface using the IHXRMEvents2::AddSaveProgressSink method.

During processing, you will receive a call to IHXProgressSink::NotifyStart, a number of IHXProgressSink::SetProgress calls, then a call to IHXProgressSink::NotifyFinish. After the return from the IHXRMEvents::Process call, you should call the IHXRMEvents2::RemoveSaveProgressSink method if you have no more processing to do on other files.

RealMedia Samples

The Helix DNA Producer SDK provides two samples containing code that demonstrate how to use the SDK's RealMedia edit and event capabilities. These samples are located in the following directories:

- \producersdk\samples\rmeditor
- \producersdk\samples\rmevents

APPENDIX

A

INTERFACE LIST

IHXTAsmConnectionProperty

Header file: `ihxtbase.h`

This interface is reserved for future use.

IHXTAsmHeaderSource

Header file: `ihxtbase.h`

This interface is reserved for future use.

IHXTAsmHeaderSink

Header file: `ihxtbase.h`

This interface is reserved for future use.

IHXTAsmHeaderTransform

Header file: `ihxtbase.h`

This interface is reserved for future use.

IHXTAudience

Purpose:	Defines a set of possible audio/video streams for a target bit rate.
Implemented by:	Encoding
Header file:	ihxtencodingjob.h

This interface manipulates a list of stream configurations (typically audio and video codecs). This interface inherits the configuration methods of IHXTConfigurationAgent, which are used to configure the audience properties.

> **Note:** For more information on the audience properties that can be configured by this interface, see "Audiences" on page 56.

The IHXTAudience interface contains the following methods:

- IHXTAudience::AddStreamConfig
- IHXTAudience::GetStreamConfig
- IHXTAudience::GetStreamConfigCount
- IHXTAudience::MoveStreamConfig
- IHXTAudience::RemoveStreamConfig

As with all Component Object Model (COM) interfaces, the IHXTAudience interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTAudience::AddStreamConfig

Adds a stream configuration to the back of the stream configuration index list.

```
STDMETHOD(AddStreamConfig) (
   THIS_
   IHXTStreamConfig* pStreamDef
) PURE;
```

pStreamDef
Pointer to an IHXTStreamConfig interface that manages the stream configuration to add.

IHXTAudience::GetStreamConfig

Retrieves the specified stream configuration at index list.

```
STDMETHOD(GetStreamConfig) (
   THIS_
   UINT32 ulIndex,
   IHXTStreamConfig** ppStreamDef
) PURE;
```

ulIndex
: The location of the stream configuration in the index list.

ppStreamDef
: Address of a pointer to an `IHXTStreamConfig` interface that manages the stream configuration information.

IHXTAudience::GetStreamConfigCount

Returns the number of stream configurations in the index list.

```
STDMETHOD_(UINT32, GetStreamConfigCount) (
   THIS
) PURE;
```

IHXTAudience::MoveStreamConfig

Moves a stream configuration from its original location in the index list to a new location.

```
STDMETHOD(MoveStreamConfig) (
   UINT32 ulOrigIndex,
   UINT32 ulDestIndex
) PURE;
```

ulOrigIndex
: The original location of the stream configuration in the index list.

ulDestIndex
: The destination to which the stream configuration is to be moved.

IHXTAudience::RemoveStreamConfig

Removes the specified stream configuration from the index list.

```
STDMETHOD(RemoveStreamConfig) (
   THIS_
   UINT32 ulIndex
) PURE;
```

ulIndex
The location in the index list of the stream configuration to be removed.

IHXTAudienceEnumerator

Purpose:	Searches a given directory for audience template files and provides an enumerator for deserialized audience objects.
Implemented by:	Encoding
Header file:	ihxtencodingjob.h

The IHXTAudienceEnumerator interface contains the following methods:

- IHXTAudienceEnumerator::GetAudience
- IHXTAudienceEnumerator::GetAudienceCount
- IHXTAudienceEnumerator::SetProfileDirectory
- IHXTAudienceEnumerator::SetProfileExtension

As with all Component Object Model (COM) interfaces, the IHXTAudienceEnumerator interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTAudienceEnumerator::GetAudience

Gets audience information from the specified file.

```
STDMETHOD(GetAudience) (
   THIS_
   UINT32 ulIndex,
   IHXTAudience** ppAudience,
   const char** ppszFilename
) PURE;
```

ulIndex
The location of the audience in the index list.

ppAudience
: Address of a pointer to an `IHXTAudience` interface that manages the audience information.

ppszFilename
: Address of a pointer to the pathname of the file that contains the audience template.

IHXTAudienceEnumerator::GetAudienceCount

Returns the number of audiences in the index list.

```
STDMETHOD_(UINT32, GetAudienceCount) (
   THIS
) PURE;
```

IHXTAudienceEnumerator::SetProfileDirectory

Sets the directory location in which the audience enumerator looks for files. The files being enumerated contain profiles (self-contained audience definitions), and generally end in a .rpad extension, such as `56k Dial-up.rpad` and `150k LAN.rpad`.

```
STDMETHOD(SetProfileDirectory) (
   THIS_
   const char* szDirectoryPath
) PURE;
```

szDirectoryPath
: Pointer to directory location of the profile files.

IHXTAudienceEnumerator::SetProfileExtension

Sets the profile filename extension. By default, the profile filename extension is `.rpad`.

```
STDMETHOD(SetProfileExtension) (
   THIS_
   const char* szProfileExtension
) PURE;
```

szProfileExtension
: Pointer to the filename extension.

IHXTAudienceEnumerator2

Purpose:	Provides an enumerator for deserialized audience objects for automatic codec selection.
Implemented by:	Encoding
Header file:	ihxtencodingjob.h

This interface provides additional enumeration capabilities in addition to those provided by IHXTAudienceEnumerator. With this interface you can force deserialization of audience files during initialization, and provide an IHXTCodecUpdater interface for automatic codec selection. This method inherits all of the methods from the IHXTAudienceEnumerator interface.

The IHXTAudienceEnumerator2 inteface contains the following methods:

- IHXTAudienceEnumerator2::GetCodecUpdater
- IHXTAudienceEnumerator2::GetForceInitialize
- IHXTAudienceEnumerator2::GetProfileDirectory
- IHXTAudienceEnumerator2::GetProfileExtension
- IHXTAudienceEnumerator2::SetCodecUpdater
- IHXTAudienceEnumerator2::SetForceInitialize

As with all Component Object Model (COM) interfaces, the IHXTAudienceEnumerator2 interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTAudienceEnumerator2::GetCodecUpdater

Returns the location of the codec updater used in the automatic codec selection.

```
STDMETHOD(GetCodecUpdater) (
   THIS_
   IHXTCodecUpdater** ppCodecUpdater
) PURE;
```

ppCodecUpdater

Address of a pointer to an IHXTCodecUpdater interface that finds the mapping file containing the codec and updates an audience with the codec information.

IHXTAudienceEnumerator2::GetForceInitialize

Determines if forced deserialization of an audience file will occur during initialization.

```
STDMETHOD(GetForceInitialize) (
   THIS_
   BOOL* pbForceInitialize
) PURE;
```

pbForceInitialize
: If false (default), the producer will not force the deserialization of any audience file that contains non-existant codecs. If this parameter is true, the producer is forced to deserialize the audience file even with a non-existant codec.

IHXTAudienceEnumerator2::GetProfileDirectory

Gets the directory location from which the audience enumerator looks for files. The files being enumerated contain profiles (self-contained audience definitions), and generally end in a .rpad extension, such as `56k Dial-up.rpad` and `150k LAN.rpad`.

```
STDMETHOD(GetProfileDirectory) (
   THIS_
   const char** cpszDirectoryPath
) PURE;
```

cpszDirectoryPath
: Address of a pointer to the directory location of the profile files.

IHXTAudienceEnumerator2::GetProfileExtension

Gets the profile filename extension. Generally, the profile filename extension is `.rpad`.

```
STDMETHOD(GetProfileExtension) (
   THIS_
   const char** cpszProfileExtension
) PURE;
```

cpszProfileExtension
: Address of a pointer to the filename extension.

IHXTAudienceEnumerator2::SetCodecUpdater

Sets the codec updater interface to be used in automatic codec selection.

```
STDMETHOD(SetCodecUpdater) (
   THIS_
   IHXTCodecUpdater* pCodecUpdater
) PURE;
```

pCodecUpdater
: Pointer to an IHXTCodecUpdater interface that manages the mapping file containing the codec, and is used to update an audience with the codec information.

IHXTAudienceEnumerator2::SetForceInitialize

Forces deserialization of an audience file during intialization.

```
STDMETHOD(SetForceInitialize) (
   THIS_
   BOOL bForceInitialize
) PURE;
```

bForceInitialize
: If set to false (default), the producer will not force the deserialization of any audience file that contains non-existant codecs. If this parameter is set to true, the producer is forced to deserialize the audience file even with a non-existant codec.

IHXTAudioLevelChannel

Purpose:	Communicates audio level data on a per-channel basis.
Implemented by:	Encoding
Header file:	ihxtbase.h

This interface exposes methods to get and set attributes of current, peak, peak hold, and clipped energy levels. This interface supports audio preview, particularly in the form of VU meters that need processed volume levels.

The IHXTAudioLevelChannel interface contains the following methods:

- IHXTAudioLevelChannel::GetClipped
- IHXTAudioLevelChannel::GetEnergy
- IHXTAudioLevelChannel::GetPeak
- IHXTAudioLevelChannel::GetPeakHold

- IHXTAudioLevelChannel::Initialize
- IHXTAudioLevelChannel::SetClipped
- IHXTAudioLevelChannel::SetEnergy
- IHXTAudioLevelChannel::SetPeak
- IHXTAudioLevelChannel::SetPeakHold

As with all Component Object Model (COM) interfaces, the IHXTAudioLevelChannel interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTAudioLevelChannel::GetClipped

Indicates whether the audio level channel is clipped. Returns TRUE if the audio is clipped, or FALSE if it is not.

```
STDMETHOD_(BOOL, GetClipped) (
) PURE;
```

IHXTAudioLevelChannel::GetEnergy

Indicates the audio level, in decibels.

```
STDMETHOD_(float, GetEnergy) (
) PURE;
```

IHXTAudioLevelChannel::GetPeak

Indicates the peak audio level, in decibels.

```
STDMETHOD_(float, GetPeak) (
) PURE;
```

IHXTAudioLevelChannel::GetPeakHold

Indicates the maximum audio level, in decibels.

```
STDMETHOD_(float, GetPeakHold) (
) PURE;
```

IHXTAudioLevelChannel::Initialize

This method is used internally by Helix DNA Producer.

```
STDMETHOD(Initialize) (
   float fCurrent,
   float fPeak,
   float fPeakHold,
   BOOL bClipped
) PURE;
```

IHXTAudioLevelChannel::SetClipped

This method is used internally by Helix DNA Producer.

```
STDMETHOD(SetClipped) (
   BOOL fClipped
) PURE;
```

IHXTAudioLevelChannel::SetEnergy

This method is used internally by Helix DNA Producer.

```
STDMETHOD(SetEnergy) (
   float fEnergy
) PURE;
```

IHXTAudioLevelChannel::SetPeak

This method is used internally by Helix DNA Producer.

```
STDMETHOD(SetPeak) (
   float fPeak
) PURE;
```

IHXTAudioLevelChannel::SetPeakHold

This method is used internally by Helix DNA Producer.

```
STDMETHOD(SetPeakHold) (
   float fPeakHold
) PURE;
```

IHXTAudioLevelChannels

Purpose:	Exposes multiple channels of audio level data.
Implemented by:	Encoding
Header file:	ihxtbase.h

Together with IHXTAudioLevelChannel, this interface facilitates audio preview support. This interface can be implemented or attached to a media sample and forwarded using existing preview sink interfaces.

The IHXTAudioLevelChannels interface contains the following methods:

- IHXTAudioLevelChannels::AddChannel
- IHXTAudioLevelChannels::GetChannel
- IHXTAudioLevelChannels::GetChannelCount
- IHXTAudioLevelChannels::RemoveChannel
- IHXTAudioLevelChannels::SetChannel

As with all Component Object Model (COM) interfaces, the IHXTAudioLevelChannels interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTAudioLevelChannels::AddChannel

Adds an audio level channel.

```
STDMETHOD(AddChannel) (
   IHXTAudioLevelChannel *pAudioLevel,
   UINT32 *puChannel = NULL
) PURE;
```

pAudioLevel
: Pointer to an IHXTAudioLevelChannel interface that manages the audio level information for the channel that was added.

puChannel
: Pointer to audio channel that was added.

IHXTAudioLevelChannels::GetChannel

Gets the audio level information for the specified channel.

```
STDMETHOD(GetChannel) (
   UINT32 uChannel,
   IHXTAudioLevelChannel **ppAudioLevel
) PURE;
```

uChannel
: The channel from which to get the audio level information.

ppAudioLevel
: Address of a pointer to an `IHXTAudioLevelChannel` interface that manages the audio level information.

IHXTAudioLevelChannels::GetChannelCount

Returns the number of channels in the audio.

```
STDMETHOD_(UINT32, GetChannelCount) (
) PURE;
```

IHXTAudioLevelChannels::RemoveChannel

Removes the specified audio level channel.

```
STDMETHOD(RemoveChannel) (
   UINT32 uChannel
) PURE;
```

uChannel
: The audio level channel to be removed.

IHXTAudioLevelChannels::SetChannel

This method is used internally by Helix DNA Producer.

```
STDMETHOD(SetChannel) (
   UINT32 uChannel,
   IHXTAudioLevelChannel *pAudioLevel
) PURE;
```

IHXTAudioPinFormat

Purpose:	Specifies the format of audio samples that will be passed to the encoding engine.
Implemented by:	Encoding
Header file:	ihxtbase.h

The IHXTAudioPinFormat interface contains the following methods:

- IHXTAudioPinFormat::GetChannelFormat
- IHXTAudioPinFormat::GetSampleFormat
- IHXTAudioPinFormat::GetSampleRate
- IHXTAudioPinFormat::SetChannelFormat
- IHXTAudioPinFormat::SetSampleFormat
- IHXTAudioPinFormat::SetSampleRate

As with all Component Object Model (COM) interfaces, the IHXTAudioPinFormat interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTAudioPinFormat::GetChannelFormat

Gets the channel format (speaker layout) in audio samples passed to the encoding manager. See the EHXTAudioChannelFormat enumerator in ihxtaudioformat.h.

```
STDMETHOD(GetChannelFormat) (
   THIS_
   EHXTAudioChannelFormat* peChannelFormat
) PURE;
```

peChannelFormat
Pointer to an EHXTAudioChannelFormat enumerator that specifies the channel format in the audio sample.

IHXTAudioPinFormat::GetSampleFormat

Gets the sample format (bits per sample, bit packing) of audio samples passed to the encoding manager. See the EHXTAudioSampleFormat enumerator in ihxtaudioformat.h.

```
STDMETHOD(GetSampleFormat) (
   THIS_
   EHXTAudioSampleFormat* peSampleFormat
) PURE;
```

peSampleFormat
: Pointer to an EHXTAudioSampleFormat enumerator that specifies the sample format of the audio sample.

IHXTAudioPinFormat::GetSampleRate

Gets the sample rate of audio samples passed to the encoding engine.

```
STDMETHOD(GetSampleRate) (
   THIS_
   UINT32* pulSamplesPerSec
) PURE;
```

pulSamplesPerSec
: Pointer to the sample rate, per second, of the audio sample.

IHXTAudioPinFormat::SetChannelFormat

Sets the channel format (speaker layout) in audio samples passed to the encoding manager. See the EHXTAudioChannelFormat enumerator in ihxtaudioformat.h.

```
STDMETHOD(SetChannelFormat) (
   THIS_
   EHXTAudioChannelFormat eChannelFormat
) PURE;
```

eChannelFormat
: An EHXTAudioChannelFormat enumerator that specifies the channel format in the audio sample.

IHXTAudioPinFormat::SetSampleFormat

Sets the sample format (bits per sample, bit packing) of audio samples passed to the encoding manager. See the EHXTAudioSampleFormat enumerator in ihxtaudioformat.h.

```
STDMETHOD(SetSampleFormat) (
   THIS_
   EHXTAudioSampleFormat eSampleFormat
) PURE;
```

eSampleFormat
An `EHXTAudioSampleFormat` enumerator that specifies the sample format of the audio sample.

IHXTAudioPinFormat::SetSampleRate

Sets the sample rate of audio samples passed to the encoding engine.

```
STDMETHOD(SetSampleRate) (
   THIS_
   UINT32 ulSamplesPerSec
) PURE;
```

ulSamplesPerSec
The sample rate, per second.

IHXTCaptureDialogControl

Purpose:	Launches capture dialog windows.
Implemented by:	GUI video capture setting's dialog
Header file:	`ihxtbase.h`

The capture dialog windows launched by this interface are provided by the operating system or capture driver vendors.

The `IHXTCaptureDialogControl` interface contains the `IHXTCaptureDialogControl::LaunchDialog` method.

As with all Component Object Model (COM) interfaces, the `IHXTCaptureDialogControl` interface inherits the following `IUnknown` methods:

- `IUnknown::AddRef`
- `IUnknown::QueryInterface`
- `IUnknown::Release`

IHXTCaptureDialogControl::LaunchDialog

Displays a dialog for changing the capture parameters. Each capture dialog has its own unique identifier. The window pointer is operating system-specific. For Windows DirectShow, it is the `HWND` window handle of the parent window. This method returns an error for certain dialogs if a capture is running. In this case, stop the capture, launch the dialog again, and restart the capture.

When mapping identifiers to dialog name, the input source can be queried for the kPropAudioCaptureDialogs and kPropVideoCaptureDialogs property bags. These property bags each contain a kPropCaptureDialogNames property bag that holds dialog name and identifier value pairs.

This method returns an error for certain dialogs if a capture is running. In this case, stop the capture, launch the dialog again, and restart the capture.

```
STDMETHOD_(HX_RESULT, LaunchDialog) (
   THIS_
   INT32 nIdentifier,
   void* window
) PURE;
```

nIdentifier
: The unique identifier for this specific capture dialog.

window
: Pointer to the operating system-specific window. For example, for Windows DirectShow this parameter would point to the HWND window handle of the parent window.

IHXTClassFactory

Purpose:	Creates RTA objects.
Implemented by:	Any component
Header file:	ihxtencodingjob.h

Any Helix DNA Producer SDK component can use this interface to create a Helix DNA Producer SDK object.This is the preferred method for creating objects used by multiple components. A component can use the C++ new operator to create objects that it alone manipulates, however. When Helix DNA Producer initializes a component, it passes the component a pointer to the system context.The component can then use this pointer to call any of the IHXClassFactory methods.

The IHXTClassFactory interface contains the following methods:

- IHXTClassFactory::BuildInstance
- IHXTClassFactory::BuildInstanceFromBuffer
- IHXTClassFactory::BuildInstanceFromFile
- IHXTClassFactory::BuildInstanceFromObject
- IHXTClassFactory::CreateInstance

As with all Component Object Model (COM) interfaces, the IHXTClassFactory interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTClassFactory::BuildInstance

Creates an instance of the specified interface, specifies the properties used by that interface, and initializes the interface.

```
STDMETHOD(BuildInstance) (
   THIS_
   REFIID riid,
   IHXTPropertyBag* pInitParams,
   IUnknown** ppNewInstance,
   BOOL bForceInitialization=FALSE,
   IHXTPropertyBag** ppInitErrorBag=NULL
) PURE;
```

riid
: Indicates the reference identifier for the interface being built.

pInitParams
: Pointer to an IHXTPropertyBag interface that manages the collection of properties used by this interface.

ppNewInstance
: Address of a pointer to an IUnknown interface that identifies the new interface.

bForceInitialization
: If set to false (default), this method will not create the object if any of the initialization properties fail to be validated. This ensures you never have an object in a bad state. However, this can have undesirable side effects, like causing a job file to fail to deserialize if the input filename does not exist, or if the specified capture device is in use. If this parameter is set to true, the object is forced to be created even with an invalid property. This allows the object to be created even though it cannot be part of an encode—the calling SDK application can then inspect the object and determine the problem that occurred during initialization.

ppInitErrorBag
: Address of a pointer to an IHXTPropertyBag interface that manages the invalid properties that caused initialization to fail.

IHXTClassFactory::BuildInstanceFromBuffer

Creates an instance of the specified interface and initializes the interface from a buffer containing XML code.

```
STDMETHOD(BuildInstanceFromBuffer) (
   THIS_
   REFIID riid,
   IHXBuffer* pXmlBuffer,
   IUnknown** ppNewInstance,
   BOOL bForceInitialization=FALSE,
   IHXTPropertyBag** ppInitErrorBag=NULL
) PURE;
```

riid
: Indicates the reference identifier for the interface being built.

pXmlBuffer
: Pointer to an IHXBuffer interface that manages the XML code.

ppNewInstance
: Address of a pointer to an IUnknown interface that identifies the new interface.

bForceInitialization
: If set to false (default), this method will not create the object if any of the initialization properties fail to be validated. This ensures you never have an object in a bad state. However, this can have undesirable side effects, like causing a job file to fail to deserialize if the input filename does not exist, or if the specified capture device is in use. If this parameter is set to true, the object is forced to be created even with an invalid property. This allows the object to be created even though it cannot be part of an encode—the calling SDK application can then inspect the object and determine the problem that occurred during initialization.

ppInitErrorBag
: Address of a pointer to an IHXTPropertyBag interface that manages the invalid properties that caused initialization to fail.

IHXTClassFactory::BuildInstanceFromFile

Creates an instance of the specified interface, along with any data required by the interface. The data is supplied in an XML file specified in this method.

```
STDMETHOD(BuildInstanceFromFile) (
   THIS_
   REFIID riid,
   const char* szPathname,
   IUnknown** ppNewInstance,
   BOOL bForceInitialization=FALSE,
   IHXTPropertyBag** ppInitErrorBag=NULL
) PURE;
```

riid
: Indicates the reference identifier for the interface being built.

szPathname
: Pointer to the pathname of the XML file used to build this instance.

ppNewInstance
: Address of a pointer to an `IUnknown` interface that identifies the new interface.

bForceInitialization
: If set to `false` (default), this method will not create the object if any of the initialization properties fail to be validated. This ensures you never have an object in a bad state. However, this can have undesirable side effects, like causing a job file to fail to deserialize if the input filename does not exist, or if the specified capture device is in use. If this parameter is set to `true`, the object is forced to be created even with an invalid property. This allows the object to be created even though it cannot be part of an encode—the calling SDK application can then inspect the object and determine the problem that occurred during initialization.

ppInitErrorBag
: Address of a pointer to an `IHXTPropertyBag` interface that manages the invalid properties that caused initialization to fail.

IHXTClassFactory::BuildInstanceFromObject

Clones an existing interface and optionally replaces some parameters in the new interface.

```
STDMETHOD(BuildInstanceFromObject) (
   THIS_
   REFIID riid,
   IUnknown* pUnkExistingObj,
   IUnknown** ppUnkNewInstance,
   IHXTPropertyBag** ppReplaceProps=NULL,
   BOOL bForceInitialization=FALSE,
   IHXTPropertyBag** ppInitErrorBag=NULL
) PURE;
```

riid
: Indicates the reference identifier for the interface being built.

pUnkExistingObj
: Pointer to an `IUnknown` interface that identifies the existing interface from which the new interface can be built.

ppUnkNewInstance
: Address of a pointer to an `IUnknown` interface that identifies the new interface.

ppReplaceProps
: Address of a pointer to an `IHXTPropertyBag` interface that manages the replacement parameters for the new interface.

bForceInitialization
: If set to `false` (default), this method will not create the object if any of the initialization properties fail to be validated. This ensures you never have an object in a bad state. However, this can have undesirable side effects, like causing a job file to fail to deserialize if the input filename does not exist, or if the specified capture device is in use. If this parameter is set to `true`, the object is forced to be created even with an invalid property. This allows the object to be created even though it cannot be part of an encode—the calling SDK application can then inspect the object and determine the problem that occurred during initialization.

ppInitErrorBag
: Address of a pointer to an `IHXTPropertyBag` interface that manages the invalid properties that caused initialization to fail.

IHXTClassFactory::CreateInstance

Creates an instance of a Helix DNA Producer SDK interface. This method does not perform any kind of initialization.

```
STDMETHOD(CreateInstance) (
   THIS_
   REFIID riid,
   IUnknown** ppNewInstance
) PURE;
```

riid
: Indicates the reference identifier for the interface being created.

ppUnk
: Address of a pointer to an IUnknown interface that identifies the new interface.

IHXTCodecUpdater

Purpose:	Automatically updates codecs.
Implemented by:	Encoding
Header file:	ihxtencodingjob.h

This interface specifies the location of a mapping file that is the source for upgrading codecs. The user can have all audiences for an encoding job updated or can select individual audiences to update.

> **For More Information:** See "Automatic Codec Selection" on page 82.

The IHXTCodecUpdater interface contains the following methods:

- IHXTCodecUpdater::GetCodecMappingFile
- IHXTCodecUpdater::SetCodecMappingFile
- IHXTCodecUpdater::UpdateAudience
- IHXTCodecUpdater::UpdateJob

As with all Component Object Model (COM) interfaces, the IHXTCodecUpdater interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTCodecUpdater::GetCodecMappingFile

Gets the location of the codec mapping file.

```
STDMETHOD(GetCodecMappingFile) (
   THIS_
   const char** ppszMappingFile
) PURE;
```

ppszMappingFile
: Address of a pointer to the location of the codec mapping file.

IHXTCodecUpdater::SetCodecMappingFile

Sets the location of the codec mapping file. The file will be parsed and errors reported as applicable.

```
STDMETHOD(SetCodecMappingFile) (
   THIS_
   const char* pszMappingFile
) PURE;
```

pszMappingFile
: Pointer to the location of the codec mapping file.

IHXTCodecUpdater::UpdateAudience

Updates a single audience based on the mappings provided in the codec mapping file. If no mappings are specified, an unexpected error is returned. If any stream update fails in the audience, the entire method fails. The IN audience is unaltered and the new updated job is returned as an OUT parameter.

```
STDMETHOD(UpdateAudience) (
   THIS_
   IHXTAudience* pAudience,
   IHXTAudience** ppNewAudience,
   BOOL* pbAudienceUpdated = NULL
) PURE;
```

pAudience
: Pointer to an `IHXTAudience` interface that manages the original audience.

ppNewAudience
: Address of a pointer to an `IHXTAudience` interface that manages the new audience that contains the updated codec.

pbAudienceUpdated
: Pointer to a Boolean expression that indicates whether the audience has been updated. If `TRUE`, the new audience has been successfully modified

from the original audience. If FALSE, the new audience has not been modified. This parameter can be NULL, in which case no value is returned.

IHXTCodecUpdater::UpdateJob

Updates all audiences contained in an encoding job based on the mappings provided in the codec mapping file. If no mappings are specified, an unexpected error is returned. If any stream update fails in the audience, the entire method fails.

```
STDMETHOD(UpdateJob) (
   THIS_
   IHXTEncodingJob* pEncodingJob,
   BOOL* pbAudienceUpdated = NULL
) PURE;
```

pEncodingJob
: Pointer to an IHXTEncodingJob interface that manages the audiences to be updated.

pbAudienceUpdated
: Pointer to a Boolean expression that indicates whether any of the audiences have been updated. If TRUE, one or more of the new audiences have been successfully modified from the original audiences. If FALSE, no audience has been modified. This parameter can be NULL, in which case no value is returned.

IHXTConfigurationAgent

Purpose:	Provides generic configuration that uses property bags.
Implemented by:	Encoding and plug-ins
Header file:	ihxtbase.h

This interface inherits the methods from the IHXTPropertyBag interface.

The IHXTConfigurationAgent method contains the IHXTConfigurationAgent::Initialize method.

As with all Component Object Model (COM) interfaces, the IHXTConfigurationAgent interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTConfigurationAgent::Initialize

Initializes an object with the specified properties. This method returns a success code if the initialization is successful.

```
STDMETHOD(Initialize) (
   THIS_
   IHXTPropertyBag* pPropBag,
   IHXTPropertyBag** ppErrorBag=NULL
) PURE;
```

pPropBag
Pointer to an IHXTPropertyBag interface that manages the properties for the object being initialized.

ppErrorBag
If this parameter is set to a value other than NULL before initialization, contains an address of a pointer to an IHXTPropertyBag interface that identifies a failed property if the initialization process fails.

IHXTConnectionAgent

Purpose:	Advertises the various formats with which a filter can connect.
Implemented by:	Plug-ins
Header file:	ihxtbase.h

This interface must be implemented by all filters.

The IHXTConnectionAgent method contains the following methods:

- IHXTConnectionAgent::GetInputStreamCount
- IHXTConnectionAgent::GetNegotiatedInputFormat
- IHXTConnectionAgent::GetNegotiatedOutputFormat
- IHXTConnectionAgent::GetOutputStreamCount
- IHXTConnectionAgent::GetPreferredInputFormat
- IHXTConnectionAgent::GetPreferredOutputFormat
- IHXTConnectionAgent::GetSupportedInputFormat
- IHXTConnectionAgent::GetSupportedOutputFormat
- IHXTConnectionAgent::SetNegotiatedInputFormat
- IHXTConnectionAgent::SetNegotiatedOutputFormat

As with all Component Object Model (COM) interfaces, the IHXTConnectionAgent interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTConnectionAgent::GetInputStreamCount

Returns the total number of possible input streams that can be connected.

```
STDMETHOD_(UINT32, GetInputStreamCount) (
   THIS
) PURE;
```

IHXTConnectionAgent::GetNegotiatedInputFormat

Retrieves the negotiated format previously set by IHXTConnectionAgent::SetNegotiatedInputFormat. Filters can implement this method by saving the property bag specified in IHXTConnectionAgent::SetNegotiatedInputFormat.

```
STDMETHOD(GetNegotiatedInputFormat) (
   THIS_
   UINT32 uIndex,
   IHXTPropertyBag** pSupportedFormats
) PURE;
```

uIndex
The index number of the stream.

pSupportedFormats
Address of a pointer to an IHXTPropertyBag interface that manages the negotiated input formats.

IHXTConnectionAgent::GetNegotiatedOutputFormat

Retrieves the negotiated format set by IHXTConnectionAgent::SetNegotiatedOutputFormat. Filters can implement this method by saving the property bag specified in IHXTConnectionAgent::SetNegotiatedOutputFormat.

```
STDMETHOD(GetNegotiatedOutputFormat) (
   THIS_
   UINT32 uIndex,
   IHXTPropertyBag** pSupportedFormats
) PURE;
```

uIndex
: The index number of the stream.

pSupportedFormats
: Address of a pointer to an `IHXTPropertyBag` interface that manages the negotiated output formats.

IHXTConnectionAgent::GetOutputStreamCount

Returns the total number of possible output streams that can be connected.

```
STDMETHOD_( UINT32, GetOutputStreamCount) (
   THIS
) PURE;
```

IHXTConnectionAgent::GetPreferredInputFormat

This method is reserved for future use

```
STDMETHOD(GetPreferredInputFormat) (
   THIS_
   UINT32 uIndex,
   UINT32 uPrefRank,
   IHXTPropertyBag** ppPreferredFormat
) PURE;
```

IHXTConnectionAgent::GetPreferredOutputFormat

Reserved for future use.

```
STDMETHOD(GetPreferredOutputFormat) (
   THIS_
   UINT32 uIndex,
   UINT32 uPrefRank,
   IHXTPropertyBag** ppPreferredFormat
) PURE;
```

IHXTConnectionAgent::GetSupportedInputFormat

Gets the format for a particular input stream. A filter can specify support for multiple values for a single property (such as multiple audio sample rates) by using list and range properties.

```
STDMETHOD(GetSupportedInputFormat) (
   THIS_
   UINT32 uIndex,
   IHXTPropertyBag** pSupportedFormats
) PURE;
```

uIndex
The index number of the stream.

pSupportedFormats
Address of a pointer to an `IHXTPropertyBag` interface that manages the supported input formats.

IHXTConnectionAgent::GetSupportedOutputFormat

Gets the format for a particular output stream. A filter can specify support for multiple values for a single property (such as multiple audio sample rates) by using list and range properties.

```
STDMETHOD(GetSupportedOutputFormat) (
   THIS_
   UINT32 uIndex,
   IHXTPropertyBag** pSupportedFormats
) PURE;
```

uIndex
The index number of the stream.

pSupportedFormats
Address of a pointer to an `IHXTPropertyBag` interface that manages the supported output formats.

IHXTConnectionAgent::SetNegotiatedInputFormat

Notifies the filter that an input stream has been connected.

```
STDMETHOD(SetNegotiatedInputFormat) (
   THIS_
   UINT32 uStreamID,
   IHXTPropertyBag* pInputFormat
) PURE;
```

uStreamID
The index number of the stream that has been connected.

pInputFormat
: Pointer to an IHXTPropertyBag interface that manages the format that was negotiated for the connection.

IHXTConnectionAgent::SetNegotiatedOutputFormat

Notifies the filter that an output stream has been connected.

```
STDMETHOD(SetNegotiatedOutputFormat) (
   THIS_
   UINT32 uStreamID,
   IHXTPropertyBag* pOutputFormat
) PURE;
```

uStreamID
: The index number of the stream that has been connected.

pOutputFormat
: Pointer to an IHXTPropertyBag interface that manages the format that was negotiated for the connection.

IHXTCustomComparison

Purpose:	Provides a means of comparing custom properties.
Implemented by:	Custom properties
Header file:	ihxtpropertybag.h

Typical connection properties are handled by the Helix DNA Producer SDK; therefore, you don't have to write any code that does any sort of property comparison. For example, if you implement an audio prefilter, you might advertise that you can handle one or two channels of incoming audio by populating an IHXTUintList with the values {1,2}. But you don't have to write any code that checks if the connecting filter has either one or two channels of audio—the SDK looks at both filters' property bags and determines if they are compatible.

However, if you are creating custom connection properties not specifically supported by the Helix DNA Producer SDK, you must use this interface to implement the code that compares and negotiates the final connected value of a property.

The IHXTCustomComparison method contains the IHXTCustomComparison::Compare method.

As with all Component Object Model (COM) interfaces, the IHXTCustomComparison interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTCustomComparison::Compare

Performs a comparison against a custom connecting property and negotiates a mutually acceptable property, if possible

```
STDMETHOD(Compare) (
   THIS_
   IUnknown *pConnectingProperty,
   IUnknown **ppNegotiatedProperty
) PURE;
```

pConnectingProperty
: Pointer to an IUnknown interface that identifies the custom connecting property.

ppNegotiatedProperty
: Address of a pointer to an IUnknown interface that identifies the property that contains the final negotiated value.

IHXTDestination

Purpose:	Represents the output from the encoding job (either file or server) and provides failover methods.
Implemented by:	Encoding
Header file:	ihxtencodingjob.h

This interface inherits the configuration methods of IHXTConfigurationAgent, which are used to configure the destination properties. In addition, it adds, deletes, and manipulates an index list of postfilters for the encoding engine.

> **Note:** For more information on the destination properties that can be configured by this interface, see "Destination" on page 48

The IHXTDestination interface contains the following methods:

- IHXTDestination::AddFailoverDestination

- IHXTDestination::AddPostfilter
- IHXTDestination::GetFailoverDestination
- IHXTDestination::GetFailoverDestinationCount
- IHXTDestination::GetPostfilter
- IHXTDestination::GetPostfilterCount
- IHXTDestination::MoveFailoverDestination
- IHXTDestination::MovePostfilter
- IHXTDestination::RemoveFailoverDestination
- IHXTDestination::RemovePostfilter

As with all Component Object Model (COM) interfaces, the IHXTDestination interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTDestination::AddFailoverDestination

Not currently implemented.

```
STDMETHOD(AddFailoverDestination) (
   THIS_
   IHXTDestination* pDestination
) PURE;
```

IHXTDestination::AddPostfilter

Adds a postfilter to the back of the postfilter list.

```
STDMETHOD(AddPostfilter) (
   THIS_
   IHXTPostfilter* pPostfilter
) PURE;
```

pPostfilter
Pointer to an IHXTPostfilter interface that manages the postfilter being added to the postfilter list.

IHXTDestination::GetFailoverDestination

Not currently implemented.

```
STDMETHOD(GetFailoverDestination) (
   THIS_
   UINT32 ulIndex,
   IHXTDestination** ppDestination
) PURE;
```

IHXTDestination::GetFailoverDestinationCount

Not currently implemented.

```
STDMETHOD_(UINT32, GetFailoverDestinationCount) (
  THIS
) PURE;
```

IHXTDestination::GetPostfilter

Retrieves the specified postfilter in the index list.

```
STDMETHOD(GetPostfilter) (
   THIS_
   UINT32 ulIndex,
   IHXTPostfilter** ppPostfilter
) PURE;
```

ulIndex
The position of the postfilter in the index list.

ppPostfilter
Address of a pointer to an `IHXTPostfilter` interface that manages the postfilter information.

IHXTDestination::GetPostfilterCount

Returns the number of postfilters in the list.

```
STDMETHOD_(UINT32, GetPostfilterCount) (
   THIS
) PURE;
```

IHXTDestination::MoveFailoverDestination

Not currently implemented.

```
STDMETHOD(MoveFailoverDestination) (
   UINT32 ulOrigIndex,
   UINT32 ulDestIndex
) PURE;
```

IHXTDestination::MovePostfilter

Moves a postfilter from its original location in the index list to another location. During encoding, audio/video samples propagate through postfilters based on their list ordering (from lowest index number to highest).

```
STDMETHOD(MovePostfilter) (
   UINT32 ulOrigIndex,
   UINT32 ulDestIndex
) PURE;
```

ulOrigIndex
: The original location of the postfilter in the index list.

ulDestIndex
: The destination in the index list to which the postfilter is to be moved.

IHXTDestination::RemoveFailoverDestination

Not currently implemented.

```
STDMETHOD(RemoveFailoverDestination) (
   THIS_
   UINT32 ulIndex
) PURE;
```

IHXTDestination::RemovePostfilter

Removes the specified postfilter in the index list.

```
STDMETHOD(RemovePostfilter) (
   THIS_
   UINT32 ulIndex
) PURE;
```

ulIndex
: The location in the index list of the postfilter to be removed.

IHXTDestinationEnumerator

Purpose:	Identifies specific broadcast destinations.
Implemented by:	Encoding
Header file:	`ihxtencodingjob.h`

The `IHXTDestinationEnumerator` interface contains the following methods:

- IHXTDestinationEnumerator::GetDestination
- IHXTDestinationEnumerator::GetDestinationCount
- IHXTDestinationEnumerator::SetProfileDirectory
- IHXTDestinationEnumerator::SetProfileExtension

As with all Component Object Model (COM) interfaces, the IHXTDestinationEnumerator interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTDestinationEnumerator::GetDestination

Retrieves the specified broadcast destination in the index list.

```
STDMETHOD(GetDestination) (
   THIS_
   UINT32 ulIndex,
   IHXTDestination** ppBroadcastDef,
   const char** pszFilename
) PURE;
```

ulIndex
The location of the destination in the index list.

ppBroadcastDef
Address of a pointer to an IHXTDestination interface that manages the destination information.

pszFilename
Address of a pointer to the broadcast destination file.

IHXTDestinationEnumerator::GetDestinationCount

Returns the number of destinations in the list.

```
STDMETHOD_(UINT32, GetDestinationCount) (
   THIS
) PURE;
```

IHXTDestinationEnumerator::SetProfileDirectory

Sets the directory for retrieving the broadcast definitions to deserialize.

```
STDMETHOD(SetProfileDirectory) (
   THIS_
   const char* szDirectoryPath
) PURE;
```

szDirectoryPath
: Pointer to the directory path of the broadcast definitions.

IHXTDestinationEnumerator::SetProfileExtension

Selects broadcast definitions by extension. At this time, this method will only select definitions from .rpsd extensions. Currently, this method is not implemented.

```
STDMETHOD(SetProfileExtension) (
   THIS_
   const char* szProfileExtension
) PURE;
```

szProfileExtension
: Pointer to the filename extension that will contain the broadcast definitions.

IHXTDoubleEnumerator

Purpose:	Enumerates the values of a double list.
Implemented by:	All components
Header file:	ihxtpropertybag.h

> **Note:** This interface does not need to be implemented by itself, but you can access this interface from a property bag.

The IHXTDoubleEnumerator interface contains the following methods:

- IHXTDoubleEnumerator::Current
- IHXTDoubleEnumerator::First
- IHXTDoubleEnumerator::GetCount
- IHXTDoubleEnumerator::Next

As with all Component Object Model (COM) interfaces, the IHXTDoubleEnumerator interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface

- IUnknown::Release

IHXTDoubleEnumerator::Current

Gets the double value at the current position of the internal iterator.

```
STDMETHOD(Current) (
   THIS_
   double *pValue
) PURE;
```

pValue
Pointer to the value of the double.

IHXTDoubleEnumerator::First

Gets the double value at the first position in the list of doubles.

```
STDMETHOD(First) (
   THIS_
   double *pValue
) PURE
```

pValue
Pointer to the value of the double.

IHXTDoubleEnumerator::GetCount

Returns the total number of properties in the list of doubles.

```
STDMETHOD_(UINT32, GetCount) (
   THIS
) const PURE;
```

IHXTDoubleEnumerator::Next

Advances the internal iterator and gets the double value at the next position in the list of doubles.

```
STDMETHOD(Next) (
   THIS_
   double *pValue
) PURE;
```

pValue
Pointer to the value of the double.

IHXTDoubleList

Purpose:	Stores a list of doubles.
Implemented by:	All components
Header file:	ihxtpropertybag.h

> **Note:** This interface does not need to be implemented by itself, but you can obtain an object that implements this interface from the class factory.

The IHXTDoubleList interface contains the following methods:

- IHXTDoubleList::Clear
- IHXTDoubleList::Compare
- IHXTDoubleList::Contains
- IHXTDoubleList::GetBack
- IHXTDoubleList::GetEnumerator
- IHXTDoubleList::GetFront
- IHXTDoubleList::GetIntersection
- IHXTDoubleList::GetSize
- IHXTDoubleList::IsEmpty
- IHXTDoubleList::PopBack
- IHXTDoubleList::PopFront
- IHXTDoubleList::PushBack
- IHXTDoubleList::PushFront

As with all Component Object Model (COM) interfaces, the IHXTDoubleList interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTDoubleList::Clear

Clears all properties from the current list.

```
STDMETHOD(Clear) (
   THIS
) PURE;
```

IHXTDoubleList::Compare

This method is obsolete and should not be used.

```
STDMETHOD(Compare) (
   THIS_
   IHXTDoubleList *pList,
   double *puValue
) const PURE;
```

IHXTDoubleList::Contains

Returns `TRUE` if the specified value is contained in the current list. Returns `FALSE` if the value is not in the current list.

```
STDMETHOD_(BOOL, Contains (
   THIS_
   double value
) const PURE;
```

value
The double value being searched for in the current list.

IHXTDoubleList::GetBack

Returns the value at the end of the list.

```
STDMETHOD_(double, GetBack) (
   THIS
) PURE;
```

IHXTDoubleList::GetEnumerator

Gets an enumerator that can be used to enumerate through all the items in the list.

```
STDMETHOD(GetEnumerator) (
   THIS_
   IHXTDoubleEnumerator **pEnumerator
) PURE;
```

pEnumerator
Address of a pointer to an `IHXTDoubleEnumerator` interface that manages the enumerator.

IHXTDoubleList::GetFront

Returns the value at the beginning of the list.

```
STDMETHOD_(double, GetFront) (
   THIS
) PURE;
```

IHXTDoubleList::GetIntersection

Gets the intersection between the current list and another specified list.

```
STDMETHOD(GetIntersection) (
   THIS_
   IHXTDoubleList *pList,
   IHXTDoubleList **ppIntersection
) const PURE;
```

pList
: Pointer to an IHXTDoubleList interface that manages the double list to compare against the current double list.

ppIntersection
: Address of a pointer to an IHXTDoubleList interface that manages the new double list. This new list contains only those properties that were the same in both of the compared lists.

IHXTDoubleList::GetSize

Returns the size of the current list.

```
STDMETHOD_(UINT32, GetSize) (
   THIS
) PURE;
```

IHXTDoubleList::IsEmpty

If TRUE, indicates the current list is empty. If FALSE, indicates there is at least one property in the current list.

```
STDMETHOD_(BOOL, IsEmpty) (
   THIS
) PURE;
```

IHXTDoubleList::PopBack

Returns the property value at the end of the list, and removes the value from the list.

```
STDMETHOD_(double, PopBack) (
   THIS
) PURE;
```

IHXTDoubleList::PopFront

Returns the property value at the beginning of the list, and removes the value from the list.

```
STDMETHOD_(double, PopFront) (
   THIS
) PURE;
```

IHXTDoubleList::PushBack

Places a property value at the end of the list.

```
STDMETHOD(PushBack) (
    THIS_
   double value
) PURE;
```

value
The value of the property to add to the end of the list.

IHXTDoubleList::PushFront

Places a property value at the beginning of the list.

```
STDMETHOD(PushFront) (
   THIS_
   double value
) PURE;
```

value
The value of the property to add to the beginning of the list.

IHXTDoubleRange

Purpose:	Represents a range of double values.
Implemented by:	All components
Header file:	ihxtpropertybag.h

This interface provides access to a range of values. For example, you might have a resampler that supports resampling to all sampling rates between 0 and 96kHz. This interface can be used to represent all of these values using a range (0, 96000) instead of a list with 96,000 discrete elements.

> **Note:** This interface does not need to be implemented by itself, but you can obtain an object that implements this interface from the class factory.

The IHXTDoubleRange interface contains the following methods:

- IHXTDoubleRange::Compare
- IHXTDoubleRange::GetError
- IHXTDoubleRange::GetMax
- IHXTDoubleRange::GetMin
- IHXTDoubleRange::GetStepSize
- IHXTDoubleRange::IsInRange
- IHXTDoubleRange::Set

As with all Component Object Model (COM) interfaces, the IHXTDoubleRange interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTDoubleRange::Compare

This method is obsolete and should not be used.

```
STDMETHOD(Compare) (
   THIS_
   IHXTDoubleRange *pRange,
   IHXTDoubleRange **ppResult
) const PURE;
```

IHXTDoubleRange::GetError

Returns an error value.

```
STDMETHOD_(double, GetError) (
   THIS
) const PURE;
```

IHXTDoubleRange::GetMax

Returns the maximum value in the current range.

```
STDMETHOD_(double, GetMax) (
   THIS
) const PURE;
```

IHXTDoubleRange::GetMin

Returns the minimum value in the current range.

```
STDMETHOD_(double, GetMin) (
   THIS
) const PURE;
```

IHXTDoubleRange::GetStepSize

Returns the step size of the range.

```
STDMETHOD_(double, GetStepSize) (
   THIS
) const PURE;
```

IHXTDoubleRange::IsInRange

Returns `TRUE` if the specified value is in range, or returns `FALSE` if the value is out of range.

```
STDMETHOD_(BOOL, IsInRange) (
   THIS_
   double dValue
) const PURE;
```

dValue
The value being tested to see if it is in range.

IHXTDoubleRange::Set

Sets the range parameters.

```
STDMETHOD(Set) (
   double dMin,
   double dMax,
   double dStepSize,
   double dError
) PURE;
```

dMin
: The minimum value to be set in the range.

dMax
: The maximum value to be set in the range.

dStepSize
: The step size of the values in the range.

dError
: This parameter is reserved and should not be used.

IHXTEncodingJob

Purpose:	Configures and manages a single encoding job.
Implemented by:	Encoding
Header file:	`ihxtencodingjob.h`

This interface exposes the main encoding methods used by the encoding manager. This interface inherits the configuration methods of `IHXTConfigurationAgent`, which are used to configure the encoding job properties. In addition, it starts, stops, or cancels encoding jobs.

> **Note:** For more information on the encoding job properties that can be set by this interface, see "Setting Up an Encoding Job" on page 33.

The `IHXTEncodingJob` interface contains the following methods:

- `IHXTEncodingJob::AddOutputProfile`
- `IHXTEncodingJob::CancelEncoding`
- `IHXTEncodingJob::GetEventManager`
- `IHXTEncodingJob::GetInput`
- `IHXTEncodingJob::GetMetadata`

- IHXTEncodingJob::GetOutputProfile
- IHXTEncodingJob::GetOutputProfileCount
- IHXTEncodingJob::MoveOutputProfile
- IHXTEncodingJob::RemoveOutputProfile
- IHXTEncodingJob::SetInput
- IHXTEncodingJob::SetMetadata
- IHXTEncodingJob::StartEncoding
- IHXTEncodingJob::StopEncoding

As with all Component Object Model (COM) interfaces, the IHXTEncodingJob interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTEncodingJob::AddOutputProfile

Adds an output profile to the back of the output profile list.

```
STDMETHOD(AddOutputProfile) (
   THIS_
   IHXTOutputProfile* pOutputProfile
) PURE;
```

pOutputProfile
: Pointer to an IHXTOutputProfile interface that manages the output profile.

IHXTEncodingJob::CancelEncoding

Cancels the currently running encoding job. Output files will not be created. Blocks until encoding has been canceled.

```
STDMETHOD(CancelEncoding) (
   THIS
) PURE;
```

IHXTEncodingJob::GetEventManager

Gets the event subsystem manager. Used to subscribe and unsubscribe for events.

```
STDMETHOD(GetEventManager) (
   THIS_
   IHXTEventManager **ppEventManager
) PURE;
```

ppEventManager
: Address of a pointer to an IHXTEventManager interface that manages the event subsystem manager.

IHXTEncodingJob::GetInput

Gets the current input source, such as a file or capture device input.

```
STDMETHOD(GetInput) (
   THIS_
   IHXTInput** ppInput
) PURE;
```

ppInput
: Address of a pointer to an IHXTInput interface that manages the input source.

IHXTEncodingJob::GetMetadata

Gets the current metadata.

```
STDMETHOD(GetMetadata) (
   THIS_
   IHXTPropertyBag** ppMetadata
) PURE;
```

ppMetadata
: Address of a pointer to an IHXTPropertyBag interface that manages the metadata.

> **Note:** Although metadata is supplied in a property bag, only strings and UINTs are supported as metadata.

IHXTEncodingJob::GetOutputProfile

Retrieves the specified output profile in the index list.

```
STDMETHOD(GetOutputProfile) (
   THIS_
   UINT32 ulIndex,
   IHXTOutputProfile** ppOutputProfile
) PURE;
```

ulIndex
: The location of the output profile in the index list.

ppOutputProfile
: Address of a pointer to an `IHXTOutputProfile` interface that manages output profile information.

IHXTEncodingJob::GetOutputProfileCount

Returns the number of output profiles in the list.

```
STDMETHOD_(UINT32, GetOutputProfileCount) (
   THIS
) PURE;
```

IHXTEncodingJob::MoveOutputProfile

Moves an output profile from its original location in the index list to a new location.

```
STDMETHOD(MoveOutputProfile) (
   UINT32 ulOrigIndex,
   UINT32 ulDestIndex
) PURE;
```

ulOrigIndex
: The original location of the output profile in the index list.

ulDestIndex
: The destination to which the output profile is moved in the index list.

IHXTEncodingJob::RemoveOutputProfile

Removes the output profile from the specified location in the index list.

```
STDMETHOD(RemoveOutputProfile) (
   THIS_
   UINT32 ulIndex
) PURE;
```

ulIndex
: The location in the index list from which the output profile is to be removed.

IHXTEncodingJob::SetInput

Sets the input source.

```
STDMETHOD(SetInput) (
   THIS_
   IHXTInput* pInput
) PURE;
```

pInput
: Pointer to an IHXTInput interface that manages the input source.

IHXTEncodingJob::SetMetadata

Sets the metadata for the encoding job.

```
STDMETHOD(SetMetadata) (
   THIS_
   IHXTPropertyBag* pMetadata
) PURE;
```

pMetadata
: Pointer to an IHXTPropertyBag interface that manages the metadata.

> **Note:** Although metadata is supplied in a property bag, only strings and UINTs are supported as metadata.

IHXTEncodingJob::StartEncoding

Starts the encoding job.

```
STDMETHOD(StartEncoding) (
   THIS_
   BOOL bBlockUntilComplete=TRUE
) PURE;
```

bBlockUntilComplete
: If this parameter is TRUE, the call will block until encoding completes. A SUCCESS return code indicates that one or more destinations completed successfully. A FAIL return code indicates that no destinations completed successfully. If this parameter is FALSE, the call will return once encoding has started. The return code indicates whether encoding successfully started or not.

IHXTEncodingJob::StopEncoding

Stops the current encoding job. Output files will be written with whatever data has already been encoded. Blocks until encoding has stopped. Stopping

an encoding job during the first pass of a two-pass encode has the same effect as canceling the encode, since no data has been encoded.

```
STDMETHOD(StopEncoding) (
   THIS
) PURE;
```

IHXTEventManager

Purpose:	Manages the event system.
Implemented by:	Encoding
Header file:	ihxtbase.h

This interface provides subscription service to events from the encoding system. To use this interface, you must have previously implemented an event sink (IHXTEventSink interface) to handle the actual events. Then provide a pointer to the IHXTEventSink interface using the IHXTEventManager::Subscribe method.

The IHXTEventManager interface contains the following methods:

- IHXTEventManager::Subscribe
- IHXTEventManager::Unsubscribe

As with all Component Object Model (COM) interfaces, the IHXTEventManager interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTEventManager::Subscribe

Begins the process of receiving events. Before using this method, you must have implemented an event sink (IHXTEventSink). Note that subscription occurs asynchronously, and might not have actually occurred by the time the call to this method returns.

```
STDMETHOD(Subscribe) (
   THIS_
   IHXTEventSink *pEventSink
) PURE;
```

pEventSink
: Pointer to an IHXTEventSink interface that manages the events.

IHXTEventManager::Unsubscribe

Stops the process of receiving events. Note that unsubscribing occurs asynchronously, and might not have actually occurred by the time the call to this method returns.

```
STDMETHOD(Unsubscribe) (
   THIS_
   IHXTEventSink *pEventSink
) PURE;
```

pEventSink
: Pointer to an IHXTEventSink interface that manages the events.

IHXTEventSample

Purpose:	Stores fields for event streams.
Implemented by:	Encoding
Header file:	ihxtbase.h

This interface is created using the IHXTClassFactory interface. Once created and filled out, this interface is sent to an IHXTMediaInputPin interface of an input each time a media packet is set to be encoded.

This interface changes the display of events, such as the display of the Title, Author, and Copyright information, in RealPlayer as the presentation plays.

> **Note:** The "events" changed by this interface are not the same as the events managed by the IHXTEventManager event subsystem.

This interface inherits methods from the IHXTMediaSample interface.

The IHXTEventSample interface contains the following unique methods:

- IHXTEventSample::GetAction
- IHXTEventSample::SetAction

As with all Component Object Model (COM) interfaces, the IHXTEventSample interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTEventSample::GetAction

Gets the event action.

```
STDMETHOD(GetAction) (
   THIS_
   UINT16* pnType,
   const char** ppString,
   IHXValues** ppOtherParam
) PURE;
```

pnType
Pointer to the action type. One of the following:

- HXEventMediaSample_Author
  Indicates an author change will occur during playback.
- HXEventMediaSample_Copyright
  Indicates a copyright change will occur during playback.
- HXEventMediaSample_Custom
  Indicates a custom event will occur during playback.
- HXEventMediaSample_Title
  Indicates a title change will occur during playback.
- HXEventMediaSample_URL
  Indicates a web browser will be launched.

ppString
Address of a pointer to a string required by the action type.

ppOtherParam
Address of a pointer to an `IHXValues` interface that manages any other values required by the action type.

IHXTEventSample::SetAction

Sets the event action.

```
STDMETHOD(SetAction) (
   THIS_
   UINT16 nType,
   const char* pString,
   IHXValues* pOtherParam
) PURE;
```

nType
The action type. One of the following:

- HXEventMediaSample_Author

  Causes an author property in a presentation to dynamically change during playback.

- HXEventMediaSample_Copyright

  Causes a copyright property in a presentation to dynamically change during playback.

- HXEventMediaSample_Custom

  Attaches one or more named key/value pairs to an event. The key value is specified in the `pString` parameter of this method and the value for the key is specified by an `IHXValues` interface in the `pOtherParam` parameter. The value can be any type supported by `IHXValues`. For example:

  ```
  "markin", "select XXX from XXX"
  ```

  or

  ```
  "markin", 1000
  ```

- HXEventMediaSample_Title

  Causes a title property in a presentation to dynamically change during playback.

- HXEventMediaSample_URL

  Launches a web browser with a URL as specified in the `pString` parameter.

pString
Pointer to a string required by the action type.

pOtherParam
Pointer to an `IHXValues` interface that manages any other values required by the action type.

IHXTEventSink

Purpose:	Handles events passed up from the media engine, including progress and percent complete events, and asynchronous errors.
Implemented by:	Encoding
Header file:	ihxtbase.h

Objects that implement this interface can process events synchronously or asynchronously.

The IHXTEventSink interface contains the IHXTEventSink::HandleEvent method.

As with all Component Object Model (COM) interfaces, the IHXTEventSink interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTEventSink::HandleEvent

Sends an event to the event sink.

```
STDMETHOD(HandleEvent) (
   THIS_
   EHXTEvent eEvent,
   UINT32 *puValue = NULL,
   const char *cszValue = NULL,
   IUnknown *pUnknown = NULL
) PURE;
```

eEvent

The event to be sent. One of the following:

- eEventEncodingStarted

  Indicates that the encoding process has started.

- eEventEncodingFinished

  Indicates that the encoding process is finished.

- eEventTwoPassAnalysisStarted

  Indicates that a two-pass encoding process analysis has started.

- eEventTwoPassAnalysisFinished

  Indicates that a two-pass encoding process analysis is finished.

- eEventTwoPassEncodingUsingAnalysisStarted

  Indicates that a two-pass encoding process based on the previous analysis has started.

- eEventTwoPassEncodingUsingAnalysisFinished

  Indicates that a two-pass encoding process based on the previous analysis is finished.

- eEventEncodeProgress

  Indicates the progress of the encoding process. The `puValue` parameter points to the percentage of the process completed (from 0 to 100). The `pUnknown` parameter points to an `IUnknown` interface that identifies the `IHXTTime` interface associated with this event.

- eEventAnalysisProgress

  Indicates the progress of the encoding analysis in a two-pass encoding process. The `puValue` parameter points to the percentage of the process completed (from 0 to 100). The `pUnknown` parameter points to an `IUnknown` interface that identifies the `IHXTTime` interface associated with this event.

- eEventInputStarted

  Indicates the input to the encoding process has started. The `pUnknown` parameter points to an `IUnknown` interface that identifies the `IHXTInput` interface associated with this event.

- eEventInputFinished

  Indicates the input to the encoding process is finished. The `pUnknown` parameter points to an `IUnknown` interface that identifies the `IHXTInput` interface associated with this event.

- eEventInputError

  Indicates an error occurred during input to the encoding process. The `puValue` parameter points to the error code. The `pUnknown` parameter points to an `IUnknown` interface that identifies the `IHXTInput` interface associated with this event.

- eEventDestinationStarted

  Indicates that output to a destination from the encoding process has started. The `pUnknown` parameter points to an `IUnknown` interface that identifies the `IHXTDestination` interface associated with this event.

- eEventDestinationPostProcessProgress

  Indicates the progress percentage of the output post processing to a destination from the encoding process. The `puValue` parameter points to the percentage of the process completed (from 0 to 100). The `pUnknown` parameter points to an `IUnknown` interface that identifies the `IHXTDestination` interface associated with this event.

- eEventDestinationFinished

  Indicates that output to a destination from the encoding process is finished. The `pUnknown` parameter points to an `IUnknown` interface that identifies the `IHXTDestination` interface associated with this event.

- eEventDestinationError

  Indicates an error occurred during output to a destination from the encoding process. The `puValue` parameter points to the error code. The `pUnknown` parameter points to an `IUnknown` interface that identifies the `IHXTDestination` interface associated with this event.

- eEventDestinationCanceled

  Indicates the output to a destination was cancelled. The `pUnknown` parameter points to an `IUnknown` interface that identifies the `IHXTDestination` interface associated with this event.

- eEventInputFilterStarted

  Indicates that input prefiltering has started. The `pUnknown` parameter points to an `IUnknown` interface that identifies the `IHXTConnectionAgent` interface associated with this event.

- eEventInputFilterFinished

  Indicates that input prefiltering is finished. The `pUnknown` parameter points to an `IUnknown` interface that identifies the `IHXTConnectionAgent` interface associated with this event.

- eEventInputFilterError

  Indicates that an error occurred during input prefiltering. The `puValue` parameter points to the error code. The `pUnknown` parameter points to an `IUnknown` interface that identifies the `IHXTConnectionAgent` interface associated with this event.

- eEventOutputFilterStarted

Indicates that output postfiltering has started. The pUnknown parameter points to an IUnknown interface that identifies the IHXTConnectionAgent interface associated with this event.

- eEventOutputFilterPostProcessProgress

  Indicates the progress percentage of the output postfiltering process. The puValue parameter points to the percentage of the process completed (from 0 to 100). The pUnknown parameter points to an IUnknown interface that identifies the IHXTConnectionAgent interface associated with this event.

- eEventOutputFilterFinished

  Indicates that the output postfiltering is finished. The pUnknown parameter points to an IUnknown interface that identifies the IHXTConnectionAgent interface associated with this event.

- eEventOutputFilterError

  Indicates an error occurred during output postfiltering. The puValue parameter points to the error code. The pUnknown parameter points to an IUnknown interface that identifies the IHXTConnectionAgent interface associated with this event.

- eEventOutputFilterCanceled

  Indicates the output postfiltering was cancelled. The pUnknown parameter points to an IUnknown interface that identifies the IHXTConnectionAgent interface associated with this event.

- eEventOutputFilterStreamDone

  Indicates the output filter stream is done.

puValue
: Pointer to a value associated with the specific event. If this parameter is not used for a particular type of event, its value is NULL.

cszValue
: Pointer to a string associated with the specific event. If this parameter is not used for a particular type of event, its value is NULL.

pUnknown
: Pointer to an IUnknown interface that identifies an interface associated with the specific event. If this parameter is not used for a particular type of event, its value is NULL.

(Continued on next page.)

IHXTFileObserver

Purpose:	Manages log messages being sent to a file.
Implemented by:	Logging system
Header file:	ihxtfileobserver.h

This interface acts as a file observer to the log system. The file observer writes received log messages to the specified file.

The IHXTFileObserver interface contains the following methods:

- IHXTFileObserver::Enable
- IHXTFileObserver::EnableSDKMessages
- IHXTFileObserver::GetFilename
- IHXTFileObserver::GetPreviousFilename
- IHXTFileObserver::Init
- IHXTFileObserver::SetCategoryFilter
- IHXTFileObserver::SetFilename
- IHXTFileObserver::SetFormat
- IHXTFileObserver::SetFuncAreaFilter
- IHXTFileObserver::SetLanguage
- IHXTFileObserver::SetPreviousFilename
- IHXTFileObserver::SetSeperator
- IHXTFileObserver::SetSizeRoll
- IHXTFileObserver::SetTimeRoll
- IHXTFileObserver::Shutdown

As with all Component Object Model (COM) interfaces, the IHXTFileObserver interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTFileObserver::Enable

Enables and disables the observer from writing to the file.

```
STDMETHOD(Enable) (
   THIS_
   BOOL bEnable
) PURE;
```

bEnable
: If set to true, the observer can write to the file. If set to false, the observer cannot write to the file.

IHXTFileObserver::EnableSDKMessages

Instructs the observer to either accept or reject messages marked as SDK.

```
STDMETHOD_(void, EnableSDKMessages) (
   BOOL bEnable
) PURE;
```

bEnable
: If set to true, messages marked as SDK are accepted and are logged. If set to false, messages marked as SDK are rejected and are not logged.

IHXTFileObserver::GetFilename

Returns a pointer to the name of the file to which log messages are currently being written.

```
STDMETHOD_(const char*, GetFilename)(
) PURE;
```

IHXTFileObserver::GetPreviousFilename

Returns a pointer to the string being used as the previous file name.

```
STDMETHOD_(const char*, GetPreviousFilename)(
) PURE;
```

IHXTFileObserver::Init

Initializes the file observer.

```
STDMETHOD(Init) (
   THIS_
   const char* szFilename
) PURE;
```

szFilename
: Pointer to the name of the file to which log messages will be written.

IHXTFileObserver::SetCategoryFilter

Sets a bitmask filter that allows only those log messages with log codes that pass the bitmask parameter to be sent.

```
STDMETHOD(SetCategoryFilter) (
   THIS_
   UINT32 nCategoryFilter
) PURE;
```

nCategoryFilter
: The bitmask filter to compare against the log code.

IHXTFileObserver::SetFilename

Sets the filename to which the observer should write.

```
STDMETHOD(SetFilename) (
   THIS_
   const char* szFilename
) PURE;
```

szFilename
: Pointer to the name of the file to which the observer writes.

IHXTFileObserver::SetFormat

Sets the format of log messages that are written to a file.

```
STDMETHOD(SetFormat) (
   THIS_
   enumLogFormat format
) PURE;
```

format
: The format for the log messages. One of the following

  - Short

    Include only the job name of the job that sent the message and the message text.

  - Detailed.

    Include the job name, message text, the message category, the functional area, and the time the message was logged.

IHXTFileObserver::SetFuncAreaFilter

Sets the log system so it only sends log messages with functional areas listed in the parameter string.

```
STDMETHOD(SetFuncAreaFilter) (
   THIS_
   const char* szFuncAreaList
) PURE;
```

szFuncAreaList
Pointer to a list of accepted functional areas. The format of this list is a set of one or more name space and functional area pairs (with differing values for the name space, the functional area, or both), separated by a comma, for example:

"*Namespace*:*FunctionalArea*, *Namespace*:*FunctionalArea*, ..."

IHXTFileObserver::SetLanguage

This method is currently not implemented.

```
STDMETHOD(SetLanguage) (
   THIS_
   const char* szLanguage
) PURE;
```

IHXTFileObserver::SetPreviousFilename

Sets the previous filename. This filename is used in some rolling calculations.

```
STDMETHOD(SetPreviousFilename) (
   THIS_
   const char* szFilename
) PURE;
```

szFilename
Pointer to the previous filename.

IHXTFileObserver::SetSeperator

Sets the separator that separates log message elements in a single log message. For example, the separator could be a colon (:), a comma (,), or any other separator character.

```
STDMETHOD(SetSeparator) (
   THIS_
   char cSep
) PURE;
```

cSep
: The character used as a separator.

IHXTFileObserver::SetSizeRoll

Sets the roll size of the log file to which the observer writes. Once the specified file size is met, the observer rolls the file.

```
STDMETHOD(SetSizeRoll) (
   THIS_
   UINT32 nNumMB
) PURE;
```

nNumMB
: The number of megabytes the file size must reach before it is rolled.

IHXTFileObserver::SetTimeRoll

Sets a specific time interval for the log file before the observer to rolls the file.

```
STDMETHOD(SetTimeRoll) (
   THIS_
   enumRollType rolltype,
   UINT32 nTime,
   UINT32 nInterval,
   INT32 nTimeZone
) PURE;
```

rolltype
: The period over which the log is rolled. One of the following:

  - Monthly
  - Weekly
  - Daily
  - Hourly

nTime
: Not currently used.

nInterval
: Not currently used.

nTimeZone
Not currently used.

IHXTFileObserver::Shutdown

Shuts down (that is, unsubscribes and closes out the file) the observer.

```
STDMETHOD(Shutdown) (
   THIS
) PURE;
```

IHXTFilter

Purpose:	Provides basic filter operations.
Implemented by:	Plug-ins
Header file:	ihxtbase.h

The IHXTFilter interface contains the following methods:

- IHXTFilter::DiscardCachedSamples
- IHXTFilter::Prime
- IHXTFilter::SetFactory
- IHXTFilter::SetGraphServices
- IHXTFilter::Teardown

As with all Component Object Model (COM) interfaces, the IHXTFilter interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTFilter::DiscardCachedSamples

Signals the filter to release any cached samples it may have. This method can be called at any time between IHXTFilter::Prime and IHXTFilter::Teardown calls. If IHXTTransformFilter::ReceiveSample or IHXTOutputFilter::ReceiveSample is called after this method, the sample will generally be marked with end-of-stream. This method returns HXR_OK if there were no cached samples or if there were cached samples and they were released. It returns HXR_FAIL if there were cached samples that could not be released.

```
STDMETHOD(DiscardCachedSamples) (
   THIS_
   UINT32 uStream
) PURE;
```

uStream
Indicates the output stream on an input filter, and the input stream on transform and output filters.

> **Note:** Input filters only have output streams, and output filters only have input streams.

IHXTFilter::Prime

Prepares the filter for subsequent data on the given stream. A call to this method always precedes calls to `IHXTTransformFilter::ReceiveSample` or `IHXTOutputFilter::ReceiveSample`. Upon a successful completion of this method, the filter must be prepared to process data for the corresponding stream.

In regard to threading behavior, this method should only return when it is prepared to process data on the associated stream. While responsive behavior is desired, it may need to block until this condition is met.

```
STDMETHOD(Prime) (
   THIS_
   UINT32 uStream
) PURE;
```

uStream
The stream to be prepared. This parameter refers to output streams on input filters, and input streams on transform and output filters.

> **Note:** Input filters only have output streams, and output filters only have input streams.

IHXTFilter::SetFactory

Provides the filter with an object factory from which to construct property bags and related types.

```
STDMETHOD(SetFactory) (
   THIS_
   IHXCommonClassFactory* pCCF
) PURE;
```

pCCF
: Pointer to an IHXCommonClassFactory interface that manages the factory from which to build objects.

IHXTFilter::SetGraphServices

Provides the filter with a service broker from which to access shared resources and services.

```
STDMETHOD(SetGraphServices) (
   THIS_
   IHXTServiceBroker* pGraphServices
) PURE;
```

pGraphServices
: Pointer to an IHXTServiceBroker interface that manages the shared resources and services.

IHXTFilter::Teardown

Signals that the filter graph will be stopped shortly. In regard to threading behavior, this method should not block to avoid unresponsive behavior in the graph.

The filter must guarantee that no samples are sent after this method returns. In addition, the filter must be written to safely handle being released after this method is called. That is, if the filter uses threads internally, and since this method should not block, the thread should gracefully exit asynchronously.

```
STDMETHOD(Teardown) (
   THIS_
   UINT32 uStream
) PURE;
```

uStream
: The stream to be stopped. This parameter refers to output streams on input filters, and input streams on transform and output filters.

> **Note:** Input filters only have output streams, and output filters only have input streams.

IHXTFuncAreaEnum

Purpose:	Enurmerates through all functional areas pre-loaded by the log system.
Implemented by:	Logging
Header file:	ihxtlogsystem.h

The IHXTFuncAreaEnum interface contains the following methods:

- IHXTFuncAreaEnum::GetFirst
- IHXTFuncAreaEnum::GetNext

As with all Component Object Model (COM) interfaces, the IHXTFuncAreaEnum interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTFuncAreaEnum::GetFirst

Retrieves details about the first pre-loaded functional area and resets the enumerator to that position.

```
STDMETHOD(GetFirst) (
   THIS_
   const char** ppszNamespace,
   UINT32* pnNum,
   const char** ppszName
) PURE;
```

ppszNamespace
: Address of a pointer that points to the text of the namespace for this functional area.

pnNum
: Pointer to the integer that is set to the numeric value for this functional area.

ppszName
: Address of a pointer that is set to the localized translation of the text representing the functional area.

IHXTFuncAreaEnum::GetNext

Retrieves details about the next pre-loaded functional area.

```
STDMETHOD(GetNext) (
   THIS_
   const char** ppszNamespace,
   UINT32* pnNum,
   const char** ppszName
) PURE;
```

ppszNamespace
: Address of a pointer that points to the text of the namespace for this functional area.

pnNum
: Pointer to the integer that is set to the numeric value for this functional area.

ppszName
: Address of a pointer that is set to the localized translation of the text representing the functional area.

IHXTInput

Purpose:	Configures an input source for an encoding job.
Implemented by:	Encoding
Header file:	`ihxtencodingjob.h`

This interface manipulates a list of prefilters associated with the input. This interface inherits the configuration methods of `IHXTConfigurationAgent`, which are used to configure the input properties.

> **Note:** For more information on the input properties that can be configured by this interface, see "Input" on page 34.

The `IHXTInput` interface contains the following methods:

- `IHXTInput::AddPrefilter`
- `IHXTInput::GetPrefilter`
- `IHXTInput::GetPrefilterCount`
- `IHXTInput::MovePrefilter`
- `IHXTInput::RemovePrefilter`

As with all Component Object Model (COM) interfaces, the `IHXTInput` interface inherits the following `IUnknown` methods:

- `IUnknown::AddRef`

- IUnknown::QueryInterface
- IUnknown::Release

IHXTInput::AddPrefilter

Adds a prefilter to the back of the prefilter list.

```
STDMETHOD(AddPrefilter) (
   THIS_
   IHXTPrefilter* pPrefilter
) PURE;
```

pPrefilter
Pointer to an IHXTPrefilter interface that manages the prefilter to be added.

IHXTInput::GetPrefilter

Retrieves the specified prefilter.

```
STDMETHOD(GetPrefilter) (
   THIS_
   UINT32 ulIndex,
   IHXTPrefilter** ppPrefilter
) PURE;
```

ulIndex
The position of the prefilter in the index list.

ppPrefilter
Address of a pointer to an IHXTPrefilter interface that manages the prefilter information.

IHXTInput::GetPrefilterCount

Returns the number of prefilters in the index list.

```
STDMETHOD_(UINT32, GetPrefilterCount) (
   THIS
) PURE;
```

IHXTInput::MovePrefilter

Moves a prefilter from one position to another on the index list. During encoding, audio/video samples propagate through prefilters based on their list ordering (from lowest index number to highest).

```
STDMETHOD(MovePrefilter) (
   UINT32 ulOrigIndex,
   UINT32 ulDestIndex
) PURE;
```

ulOrigIndex
: The original position of the prefilter in the index list.

ulDestIndex
: The destination position of the prefilter in the index list.

IHXTInput::RemovePrefilter

Removes the specified prefilter in the index list.

```
STDMETHOD(RemovePrefilter) (
   THIS_
   UINT32 ulIndex
) PURE;
```

ulIndex
: The location in the index list of the prefilter to be removed.

IHXTInput2

Purpose:	Configures input sources and input groups for an encoding job.
Implemented by:	Encoding
Header file:	ihxtencodingjob.h

This interface provides a means of adding and configuring multiple inputs for an encoding job. This interface inherits the input configuration methods from IHXTInput, which in turn inherits the configuration methods of IHXTConfigurationAgent, which are used to configure the input properties.

> **For More Information:** See "Creating Parallel Inputs" on page 38 for more information on using this interface.

The IHXTInput2 interface contains the following methods:

- IHXTInput2::AddInput
- IHXTInput2::GetInput
- IHXTInput2::GetInputCount
- IHXTInput2::MoveInput
- IHXTInput2::RemoveInput

As with all Component Object Model (COM) interfaces, the IHXTInput2 interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTInput2::AddInput

Add an input to the back of the input list.

```
STDMETHOD(AddInput) (
   THIS_
   IHXTInput* pSource
) PURE;
```

pSource
Pointer to an IHXTInput interface that manages the input that is added.

IHXTInput2::GetInput

Retrieves the specified input in the index list.

```
STDMETHOD(GetInput) (
   THIS_
   UINT32 ulIndex,
   IHXTInput** ppSource
) PURE;
```

ulIndex
The location of the input in the index list.

ppSource
Address of a pointer to an IHXTInput interface that manages the input information.

IHXTInput2::GetInputCount

Returns the number of inputs in the list.

```
STDMETHOD_(UINT32, GetInputCount) (
   THIS
) PURE;
```

IHXTInput2::MoveInput

Moves the specified input from its original location on the index list to a new location in the index list.

```
STDMETHOD(MoveInput) (
   UINT32 ulOrigIndex,
   UINT32 ulDestIndex
) PURE;
```

ulOrigIndex
: The original location of the input in the index list.

ulDestIndex
: The destination in the index list to which the input is to be moved.

IHXTInput2::RemoveInput

Removes the input at the specified location in the index list.

```
STDMETHOD(RemoveInput) (
   THIS_
   UINT32 ulIndex
) PURE;
```

ulIndex
: The location in the index list of the input to be removed.

IHXTInputFilter

Purpose:	Provides input filter operations on media samples.
Implemented by:	Plug-ins
Header file:	ihxtbase.h

This interface inherits methods from the IHXTFilter interface.

The IHXTInputFilter interface contains the following unique methods:

- IHXTInputFilter::ReadSample
- IHXTInputFilter::SetAllocator

As with all Component Object Model (COM) interfaces, the IHXTInputFilter interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTInputFilter::ReadSample

Requests the next media sample for a particular stream. Filters will typically use the allocator provided in IHXTInputFilter::SetAllocator to create the media samples returned by the call to this method. If no sample is currently available for a particular stream, but should be shortly (such as in the case of audio/video capture), the filter should return HXR_S_NOT_HANDLED.

If a valid sample can be produced, the filter should assign the sample to the ppSample in/out parameter and return HXR_OK. The sample does not need to contain data, but it must have valid timestamps and flags. If data is available, the filter should create a sample and return it. If no data is available, the filter does not need to create the sample. Instead, the system will create one and send it downstream. When the end of the stream has been reached, the filter must either return HXR_S_END_OF_STREAM or set the HXT_SAMPLE_ENDOFSTREAM flag on the sample.

The filter should return HXR_S_NOT_HANDLED if no data is available but the end of the stream has not been reached. This method will continue to be called after this result.

During threading, a call to this method must not block. If no sample is available, but there will be in the future, this method should return HXR_S_NOT_HANDLED.

> **Note:** Do not reuse (such as read from or write to the data buffer) the media sample after passing it to this method. The media sample is not automatically copied to memory (memcpy), so accessing it might result in undefined behavior. For example, some other object might have a reference count on it and modify the buffer on another thread.

```
STDMETHOD(ReadSample) (
   THIS_
   UINT32 uStreamID,
   IHXTMediaSample** ppSample
) PURE;
```

uStreamID
: The identity of the stream that contains the media sample.

ppSample
: Address of a pointer to an IHXTMediaSample interface that manages the media sample to be read.

IHXTInputFilter::SetAllocator

Provides an allocator for each of a filter's output streams. The allocator can be reset at any time. The filter is also free to ignore the allocator (for example, in a filter that supports in-place transforms). This method is always called before data flow calls are made.

```
STDMETHOD(SetAllocator) (
   THIS_
   UINT32 uStreamID,
   IHXTSampleAllocator *pAllocator
) PURE;
```

uStreamID
: The identity of the stream for which the allocator will be provided.

pAllocator
: Pointer to an `IHXTSampleAllocator` interface that manages the allocation of media sample data buffers.

IHXTInputPreviewControl

Purpose:	Controls the preview of an input media sample.
Implemented by:	Encoding
Header file:	ihxtpreviewsink.h

The `IHXTInputPreviewControl` interface contains the following methods:

- `IHXTInputPreviewControl::Close`
- `IHXTInputPreviewControl::Open`

As with all Component Object Model (COM) interfaces, the `IHXTInputPreviewControl` interface inherits the following `IUnknown` methods:

- `IUnknown::AddRef`
- `IUnknown::QueryInterface`
- `IUnknown::Release`

IHXTInputPreviewControl::Close

Stops passing media samples to the all registered sinks and tears down the input graph.

```
STDMETHOD (Close) (
   THIS
) PURE;
```

IHXTInputPreviewControl::Open

Builds the input graph for the purposes of preview and begins passing media samples to registered sinks.

```
STDMETHOD (Open) (
   THIS
) PURE;
```

IHXTInt64Enumerator

Purpose:	Enumerates the values of a 64-bit integer list.
Implemented by:	All components
Header file:	ihxtpropertybag.h

> **Note:** This interface does not need to be implemented by itself, but you can access this interface from a property bag.

The IHXT64IntEnumerator interface contains the following methods:

- IHXTInt64Enumerator::Current
- IHXTInt64Enumerator::First
- IHXTInt64Enumerator::GetCount
- IHXTInt64Enumerator::Next

As with all Component Object Model (COM) interfaces, the IHXTInt64Enumerator interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTInt64Enumerator::Current

Gets the 64-bit integer value at the current position of the internal iterator.

```
STDMETHOD(Current) (
   THIS_
   INT64 *pValue
) PURE;
```

pValue
: Pointer to the value of the 64-bit integer.

IHXTInt64Enumerator::First

Gets the 64-bit integer value at the first position in the list of 64-bit integers.

```
STDMETHOD(First) (
   THIS_
   INT64 *pValue
) PURE;
```

pValue
Pointer to the value of the 64-bit integer.

IHXTInt64Enumerator::GetCount

Returns the total number of properties in the list of 64-bit integers.

```
STDMETHOD_(UINT32, GetCount) (
   THIS
) const PURE;
```

IHXTInt64Enumerator::Next

Advances the internal iterator and gets the 64-bit integer value at the next position in the list of 64-bit integers.

```
STDMETHOD(Next) (
   THIS_
   INT64 *pValue
) PURE;
```

pValue
Pointer to the value of the 64-bit integer.

IHXTInt64List

Purpose:	Stores a list of 64-bit integer values.
Implemented by:	All components
Header file:	`ihxtpropertybag.h`

> **Note:** This interface does not need to be implemented by itself, but you can obtain an object that implements this interface from the class factory.

The `IHXTInt64List` interface contains the following methods:

- `IHXTInt64List::Clear`

- IHXTInt64List::Compare
- IHXTInt64List::Contains
- IHXTInt64List::GetBack
- IHXTInt64List::GetEnumerator
- IHXTInt64List::GetFront
- IHXTInt64List::GetIntersection
- IHXTInt64List::GetSize
- IHXTInt64List::IsEmpty
- IHXTInt64List::PopBack
- IHXTInt64List::PopFront
- IHXTInt64List::PushBack
- IHXTInt64List::PushFront

As with all Component Object Model (COM) interfaces, the IHXTInt64List interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTInt64List::Clear

Clears all properties from the current list.

```
STDMETHOD(Clear) (
   THIS
) PURE;
```

IHXTInt64List::Compare

This method is obsolete and should not be used.

```
STDMETHOD(Compare) (
   THIS_
   IHXTInt64List *pList,
   INT64 *puValue
) const PURE;
```

IHXTInt64List::Contains

Returns TRUE if the specified value is contained in the current list. Returns FALSE if the value is not in the current list.

```
STDMETHOD_(BOOL, Contains) (
   THIS_
   INT64 value
) const PURE;
```

value
: The 64-bit integer value being searched for in the current list.

IHXTInt64List::GetBack

Returns the value of the property at the end of the current list.

```
STDMETHOD_(INT64, GetBack) (
   THIS
) PURE;
```

IHXTInt64List::GetEnumerator

Gets an enumerator that can be used to enumerate through all the items in the list.

```
STDMETHOD(GetEnumerator) (
   THIS_
   IHXTInt64Enumerator **pEnumerator
) PURE;
```

pEnumerator
: Address of a pointer to an `IHXTInt64Enumerator` interface that manages the enumerator.

IHXTInt64List::GetFront

Returns the value of the property at the beginning of the current list.

```
STDMETHOD_(INT64, GetFront) (
   THIS
) PURE;
```

IHXTInt64List::GetIntersection

Gets the intersection between the current list and another specified list.

```
STDMETHOD(GetIntersection) (
   THIS_
   IHXTInt64List *pList,
   IHXTInt64List **ppIntersection
) const PURE;
```

pList
: Pointer to an IHXTInt64List interface that manages the 64-bit integer list to compare against the current 64-bit integer list.

ppIntersection
: Address of a pointer to an IHXTInt64List interface that manages the new 64-bit integer list. This new list contains only those properties that were the same in both of the compared lists.

IHXTInt64List::GetSize

Returns the size of the current list.

```
STDMETHOD_(UINT32, GetSize) (
   THIS
) PURE;
```

IHXTInt64List::IsEmpty

If TRUE, indicates the current list is empty. If FALSE, indicates there is at least one property in the current list.

```
STDMETHOD_(BOOL, IsEmpty) (
   THIS
) PURE;
```

IHXTInt64List::PopBack

Returns the property value at the end of the list, and removes the value from the list.

```
STDMETHOD_(INT64, PopBack) (
   THIS
) PURE;
```

IHXTInt64List::PopFront

Returns the property value at the beginning of the list, and removes the value from the list.

```
STDMETHOD_(INT64, PopFront) (
   THIS
) PURE;
```

IHXTInt64List::PushBack

Places a property value at the end of the list.

```
STDMETHOD(PushBack) (
   THIS_
   INT64 value
) PURE;
```

value
The value of the property to add to the end of the list.

IHXTInt64List::PushFront

Places a property value at the beginning of the list.

```
STDMETHOD(PushFront) (
   THIS_
   INT64 value
) PURE;
```

value
The value of the property to add to the beginning of the list.

IHXTInt64Range

Purpose:	Represents a range of 64-bit integer values.
Implemented by:	All components
Header file:	ihxtpropertybag.h

> **Note:** This interface does not need to be implemented by itself, but you can obtain an object that implements this interface from the class factory.

The IHXTInt64Range interface contains the following methods:

- IHXTInt64Range::Compare
- IHXTInt64Range::GetMax
- IHXTInt64Range::GetMin
- IHXTInt64Range::GetStepSize
- IHXTInt64Range::IsInRange
- IHXTInt64Range::Set

As with all Component Object Model (COM) interfaces, the IHXTInt64Range interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTInt64Range::Compare

This method is obsolete and should not be used.

```
STDMETHOD(Compare) (
   THIS_
   IHXTInt64Range *pRange,
   IHXTInt64Range **ppResult
) const PURE;
```

IHXTInt64Range::GetMax

Returns the maximum value in the current range.

```
STDMETHOD_(INT64, GetMax) (
   THIS
) const PURE;
```

IHXTInt64Range::GetMin

Returns minimum value in the current range.

```
STDMETHOD_(INT64, GetMin) (
   THIS
) const PURE;
```

IHXTInt64Range::GetStepSize

Returns the step size of the current range.

```
STDMETHOD_(INT64, GetStepSize) (
   THIS
) const PURE;
```

IHXTInt64Range::IsInRange

Returns TRUE if the specified value is in range or FALSE if the specified value is out of range.

```
STDMETHOD_(BOOL, IsInRange) (
   THIS_
   INT64 uValue
) const PURE;
```

uValue
The value being tested to see if it is in range.

IHXTInt64Range::Set

Sets the range parameters.

```
STDMETHOD(Set) (
   THIS_
   INT64 uMin,
   INT64 uMax,
   INT64 uStepSize = 1
) PURE;
```

uMin
The minimum value to be set in the range.

uMax
The maximum value to be set in the range.

uStepSize
The step size of the values in the range.

IHXTIntEnumerator

Purpose:	Enumerates the values of an integer list.
Implemented by:	All components
Header file:	ihxtpropertybag.h

Note: This interface does not need to be implemented by itself, but you can access this interface from a property bag.

The IHXTIntEnumerator interface contains the following methods:

- IHXTIntEnumerator::Current
- IHXTIntEnumerator::First
- IHXTIntEnumerator::GetCount
- IHXTIntEnumerator::Next

As with all Component Object Model (COM) interfaces, the IHXTIntEnumerator interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTIntEnumerator::Current

Gets the integer value at the current position of the internal iterator.

```
STDMETHOD(Current) (
   THIS_
   INT32 *pValue
) PURE;
```

pValue
: Pointer to the value of the integer.

IHXTIntEnumerator::First

Gets the integer value at the first position in the list of integers.

```
STDMETHOD(First) (
   THIS_
   INT32 *pValue
) PURE;
```

pValue
: Pointer to the value of the integer.

IHXTIntEnumerator::GetCount

Returns the total number of properties in the list of integers.

```
STDMETHOD_(UINT32, GetCount) (
   THIS
) const PURE;
```

IHXTIntEnumerator::Next

Advances the internal iterator and gets the integer value at the next position in the list of integers.

```
STDMETHOD(Next) (
   THIS_
   INT32 *pValue
) PURE;
```

pValue
Pointer to value of the integer.

IHXTIntList

Purpose:	Stores a list of integer values.
Implemented by:	All components
Header file:	ihxtpropertybag.h

> **Note:** This interface does not need to be implemented by itself, but you can obtain an object that implements this interface from the class factory.

The IHXTIntList interface contains the following methods:

- IHXTIntList::Clear
- IHXTIntList::Compare
- IHXTIntList::Contains
- IHXTIntList::GetBack
- IHXTIntList::GetEnumerator
- IHXTIntList::GetFront
- IHXTIntList::GetIntersection
- IHXTIntList::GetSize
- IHXTIntList::IsEmpty
- IHXTIntList::PopBack
- IHXTIntList::PopFront
- IHXTIntList::PushBack
- IHXTIntList::PushFront

As with all Component Object Model (COM) interfaces, the IHXTIntList interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTIntList::Clear

Clears all properties from the current list.

```
STDMETHOD(Clear) (
   THIS
) PURE;
```

IHXTIntList::Compare

This method is obsolete and should not be used.

```
STDMETHOD(Compare) (
   THIS_
   IHXTIntList *pList,
   INT32 *puValue
) const PURE;
```

IHXTIntList::Contains

Returns `TRUE` if the specified value is contained in the current list. Returns `FALSE` if the value is not in the current list.

```
STDMETHOD_(BOOL, Contains) (
   THIS_
   INT32 value
) const PURE;
```

value
The integer value being searched for in the current list.

IHXTIntList::GetBack

Returns the value of the property at the end of the current list.

```
STDMETHOD_(INT32, GetBack) (
   THIS
) PURE;
```

IHXTIntList::GetEnumerator

Gets an enumerator that can be used to enumerate through all the items in the list.

```
STDMETHOD(GetEnumerator) (
   THIS_
   IHXTIntEnumerator **pEnumerator
) PURE;
```

pEnumerator
: Address of a pointer to an IHXTIntEnumerator interface that manages the enumerator.

IHXTIntList::GetFront

Returns the value of the property at the beginning of the current list.

```
STDMETHOD_(INT32, GetFront) (
   THIS
) PURE;
```

IHXTIntList::GetIntersection

Gets the intersection between the current list and another specified list.

```
STDMETHOD(GetIntersection) (
   THIS_
   IHXTIntList *pList,
   IHXTIntList **ppIntersection
) const PURE;
```

pList
: Pointer to an IHXTIntList interface that manages the integer list to compare against the current integer list.

ppIntersection
: Address of a pointer to and IHXTIntList interface that manages the new integer list. This new list contains only those properties that were the same in both of the compared lists.

IHXTIntList::GetSize

Returns the size of the current list.

```
STDMETHOD_(UINT32, GetSize) (
   THIS
) PURE;
```

IHXTIntList::IsEmpty

If TRUE, indicates the current list is empty. If FALSE, indicates there is at least one property in the current list.

```
STDMETHOD_(BOOL, IsEmpty) (
   THIS
) PURE;
```

IHXTIntList::PopBack

Returns the property value at the end of the list, and removes the value from the list.

```
STDMETHOD_(INT32, PopBack) (
   THIS
) PURE;
```

IHXTIntList::PopFront

Returns the property value at the beginning of the list, and removes the value from the list.

```
STDMETHOD_(INT32, PopFront) (
   THIS
) PURE;
```

IHXTIntList::PushBack

Places a property value at the end of the list.

```
STDMETHOD(PushBack) (
   THIS_
   INT32 value
) PURE;
```

value
: The value of the property to add to the end of the list.

IHXTIntList::PushFront

Places a property value at the beginning of the list.

```
STDMETHOD(PushFront) (
   THIS_
   INT32 value
) PURE;
```

value
The value of the property to add to the beginning of the list.

IHXTIntRange

Purpose:	Represents a range of integer values.
Implemented by:	All components
Header file:	ihxtpropertybag.h

Note: This interface does not need to be implemented by itself, but you can obtain an object that implements this interface from the class factory.

The IHXTIntRange interface contains the following methods:

- IHXTIntRange::Compare
- IHXTIntRange::GetMax
- IHXTIntRange::GetMin
- IHXTIntRange::GetStepSize
- IHXTIntRange::IsInRange
- IHXTIntRange::Set

As with all Component Object Model (COM) interfaces, the IHXTIntRange interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTIntRange::Compare

This method is obsolete and should not be used.

```
STDMETHOD(Compare) (
   THIS_
   IHXTIntRange *pRange,
   IHXTIntRange **ppResult
) const PURE;
```

IHXTIntRange::GetMax

Returns the maximum value in the current range.

```
STDMETHOD_(INT32, GetMax) (
   THIS
) const PURE;
```

IHXTIntRange::GetMin

Returns the minimum value in the current range.

```
STDMETHOD_(INT32, GetMin) (
   THIS
) const PURE;
```

IHXTIntRange::GetStepSize

Returns the step size of the current range.

```
STDMETHOD_(INT32, GetStepSize) (
   THIS
) const PURE;
```

IHXTIntRange::IsInRange

Returns TRUE if the specified value is in range or FALSE if the specified value is out of range.

```
STDMETHOD_(BOOL, IsInRange) (
   THIS_
   INT32 uValue
) const PURE;
```

uValue
The value being tested to see if it is in range.

IHXTIntRange::Set

Sets the range parameters.

```
STDMETHOD(Set) (
   THIS_
   INT32 uMin,
   INT32 uMax,
   INT32 uStepSize = 1
) PURE;
```

uMin
The minimum value to be set in the range.

uMax
: The maximum value to be set in the range.

uStepSize
: The step size of the values in the range.

IHXTLoadAdjustment

Purpose:	Adjusts load levels when the system is overloaded.
Implemented by:	Plug-ins
Header file:	ihxtbase.h

This interface is implemented by any plug-in that consumes significant CPU load. It provides a means of scaling load to meet external constraints, which is necessary to gracefully degrade quality when the system is overloaded.

The IHXTLoadAdjustment interface contains the following methods:

- IHXTLoadAdjustment::GetLoadLevel
- IHXTLoadAdjustment::SetLoadLevel

As with all Component Object Model (COM) interfaces, the IHXTLoadAdjustment interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTLoadAdjustment::GetLoadLevel

Retrieves the current load level the plug-in should try to achieve. Returns HXR_OK if the puLoadLevel parameter has been successfully updated. Returns HXR_NOTIMPL if the method is not implemented. This method is optional.

```
STDMETHOD(GetLoadLevel) (
   UINT32 *puLoadLevel
) PURE;
```

puLoadLevel
: Pointer to the current load level, in percent. The value of this parameter can be between 0 and 100.

IHXTLoadAdjustment::SetLoadLevel

Sets the requested load level the plug-in should try to achieve. Returns HXR_OK if the load level was successfully updated, HXR_FAIL if the load level could not be updated, or HXR_NOTIMPL if the method is not implemented.

```
STDMETHOD(SetLoadLevel) (
   UINT32 uLoadLevel
) PURE;
```

uLoadLevel

The load level to be set, in percent. A load level of 100 indicates the component should do what it normally does. A load level of 50 indicates the component should do half of what it normally does. The value of this parameter can be from 0 to 100.

IHXTLogObserver

Purpose:	Communicates with all observers connected to the logging system.
Implemented by:	Logging
Header file:	ihxtlogsystem.h

This interface is used by the logging system to send log messages and the end service notification to all observers connected to the logging system. IHXTLogSystem must be implemented by any observer that intends to subscribe to the log system.

The IHXTLogObserver interface contains the following methods:

- IHXTLogObserver::OnEndService
- IHXTLogObserver::ReceiveMsg

As with all Component Object Model (COM) interfaces, the IHXTLogObserver interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTLogObserver::OnEndService

Notifies the observer that the logging system is shutting down, and that all log messages have been delivered.

```
STDMETHOD(OnEndService) (
   THIS
) PURE;
```

IHXTLogObserver::ReceiveMsg

Method called on an observer when a log message is sent to the log system that passes the observers filter.

```
STDMETHOD(ReceiveMsg) (
   THIS_
   const char* szNamespace,
   EHXTLogCode nCode,
   UINT32 unFuncArea,
   const char* szFuncArea,
   INT32 nTimeStamp,
   UINT32 nMsg,
   const char* szMsg,
   const char* szJobName
) PURE;
```

szNamespace
: Pointer to the namespace used to qualify the functional area and translated message.

nCode
: The importance level of the message. One of the following:

  - LC_APP_DIAG

    Diagnostic messages of less importance than INFO messages (only important if something goes wrong). These messages can be seen by application end users and third-party developers.

  - LC_APP_INFO

    Very important messages (always want to see these messages). These messages can be seen by application end users and third-party developers.

  - LC_APP_WARN

    There was a problem, but it was handled and everything is probably all right. These messages can be seen by application end users and third-party developers.

- LC_APP_ERROR

  There was a problem and it wasn't handled. These messages can be seen by application end users and third-party developers.

- LC_SDK_DIAG

  Diagnostic messages of less importance than INFO messages (only important if something goes wrong). These messages can be seen by third-party developers.

- LC_SDK_INFO

  Very important messages (always want to see these messages). These messages can be seen by third-party developers.

- LC_SDK_WARN

  There was a problem, but it was handled and everything is probably all right. These messages can be seen by third-party developers.

- LC_SDK_ERROR

  There was a problem and it wasn't handled. These messages can be seen by third-party developers.

unFuncArea
: The numeric value used in the message as the functional area.

szFuncArea
: Pointer to the translated string for the numeric functional area, if one was found.

nTimeStamp
: The time (in milliseconds since midnight, January 1, 1970) when the log message was sent to the log.

nMsg
: The numeric value used to identify the message for translation.

szMsg
: Pointer to the actual message text; either the translation or the text sent to the `IHXTLogWriter::LogMessage` method.

szJobName
: Pointer to the name of the job from which the message originated.

IHXTLogObserver2

Purpose:	Communicates with all observers connected to the logging system.
Implemented by:	Logging
Header file:	ihxtlogsystem.h

This interface is used by the logging system to send log messages and the end service notification to all observers connected to the logging system. In addition, this interface supports flushing of the logging buffer. IHXTLogSystem must be implemented by any observer that intends to subscribe to the log system. This interface inherits all of the methods from the IHXTLogObserver interface.

The IHXTLogObserver2 interface contains the IHXTLogObserver2::Flush method.

As with all Component Object Model (COM) interfaces, the IHXTLogObserver2 interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTLogObserver2::Flush

Flushes any internal buffers when called by the log observer manager.

```
STDMETHOD(Flush) (
) PURE;
```

IHXTLogObserverManager

Purpose:	Manages observers attached to the log system.
Implemented by:	Logging
Header file:	ihxtlogsystem.h

This interface manages the subscription of observer objects to the log system.

The IHXTLogObserverManager interface contains the following methods:

- IHXTLogObserverManager::SetFilter
- IHXTLogObserverManager::SetLanguage
- IHXTLogObserverManager::Subscribe
- IHXTLogObserverManager::Unsubscribe

As with all Component Object Model (COM) interfaces, the IHXTLogObserverManager interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTLogObserverManager::SetFilter

Applies the specified filter to all future log messages delivered to the specified observer. This method always returns HXR_OK since the filter is applied asynchronously.

```
STDMETHOD(SetFilter) (
   THIS_
   const char* szFilterStr,
  IUnknown* pObserver
) PURE;
```

szFilterStr
: Pointer to an XML string specifying a filter for this observer.

pObserver
: Pointer to an IUnknown interface that identifies a previously-subscribed observer to which the filter will be applied.

IHXTLogObserverManager::SetLanguage

Sets the language that is used for translatable messages when messages are delivered to the specified observer. This method is intended for future use.

```
STDMETHOD(SetLanguage) (
   THIS_
   const char* szLanguage,
   IUnknown* pObserver
) PURE;
```

szLanguage
: Pointer to the language in which the observer will receive log messages. This parameter is currently ignored.

pObserver
: Pointer to an IUnknown interface that identifies the observer that will have its language value set.

IHXTLogObserverManager::Subscribe

Adds an observer to the log system that receives log messages, and initializes it with the specified values.

```
STDMETHOD(Subscribe) (
   THIS_
   IUnknown* pUnknown,
   const char* szFilterStr,
   const char* szLocale,
   BOOL bCatchUp
) PURE;
```

pUnknown
: Pointer to an `IUnknown` interface that identifies the observer object that must support a `QueryInterface` for the `IHXTLogObserver` interface.

szFilterStr
: Pointer to an XML string specifying an initial filter for this observer.

szLocale
: Pointer to the language in which the observer will receive log messages. This parameter is currently ignored.

bCatchUp
: If set to `true`, the observer will receive all log messages (up to 1000) previously delivered by the log system prior to this observer's subscription.

IHXTLogObserverManager::Unsubscribe

Removes an observer from the log system.

```
STDMETHOD(Unsubscribe) (
   THIS_
   IUnknown* pObserver,
   BOOL bReceiveUnsentMessages
) PURE;
```

pObserver
: Pointer to an `IUnknown` interface that identifies the observer object to be removed.

bReceiveUnsentMessages
: If set to `true`, the observer will have all remaining messages delivered that had been received by the log system, but were not yet delivered to this observer.

IHXTLogObserverManager2

Purpose:	Manages observers attached to the log system.
Implemented by:	Logging
Header file:	ihxtlogsystem.h

This interface manages the subscription of observer objects to the log system. This interface inherits the methods of the IHXTLogObserverManager interface, and includes a new method to flush the log messages from the logging queue.

The IHXTLogObserverManager2 interface contains the IHXTLogObserverManager2::FlushObservers method.

As with all Component Object Model (COM) interfaces, the IHXTLogObserverManager2 interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTLogObserverManager2::FlushObservers

Flushes the log messages from logging queue and calls IHXTLogObserver2::Flush on all observers.

```
STDMETHOD(FlushObservers) (
   THIS
) PURE;
```

IHXTLogSystem

Purpose:	Provides access to various areas of the log system.
Implemented by:	Logging
Header file:	ihxtlogsystem.h

The IHXTLogSystem interface contains the following methods:

- IHXTLogSystem::GetFunctionalAreaEnumerator
- IHXTLogSystem::GetObserverManagerInterface
- IHXTLogSystem::GetWriterInterface
- IHXTLogSystem::SetTranslationFileDirectory
- IHXTLogSystem::Shutdown

As with all Component Object Model (COM) interfaces, the IHXTLogSystem interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTLogSystem::GetFunctionalAreaEnumerator

Retrieves an interface to an enumerator that will enumerate through all functional areas in all namespaces in the specified language loaded during log system initialization.

```
STDMETHOD(GetFunctionalAreaEnumerator) (
   IHXTFuncAreaEnum** pIEnum,
   const char* szLanguage
) PURE;
```

pIEnum
Address of a pointer to an IHXTFuncAreaEnum interface that manages the functional area enumerator.

szLanguage
Pointer to the language of the functional areas to be enumerated.

IHXTLogSystem::GetObserverManagerInterface

Retrieves an interface to the observer manager, which subscribes, manages, and unsubscribes listening observers that receive log messages.

```
STDMETHOD(GetObserverManagerInterface) (
   THIS_
   IHXTLogObserverManager** ppILogObserverManager
) PURE;
```

ppILogObserverManager
Address of a pointer to an IHXTLogObserverManager interface that manages the observer manager.

IHXTLogSystem::GetWriterInterface

Retrieves an interface to the log writer, which sends messages to the log system.

```
STDMETHOD(GetWriterInterface) (
   THIS_
   IHXTLogWriter** ppIWriter
) PURE;
```

ppIWriter
: Address of a pointer to an IHXTLogWriter interface that manages the log writer.

IHXTLogSystem::SetTranslationFileDirectory

Sets the translation file directory for the log system. The files at this location will be used to translate message numbers to text strings.

```
STDMETHOD(SetTranslationFileDirectory) (
   THIS_
   const char* szTranslationFileDir
) PURE;
```

szTranslationFileDir
: Pointer to all the log system translation files.

IHXTLogSystem::Shutdown

Shuts down the log system properly.

> **Note:** Under Windows, this method should not be called from within DllMain.

```
STDMETHOD(Shutdown) (
   THIS
) PURE;
```

IHXTLogWriter

Purpose:	Sends log messages to the log system.
Implemented by:	Logging
Header file:	ihxtlogsystem.h

The IHXTLogWriter interface contains the following methods:

- IHXTLogWriter::GetTranslatedMessage
- IHXTLogWriter::LogMessage

As with all Component Object Model (COM) interfaces, the IHXTLogWriter interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTLogWriter::GetTranslatedMessage

Retrieves the translated string for the message number provided by the log system.

```
STDMETHOD(GetTranslatedMessage) (
   THIS_
   UINT32 nMessageNumber,
   const char* szNamespace,
   const char* szLanguage,
   const char** szMessage
) PURE;
```

nMessageNumber
Message number to be translated.

szNamespace
Pointer to the name space of the message to be translated.

szLanguage
Not currently implemented.

szMessage
Address of a pointer to the translated message string.

IHXTLogWriter::LogMessage

Logs a message in the log system with the specified information that will be delivered to all observers.

```
STDMETHOD(LogMessage) (
   THIS_
   const char* szNamespace,
   EHXTLogCode nLogCode,
   EHXTLogFuncArea nFuncArea,
   UINT32 nMsg,
   const char* szMsg,
   va_list args
) PURE;
```

szNamespace
: Pointer to the text identifier to qualify the functional area and numeric message parameter.

nLogCode
: The importance level of the message. One of the following:

- LC_APP_DIAG

  Diagnostic messages of less importance than INFO messages (only important if something goes wrong). These messages can be seen by application end users and third-party developers.

- LC_APP_INFO

  Very important messages (always want to see these messages). These messages can be seen by application end users and third-party developers.

- LC_APP_WARN

  There was a problem, but it was handled and everything is probably all right. These messages can be seen by application end users and third-party developers.

- LC_APP_ERROR

  There was a problem and it wasn't handled. These messages can be seen by application end users and third-party developers.

- LC_SDK_DIAG

  Diagnostic messages of less importance than INFO messages (only important if something goes wrong). These messages can be seen by third-party developers.

- LC_SDK_INFO

  Very important messages (always want to see these messages). These messages can be seen by third-party developers.

- LC_SDK_WARN

  There was a problem, but it was handled and everything is probably all right. These messages can be seen by third-party developers.

- LC_SDK_ERROR

  There was a problem and it wasn't handled. These messages can be seen by third-party developers.

nFuncArea

The general area of the system where the message originated. One of the following:

- NONE

  The message either originated in an unknown area, or no functional area is required.

- ACTIVEX

  The message originated in the ActiveX control.

- AUDCODEC

  The message originated in the audio codec.

- AUDPREFI

  The message originated in the audio prefilter plug-in.

- BCAST

  The message originated during the transmission to a server.

- CAPTURE

  The message originated during capture.

- CMDLINE

  The message originated in command line logic.

- FILEOUT

  The message originated during a write to a file.

- FILEREAD

  The message originated during the reading of audio files or the audio component of a video file. This includes reading from a general file, a DirectShow file, an uncompressed AVI file, an uncompressed QuickTime file, a compressed QuickTime file, a WAV file, or an MP3 file.

- GUI

  The message originated in the graphical user interface.

- JOBFILE

  The message originated during job file processing, that is, during serialization and deserialization.

- LIC

The message occurred because of license key activity, such as expiration of the licence key.

- POSFIL

  The message originated in the post filter.

- PUB

  This member is intended for future use.

- REMOTE

  This member is intended for future use.

- FA_SDK_CONFIG

  The message originated during SDK configuration.

- FA_SDK_ENCODE

  The message originated during SDK encoding.

- FA_SDK_CORE

  The message originated in the SDK core.

- FA_GEN_FILTER

  The message originated in some generic filter.

- STATS

  The message originated during statistics processing.

- VIDCODEC

  The message originated in the video codec.

- VIDPREFIL

  The message originated in a video prefilter plug-in.

- VIDRENDR

  The message originated during the rendering of video in a preview.

- MEDIASAMPLES

  The message originated in the media sample filter.

nMsg

Identifies the log message to be used from the translation XML files loaded by the log system upon startup. To use the `szMsg` parameter of this method for the message instead, specify `0xFFFFFFFF` for this parameter's value.

szMsg
: Pointer to the text that will be used for the log message if the `nMsg` parameter is set to `0xFFFFFFFF`.

args
: A list of variable arguments that will be substituted into the log message by the log system using `sprintf`.

(Continued on next page.)

IHXTMediaInputPin

Purpose:	Passes uncompressed media samples to the encoding engine.
Implemented by:	Encoding
Header file:	ihxtbase.h

The IHXTMediaInputPin interface can query for both IHXTVideoPinFormat and IHXTAudioPinFormat to set encoding format.

The IHXTMediaInputPin interface contains the following methods:

- IHXTMediaInputPin::EncodeSample
- IHXTMediaInputPin::GetPinEnabled
- IHXTMediaInputPin::SetPinEnabled

As with all Component Object Model (COM) interfaces, the IHXTMediaInputPin interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTMediaInputPin::EncodeSample

Passes a media sample to the encoding engine.

```
STDMETHOD(EncodeSample) (
   THIS_
   IHXTMediaSample* pMediaSample
) PURE;
```

pMediaSample
Pointer to an IHXTMediaSample interface that manages the media sample.

> **Note:** IHXTMediaSample interfaces are reference counted, and should not be re-used.

IHXTMediaInputPin::GetPinEnabled

Determines if the media pin is enabled. By default, only the audio and video media pins are enabled.

```
STDMETHOD(GetPinEnabled) (
   THIS_
   BOOL* pbEnablePin
) PURE;
```

pbEnablePin
: Pointer to a boolean value that, if true, indicates the media pin is enabled. If false, the media pin is disabled.

IHXTMediaInputPin::SetPinEnabled

Sets whether the media pin is enabled or disabled.

```
STDMETHOD(SetPinEnabled) (
   THIS_
   BOOL bEnablePin
) PURE;
```

bEnablePin
: If set to true, the media pin is enabled. If set to false, the media pin is disabled.

IHXTMediaProfile

Purpose:	Provides a collection of audiences and post-filters for an output profile.
Implemented by:	Encoding
Header file:	ihxtencodingjob.h

This interface manipulates a list of audiences. This interface inherits the configuration methods of IHXTConfigurationAgent, which are used to configure the media profile properties.

> **Note:** For more information on the media profile properties that can be configured by this interface, see "Media Profile" on page 55.

The IHXTMediaProfile interface contains the following methods:

- IHXTMediaProfile::AddAudience
- IHXTMediaProfile::GetAudience
- IHXTMediaProfile::GetAudienceCount
- IHXTMediaProfile::MoveAudience
- IHXTMediaProfile::RemoveAudience

As with all Component Object Model (COM) interfaces, the IHXTMediaProfile interface inherits the following IUnknown methods:

- IUnknown::AddRef

- IUnknown::QueryInterface
- IUnknown::Release

IHXTMediaProfile::AddAudience

Adds an audience to the back of the audience list.

```
STDMETHOD(AddAudience) (
   THIS_
   IHXTAudience* pAudience
) PURE;
```

pAudience
: Pointer to an IHXTAudience interface that manages the audience being added.

IHXTMediaProfile::GetAudience

Retrieves the specified audience in the index list.

```
STDMETHOD(GetAudience) (
   THIS_
   UINT32 ulIndex,
   IHXTAudience** ppAudience
) PURE;
```

ulIndex
: The location of the audience in the index list.

ppAudience
: Address of a pointer to an IHXTAudience interface that manages the audience information.

IHXTMediaProfile::GetAudienceCount

Returns the number of audiences in the list.

```
STDMETHOD_(UINT32, GetAudienceCount) (
   THIS
) PURE;
```

IHXTMediaProfile::MoveAudience

Moves the specified audience from its original location in the index list to a new location in the index list.

```
STDMETHOD(MoveAudience) (
   UINT32 ulOrigIndex,
   UINT32 ulDestIndex
) PURE;
```

ulOrigIndex
: The original location of the audience in the index list.

ulDestIndex
: The destination in the index list to which the audience is to be moved.

IHXTMediaProfile::RemoveAudience

Removes the audience at the specified location in the index list.

```
STDMETHOD(RemoveAudience) (
   THIS_
   UINT32 ulIndex
) PURE;
```

ulIndex
: The location in the index list of the audience to be removed.

IHXTMediaSample

Purpose:	Provides access to a data buffer, as well as flags, fields, and times associated with the data buffer.
Implemented by:	Plug-ins
Header file:	ihxtbase.h

The IHXTMediaSample interface contains the following methods:

- IHXTMediaSample::Clone
- IHXTMediaSample::CopyProperties
- IHXTMediaSample::GetDataSize
- IHXTMediaSample::GetDataStartForReading
- IHXTMediaSample::GetDataStartForWriting
- IHXTMediaSample::GetSampleField
- IHXTMediaSample::GetSampleFlags
- IHXTMediaSample::GetTime
- IHXTMediaSample::Initialize
- IHXTMediaSample::SetDataSize
- IHXTMediaSample::SetDataStart

- IHXTMediaSample::SetSampleField
- IHXTMediaSample::SetSampleFlags
- IHXTMediaSample::SetTime

As with all Component Object Model (COM) interfaces, the IHXTMediaSample interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTMediaSample::Clone

Creates an independent copy of the current media sample.

```
STDMETHOD(Clone) (
   THIS_
   IHXTMediaSample **ppMediaSample
) PURE;
```

ppMediaSample
: Address of a pointer to an IHXTMediaSample interface that manages the media sample created in the cloning process.

IHXTMediaSample::CopyProperties

Copies various properties of the media sample, such as flags, fields, and start and end times.

```
STDMETHOD(CopyProperties) (
   THIS_
   IHXTMediaSample* pInSample
) PURE;
```

pInSample
: Pointer to an IHXTMediaSample interface that manages the media sample properties being copied.

IHXTMediaSample::GetDataSize

Returns the size of the data buffer.

```
STDMETHOD_(UINT32, GetDataSize) (
   THIS_
) PURE;
```

IHXTMediaSample::GetDataStartForReading

Returns a read-only pointer to the beginning of the data buffer. The implementation of IHXTMediaSample uses a copy-on-write scheme, therefore if you intend to write to the data buffer, you should call the IHXTMediaSample::GetDataStartForWriting method.

```
STDMETHOD_(const UCHAR*, GetDataStartForReading) (
) PURE;
```

IHXTMediaSample::GetDataStartForWriting

Returns a read/write pointer to the beginning of the data buffer.

```
STDMETHOD_(UCHAR*, GetDataStartForWriting)(
) PURE;
```

IHXTMediaSample::GetSampleField

Gets a specific sample field.

```
STDMETHOD(GetSampleField) (
   THIS_
   UINT32 uFieldId,
   UINT32* puFieldValue
) PURE;
```

uFieldId

The type of sample field. One of the following:

- HXT_FIELD_ASM_RULE

  Indicates the sample field contains an ASM rule number.

- HXT_FIELD_ASM_FLAGS

  Indicates the sample field contains ASM flags.

- HXT_FIELD_STREAM_ID

  Indicates the sample field contains a stream identifier.

- HXT_FIELD_LOGICAL_STREAM_ID

  Indicates the sample field contains a logical stream identifier.

- HXT_FIELD_MULTI_FRAME_SIZES

  Reserved for future use.

puFieldValue
: A pointer to the value contained in the field specified by the `uFieldID` parameter.

IHXTMediaSample::GetSampleFlags

Returns the sample flags. One of the following:

- HXT_SAMPLE_DATADISCONTINUITY

  Indicates the start of a new segment.

- HXT_SAMPLE_KEYFRAME

  Indicates the sample is a keyframe.

- HXT_SAMPLE_ENDOFSTREAM

  Indicates the last sample for the current stream.

- HXT_STREAM_ACCURATE_END_TIME

  Indicates the actual end time of a particular stream (used when there is extraneous padding at the end of a stream).

```
STDMETHOD_(UINT32, GetSampleFlags) (
   THIS
) PURE;
```

IHXTMediaSample::GetTime

Gets the start and end times of the data buffer.

```
STDMETHOD(GetTime) (
   THIS_
   HXT_TIME* pTimeStart,
   HXT_TIME* pTimeEnd
) PURE;
```

pTimeStart
: Pointer to the start time (in milliseconds) of the data buffer.

pTimeEnd
: Pointer to the end time (in milliseconds) of the data buffer.

IHXTMediaSample::Initialize

Initializes the data buffer.

```
STDMETHOD(Initialize) (
   THIS_
   UCHAR *pBuffer,
   UINT32 uSize
) PURE;
```

pBuffer
: Pointer to data buffer being initialized.

uSize
: Indicates the size (in bytes) of the data buffer being initialized.

IHXTMediaSample::SetDataSize

Sets the size of the data buffer. Calling this method will not cause the data buffer to be re-allocated.

```
STDMETHOD(SetDataSize) (
   UINT32 uSize
) PURE;
```

uSize
: The size of the data buffer (in bytes) being set. This parameter should not exceed the size of the data buffer that was originally allocated.

IHXTMediaSample::SetDataStart

Sets the start of the data buffer. Calling this method will not cause the data buffer to be re-allocated.

```
STDMETHOD(SetDataStart) (
   THIS_
   UCHAR *pDataStart
) PURE;
```

pDataStart
: Pointer to start of the data buffer. This parameter should not precede the start of the data buffer that was originally allocated.

IHXTMediaSample::SetSampleField

Sets a specific sample field.

```
STDMETHOD(SetSampleField) (
   THIS_
   UINT32 uFieldId,
   UINT32 uFieldValue
) PURE;
```

uFieldId
: The type of sample field. One of the following:

- HXT_FIELD_ASM_RULE

  Indicates the sample field contains an ASM rule number.

- HXT_FIELD_ASM_FLAGS

  Indicates the sample field contains ASM flags.

- HXT_FIELD_STREAM_ID

  Indicates the sample field contains a stream identifier.

- HXT_FIELD_LOGICAL_STREAM_ID

  Indicates the sample field contains a logical stream identifier.

- HXT_FIELD_MULTI_FRAME_SIZES

  Reserved for future use.

puFieldValue
: The value to set in the field specified by the `uFieldID` parameter.

IHXTMediaSample::SetSampleFlags

Sets the specified sample flag.

```
STDMETHOD(SetSampleFlags) (
   THIS_
   UINT32 uFlags
) PURE;
```

uFlags
: The flag to be set. One of the following:

- HXT_SAMPLE_DATADISCONTINUITY

  Indicates the start of a new segment.

- HXT_SAMPLE_KEYFRAME

  Indicates the sample is a keyframe.

- HXT_SAMPLE_ENDOFSTREAM

Indicates the last sample for the current stream.

- HXT_STREAM_ACCURATE_END_TIME

  Indicates the actual end time of a particular stream (used when there is extraneous padding at the end of a stream).

IHXTMediaSample::SetTime

Sets the start and end times of the data buffer.

```
STDMETHOD(SetTime) (
   THIS_
   HXT_TIME* pTimeStart,
   HXT_TIME* pTimeEnd
) PURE;
```

pTimeStart
: Pointer to the start time (in milliseconds) of the data buffer.

pTimeEnd
: Pointer to the end time (in milliseconds) of the data buffer. Setting this parameter to `0xFFFFFFFF` indicates infinity.

IHXTOutputFilter

Purpose:	Performs the final processing of a media sample.
Implemented by:	Plug-ins
Header file:	`ihxtbase.h`

This interface inherits methods from the `IHXTFilter` interface.

The `IHXTOutputFilter` interface contains the `IHXTOutputFilter::ReceiveSample` method.

As with all Component Object Model (COM) interfaces, the `IHXTOutputFilter` interface inherits the following `IUnknown` methods:

- `IUnknown::AddRef`
- `IUnknown::QueryInterface`
- `IUnknown::Release`

IHXTOutputFilter::ReceiveSample

Sends the filter the next media sample to process.

```
STDMETHOD(ReceiveSample) (
   THIS_
   UINT32 uStreamID,
   IHXTMediaSample* pSample
) PURE;
```

uStreamID
The identity of the stream that contains the media sample

pSample
Pointer to an IHXTMediaSample interface that manages the media sample data buffer sent to the filter.

IHXTOutputProfile

Purpose:	Specifies one media profile and multiple output destinations.
Implemented by:	Encoding
Header file:	ihxtencodingjob.h

The output profile associates a list of destinations (that is, file writers or broadcast transmitters) with a media profile (how something will be encoded). This interface inherits the configuration methods of IHXTConfigurationAgent, which are used to configure the output profile properties.

> **Note:** For more information on the output profile properties that can be configured by this interface, see "Output Profile" on page 43.

The IHXTOutputProfile interface contains the following methods:

- IHXTOutputProfile::AddDestination
- IHXTOutputProfile::GetDestination
- IHXTOutputProfile::GetDestinationCount
- IHXTOutputProfile::GetMediaProfile
- IHXTOutputProfile::MoveDestination
- IHXTOutputProfile::RemoveDestination
- IHXTOutputProfile::SetMediaProfile

As with all Component Object Model (COM) interfaces, the IHXTOutputProfile interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface

- IUnknown::Release

IHXTOutputProfile::AddDestination

Adds a destination to the back of the destination list.

```
STDMETHOD(AddDestination) (
   THIS_
   IHXTDestination* pDestination
) PURE;
```

pDestination
: Pointer to an IHXTDestination interface that manages the destination to be added.

IHXTOutputProfile::GetDestination

Retrieves the specified destination in the index list.

```
STDMETHOD(GetDestination) (
   THIS_
   UINT32 ulIndex,
   IHXTDestination** ppDestination
) PURE;
```

ulIndex
: The number of the destination in the index list.

ppDestination
: Address of a pointer to an IHXTDestination interface that manages the destination.

IHXTOutputProfile::GetDestinationCount

Returns the number of destinations in the list.

```
STDMETHOD_(UINT32, GetDestinationCount) (
   THIS
) PURE;
```

IHXTOutputProfile::GetMediaProfile

Gets the current media profile.

```
STDMETHOD(GetMediaProfile) (
   THIS_
   IHXTMediaProfile** ppMediaProfile
) PURE;
```

ppMediaProfile
: Address of a pointer to an IHXTMediaProfile interface that manages the media profile.

IHXTOutputProfile::MoveDestination

Moves a destination from its original location in the index list to a new location. During encoding, audio/video samples propagate through destinations based on their list ordering (from lowest index number to highest).

```
STDMETHOD(MoveDestination) (
   UINT32 ulOrigIndex,
   UINT32 ulDestIndex
) PURE;
```

ulOrigIndex
: The original location of the destination in the index list.

ulDestIndex
: The location to which the destination is to be moved.

IHXTOutputProfile::RemoveDestination

Removes the specified destination in the index list.

```
STDMETHOD(RemoveDestination) (
   THIS_
   UINT32 ulIndex
) PURE;
```

ulIndex
: The location in the index list of the destination to be removed.

IHXTOutputProfile::SetMediaProfile

Sets the media profile.

```
STDMETHOD(SetMediaProfile) (
   THIS_
   IHXTMediaProfile* pMediaProfile
) PURE;
```

pMediaProfile
: Pointer to an IHXTMediaProfile interface that manages the media profile.

IHXTOutputProfile2

Purpose:	Provides metadata support for output profiles.
Implemented by:	Encoding
Header file:	ihxtencodingjob.h

This interface inherits the methods of the IHXTOutputProfile interface, which in turn inherits the configuration methods of IHXTConfigurationAgent, which are used to configure the output profile properties.

The IHXTOutputProfile2 interface contains the following methods:

- IHXTOutputProfile2::GetMetadata
- IHXTOutputProfile2::SetMetadata

As with all Component Object Model (COM) interfaces, the IHXTOutputProfile2 interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTOutputProfile2::GetMetadata

Retrieves the metadata information.

```
STDMETHOD(GetMetadata) (
   THIS_
   IHXTPropertyBag** ppMetadata
) PURE;
```

ppMetadata
: Address of a pointer to an IHXTPropertyBag interface that manages the metadata to be retrieved.

IHXTOutputProfile2::SetMetadata

Sets the metadata information.

```
STDMETHOD(SetMetadata) (
   THIS_
   IHXTPropertyBag* pMetadata
) PURE;
```

pMetadata
: Pointer to an IHXTPropertyBag interface that manages the metadata to be set.

IHXTPacketSource

This interface is reserved for future use.

IHXTPluginInfoEnum

Purpose:	Enumerates a set of plug-in object instances.
Implemented by:	Encoding
Header file:	ihxtbase.h

The IHXTPluginInfoEnum interface contains the following methods:

- IHXTPluginInfoEnum::GetCount
- IHXTPluginInfoEnum::GetPluginInfoAt

As with all Component Object Model (COM) interfaces, the IHXTPluginInfoEnum interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTPluginInfoEnum::GetCount

Returns the total number of plug-in instances.

```
STDMETHOD_(UINT32, GetCount) (
   THIS
) PURE;
```

IHXTPluginInfoEnum::GetPluginInfoAt

Gets plug-in-specific information about a particular plug-in instance.

```
STDMETHOD(GetPluginInfoAt) (
   THIS_
   UINT32 ulIndex,
   IHXTPropertyBag ** ppIPluginInfo
) PURE;
```

ulIndex
: The index number of the property bag that contains the plug-in information.

ppIPluginInfo
: Address of a pointer to an IHXTPropertyBag interface that manages plug-in information.

IHXTPluginInfoManager

Purpose:	Provides access to information about plug-ins installed in the system.
Implemented by:	Encoding
Header file:	ihxtencodingjob.h

The IHXTPluginInfoManager interface contains the IHXTPluginInfoManager::GetPluginInfoEnum method.

As with all Component Object Model (COM) interfaces, the IHXTPluginInfoManager interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTPluginInfoManager::GetPluginInfoEnum

Gets an enumerator interface that contains a list of "info-bags" describing each of the plug-ins you can choose for encoding. The "info-bag" is a property bag containing metadata about the plug-in, such as the name, sampling rate, bitrate, and so on. The list of plug-ins returned are limited to the ones containing the properties you supply.

> **Note:** Currently, this method only enumerates codecs and capture device plug-ins.

```
STDMETHOD(GetPluginInfoEnum) (
   IHXTPropertyBag *pIQueryPropertyBag,
   IHXTPluginInfoEnum ** ppIPluginInfoEnum
) PURE;
```

pIQueryPropertyBag
: Pointer to an IHXTPropertyBag interface that manages the properties used to query for the list of plug-ins. For example, to get a list of video codecs you could put the property kPropPluginType with a value of kValuePluginTypeVideoStream in this query property bag. Then only codecs with this property would be returned in the list. Passing NULL to this parameter will return information about all the codecs.

ppIPluginInfoEnum
: Address of a pointer to an IHXTPluginInfoEnum interface that manages the list of "info-bags" describing all plug-ins in the system.

IHXTPostfilter

Purpose:	Configure a postfilter's configuration properties.
Implemented by:	Encoding
Header file:	ihxtencodingjob.h

The IHXTPostfilter interface contains no methods of its own. This interface inherits the configuration methods of IHXTConfigurationAgent, which are used to configure the postfilter properties.

As with all Component Object Model (COM) interfaces, the IHXTPostfilter interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTPrefilter

Purpose:	Configure a prefilter's configuration properties.
Implemented by:	Encoding
Header file:	ihxtencodingjob.h

The IHXTPrefilter interface contains no methods of its own. This interface inherits the configuration methods of IHXTConfigurationAgent, which are used to configure the prefilter properties.

> **Note:** For more information on the prefilter properties that can be configured by this interface, see "Prefilters" on page 39.

As with all Component Object Model (COM) interfaces, the IHXTPrefilter interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTPreviewSink

Purpose:	Receives preview media samples.
Implemented by:	Encoding
Header file:	ihxtpreviewsink.h

The IHXTPreviewSink interface contains the following methods:

- IHXTPreviewSink::OnFormatChanged
- IHXTPreviewSink::OnSample

As with all Component Object Model (COM) interfaces, the IHXTPreviewSink interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTPreviewSink::OnFormatChanged

Indicates the next sample sent using the IHXTPreviewSink::OnSample call is going to have a different format.

```
STDMETHOD(OnFormatChanged) (
   THIS_
   IHXTPropertyBag* pProps
) PURE;
```

pProps
Pointer to an IHXTPropertyBag interface that manages the new format.

IHXTPreviewSink::OnSample

Indicates a new media sample is available.

```
STDMETHOD(OnSample) (
   THIS_
   IHXTMediaSample* pSample
) PURE;
```

pSample
Pointer to an IHXTMediaSample interface that manages the new media sample.

IHXTPreviewSinkControl

Purpose:	Registers preview sinks.
Implemented by:	Encoding
Header file:	ihxtpreviewsink.h

The IHXTPreviewSinkControl interface contains the following methods:

- IHXTPreviewSinkControl::AddSink
- IHXTPreviewSinkControl::GetOptimalSinkProperties
- IHXTPreviewSinkControl::RemoveSink

As with all Component Object Model (COM) interfaces, the IHXTPreviewSinkControl interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTPreviewSinkControl::AddSink

Registers a preview sink for media samples. If this method return a success code, then as soon as encoding begins, the caller will begin to receive media sample callbacks (or if the sink is being added to input, samples will be received as soon as IHXTInputPreviewControl::Open is called).

```
STDMETHOD(AddSink) (
   THIS_
   IHXTPreviewSink* pSink,
   IHXTPropertyBag* pProps
) PURE;
```

pSink
Pointer to an IHXTPreviewSink interface that manages the sink interface to be called when new media samples are available.

pProps
: Pointer to an IHXTPropertyBag interface that manages the properties of the format that the caller wants to receive.

IHXTPreviewSinkControl::GetOptimalSinkProperties

Gets the optimal (the least CPU intensive) properties to use when registering as a preview sink (by calling IHXTPreviewSinkControl::AddSink). If this call is successful, the passed-in property bag is filled with the relevant property information.

```
STDMETHOD(GetOptimalSinkProperties) (
   THIS_
   IHXTPropertyBag** ppProps
) PURE;
```

ppProps
: Address of a pointer to an IHXTPropertyBag interface that manages optimal properties.

IHXTPreviewSinkControl::RemoveSink

Unregisters the preview sink so that samples are no longer sent to the specified preview sink.

```
STDMETHOD(RemoveSink) (
   THIS_
   IHXTPreviewSink* pSink
) PURE;
```

pSink
: Pointer to an IHXTPreviewSink interface that manages the preview sink to be removed.

IHXTPreviewSinkControl3

Purpose:	Enables or disables preview sampling.
Implemented by:	Encoding
Header file:	ihxtpreviewsink.h

This interface inherits methods from the IHXTPreviewSinkControl interface.

> **For More Information:** See "Audio and Video Preview" on page 71 for more information on enabling and disabling preview sampling.

The IHXTPreviewSinkControl3 interface contains the following methods:

- IHXTPreviewSinkControl3::DisableSink
- IHXTPreviewSinkControl3::EnableSink

As with all Component Object Model (COM) interfaces, the IHXTPreviewSinkControl3 interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTPreviewSinkControl3::DisableSink

Disables the specified preview sink.

```
STDMETHOD(DisableSink) (
   THIS_
   IHXTPreviewSink* pSink
) PURE;
```

pSink
Pointer to an IHXTPreviewSink interface that manages the preview sink to disable.

IHXTPreviewSinkControl3::EnableSink

Enables the specified preview sink.

```
STDMETHOD(EnableSink) (
   THIS_
   IHXTPreviewSink* pSink
) PURE;
```

pSink
Pointer to an IHXTPreviewSink interface that manages the preview sink to enable.

IHXTProperty

Purpose:	Generic property interface that supports a single name and value pair.
Implemented by:	Encoding
Header file:	ihxtpropertybag.h

This interface manages a single property; the interface for handling multiple properties is IHXTPropertyBag.

The IHXTProperty interface contains the following methods:

- IHXTProperty::GetBool
- IHXTProperty::GetDouble
- IHXTProperty::GetDoubleList
- IHXTProperty::GetDoubleRange
- IHXTProperty::GetInt
- IHXTProperty::GetInt64
- IHXTProperty::GetInt64List
- IHXTProperty::GetInt64Range
- IHXTProperty::GetIntList
- IHXTProperty::GetIntRange
- IHXTProperty::GetKey
- IHXTProperty::GetPropertyBag
- IHXTProperty::GetString
- IHXTProperty::GetType
- IHXTProperty::GetUint
- IHXTProperty::GetUintList
- IHXTProperty::GetUintRange
- IHXTProperty::GetUnknown
- IHXTProperty::SetBool
- IHXTProperty::SetDouble
- IHXTProperty::SetDoubleList
- IHXTProperty::SetDoubleRange
- IHXTProperty::SetInt
- IHXTProperty::SetInt64
- IHXTProperty::SetInt64List
- IHXTProperty::SetInt64Range

- IHXTProperty::SetIntList
- IHXTProperty::SetIntRange
- IHXTProperty::SetPropertyBag
- IHXTProperty::SetString
- IHXTProperty::SetUint
- IHXTProperty::SetUintList
- IHXTProperty::SetUintRange
- IHXTProperty::SetUnknown

As with all Component Object Model (COM) interfaces, the IHXTProperty interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTProperty::GetBool

Gets the boolean value of the current property.

```
STDMETHOD(GetBool) (
   BOOL *pbValue
) const PURE;
```

pbValue
: Pointer to the value of the current property.

IHXTProperty::GetDouble

Gets the double value of the current property.

```
STDMETHOD(GetDouble) (
   double *pdValue
) const PURE;
```

pdValue
: Pointer to the value of the current property.

IHXTProperty::GetDoubleList

Gets the double list of the current property.

```
STDMETHOD(GetDoubleList) (
   IHXTDoubleList **ppValue
) const PURE;
```

ppValue
: Address of a pointer to an IHXTDoubleList interface that manages the double list.

IHXTProperty::GetDoubleRange

Gets the double range of the current property.

```
STDMETHOD(GetDoubleRange) (
   IHXTDoubleRange **ppValue
) const PURE;
```

ppValue
: Address of a pointer to an IHXTDoubleRange interface that manages the double range.

IHXTProperty::GetInt

Gets the integer value of the current property.

```
STDMETHOD(GetInt) (
   INT32 *pnValue
) const PURE;
```

pnValue
: Pointer to the value of the current property.

IHXTProperty::GetInt64

Gets the 64-bit integer value of the current property.

```
STDMETHOD(GetInt64) (
   INT64 *pn64Value
) const PURE;
```

pn64Value
: Pointer to the value of the current property.

IHXTProperty::GetInt64List

Gets the 64-bit integer list of the current property.

```
STDMETHOD(GetInt64List) (
   IHXTInt64List **ppValue
) const PURE;
```

ppValue
: Address of a pointer to an IHXTInt64List interface that manages the 64-bit integer list.

IHXTProperty::GetInt64Range

Gets the 64-bit integer range of the current property.

```
STDMETHOD(GetInt64Range) (
   IHXTInt64Range **ppValue
) const PURE;
```

ppValue
: Address of a pointer to an IHXTInt64Range interface that manages the 64-bit integer range.

IHXTProperty::GetIntList

Gets the integer list of the current property.

```
STDMETHOD(GetIntList) (
   IHXTIntList **ppValue
) const PURE;
```

ppValue
: Address of a pointer to an IHXTIntList interface that manages the integer list.

IHXTProperty::GetIntRange

Gets the integer range of the current property.

```
STDMETHOD(GetIntRange) (
   IHXTIntRange **ppValue
) const PURE;
```

ppValue
: Address of a pointer to an IHXTIntRange interface that manages the integer range.

IHXTProperty::GetKey

Returns the name of the current property.

```
STDMETHOD_(const CHAR*, GetKey) (
   THIS
) const PURE;
```

IHXTProperty::GetPropertyBag

Gets the property bag values of the current property.

```
STDMETHOD(GetPropertyBag) (
   IHXTPropertyBag **ppValue
) const PURE;
```

ppValue
: Address of a pointer to an `IHXTPropertyBag` interface that manages the property bag.

IHXTProperty::GetString

Gets the string value of the current property.

```
STDMETHOD(GetString) (
   const CHAR **pcszValue
) const PURE;
```

pcszValue
: Address of a pointer to the string value of the current property.

IHXTProperty::GetType

Returns the type (such as `UINT`, `double`, and so on) of the current property.

```
STDMETHOD_(UINT32, GetType) (
   THIS
) const PURE;
```

IHXTProperty::GetUint

Gets the unsigned integer value of the current property.

```
STDMETHOD(GetUint) (
   UINT32 *puValue
) const PURE;
```

puValue
: Pointer to the value of the current property.

IHXTProperty::GetUintList

Gets the unsigned integer list of the current property.

```
STDMETHOD(GetUintList) (
   IHXTUintList **ppValue
) const PURE;
```

ppValue
: Address of a pointer to an IHXTUintList interface that manages the unsigned integer list.

IHXTProperty::GetUintRange

Gets the unsigned integer range of the current property.

```
STDMETHOD(GetUintRange) (
   IHXTUintRange **ppValue
) const PURE;
```

ppValue
: Address of a pointer to an IHXTUintRange interface that manages the unsigned integer range.

IHXTProperty::GetUnknown

Gets the interface value of the current property.

```
STDMETHOD(GetUnknown) (
   IUnknown **ppType
) const PURE;
```

ppType
: Address of a pointer to an IUnknown interface that identifies the interface.

IHXTProperty::SetBool

Sets the boolean value of the current property.

```
STDMETHOD(SetBool) (
   BOOL bValue
) PURE;
```

bValue
: The value to set the current property.

IHXTProperty::SetDouble

Sets the double value of the current property.

```
STDMETHOD(SetDouble) (
   double dValue
) PURE;
```

dValue
: The value to set the current property.

IHXTProperty::SetDoubleList

Sets the double list of the current property.

```
STDMETHOD(SetDoubleList) (
   IHXTDoubleList *pValue
) PURE;
```

pValue
: Pointer to an `IHXTDoubleList` interface that manages the double list value to be set.

IHXTProperty::SetDoubleRange

Sets the double range of the current property.

```
STDMETHOD(SetDoubleRange) (
   IHXTDoubleRange *pValue
) PURE;
```

pValue
: Pointer to an `IHXTDoubleRange` interface that manages the double range value to be set.

IHXTProperty::SetInt

Sets the integer value of the current property.

```
STDMETHOD(SetInt) (
   INT32 nValue
) PURE;
```

nValue
: The value to set the current property.

IHXTProperty::SetInt64

Sets the 64-bit integer value of the current property.

```
STDMETHOD(SetInt64) (
   INT64 n64Value
) PURE;
```

n64Value
: The value to set the current property.

IHXTProperty::SetInt64List

Sets the 64-bit integer list of the current property.

```
STDMETHOD(SetInt64List) (
   IHXTInt64List *pValue
) PURE;
```

pValue
: Pointer to an IHXTInt64List interface that manages the 64-bit list value to be set.

IHXTProperty::SetInt64Range

Sets the 64-bit integer range of the current property.

```
STDMETHOD(SetInt64Range) (
   IHXTInt64Range *pValue
) PURE;
```

pValue
: Pointer to an IHXTInt64Range interface that manages the 64-bit integer range value to be set.

IHXTProperty::SetIntList

Sets the integer list of the current property.

```
STDMETHOD(SetIntList) (
   IHXTIntList *pValue
) PURE;
```

pValue
: Pointer to an IHXTIntList interface that manages the integer list value to be set.

IHXTProperty::SetIntRange

Sets the integer range of the current property.

```
STDMETHOD(SetIntRange) (
   IHXTIntRange *pValue
) PURE;
```

pValue
: Pointer to an IHXTIntRange interface that manages the integer range value to be set.

IHXTProperty::SetPropertyBag

Sets the property bag of the current property.

```
STDMETHOD(SetPropertyBag) (
   IHXTPropertyBag *pValue
) PURE;
```

pValue
: Pointer to an IHXTPropertyBag interface that manages the property bag.

IHXTProperty::SetString

Sets the string value of the current property.

```
STDMETHOD(SetString) (
   const CHAR *cszValue
) PURE;
```

cszValue
: Pointer to the value to set the current property.

IHXTProperty::SetUint

Sets the unsigned integer value of the current property.

```
STDMETHOD(SetUint) (
   UINT32 uValue
) PURE;
```

uValue
: The value to set the current property.

IHXTProperty::SetUintList

Sets the unsigned integer list of the current property.

```
STDMETHOD(SetUintList) (
   IHXTUintList *pValue
) PURE;
```

pValue
: Pointer to an IHXTUintList interface that manages the unsigned integer list value to be set.

IHXTProperty::SetUintRange

Sets the unsigned integer range of the current property.

```
STDMETHOD(SetUintRange) (
   IHXTUintRange *pValue
) PURE;
```

pValue
: Pointer to an IHXTUintRange interface that manages the unsigned integer range value to be set.

IHXTProperty::SetUnknown

Sets the interface of the current property.

```
STDMETHOD(SetUnknown) (
   IUnknown *pType
) PURE;
```

pType
: Pointer to an IUnknown interface that identifies the interface to be set.

IHXTPropertyBag

Purpose:	Contains sets of properties including simple (UINT, string) and complex (ranges, lists) data types.
Implemented by:	All components
Header file:	ihxtpropertybag.h

A property bag stores a collection of properties. Each property has a name and a value. The property name is always a string. The values are a distinct data type, such as a sting, an unsigned integer, a double, and so on.

The IHXTPropertyBag interface contains the following methods:

- IHXTPropertyBag::GetBool
- IHXTPropertyBag::GetCount
- IHXTPropertyBag::GetDouble
- IHXTPropertyBag::GetDoubleList

- IHXTPropertyBag::GetDoubleRange
- IHXTPropertyBag::GetInt
- IHXTPropertyBag::GetInt64
- IHXTPropertyBag::GetInt64List
- IHXTPropertyBag::GetInt64Range
- IHXTPropertyBag::GetIntList
- IHXTPropertyBag::GetIntRange
- IHXTPropertyBag::GetProperty
- IHXTPropertyBag::GetPropertyBag
- IHXTPropertyBag::GetPropertyBagEnumerator
- IHXTPropertyBag::GetPropertyEnumerator
- IHXTPropertyBag::GetString
- IHXTPropertyBag::GetUint
- IHXTPropertyBag::GetUintList
- IHXTPropertyBag::GetUintRange
- IHXTPropertyBag::GetUnknown
- IHXTPropertyBag::Remove
- IHXTPropertyBag::SetBool
- IHXTPropertyBag::SetDouble
- IHXTPropertyBag::SetDoubleList
- IHXTPropertyBag::SetDoubleRange
- IHXTPropertyBag::SetInt
- IHXTPropertyBag::SetInt64
- IHXTPropertyBag::SetInt64List
- IHXTPropertyBag::SetInt64Range
- IHXTPropertyBag::SetIntList
- IHXTPropertyBag::SetIntRange
- IHXTPropertyBag::SetProperty
- IHXTPropertyBag::SetPropertyBag
- IHXTPropertyBag::SetString
- IHXTPropertyBag::SetUint
- IHXTPropertyBag::SetUintList
- IHXTPropertyBag::SetUintRange
- IHXTPropertyBag::SetUnknown

As with all Component Object Model (COM) interfaces, the IHXTPropertyBag interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTPropertyBag::GetBool

Gets a boolean value from the specified property.

```
STDMETHOD(GetBool) (
   THIS_
   const CHAR* pName,
   BOOL *pbValue
) PURE;
```

pName
: Pointer to the name of the property from which to get the boolean value.

pbValue
: Pointer to the boolean value of the property.

IHXTPropertyBag::GetCount

Returns the number of properties in the current property bag.

```
STDMETHOD_(UINT32, GetCount) (
   THIS
) PURE;
```

IHXTPropertyBag::GetDouble

Gets a double value from the specified property.

```
STDMETHOD(GetDouble) (
   THIS_
   const CHAR* pName,
   double *puValue
) PURE;
```

pName
: Pointer to the name of the property from which to get the double value.

puValue
: Pointer to value of the property.

IHXTPropertyBag::GetDoubleList

Gets a double list from the specified property.

```
STDMETHOD(GetDoubleList) (
   THIS_
   const CHAR* pName,
   IHXTDoubleList **ppValue
) PURE;
```

pName
: Pointer to name of the property from which to get the double list.

ppValue
: Address of a pointer to an `IHXTDoubleList` interface that manages the double list value.

IHXTPropertyBag::GetDoubleRange

Gets a double range from the specified property.

```
STDMETHOD(GetDoubleRange) (
   THIS_
   const CHAR* pName,
   IHXTDoubleRange **ppValue
) PURE;
```

pName
: Pointer to the name of the property from which to get the double range.

ppValue
: Address of a pointer to an `IHXTDoubleRange` interface that manages the double range value.

IHXTPropertyBag::GetInt

Gets the integer value of the specified property.

```
STDMETHOD(GetInt) (
   THIS_
   const CHAR* pName,
   INT32 *puValue
) PURE;
```

pName
: Pointer to the name of the property from which to get the integer value.

puValue
: Pointer to the value of the property.

IHXTPropertyBag::GetInt64

Gets the 64-bit integer value of the specified property.

```
STDMETHOD(GetInt64) (
   THIS_
   const CHAR* pName,
   INT64 *puValue
) PURE;
```

pName
: Pointer to the name of the property from which to get the 64-bit integer value.

puValue
: Pointer to value of the property.

IHXTPropertyBag::GetInt64List

Gets the 64-bit integer list of the specified property.

```
STDMETHOD(GetInt64List) (
   THIS_
   const CHAR* pName,
   IHXTInt64List **ppValue
) PURE;
```

pName
: Pointer to name of the property from which to get the 64-bit integer list.

ppValue
: Address of a pointer to an `IHXTInt64List` interface that manages the 64-bit integer list value.

IHXTPropertyBag::GetInt64Range

Gets the 64-bit integer range of the specified property.

```
STDMETHOD(GetInt64Range) (
   THIS_
   const CHAR* pName,
   IHXTInt64Range **ppValue
) PURE;
```

pName
: Pointer to the name of the property from which to get the 64-bit integer range.

ppValue
: Address of a pointer to an `IHXTInt64Range` interface that manages the 64-bit integer range value.

IHXTPropertyBag::GetIntList

Gets the integer list of the specified property.

```
STDMETHOD(GetIntList) (
   THIS_
   const CHAR* pName,
   IHXTIntList **ppValue
) PURE;
```

pName
: Pointer to the name of the property from which to get the integer list.

ppValue
: Address of a pointer to an `IHXTIntList` interface that manages the integer list value.

IHXTPropertyBag::GetIntRange

Gets the integer range of the specified property.

```
STDMETHOD(GetIntRange) (
   THIS_
   const CHAR* pName,
   IHXTIntRange **ppValue
) PURE;
```

pName
: Pointer to the name of the property from which to get the integer range.

ppValue
: Address of a pointer to an `IHXTIntRange` interface that manages the integer range value.

IHXTPropertyBag::GetProperty

Gets the property value of the specified property.

```
STDMETHOD(GetProperty) (
   THIS_
   const CHAR* pName,
   IHXTProperty **pProperty
) PURE;
```

pName
: Pointer to the name of the property from which to get the property value.

pProperty
: Address of a pointer to an IHXTProperty interface that manages the property value.

IHXTPropertyBag::GetPropertyBag

Gets the property bag of the specified property. This method is provided for nested property elements.

```
STDMETHOD(GetPropertyBag) (
   THIS_
   const CHAR* pName,
   IHXTPropertyBag **ppValue
) PURE;
```

pName
: Pointer to the name of the property from which to get the property bag.

ppValue
: Address of a pointer to an IHXTPropertyBag interface that manages the property bag.

IHXTPropertyBag::GetPropertyBagEnumerator

Gets the property bag enumerator. The property bag enumerator can then be used to enumerate through the nested property bags.

```
STDMETHOD(GetPropertyBagEnumerator) (
   THIS_
   IHXTPropertyEnumerator **ppEnumerator
) PURE;
```

ppEnumerator
: Address of a pointer to an IHXTPropertyEnumerator interface that manages the property bag enumerator.

IHXTPropertyBag::GetPropertyEnumerator

Gets the property enumerator. The property enumerator lets you generically enumerate through all the properties in a property bag. For example, you could use this enumerator if you have some code that prints out the name of each property within a property bag.

```
STDMETHOD(GetPropertyEnumerator) (
   THIS_
   IHXTPropertyEnumerator **ppEnumerator
) PURE;
```

ppEnumerator
: Address of a pointer to an `IHXTPropertyEnumerator` interface that manages the property enumerator.

IHXTPropertyBag::GetString

Gets the string value for the specified property.

```
STDMETHOD(GetString) (
   THIS_
   const CHAR* pName,
   const CHAR **pcszValue
) PURE;
```

pName
: Pointer to the name of the property from which to get the string value.

pcszValue
: Address of a pointer to string associated with the property.

IHXTPropertyBag::GetUint

Gets the unsigned integer value of the specified property.

```
STDMETHOD(GetUint) (
   THIS_
   const CHAR* pName,
   UINT32 *puValue
) PURE;
```

pName
: Pointer to the name of the property from which to get the unsigned integer value.

puValue
: Pointer to the value of the property.

IHXTPropertyBag::GetUintList

Gets the unsigned integer list of the specified property.

```
STDMETHOD(GetUintList) (
   THIS_
   const CHAR* pName,
   IHXTUintList **ppValue
) PURE;
```

pName
: Pointer to the name of the property from which to get the unsigned integer list.

ppValue
: Address of a pointer to an `IHXTUintList` interface that manages the unsigned integer list value.

IHXTPropertyBag::GetUintRange

Gets the unsigned integer range of the specified property.

```
STDMETHOD(GetUintRange) (
   THIS_
   const CHAR* pName,
   IHXTUintRange **ppValue
) PURE;
```

pName
: Pointer to name of the property from which to get the unsigned integer range.

ppValue
: Address of a pointer to an `IHXTUintRange` interface that manages the unsigned integer range value.

IHXTPropertyBag::GetUnknown

Gets the interface of the specified property.

```
STDMETHOD(GetUnknown) (
   THIS_
   const CHAR* pName,
   IUnknown **ppType
) PURE;
```

pName
: Pointer to name of the property from which to get the interface.

ppType
: Address of a pointer to an `IUnknown` interface that identifies the interface.

IHXTPropertyBag::Remove

Removes the specified property from the property bag.

```
STDMETHOD(Remove) (
   THIS_
   const CHAR* pName
) PURE;
```

pName
: Pointer to the name of the property to be removed.

IHXTPropertyBag::SetBool

Sets the specified property to a boolean value.

```
STDMETHOD(SetBool) (
   THIS_
   const CHAR* pName,
   BOOL bValue
) PURE;
```

pName
: Pointer to the name of the property to be set.

bValue
: The value to which the property is set.

IHXTPropertyBag::SetDouble

Sets the specified property to a double value.

```
STDMETHOD(SetDouble) (
   THIS_
   const CHAR* pName,
   double nValue
) PURE;
```

pName
: Pointer to the name of the property to be set.

nValue
: The value to which the property is set.

IHXTPropertyBag::SetDoubleList

Sets the specified property with a double list.

```
STDMETHOD(SetDoubleList) (
   THIS_
   const CHAR* pName,
   IHXTDoubleList *pValue
) PURE;
```

pName
: Pointer to the name of the property to be set.

pValue
: Pointer to an `IHXTDoubleList` interface that manages the double list.

IHXTPropertyBag::SetDoubleRange

Sets the specified property with a double range.

```
STDMETHOD(SetDoubleRange) (
   THIS_
   const CHAR* pName,
   IHXTDoubleRange *pValue
) PURE;
```

pName
: Pointer to the name of the property to be set.

pValue
: Pointer to an `IHXTDoubleRange` interface that manages the double range.

IHXTPropertyBag::SetInt

Sets the specified property to an integer.

```
STDMETHOD(SetInt) (
   THIS_
   const CHAR* pName,
   INT32 nValue
) PURE;
```

pName
Pointer to name of the property to be set.

nValue
The value to which the property is set.

IHXTPropertyBag::SetInt64

Sets the specified property to a 64-bit integer.

```
STDMETHOD(SetInt64) (
   THIS_
   const CHAR* pName,
   INT64 nValue
) PURE;
```

pName
Pointer to the name of the property to be set.

nValue
The value to which the property is set.

IHXTPropertyBag::SetInt64List

Sets the specified property with a 64-bit integer list.

```
STDMETHOD(SetInt64List) (
   THIS_
   const CHAR* pName,
   IHXTInt64List *pValue
) PURE;
```

pName
Pointer to the name of the property to be set.

pValue
Pointer to an `IHXTInt64List` interface that manages the 64-bit integer list.

IHXTPropertyBag::SetInt64Range

Sets the specified property with a 64-bit integer range.

```
STDMETHOD(SetInt64Range) (
   THIS_
   const CHAR* pName,
   IHXTInt64Range *pValue
) PURE;
```

pName
: Pointer to the name of the property to be set.

pValue
: Pointer to an IHXTInt64Range interface that manages the 64-bit integer range.

IHXTPropertyBag::SetIntList

Sets the specified property with an integer list.

```
STDMETHOD(SetIntList) (
   THIS_
   const CHAR* pName,
   IHXTIntList *pValue
) PURE;
```

pName
: Pointer to the name of the property to be set.

pValue
: Pointer to an IHXTIntList interface that manages the integer list.

IHXTPropertyBag::SetIntRange

Sets the specified property with an integer range.

```
STDMETHOD(SetIntRange) (
   THIS_
   const CHAR* pName,
   IHXTIntRange *pValue
) PURE;
```

pName
: Pointer to the name of the property to be set.

pValue
: Pointer to an IHXTIntRange interface that manages the integer range.

IHXTPropertyBag::SetProperty

Sets a property using the IHXTProperty interface.

```
STDMETHOD(SetProperty) (
   THIS_
   IHXTProperty *pProperty
) PURE;
```

pProperty
: Pointer to an IHXTProperty interface that manages the property being set.

IHXTPropertyBag::SetPropertyBag

Sets the specified property with a property bag. This method is used to nest property bags.

```
STDMETHOD(SetPropertyBag) (
   THIS_
   const CHAR* pName,
   IHXTPropertyBag *pValue
) PURE;
```

pName
: Pointer to name of the property to be set.

pValue
: Pointer to an IHXTPropertyBag interface that manages the property bag.

IHXTPropertyBag::SetString

Sets the specified property to a string.

```
STDMETHOD(SetString) (
   THIS_
   const CHAR* pName,
   const CHAR *cszValue
) PURE;
```

pName
: Pointer to name of the property to be set.

cszValue
: Pointer to a string that is associated with the property.

IHXTPropertyBag::SetUint

Sets the specified property to an unsigned integer value.

```
STDMETHOD(SetUint) (
   THIS_
   const CHAR* pName,
   UINT32 uValue
) PURE;
```

pName
: Pointer to the name of the property to be set.

uValue
: The value to which the property is set.

IHXTPropertyBag::SetUintList

Sets the specified property with an unsigned integer list.

```
STDMETHOD(SetUintList) (
   THIS_
   const CHAR* pName,
   IHXTUintList *pValue
) PURE;
```

pName
: Pointer to the name of the property to be set.

pValue
: Pointer to an `IHXTUintList` interface that manages the unsigned integer list.

IHXTPropertyBag::SetUintRange

Sets the specified property with an unsigned integer range.

```
STDMETHOD(SetUintRange) (
   THIS_
   const CHAR* pName,
   IHXTUintRange *pValue
) PURE;
```

pName
: Pointer to the name of the property to be set.

pValue
: Pointer to an `IHXTUintRange` interface that manages the unsigned integer range.

IHXTPropertyBag::SetUnknown

Sets the specified property to an interface.

```
STDMETHOD(SetUnknown) (
   THIS_
   const CHAR* pName,
   IUnknown *pType
) PURE;
```

pName
: Pointer to then name of the property to be set.

pType
: Pointer to an IUnknown interface that identifies the interface to which the property is to be set.

(Continued on next page.)

IHXTPropertyEnumerator

Purpose:	Provides generic enumeration of all properties in a property bag.
Implemented by:	Encoding, plug-ins
Header file:	ihxtpropertybag.h

The IHXTPropertyEnumerator interface contains the following methods:

- IHXTPropertyEnumerator::Current
- IHXTPropertyEnumerator::First
- IHXTPropertyEnumerator::GetCount
- IHXTPropertyEnumerator::Next

As with all Component Object Model (COM) interfaces, the IHXTPropertyEnumerator interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTPropertyEnumerator::Current

Gets the property value at the current position of the internal iterator.

```
STDMETHOD(Current) (
   THIS_
   IHXTProperty **pValue
) PURE;
```

pValue
: Address of a pointer to an IHXTProperty interface that manages the property.

IHXTPropertyEnumerator::First

Gets the property value at the first position in the list of property values.

```
STDMETHOD(First) (
   THIS_
   IHXTProperty **pValue
) PURE;
```

pValue
: Address of a pointer to an IHXTProperty interface that manages property.

IHXTPropertyEnumerator::GetCount

Returns the total number of properties in the list of properties.

```
STDMETHOD_(UINT32, GetCount) (
   THIS
) const PURE;
```

IHXTPropertyEnumerator::Next

Advances the internal iterator and gets the property value at the next position in the list of properties.

```
STDMETHOD(Next) (
   THIS_
   IHXTProperty **pValue
) PURE;
```

pValue
: Address of a pointer to an IHXTProperty interface that manages the property.

IHXTPropertyUtility

Purpose:	Compares and clones properties and property bags.
Implemented by:	Encoding, plug-ins
Header file:	ihxtpropertybag.h

This interface is a utility class for managing property bags. Property bags are used throughout the Helix DNA Producer SDK to initialize and configure plug-ins with their set of properties. The common functionality of this interface is to duplicate a property bag onto a new property bag and to perform property bag comparisons.

The IHXTPropertyUtility interface contains the following methods:

- IHXTPropertyUtility::ArePropertiesEquivalent
- IHXTPropertyUtility::ArePropertyBagsEquivalent
- IHXTPropertyUtility::CloneProperty
- IHXTPropertyUtility::ClonePropertyBag
- IHXTPropertyUtility::IsPropertyBagCompatibleWith
- IHXTPropertyUtility::IsPropertyCompatibleWith

As with all Component Object Model (COM) interfaces, the IHXTPropertyUtility interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTPropertyUtility::ArePropertiesEquivalent

Compares two properties to determine if they are alike.

```
STDMETHOD(ArePropertiesEquivalent) (
   IHXTProperty *pProperty1,
   IHXTProperty *pProperty2
) PURE;
```

pProperty1
Pointer to an IHXTProperty interface that manages the first property to compare.

pProperty2
Pointer to an IHXTProperty interface that manages the second property to compare against the first property.

IHXTPropertyUtility::ArePropertyBagsEquivalent

Compares two property bags to determine if they are alike.

```
STDMETHOD(ArePropertyBagsEquivalent) (
   IHXTPropertyBag *pPropertyBag1,
   IHXTPropertyBag *pPropertyBag2
) PURE;
```

pPropertyBag1
Pointer to an IHXTPropertyBag interface that manages the first property bag to compare.

pPropertyBag2
Pointer to an IHXTPropertyBag interface that manages the second property bag to compare against the first property bag.

IHXTPropertyUtility::CloneProperty

Creates an exact copy of the specified property.

```
STDMETHOD(CloneProperty) (
   IHXTProperty *pSourceProp,
   IHXTProperty **ppClonedProp = NULL
) PURE;
```

pSourceProp
: Pointer to an IHXTProperty interface that manages the property to be cloned.

ppClonedProp
: Address of a pointer to an IHXTProperty interface that manages the copy of the property.

IHXTPropertyUtility::ClonePropertyBag

Creates an exact copy of the specified property bag.

```
STDMETHOD(ClonePropertyBag) (
   IHXTPropertyBag *pSourceBag,
   IHXTPropertyBag **ppClonedBag = NULL
) PURE;
```

pSourceBag
: Pointer to an IHXTPropertyBag interface that manages the property bag to be cloned.

ppClonedBag
: Address of a pointer to an IHXTPropertyBag interface that manages the copy of the property bag.

IHXTPropertyUtility::IsPropertyBagCompatibleWith

Compares two property bags and determines if they are compatible with each other. That is, this method determines if all properties are present in both property bags and that the values are equal. If the property bags contain ranges or lists, this method determines if the first property bags's value is within the range or list of the other.

```
STDMETHOD(IsPropertyBagCompatibleWith) (
   IHXTPropertyBag *pPropertyBag1,
   IHXTPropertyBag *pPropertyBag2,
   IHXTPropertyBag **ppResult = NULL,
   IHXTPropertyBag **ppErrors = NULL
) PURE;
```

pPropertyBag1
: Pointer to an IHXTPropertyBag interface that manages the first property bag to compare.

pPropertyBag2
: Pointer to an IHXTPropertyBag interface that manages the second property bag to compare against the first property bag.

ppResult
: Address of a pointer to an IHXTPropertyBag interface that manages the result of the comparison between the property bags.

ppErrors
: Address of a pointer to an IHXTPropertyBag interface that manages the property name(s) in the case of either an error in atomic comparison, or if the property was not present at all.

IHXTPropertyUtility::IsPropertyCompatibleWith

Compares two properties and determines if they are compatible with each other. That is, this method determines if the value of all properties are equal. If the properties contain ranges or lists, this method determines if the first property's value is within the range or list of the other.

```
STDMETHOD(IsPropertyCompatibleWith) (
   IHXTProperty *pProperty1,
   IHXTProperty *pProperty2,
   IHXTProperty **ppResult = NULL
) PURE;
```

pProperty1
: Pointer to an IHXTProperty interface that manages the first property to compare.

pProperty2
: Pointer to an IHXTProperty interface that manages the second property to compare against the first property.

ppResult
: Address of a pointer to an IHXTProperty interface that manages the result of the comparison between the two properties.

IHXTSampleAllocator

Purpose:	Allocates media sample data buffers.
Implemented by:	Filters
Header file:	ihxtbase.h

The IHXTSampleAllocator interface contains the IHXTSampleAllocator::GetMediaSampleOfSize methods.

As with all Component Object Model (COM) interfaces, the IHXTSampleAllocator interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTSampleAllocator::GetMediaSampleOfSize

Gets a media sample data buffer of the specified size.

```
STDMETHOD(GetMediaSampleOfSize) (
   THIS_
   UINT32 uSize,
   IHXTMediaSample** ppMediaSample
) PURE;
```

uSize
The size of the data buffer in bytes.

ppMediaSample
Address of a pointer to an IHXTMediaSample interface that manages the data buffer.

IHXTSampleSink

Purpose:	Receives the media samples.
Implemented by:	Filters
Header file:	ihxtbase.h

This interface is implemented by any component that can receive media samples, for example filters.

The IHXTSampleSink interface contains the IHXTSampleSink::ReceiveSample method.

As with all Component Object Model (COM) interfaces, the IHXTSampleSink interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTSampleSink::ReceiveSample

Retrieves the media sample data buffer.

> **Note:** Do not reuse (such as read from or write to the data buffer) the media sample after passing it to this method. The media sample is not automatically copied to memory (memcpy) so accessing it may result in undefined behavior. For example, some other object might have a reference count on it and modify the buffer on another thread.

```
STDMETHOD(ReceiveSample) (
   THIS_
   IHXTMediaSample *pSample
) PURE;
```

pSample
Pointer to an IHXTMediaSample interface that manages media sample data buffer.

IHXTSerializeBuffer

Purpose:	Serializes objects.
Implemented by:	Encoding
Header file:	ihxtencodingjob.h

This interface serializes objects. For example, you could have an IHXTEncodingJob interface that is fully configured, serialize it to a buffer that you write to disk, and quit your application. Then you run the application again, deserialize the data from the disk, and you would have an IHXTEncodingJob interface exactly like the interface you previously set.

The IHXTSerializeBuffer interface contains the following methods:

- IHXTSerializeBuffer::ReadFromBuffer
- IHXTSerializeBuffer::WriteToBuffer

As with all Component Object Model (COM) interfaces, the IHXTSerializeBuffer interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTSerializeBuffer::ReadFromBuffer

Reads some XML code from a buffer.

```
STDMETHOD(ReadFromBuf) (
   THIS_
   IHXBuffer* pszXmlBody,
   BOOL bForceInitialization=FALSE,
   IHXTPropertyBag** ppInitErrorBag=NULL
) PURE;
```

pszXmlBody
: Pointer to an IHXBuffer interface that manages the XML code to be read.

bForceInitialization
: If set to false (default), this method will not create the object if any of the initialization properties fail to be validated. This ensures you never have an object in a bad state. However, this can have undesirable side effects, like causing a job file to fail to deserialize if the input filename does not exist, or if the specified capture device is in use. If this parameter is set to true, the object is forced to be created even with an invalid property. This allows the object to be created even though it cannot be part of an encode—the calling SDK application can then inspect the object and determine the problem that occurred during initialization.

ppInitErrorBag
: Address of a pointer to an IHXTPropertyBag interface that manages the invalid properties that caused initialization to fail.

IHXTSerializeBuffer::WriteToBuffer

Writes some XML code to a buffer.

```
STDMETHOD(WriteToBuffer) (
   THIS_
   IHXBuffer* pszXmlBody,
   IHXTSerializationCallback* pSerialCallback=NULL
) PURE;
```

pszXmlBody
: Pointer to an IHXBuffer interface that manages the XML code to be written to the buffer.

pSerialCallback
: Pointer to an IHXTSerializationCallback interface that manages the callback of the object being serialized. If this parameter is NULL, no callbacks will occur.

IHXTSerializationCallback

Purpose:	Manages the callbacks of an object being serialized.
Implemented by:	Custom serialization objects
Header file:	ihxtencodingjob.h

If the pSerialCallback parameter in either the IHXTSerializeBuffer::WriteToBuffer or the IHXTUserConfigFile::WriteToFile method is not set to NULL, this interface is called before a particular object is serialized. This gives you the opportunity to change the properties that are actually serialized or skip the serialization of certain objects (for example, you may not want the audio gain settings to be serialized—you could use this interface to prevent that from happening).

The IHXTSerializationCallback interface contains the IHXTSerializationCallback::OnSerializeObject method.

As with all Component Object Model (COM) interfaces, the IHXTServiceBroker interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTSerializationCallback::OnSerializeObject

Provides an opportunity to modify or delete a particular object being serialized. This method is called immediately before a particular object is serialized. For example, if you have an encoding job that is being serialized, you will get a callback for each child object, that is, a callback for the input, a callback for each prefilter, and so on.

```
STDMETHOD(OnSerializeObject) (
   THIS_
   const IUnknown* pObject,
   IHXTPropertyBag* pClonedBag,
   BOOL* pbIsOkToSerialize
) PURE;
```

pObject
: Pointer to an IUnknown interface that identifies user-specified properties of the property bag for the object currently being serialized.

pClonedBag
: Pointer to an IHXTPropertyBag interface that manages a copy of the individual properties of the object. Provided for future use.

pbIsOkToSerialize
: Pointer to a boolean value that, if set to true, allows serialization of the object to continue. If set to false, serialization of the current object will not occur.

IHXTServiceBroker

Purpose:	Requests services using the interface GUID.
Implemented by:	Helix DNA Producer
Header file:	ihxtbase.h

Information provided by this interface is passed back to the user through the IHXTFilter::SetGraphServices method.

The IHXTServiceBroker interface contains the IHXTServiceBroker::GetService method.

As with all Component Object Model (COM) interfaces, the IHXTServiceBroker interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTServiceBroker::GetService

Requests an object that implements the interface specified by the GUID.

```
STDMETHOD(GetService) (
   THIS_
   const GUID &guid,
   IUnknown **ppUnknown
) PURE;
```

guid
: The GUID that corresponds to the required service.

ppUnknown
: Address of a pointer to an IUnknown interface that identifies the interface to use for this service.

IHXTStatistics

Purpose:	Gets statistics.
Implemented by:	Encoding
Header file:	ihxtbase.h

This interface is implemented by filter, stream, and destination objects that give out statistics.

> **Note:** Currently, only streams give out statistics. See "Statistics" on page 61 for more information.

The IHXTStatistics interface contains the following methods:

- IHXTStatistics::GetCurrentStatistics
- IHXTStatistics::GetLifeTimeStatistics

As with all Component Object Model (COM) interfaces, the IHXTStatistics interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTStatistics::GetCurrentStatistics

Gets the current statistics.

```
STDMETHOD(GetCurrentStatistics) (
   IHXTPropertyBag* pStatsBag
) PURE;
```

pStatsBag
Pointer to an IHXTPropertyBag interface that manages the current statistics.

IHXTStatistics::GetLifeTimeStatistics

Gets all statistics available from the beginning of the encoding job.

```
STDMETHOD(GetLifeTimeStatistics) (
   IHXTPropertyBag* pStatsBag
) PURE;
```

pStatsBag
Pointer to an IHXTPropertyBag interface that manages the lifetime statistics.

IHXTStreamConfig

Purpose:	Sets media-specific encoding properties.
Implemented by:	Encoding
Header file:	ihxtencodingjob.h

Sets media-specific encoding properties, such as an audio or video codec. An audience contains multiple audio stream configuration objects. When encoding, one stream configuration is chosen based on the audio format (music, voice) and whether encoding is audio only, or audio and video.

The IHXTStreamConfig interface currently contains no methods; it inherits all of its methods from IHXTConfigurationAgent.

> **Note:** For more information on the stream properties that can be configured by this interface, see "Streams" on page 57.

As with all Component Object Model (COM) interfaces, the IHXTStreamConfig interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTStringEnumerator

Purpose:	Enumerates the values of an string list.
Implemented by:	All components
Header file:	ihxtpropertybag.h

> **Note:** This interface does not need to be implemented by itself, but you can obtain an object that implements this interface from the class factory.

The IHXTStringEnumerator interface contains the following methods:

- IHXTStringEnumerator::Current
- IHXTStringEnumerator::First
- IHXTStringEnumerator::GetCount
- IHXTStringEnumerator::Next

As with all Component Object Model (COM) interfaces, the IHXTStringEnumerator interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTStringEnumerator::Current

Gets the string at the current position of the internal iterator.

```
STDMETHOD(Current) (
   THIS_
   PCSTR *pValue
) PURE;
```

pValue
: Pointer to the contents of the string.

IHXTStringEnumerator::First

Gets the string at the first position in the list of strings.

```
STDMETHOD(First) (
   THIS_
   PCSTR *pValue
) PURE;
```

pValue
Pointer to the contents of the string.

IHXTStringEnumerator::GetCount

Returns the total number of properties in the list of strings.

```
STDMETHOD_(UINT32, GetCount) (
   THIS
) const PURE;
```

IHXTStringEnumerator::Next

Advances the internal iterator and gets the string at the next position in the list of strings.

```
STDMETHOD(Next) (
   THIS_
   PCSTR *pValue
) PURE;
```

pValue
Pointer to the contents of the string.

IHXTTime

Purpose:	Provides a general-purpose time interface.
Implemented by:	Helix DNA Producer
Header file:	ihxtbase.h

The time being manipulated by this interface generally represents a duration, but could also be the time on a time line.

The IHXTTime interface contains the following methods:

- IHXTTime::GetMilliSeconds
- IHXTTime::GetTime
- IHXTTime::GetTimeString
- IHXTTime::SetMilliSeconds
- IHXTTime::SetTime
- IHXTTime::SetTimeString

As with all Component Object Model (COM) interfaces, the IHXTTime interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTTime::GetMilliSeconds

Gets the time in milliseconds. Returns whether the call to this method was successful or failed.

```
STDMETHOD_(HX_RESULT, GetMilliSeconds) (
   THIS_
   INT64* pnMilliSeconds
) const PURE;
```

pnMilliSeconds
: Pointer to the integer value of the time in milliseconds.

IHXTTime::GetTime

Gets the time in milliseconds. Returns whether the call to this method was successful or failed.

```
STDMETHOD_(HX_RESULT, GetTime) (
   THIS_
   double* pdMilliSeconds
) const PURE;
```

pdMilliSeconds
: Pointer to the double value of the time in milliseconds.

IHXTTime::GetTimeString

Returns a pointer to the time string in the format ddd:hh:mm:ss.ms.

```
STDMETHOD_(const char*, GetTimeString) (
   THIS_
   BOOL bTerse = FALSE
) PURE;
```

bTerse
: Determines how brief the time string is. If false (default), the time string provides a more precise display of the time. If true, the time sting provides less precision in the time displayed.

IHXTTime::SetMilliSeconds

Sets the time in milliseconds. Use this method if you are using int values. Returns whether the call to this method was successful or failed.

```
STDMETHOD_(HX_RESULT, SetMilliSeconds) (
   THIS_
   const INT64 nMilliSeconds
) PURE;
```

nMilliSeconds
: The time to be set, in milliseconds.

IHXTTime::SetTime

Sets the time in milliseconds. Use this method if you are using double values. Returns whether the call to this method was successful or failed.

```
STDMETHOD_(HX_RESULT, SetTime) (
   THIS_
   const double dMilliSeconds
) PURE;
```

dMilliSeconds
: The time to be set, in milliseconds.

IHXTTime::SetTimeString

Sets the time using a string. Returns whether the call to this method was successful or failed.

```
STDMETHOD_(HX_RESULT, SetTimeString) (
   THIS_
   const char* chTime
) PURE;
```

chTime
: Pointer to the time string in the format `ddd:hh:mm:ss.ms`.

IHXTTransformFilter

Purpose:	Provides transform filter operations on media samples.
Implemented by:	Plug-ins
Header file:	ihxtbase.h

A typical transform filter will receive an incoming media sample through IHXTTransformFilter::ReceiveSample, allocate a new sample using the allocator passed through IHXTTransformFilter::SetAllocator, and deliver the outgoing sample to the IHXTSampleSink interface specified by IHXTTransformFilter::SetSampleSink. This interface inherits methods from the IHXTFilter interface.

The IHXTTransformFilter interface contains the following unique methods:

- IHXTTransformFilter::ReceiveSample
- IHXTTransformFilter::SetAllocator
- IHXTTransformFilter::SetSampleSink

As with all Component Object Model (COM) interfaces, the IHXTTransformFilter interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTTransformFilter::ReceiveSample

Sends the filter the next media sample to process. A filter should only send a sample to the IHXTSampleSink interface within the scope of the call to this method (that is, if a filter processes samples on a separate thread, it must wait until the next IHXTTransformFilter::ReceiveSample call to propagate the sample).

```
STDMETHOD(ReceiveSample) (
   THIS_
   UINT32 uStreamID,
   IHXTMediaSample* pSample
) PURE;
```

uStreamID
The identity of the stream that contains the media sample.

pSample
Pointer to an IHXTMediaSample interface that manages the media sample to be processed.

IHXTTransformFilter::SetAllocator

Provides the filter with a media sample allocator. The allocator can be reset at any time. The filter is also free to ignore the allocator (for example, in a filter that supports in-place transforms). This method is always called before data flow calls are made.

```
STDMETHOD(SetAllocator) (
   THIS_
   UINT32 uStreamID,
   IHXTSampleAllocator* pAllocator
) PURE;
```

uStreamID
The identity of the stream for which the allocator will be provided.

pAllocator
Pointer to an `IHXTSampleAllocator` interface that manages the allocation of media sample data buffers.

IHXTTransformFilter::SetSampleSink

Sets the media sample destination.

```
STDMETHOD(SetSampleSink) (
   THIS_
   UINT32 uStreamID,
   IHXTSampleSink* pOutputSink
) PURE;
```

uStreamID
The identity of the stream that contains the media sample.

pOutputSink
Pointer to an `IHXTSampleSink` interface that manages the destination's media sample data buffer.

IHXTUintEnumerator

Purpose:	Enumerates the values of an unsigned integer list.
Implemented by:	All components
Header file:	`ihxtpropertybag.h`

Note: This interface does not need to be implemented by itself, but you can access this interface from a property bag.

The IHXTUintEnumerator interface contains the following methods:

- IHXTUintEnumerator::Current
- IHXTUintEnumerator::First
- IHXTUintEnumerator::GetCount
- IHXTUintEnumerator::Next

As with all Component Object Model (COM) interfaces, the IHXTUintEnumerator interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTUintEnumerator::Current

Gets the unsigned integer value at the current position of the internal iterator.

```
STDMETHOD(Current) (
   THIS_
   UINT32 *pValue
) PURE;
```

pValue
: Pointer to the value of the unsigned integer.

IHXTUintEnumerator::First

Gets the unsigned integer value at the first position in the list of unsigned integers.

```
STDMETHOD(First) (
   THIS_
   UINT32 *pValue
) PURE;
```

pValue
: Pointer to the value of the unsigned integer.

IHXTUintEnumerator::GetCount

Returns the total number of properties in the list of unsigned integers.

```
STDMETHOD_(UINT32, GetCount) (
   THIS
) const PURE;
```

IHXTUintEnumerator::Next

Advances the internal iterator and gets the unsigned integer value at the next position in the list of unsigned integers.

```
STDMETHOD(Next) (
   THIS_
   UINT32 *pValue
) PURE;
```

pValue
: Pointer to the value of the unsigned integer.

IHXTUintList

Purpose:	Stores a list of unsigned integer values.
Implemented by:	All components
Header file:	ihxtpropertybag.h

> **Note:** This interface does not need to be implemented by itself, but you can obtain an object that implements this interface from the class factory.

The IHXTUintList interface contains the following methods:

- IHXTUintList::Clear
- IHXTUintList::Compare
- IHXTUintList::Contains
- IHXTUintList::GetBack
- IHXTUintList::GetEnumerator
- IHXTUintList::GetFront
- IHXTUintList::GetIntersection
- IHXTUintList::GetSize
- IHXTUintList::IsEmpty
- IHXTUintList::PopBack
- IHXTUintList::PopFront
- IHXTUintList::PushBack
- IHXTUintList:PushFront

As with all Component Object Model (COM) interfaces, the IHXTUintList interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTUintList::Clear

Clears all properties from the current list.

```
STDMETHOD(Clear) (
   THIS
) PURE;
```

IHXTUintList::Compare

This method is obsolete and should not be used.

```
STDMETHOD(Compare) (
   THIS_
   IHXTUintList *pList,
   UINT32 *puValue
) const PURE;
```

IHXTUintList::Contains

Returns TRUE if the specified value is contained in the current list. Returns FALSE if the value is not in the current list.

```
STDMETHOD_(BOOL, Contains) (
   THIS_
   UINT32 value
) const PURE;
```

value
: The unsigned integer value being searched for in the current list.

IHXTUintList::GetBack

Returns the value of the property at the end of the current list.

```
STDMETHOD_(UINT32, GetBack) (
   THIS
) PURE;
```

IHXTUintList::GetEnumerator

Gets an enumerator that can be used to enumerate through all the items in the list.

```
STDMETHOD(GetEnumerator) (
   THIS_
   IHXTUintEnumerator **pEnumerator
) PURE;
```

pEnumerator
Address of a pointer to an IHXTUintEnumerator interface that manages the enumerator.

IHXTUintList::GetFront

Returns the value of the property at the beginning of the current list.

```
STDMETHOD_(UINT32, GetFront) (
   THIS
) PURE;
```

IHXTUintList::GetIntersection

Gets the intersection between the current list and another specified list.

```
STDMETHOD(GetIntersection) (
   THIS_
   IHXTUintList *pList,
   IHXTUintList **ppIntersection
) const PURE;
```

pList
Pointer to an IHXTUintList interface that manages the unsigned integer list to compare against the current unsigned integer list.

ppIntersection
Address of a pointer to an IHXTUintList interface that manages the new unsigned integer list. This new list contains only those properties that were the same in both of the compared lists.

IHXTUintList::GetSize

Returns the size of the current list.

```
STDMETHOD_(UINT32, GetSize) (
   THIS
) PURE;
```

IHXTUintList::IsEmpty

If TRUE, indicates the current list is empty. If FALSE, indicates there is at least one property in the current list.

```
STDMETHOD_(BOOL, IsEmpty) (
   THIS
) PURE
```

IHXTUintList::PopBack

Returns the property value at the end of the list, and removes the value from the list.

```
STDMETHOD_(UINT32, PopBack) (
   THIS
) PURE;
```

IHXTUintList::PopFront

Returns the property value at the beginning of the list, and removes the value from the list.

```
STDMETHOD_(UINT32, PopFront) (
   THIS
) PURE;
```

IHXTUintList::PushBack

Places a property value at the end of the list.

```
STDMETHOD(PushBack) (
   UINT32 value
) PURE;
```

value
The value of the property to add to the end of the list.

IHXTUintList:PushFront

Places a property value at the beginning of the list.

```
STDMETHOD(PushFront) (
   THIS_
   UINT32 value
) PURE;
```

value
The value of the property to add to the beginning of the list.

IHXTUintRange

Purpose:	Represents a range of unsigned integer values.
Implemented by:	All components
Header file:	ihxtpropertybag.h

> **Note:** This interface does not need to be implemented by itself, but you can obtain an object that implements this interface from the class factory.

The IHXTUintRange interface contains the following methods:

- IHXTUintRange::Compare
- IHXTUintRange::GetMax
- IHXTUintRange::GetMin
- IHXTUintRange::GetStepSize
- IHXTUintRange::IsInRange
- IHXTUintRange::Set

As with all Component Object Model (COM) interfaces, the IHXTUintRange interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTUintRange::Compare

This method is obsolete and should not be used.

```
STDMETHOD(Compare) (
   THIS_
   IHXTUintRange *pRange,
   IHXTUintRange **ppResult
) const PURE;
```

IHXTUintRange::GetMax

Returns the maximum value in the current range.

```
STDMETHOD_(UINT32, GetMax) (
   THIS
) const PURE;
```

IHXTUintRange::GetMin

Returns the minimum value in the current range.

```
STDMETHOD_(UINT32, GetMin) (
   THIS
) const PURE;
```

IHXTUintRange::GetStepSize

Returns the step size of the current range.

```
STDMETHOD_(UINT32, GetStepSize) (
   THIS
) const PURE;
```

IHXTUintRange::IsInRange

Returns `TRUE` if the specified value is in range or `FALSE` if the specified value is out of range.

```
STDMETHOD_(BOOL, IsInRange) (
   THIS_
   UINT32 uValue
) const PURE;
```

uValue
The value being tested to see if it is in range.

IHXTUintRange::Set

Sets the range parameters.

```
STDMETHOD(Set) (
   THIS_
   UINT32 uMin,
   UINT32 uMax,
   UINT32 uStepSize = 1
) PURE;
```

uMin
The minimum value to be set in the range.

uMax
The maximum value to be set in the range.

uStepSize
The step size of the values in the range.

IHXTUserConfigFile

Purpose:	Serializes objects.
Implemented by:	Encoding
Header file:	ihxtencodingjob.h

This interface is similar to IHXTSerializeBuffer. However, whereas the IHXTSerializeBuffer interface's behavior is strict serialization, this interface has some custom behavior.

The IHXTUserConfigFile interface contains the following methods:

- IHXTUserConfigFile::ReadFromFile
- IHXTUserConfigFile::WriteToFile

As with all Component Object Model (COM) interfaces, the IHXTUserConfigFile interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTUserConfigFile::ReadFromFile

Reads some XML code from either a job file (.rpjf) or an audience file (.rpad).

```
STDMETHOD(ReadFromFile) (
   THIS_
   const char* szPathname,
   BOOL bForceInitialization=FALSE,
   IHXTPropertyBag** ppInitErrorBag=NULL
) PURE;
```

szPathname
Pointer to the file from which to read.

bForceInitialization
: If set to false (default), this method will not create the object if any of the initialization properties fail to be validated. This ensures you never have an object in a bad state. However, this can have undesirable side effects, like causing a job file to fail to deserialize if the input filename does not exist, or if the specified capture device is in use. If this parameter is set to true, the object is forced to be created even with an invalid property. This allows the object to be created even though it cannot be part of an encode—the calling SDK application can then inspect the object and determine the problem that occurred during initialization.

ppInitErrorBag
: Address of a pointer to an IHXTPropertyBag interface that manages the invalid properties that caused initialization to fail.

IHXTUserConfigFile::WriteToFile

Writes XML code to either a job file (.rpjf) or an audience file (.rpad).

```
STDMETHOD(WriteToFile) (
   THIS_
   const char* szPathname,
   IHXTSerializationCallback* pSerialCallback=NULL
) PURE;
```

szPathname
: Pointer to the file to which data will be written.

pSerialCallback
: Pointer to an IHXTSerializationCallback method that manages the callback of the object being serialized. If this parameter is NULL, no callbacks will occur.

IHXTVideoPinFormat

Purpose:	Specifies the format of video samples that will be passed to the encoding engine.
Implemented by:	Encoding
Header file:	ihxtbase.h

The IHXTVideoPinFormat interface contains the following methods:

- IHXTVideoPinFormat::GetColorFormat
- IHXTVideoPinFormat::GetFrameDimensions

- IHXTVideoPinFormat::GetFrameRate
- IHXTVideoPinFormat::SetColorFormat
- IHXTVideoPinFormat::SetFrameDimensions
- IHXTVideoPinFormat::SetFrameRate

As with all Component Object Model (COM) interfaces, the IHXTVideoPinFormat interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXTVideoPinFormat::GetColorFormat

Gets the video color format of video samples passed to the encoding engine.

```
STDMETHOD(GetColorFormat) (
   THIS_
   EHXTVideoColorFormat* peVideoFormat
) PURE;
```

peVideoFormat

Pointer to the specific color format of the video sample. One of the following:

- HXT_VIDEO_FORMAT_BGR24_INVERTED—default 24-bit format on Windows platforms
- HXT_VIDEO_FORMAT_BGR24_NONINVERTED
- HXT_VIDEO_FORMAT_BGRA32_INVERTED—default-32 bit format on Windows platforms
- HXT_VIDEO_FORMAT_BGRA32_NONINVERTED
- HXT_VIDEO_FORMAT_LE_ARGB555_INVERTED—default 16-bit format on Windows platforms
- HXT_VIDEO_FORMAT_LE_ARGB555_NONINVERTED
- HXT_VIDEO_FORMAT_LE_RGB565_INVERTED—default 16-bit format on Windows platforms
- HXT_VIDEO_FORMAT_LE_RGB565_NONINVERTED
- HXT_VIDEO_FORMAT_RGB24_INVERTED
- HXT_VIDEO_FORMAT_RGB24_NONINVERTED—default 24-bit format on a MacIntosh

- HXT_VIDEO_FORMAT_ARGB32_INVERTED
- HXT_VIDEO_FORMAT_ARGB32_NONINVERTED—default 32-bit format on a MacIntosh
- HXT_VIDEO_FORMAT_RGBA32_INVERTED
- HXT_VIDEO_FORMAT_RGBA32_NONINVERTED
- HXT_VIDEO_FORMAT_ABGR32_INVERTED
- HXT_VIDEO_FORMAT_ABGR32_NONINVERTED
- HXT_VIDEO_FORMAT_BE_ARGB555_INVERTED
- HXT_VIDEO_FORMAT_BE_ARGB555_NONINVERTED—default 16-bit format on a MacIntosh
- HXT_VIDEO_FORMAT_BE_RGB565_INVERTED
- HXT_VIDEO_FORMAT_BE_RGB565_NONINVERTED
- HXT_VIDEO_FORMAT_LE_RGBA555_INVERTED
- HXT_VIDEO_FORMAT_LE_RGBA555_NONINVERTED
- HXT_VIDEO_FORMAT_BGR8_INVERTED
- HXT_VIDEO_FORMAT_BGR8_NONINVERTED

Planar YUV formats

- HXT_VIDEO_FORMAT_I420—the codec's format; planar 4:2:0 (covers fourCC 'IYUV' as well)
- HXT_VIDEO_FORMAT_YV12—same as I420, but with UV planes swapped
- HXT_VIDEO_FORMAT_YVU9—planar 16:2:0 format
- HXT_VIDEO_FORMAT_IF09—same as YVU9 but an additional 4x4 subsampled plane is appended containing delta information relative to the last frame

Packed YUV formats

- HXT_VIDEO_FORMAT_YUY2—packed 4:2:2 format
- HXT_VIDEO_FORMAT_YUY2_INVERTED—packed 4:2:2 format, inverted (Winnov Videum)
- HXT_VIDEO_FORMAT_UYVY—packed 4:2:2 format, different ordering

- HXT_VIDEO_FORMAT_IUYV
- HXT_VIDEO_FORMAT_IY41
- HXT_VIDEO_FORMAT_IYU1
- HXT_VIDEO_FORMAT_IYU2
- HXT_VIDEO_FORMAT_CYUV
- HXT_VIDEO_FORMAT_YVYU
- HXT_VIDEO_FORMAT_Y211
- HXT_VIDEO_FORMAT_Y41T
- HXT_VIDEO_FORMAT_Y42T
- HXT_VIDEO_FORMAT_CLJR

MacIntosh-specific YUVs and Y'CbCr

- HXT_VIDEO_FORMAT_YUV2
- HXT_VIDEO_FORMAT_V308
- HXT_VIDEO_FORMAT_V408
- HXT_VIDEO_FORMAT_V216
- HXT_VIDEO_FORMAT_V410
- HXT_NUM_VIDEO_FORMATS

IHXTVideoPinFormat::GetFrameDimensions

Gets the frame dimensions of video samples passed to the encoding engine.

```
STDMETHOD(GetFrameDimensions) (
   THIS_
   UINT32* pulWidth,
   UINT32* pulHeight
) PURE;
```

pulWidth
Pointer to the width of the frame, in pixels.

pulHeight
Pointer to the height of the frame, in pixels.

IHXTVideoPinFormat::GetFrameRate

Gets the frame rate of video samples passed to the encoding engine.

```
STDMETHOD(GetFrameRate) (
   THIS_
   double* pdFrameRate
) PURE;
```

pdFrameRate
: The frame rate of the video sample, in frames per second.

IHXTVideoPinFormat::SetColorFormat

Sets the video color format of video samples passed to the encoding engine.

```
STDMETHOD(SetColorFormat) (
   THIS_
   EHXTVideoColorFormat eVideoFormat
) PURE;
```

eVideoFormat
: The specific color format of the video sample. One of the following:

- HXT_VIDEO_FORMAT_BGR24_INVERTED—default 24-bit format on Windows platforms
- HXT_VIDEO_FORMAT_BGR24_NONINVERTED
- HXT_VIDEO_FORMAT_BGRA32_INVERTED—default-32 bit format on Windows platforms
- HXT_VIDEO_FORMAT_BGRA32_NONINVERTED
- HXT_VIDEO_FORMAT_LE_ARGB555_INVERTED—default 16-bit format on Windows platforms
- HXT_VIDEO_FORMAT_LE_ARGB555_NONINVERTED
- HXT_VIDEO_FORMAT_LE_RGB565_INVERTED—default 16-bit format on Windows platforms
- HXT_VIDEO_FORMAT_LE_RGB565_NONINVERTED
- HXT_VIDEO_FORMAT_RGB24_INVERTED
- HXT_VIDEO_FORMAT_RGB24_NONINVERTED—default 24-bit format on a MacIntosh
- HXT_VIDEO_FORMAT_ARGB32_INVERTED
- HXT_VIDEO_FORMAT_ARGB32_NONINVERTED—default 32-bit format on a MacIntosh
- HXT_VIDEO_FORMAT_RGBA32_INVERTED

- HXT_VIDEO_FORMAT_RGBA32_NONINVERTED
- HXT_VIDEO_FORMAT_ABGR32_INVERTED
- HXT_VIDEO_FORMAT_ABGR32_NONINVERTED
- HXT_VIDEO_FORMAT_BE_ARGB555_INVERTED
- HXT_VIDEO_FORMAT_BE_ARGB555_NONINVERTED—default 16-bit format on a MacIntosh
- HXT_VIDEO_FORMAT_BE_RGB565_INVERTED
- HXT_VIDEO_FORMAT_BE_RGB565_NONINVERTED
- HXT_VIDEO_FORMAT_LE_RGBA555_INVERTED
- HXT_VIDEO_FORMAT_LE_RGBA555_NONINVERTED
- HXT_VIDEO_FORMAT_BGR8_INVERTED
- HXT_VIDEO_FORMAT_BGR8_NONINVERTED

Planar YUV formats

- HXT_VIDEO_FORMAT_I420—the codec's format; planar 4:2:0 (covers fourCC 'IYUV' as well)
- HXT_VIDEO_FORMAT_YV12—same as I420, but with UV planes swapped
- HXT_VIDEO_FORMAT_YVU9—planar 16:2:0 format
- HXT_VIDEO_FORMAT_IF09—same as YVU9 but an additional 4x4 subsampled plane is appended containing delta information relative to the last frame

Packed YUV formats

- HXT_VIDEO_FORMAT_YUY2—packed 4:2:2 format
- HXT_VIDEO_FORMAT_YUY2_INVERTED—packed 4:2:2 format, inverted (Winnov Videum)
- HXT_VIDEO_FORMAT_UYVY—packed 4:2:2 format, different ordering
- HXT_VIDEO_FORMAT_IUYV
- HXT_VIDEO_FORMAT_IY41
- HXT_VIDEO_FORMAT_IYU1
- HXT_VIDEO_FORMAT_IYU2

- HXT_VIDEO_FORMAT_CYUV
- HXT_VIDEO_FORMAT_YVYU
- HXT_VIDEO_FORMAT_Y211
- HXT_VIDEO_FORMAT_Y41T
- HXT_VIDEO_FORMAT_Y42T
- HXT_VIDEO_FORMAT_CLJR

MacIntosh-specific YUVs and Y'CbCr

- HXT_VIDEO_FORMAT_YUV2
- HXT_VIDEO_FORMAT_V308
- HXT_VIDEO_FORMAT_V408
- HXT_VIDEO_FORMAT_V216
- HXT_VIDEO_FORMAT_V410
- HXT_NUM_VIDEO_FORMATS

IHXTVideoPinFormat::SetFrameDimensions

Sets the frame dimensions of video samples passed to the encoding engine.

```
STDMETHOD(SetFrameDimensions) (
   THIS_
   UINT32 ulWidth,
   UINT32 ulHeight
) PURE;
```

ulWidth
The width of the video frame, in pixels.

ulHeight
The height of the video frame, in pixels.

IHXTVideoPinFormat::SetFrameRate

Sets the frame rate of video samples passed to the encoding engine.

```
STDMETHOD(SetFrameRate) (
   THIS_
   double dFrameRate
) PURE;
```

dFrameRate
The frame rate of the video sample, in frames per second.

IUnknown

The IUnknown interface is the basis of all Component Object Model (COM) interfaces. This interface contains a set of methods that control the lifetime of a specific object, and provides a means of querying for the interfaces used by an object. The IUnknown interface includes the following methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IUnknown::AddRef

Increases the object's reference count by one. Whenever an object is created, its reference count begins at 1. If an application calls IUnknown::AddRef, queries an interface belonging to a specific object, or uses a creation function like HXTCreateJobFactory, the reference count is incremented by 1.

```
STDMETHOD_(ULONG32,AddRef) (
   THIS
) PURE;
```

Return Values
Returns the new reference count.

> **Note:** Use the IUnknown::Release method to decrease the reference count by 1.

IUnknown::QueryInterface

Queries an object to determine whether it supports a specific interface. If the call succeeds, you can then use the methods belonging to that interface.

```
STDMETHOD(QueryInterface) (
   THIS_
   REFIID riid,
   void** ppvObj
) PURE;
```

riid
Indicates the reference identifier of the interface being queried.

ppvObj
: Points to an interface pointer that is filled in if the query succeeds.

Return Values

Returns HXR_OK if successful, or one of the following values:

- HXR_FAIL
- HXR_NOINTERFACE
- HXR_NOTIMPL
- HXR_OUTOFMEMORY

IUnknown::Release

Decreases the object's reference count by one. Every call to IUnknown::AddRef IUnknown::QueryInterface, or a creation function such as HXTCreateJobFactory must have a corresponding call to IUnknown::Release. When the reference count reaches 0 (zero), the object is destroyed.

```
STDMETHOD_(ULONG32,Release) (
   THIS
) PURE;
```

Return Values

Returns the new reference count.

APPENDIX

B

REALMEDIA EDIT INTERFACE LIST

IHXProgressSink

Purpose:	Supports callback notification about a job's progress.
Implemented by:	Helix DNA Producer
Header file:	progsink.h

This interface supplies callbacks to your application indicating the status and progress of a particular job. You can receive these callbacks for the object that performs editing of .rm files, and the object that encodes events and image maps into a .rm file. If the object performs editing of .rm file, create an instance of the IHXRMEdit3 interface, then use the IHXRMEdit3::AddSaveProgressSink method to create and enable the progress sink. If the object encodes events and image maps into a .rm file, create an instance of the IHXRMEvents2 interface, then use the IHXRMEvents2::AddSaveProgressSink method to create and enable the progress sink. Once you have finished the operation, use either the IHXRMEdit3::RemoveSaveProgressSink or the IHXRMEvents2::RemoveSaveProgressSink methods to remove the progress sink.

The IHXProgressSink interface contains the following methods:

- IHXProgressSink::NotifyFinish
- IHXProgressSink::NotifyStart
- IHXProgressSink::SetProgress

As with all COM interfaces, the IHXProgressSink interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXProgressSink::NotifyFinish

Notifies the caller that the job is complete.

```
STDMETHOD(NotifyFinish) (
   THIS
) PURE;
```

IHXProgressSink::NotifyStart

Notifies the caller that the job is starting.

```
STDMETHOD(NotifyStart) (
   THIS
) PURE;
```

IHXProgressSink::SetProgress

Sets the percentage of the merge that is complete.

```
STDMETHOD(SetProgress) (
   THIS_
   UINT32 ulPercentComplete
) PURE;
```

ulPercentComplete
The amount of the merge that has completed so far, in percent.

IHXProgressSinkControl

Purpose:	Adds and removes the progress sink interface.
Implemented by:	None
Header file:	progsink.h

This interface is obsolete and should not be used.

IHXRMEdit

Purpose:	Edits a .rm file.
Implemented by:	RealMedia file editors
Header file:	ihxtedit.h

The IHXRMEdit interface contains the following methods:

- IHXRMEdit::AddInputFile
- IHXRMEdit::CloseLogFile
- IHXRMEdit::CreateIRMABuffer

- IHXRMEdit::GetAuthor
- IHXRMEdit::GetComment
- IHXRMEdit::GetCopyright
- IHXRMEdit::GetEndTime
- IHXRMEdit::GetErrorString
- IHXRMEdit::GetFileVersion
- IHXRMEdit::GetIndexedInputFile
- IHXRMEdit::GetMobilePlayback
- IHXRMEdit::GetNumInputFiles
- IHXRMEdit::GetOutputFile
- IHXRMEdit::GetPerfectPlay
- IHXRMEdit::GetSelectiveRecord
- IHXRMEdit::GetStartTime
- IHXRMEdit::GetTitle
- IHXRMEdit::Log
- IHXRMEdit::OpenLogFile
- IHXRMEdit::Process
- IHXRMEdit::RemoveRMFileSink
- IHXRMEdit::SetAuthor
- IHXRMEdit::SetComment
- IHXRMEdit::SetCopyright
- IHXRMEdit::SetEndTime (in milliseconds)
- IHXRMEdit::SetEndTime (in days: hours: minutes: seconds: milliseconds)
- IHXRMEdit::SetInputFile
- IHXRMEdit::SetMobilePlayback
- IHXRMEdit::SetOutputFile
- IHXRMEdit::SetPerfectPlay
- IHXRMEdit::SetRMFileSink
- IHXRMEdit::SetSelectiveRecord
- IHXRMEdit::SetStartTime (in milliseconds)
- IHXRMEdit::SetStartTime (in days: hours: minutes: seconds: milliseconds)
- IHXRMEdit::SetTitle

As with all COM interfaces, the IHXRMEdit interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXRMEdit::AddInputFile

Indicates the file name of a .rm file to paste to the end of the input file specified in IHXRMEdit::SetInputFile. Call IHXRMEdit::SetInputFile before using this method.

```
STDMETHOD(AddInputFile) (
   THIS_
   const char szFileName
) PURE;
```

szFileName
The path to the .rm file to be pasted to the end of the file that was designated by IHXRMEdit::SetInputFile.

IHXRMEdit::CloseLogFile

Closes the log file opened by IHXRMEdit::OpenLogFile.

```
STDMETHOD(CloseLogFile) (
   THIS
) PURE;
```

IHXRMEdit::CreateIRMABuffer

Creates an instance of an IHXBuffer interface.

```
STDMETHOD(CreateIRMABuffer) (
   THIS_
   IHXBuffer** pBuffer
) PURE;
```

pBuffer
Returns a pointer to the IHXBuffer interface.

IHXRMEdit::GetAuthor

Returns the current author string.

```
STDMETHOD(GetAuthor) (
   THIS_
   char* szAuthor,
   UINT32 ulSize
) PURE;
```

szAuthor
: Pointer to the author string. The size of the buffer for the author string must be preallocated by the caller in `ulSize`.

ulSize
: Indicates the size of the author string to be returned in `szAuthor`.

IHXRMEdit::GetComment

Returns the current comment string.

```
STDMETHOD(GetComment) (
THIS_
   char* szComment,
   UINT32 ulSize
) PURE;
```

szComment
: Pointer to the comment string. The size of the buffer for the comment string must be preallocated by the caller in `ulSize`.

ulSize
: Indicates the size of the comment string to be returned in `szComment`.

IHXRMEdit::GetCopyright

Returns the current copyright string.

```
STDMETHOD(GetCopyright) (
   THIS_
   char* szCopyright,
   UINT32 ulSize
) PURE;
```

szCopyright
: Pointer to the copyright string. The size of the buffer for the copyright string must be preallocated by the caller in `ulSize`.

ulSize
: Indicates the size of the copyright string to be returned in `szCopyright`.

IHXRMEdit::GetEndTime

Returns the current end time in milliseconds.

```
STDMETHOD(GetEndTime) (
   THIS_
    UINT32* pulEndTime
) PURE;
```

pulEndTime
Pointer to the current end time.

IHXRMEdit::GetErrorString

Returns the error string associated with the specified result value.

```
STDMETHOD(GetErrorString) (
   THIS_
   HX_RESULT res,
   char* szErrString,
   UINT16 unMaxSize
) PURE;
```

res
The HX_RESULT value for which you want an error string.

szErrString
Pointer to the error string. The size of the buffer for the error string must be preallocated by the caller in ulMaxSize.

unMaxSize
Indicates the size of the result string returned in szErrString.

IHXRMEdit::GetFileVersion

Returns the version number of the input file.

```
STDMETHOD(GetFileVersion) (
   THIS_
   UINT32* pulVersion
) PURE;
```

pulVersion
Pointer to the version information. If the value of this parameter is 1, the file is a .rm1 file (Single Rate). If the value of this parameter is 2, the file is a .rm2 file (Sure Stream).

IHXRMEdit::GetIndexedInputFile

Returns the file name of the input file specified by index. Use IHXRMEdit::GetNumInputFiles to determine how many files have been added to the IHXRMEditor interface.

```
STDMETHOD(GetIndexedInputFile) (
   THIS_
   UINT32 index,
   char* szFileName,
   UINT32 ulMaxBufSize
) PURE;
```

index
: The index of the required input file. This parameter must be in the range of 0 to pulNumInputFiles - 1. (The value of pulNumInputFiles is returned in IHXRMEdit::GetNumInputFiles.)

szFileName
: Pointer to the path to the input file. The size of the buffer for the path string must be preallocated by the caller in ulMaxBufSize.

ulMaxBufSize
: The size of the path string returned in szFileName.

IHXRMEdit::GetMobilePlayback

Returns the current state of the Mobile Playback (Allow Download) flag.

```
STDMETHOD(GetMobilePlayback) (
   THIS_
   BOOL* bEnabled
) PURE;
```

bEnabled
: Indicates the state of the mobile playback flag.

IHXRMEdit::GetNumInputFiles

Returns the number of input files that have be added using the IHXRMEdit::SetInputFile and IHXRMEdit::AddInputFile methods.

```
STDMETHOD(GetNumInputFiles) (
   THIS_
   UINT32* pulNumInputFiles
) PURE;
```

pulNumInputFiles
: Pointer to the current number of input files.

IHXRMEdit::GetOutputFile

Returns the file name of the output file.

```
STDMETHOD(GetOutputFile) (
   THIS_
   char* szFileName,
   UINT32 ulMaxBufSize
) PURE;
```

szFileName
: Pointer to the output file name. The size of the buffer for the file name string must be preallocated by the caller in `ulMaxBufSize`.

ulMaxBufSize
: The size of the path string returned in `szFileName`.

IHXRMEdit::GetPerfectPlay

Returns the current state of the PerfectPlay (Buffered Playback) flag.

```
STDMETHOD(GetPerfectPlay) (
   THIS_
   BOOL* bEnabled
) PURE;
```

bEnabled
: Indicates the state of the PerfectPlay flag.

IHXRMEdit::GetSelectiveRecord

Returns the current state of the Selective Record (Allow Recording) flag.

```
STDMETHOD(GetSelectiveRecord) (
   THIS_
   BOOL* bEnabled
) PURE;
```

bEnabled
: Indicates the state of the selective record flag.

IHXRMEdit::GetStartTime

Returns the current start time in milliseconds.

```
STDMETHOD(GetStartTime) (
   THIS_
   UINT32* pulStartTime
) PURE;
```

pulStartTime
: Pointer to the current start time.

IHXRMEdit::GetTitle

Returns the current title string.

```
STDMETHOD(GetTitle) (
   THIS_
   char* szTitle,
   UINT32 ulSize
) PURE;
```

szTitle
: Pointer to the current title string. The size of the buffer for the title string must be preallocated by the caller in `ulSize`.

ulSize
: The size of the title string returned in `szTitle`.

IHXRMEdit::Log

Logs the string to the log file.

```
STDMETHOD(Log) (
   THIS_
   const char* pLogString
) PURE;
```

pLogString
: Pointer to the string to be logged to the log file.

IHXRMEdit::OpenLogFile

Opens the specified file for logging. All status and error messages are logged to this file.

```
STDMETHOD(OpenLogFile) (
   THIS_
   const char* pFileName
) PURE;
```

pFileName
Pointer to the file name of the log file.

IHXRMEdit::Process

Processes the edit using the current settings. Creates and writes to the output file.

```
STDMETHOD(Process) (
   THIS
) PURE;
```

IHXRMEdit::RemoveRMFileSink

Removes the specified IHXRMFileSink interface from the IHXRMEditor interface.

```
STDMETHOD(RemoveRMFileSink) (
   THIS_
   IHXRMFileSink* pRMFileSink
) PURE;
```

pRMFileSink
Pointer to the IHXRMFileSink interface to be removed.

IHXRMEdit::SetAuthor

Sets the author string.

```
STDMETHOD(SetAuthor) (
   THIS_
   const char* szAuthor
) PURE;
```

szAuthor
Pointer to the author string for the file.

IHXRMEdit::SetComment

Sets the comment string.

```
STDMETHOD(SetComment) (
   THIS_
   const char* szComment
) PURE;
```

szComment
Pointer to the comment string for the file.

IHXRMEdit::SetCopyright

Sets the copyright string.

```
STDMETHOD(SetCopyright) (
   THIS_
   const char* szCopyright
) PURE;
```

szCopyright
: Pointer to the copyright string for the file.

IHXRMEdit::SetEndTime

Specifes the end time for the edit operation in milliseconds.

```
STDMETHOD(SetEndTime) (
   THIS_
   UINT32 ulEndTime
) PURE;
```

ulEndTime
: The end time, in milliseconds.

> **Note:** If you do not call this method, the default end time will be the end of the file (EOF).

IHXRMEdit::SetEndTime

Specifes the end time in days:hours:minutes:seconds:milliseconds format for the edit operation.

```
STDMETHOD(SetEndTime) (
   THIS_
   const char* szEndTime
) PURE;
```

szEndTime
: Pointer to the end time, in days:hours:minutes:seconds:milliseconds format (0:0:0:0:0).

> **Note:** If you do not call this method, the default end time will be the end of the file (EOF).

IHXRMEdit::SetInputFile

Specifies the file name of the input .rm file. If you are pasting several .rm files, call this method with the name of the first file, then IHXRMEdit::AddInputFile for the remaining files.

```
STDMETHOD(SetInputFile) (
   THIS_
   const char* szFileName,
   BOOL bLoadFileInfo
) PURE;
```

szFileName
: Pointer to the file name of the input file.

bLoadFileInfo
: Indicates whether you want the RealMedia Edit API to load the input file's content information and property flags. If this parameter is set to TRUE, the input file's content information (Title, Author, Copyright, Comment) and property flags (Selective Record, Mobile Play, and so on) are loaded. You can then access this information using the Get methods (that is, IHXRMEdit::GetTitle, IHXRMEdit::GetAuthor, and so on).

IHXRMEdit::SetMobilePlayback

Enables or disables the Mobile Playback (Allow Download) flag.

```
STDMETHOD(SetMobilePlayback) (
   THIS_
   BOOL bEnable
) PURE;
```

bEnable
: If this parameter is set to TRUE, Allow Download is enabled. If this parameter is set to FALSE, Allow Download is disabled.

IHXRMEdit::SetOutputFile

Specifies the file name of the output file. This .rm file contains the results of the edit operation.

```
STDMETHOD(SetOutputFile) (
   THIS_
   const char* szFileName
) PURE;
```

szFileName
: Pointer to the output .rm file. If the file already exists, it will be replaced. If the file does not exist it will be created.

IHXRMEdit::SetPerfectPlay

Enables or disables the Perfect Play (Buffered Playback) flag.

```
STDMETHOD(SetPerfectPlay) (
   THIS_
   BOOL bEnable
) PURE;
```

bEnable
: If this parameter is set to TRUE, Buffered Playback is enabled. If this parameter is set to FALSE, Buffered Playback is disabled.

IHXRMEdit::SetRMFileSink

Adds an IHXRMFileSink interface to the IHXRMEditor interface. The IHXRMFileSink interface is then notified whenever a media properties header or data packet is written to the output file.

```
STDMETHOD(SetRMFileSink) (
   THIS_
   IHXRMFileSink* pRMFileSink
) PURE;
```

pRMFileSink
: Pointer to the IHXRMFileSink interface that was added.

IHXRMEdit::SetSelectiveRecord

Enables or disables the Selective Record (Allow Recording) flag.

```
STDMETHOD(SetSelectiveRecord) (
   THIS_
   BOOL bEnable
) PURE;
```

bEnable
: If this parameter is set to TRUE, Allow Recording is enabled. If this parameter is set to FALSE, Allow Recording is disabled.

IHXRMEdit::SetStartTime

Specifes the start time for the edit operation in milliseconds.

```
STDMETHOD(SetStartTime) (
   THIS_
   UINT32 ulStartTime
) PURE;
```

ulStartTime
The start time, in milliseconds.

> **Note:** If you do not call this method, the default start time will be start of the file.

IHXRMEdit::SetStartTime

Specifes the start time for the edit operation in days:hours:minutes:seconds:milliseconds format.

```
STDMETHOD(SetStartTime) (
   THIS_
   const char* szStartTime
) PURE;
```

szStartTime
Pointer to the start time in days:hours:minutes:seconds:milliseconds format (0:0:0:0:0).

> **Note:** If you do not call this method, the default start time will be start of the file.

IHXRMEdit::SetTitle

Sets the title string.

```
STDMETHOD(SetTitle) (
   THIS_
   const char* szTitle
) PURE;
```

szTitle
Pointer to the title string for the file.

IHXRMEdit2

Purpose:	Provides additional information about a .rm file.
Implemented by:	RealMedia file editors
Header file:	ihxtedit2.h

The IHXRMEdit2 interface contains the following methods:

- IHXRMEdit2::GetMetaInformation
- IHXRMEdit2::GetVideoSize
- IHXRMEdit2::HasAudio
- IHXRMEdit2::HasEvents
- IHXRMEdit2::HasImageMaps
- IHXRMEdit2::HasVideo

As with all COM interfaces, the IHXRMEdit2 interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXRMEdit2::GetMetaInformation

Gets the meta information currently stored in the active input file. This method returns a pointer to the IHXValues interface that manages all of the properties. You can then change the meta information fields in the returned IHXValues interface.

```
STDMETHOD(GetMetaInformation) (
   THIS_
   IHXValues** ppValues
) PURE;
```

ppValues
: Address of a pointer to the IHXValues interface that manages the current meta information.

IHXRMEdit2::GetVideoSize

Returns the height and width of the video in pixels.

```
STDMETHOD(GetVideoSize) (
   THIS_
   UINT16* pHeight,
   UINT16* pWidth
) PURE;
```

pHeight
: Pointer to the height of the video.

pWidth
: Pointer to the width of the video.

IHXRMEdit2::HasAudio

Indicates whether the current .rm file contains audio.

```
STDMETHOD(HasAudio) (
   THIS_
   BOOL* pbHasAudio
) PURE;
```

pbHasAudio
: If this parameter is set to TRUE, the file contains audio.

IHXRMEdit2::HasEvents

Indicates whether the current .rm file contains events.

```
STDMETHOD(HasEvents) (
   THIS_
   BOOL* pbHasEvents
) PURE;
```

pbHasEvents
: If this parameter is set to TRUE, the file contains events.

IHXRMEdit2::HasImageMaps

Indicates whether the current .rm file contains image maps.

```
STDMETHOD(HasImageMaps) (
   THIS_
   BOOL* pbHasImageMaps
) PURE;
```

pbHasImageMaps
: If this parameter is set to TRUE, the file contains image maps.

IHXRMEdit2::HasVideo

Indicates whether the current .rm file contains video.

```
STDMETHOD(HasVideo) (
   THIS_
   BOOL* pbHasVideo
) PURE;
```

pbHasVideo
If this parameter is set to TRUE, the file contains video.

IHXRMEdit3

Purpose:	Adds and removes a progress sink for the editing session.
Implemented by:	RealMedia file editors
Header file:	ihxtedit.h

This interface is queried from the IHXRMEdit interface. You then call the IHXRMEdit3::AddSaveProgressSink method to turn on status and progress callbacks from the system. These callbacks include notifying the caller that the processing of events has started or stopped, and a continuous callback that provides the percent complete. Once your application returns from the IHXRMEdit::Process call, use the IHXRMEdit3::RemoveSaveProgressSink method to remove the progress sink if there are no other editing sessions to process.

The IHXRMEdit3 interface contains the following methods:

- IHXRMEdit3::AddSaveProgressSink
- IHXRMEdit3::RemoveSaveProgressSink

As with all COM interfaces, the IHXRMEdit3 interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXRMEdit3::AddSaveProgressSink

Adds a sink interface. This sink interface receives callbacks that indicate the progress of the save operation, in percent.

```
STDMETHOD(AddSaveProgressSink) (
   IHXProgressSink* pProgressSink
) PURE;
```

pProgressSink
Pointer to an IHXProgressSink interface that manages the callback notification about a job's progress.

IHXRMEdit3::RemoveSaveProgressSink

Removes the progress-during-save sink interface.

```
STDMETHOD(RemoveSaveProgressSink) (
   IHXProgressSink* pProgressSink
) PURE;
```

pProgressSink
Pointer to the IHXProgressSink interface to be removed.

IHXRMEvents

Purpose:	Modifies events and image maps in a .rm file.
Implemented by:	RealMedia file editors
Header file:	ihxtevnts.h

The IHXRMEvents interface contains the following methods:

- IHXRMEvents::CloseLogFile
- IHXRMEvents::GetDumpFile
- IHXRMEvents::GetErrorString
- IHXRMEvents::GetEventFile
- IHXRMEvents::GetImageMapFile
- IHXRMEvents::GetInputFile
- IHXRMEvents::GetOutputFile
- IHXRMEvents::Log
- IHXRMEvents::OpenLogFile
- IHXRMEvents::Process
- IHXRMEvents::SetDumpFile
- IHXRMEvents::SetEventFile
- IHXRMEvents::SetImageMapFile
- IHXRMEvents::SetInputFile

- IHXRMEvents::SetOutputFile

As with all COM interfaces, the IHXRMEvents interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXRMEvents::CloseLogFile

Closes the log file.

```
STDMETHOD(CloseLogFile) (
   THIS
) PURE;
```

IHXRMEvents::GetDumpFile

Returns the name of the dump file root name.

```
STDMETHOD(GetDumpFile) (
   THIS_
   char* szFileName,
   UINT32 ulMaxBufSize
) PURE;
```

szFileName
Pointer to the dump file root name. The size of the buffer for the file name string must be preallocated by the caller in ulMaxBufSize.

ulMaxBufSize
Indicates the size of the file name string to be returned in szFileName.

IHXRMEvents::GetErrorString

Returns the error string associated with the specified result value.

```
STDMETHOD(GetErrorString) (
   THIS_
   HX_RESULT res,
   char* szErrString,
   UINT16 unMaxSize
) PURE;
```

res
The HX_RESULT value for which you want an error string.

szErrString
: Pointer to the error string. The size of the buffer for the error string must be preallocated by the caller in `unMaxSize`.

unMaxSize
: Indicates the size of the result string returned in `szErrString`.

IHXRMEvents::GetEventFile

Returns the file name of the event text file.

```
STDMETHOD(GetEventFile) (
   THIS_
   char* szFileName,
   UINT32 ulMaxBufSize
) PURE;
```

szFileName
: Pointer to the event text file name. The size of the buffer for the file name string must be preallocated by the caller in `ulMaxBufSize`.

ulMaxBufSize
: Indicates the size of the file name string to be returned in `szFileName`.

IHXRMEvents::GetImageMapFile

Returns the file name of the image map text file.

```
STDMETHOD(GetImageMapFile) (
   THIS_
   char* szFileName,
   UINT32 ulMaxBufSize
) PURE;
```

szFileName
: Pointer to the image map text file name. The size of the buffer for the file name string must be preallocated by the caller in `ulMaxBufSize`.

ulMaxBufSize
: Indicates the size of the file name string to be returned in `szFileName`.

IHXRMEvents::GetInputFile

Returns the file name of the input file.

```
STDMETHOD(GetInputFile) (
   THIS_
   char* szFileName,
   UINT32 ulMaxBufSize
) PURE;
```

szFileName
: Pointer to the input file name. The size of the buffer for the file name string must be preallocated by the caller in `ulMaxBufSize`.

ulMaxBufSize
: Indicates the size of the file name string to be returned in `szFileName`.

IHXRMEvents::GetOutputFile

Returns the file name of the output file.

```
STDMETHOD(GetOutputFile) (
   THIS_
   char* szFileName,
   UINT32 ulMaxBufSize
) PURE;
```

szFileName
: Pointer to the output file name. The size of the buffer for the file name string must be preallocated by the caller in `ulMaxBufSize`.

ulMaxBufSize
: Indicates the size of the file name string to be returned in `szFileName`.

IHXRMEvents::Log

Logs the specified string to the log file.

```
STDMETHOD(Log) (
   THIS_
   const char* pLogString
) PURE;
```

pLogString
: Pointer to the string to be logged to the log file.

IHXRMEvents::OpenLogFile

Opens the specified file for logging. All status and error messages are then logged to this file.

```
STDMETHOD(OpenLogFile) (
   THIS_
   const char* pFileName
) PURE;
```

pFileName
: Pointer to the file name of the log file.

IHXRMEvents::Process

Merges the events and image maps with the input file. Creates and writes the output file.

```
STDMETHOD(Process) (
   THIS
) PURE;
```

IHXRMEvents::SetDumpFile

Specifies the root file name of the dump file. All events in the input file are dumped into *szFileName*_evt.txt. All image maps in the input file are dumped into *szFileName*_imap.txt.

```
STDMETHOD(SetDumpFile) (
   THIS_
    const char* szFileName
) PURE;
```

szFileName
: Pointer to the root file name of the dump file.

IHXRMEvents::SetEventFile

Specifies the file name of the event text file.

```
STDMETHOD(SetEventFile) (
   THIS_
   const char* szFileName
) PURE;
```

szFileName
: Pointer to the file name of the event text file.

IHXRMEvents::SetImageMapFile

Specifies the file name of the image map text file.

```
STDMETHOD(SetImageMapFile) (
   THIS_
   const char* szFileName
) PURE;
```

szFileName
: Pointer to the file name of the image map text file.

IHXRMEvents::SetInputFile

Specifies the file name of the input .rm file.

```
STDMETHOD(SetInputFile) (
   THIS_
   const char* szFileName
) PURE;
```

szFileName
: Pointer to the file name of the input file.

IHXRMEvents::SetOutputFile

Specifies the file name of the output file. This .rm file contains the results of the merge operation.

```
STDMETHOD(SetOutputFile) (
   THIS_
   const char* szFileName
) PURE;
```

szFileName
: Pointer to the file name of the output .rm file. If the file already exists, it will be replaced. If the file does not exist, it will be created.

IHXRMEvents2

Purpose:	Adds and removes a progress sink during processing of events.
Implemented by:	RealMedia file editors
Header file:	ihxtevnts.h

This interface is queried from the IHXRMEvents interface. You then call the IHXRMEvents2::AddSaveProgressSink method to turn on status and progress callbacks from the system. These callbacks include notifying the caller that the processing of events has started or stopped, and a continuous callback

that provides the percent complete. Once your application returns from the IHXRMEvents::Process call, use the IHXRMEvents2::RemoveSaveProgressSink method to remove the progress sink if there are no other events to process.

The IHXRMEvents2 interface contains the following methods:

- IHXRMEvents2::AddSaveProgressSink
- IHXRMEvents2::RemoveSaveProgressSink

As with all COM interfaces, the IHXRMEvents2 interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXRMEvents2::AddSaveProgressSink

Adds a sink interface. This sink interface receives callbacks that indicate the progress of the save operation, in percent.

```
STDMETHOD(AddSaveProgressSink) (
IHXProgressSink* pProgressSink
) PURE;
```

pProgressSink
Pointer to an IHXProgressSink interface that manages the callback notification about a job's progress.

IHXRMEvents2::RemoveSaveProgressSink

Removes the progress-during-save sink interface.

```
STDMETHOD(RemoveSaveProgressSink) (
   IHXProgressSink* pProgressSink
) PURE;
```

pProgressSink
Pointer to the IHXProgressSink interface to be removed.

IHXRMFFDump

Purpose:	Dumps the contents of an existing .rm file to a text file format.
Implemented by:	RealMedia file editors
Header file:	ihxtfdump.h

An entry point, RMACreateRMFFDump, is exported by the RealMedia tools DLL (rmto3260.dll on Windows and rmtools.6.0 on Linux). The object created with this function can then be queried (IUnknown::QueryInterface) for this interface.

The IHXRMFFDump interface contains the following methods:

- IHXRMFFDump::Process
- IHXRMFFDump::SetEndTime
- IHXRMFFDump::SetInputFile
- IHXRMFFDump::SetOutputFile
- IHXRMFFDump::SetStartTime

As with all COM interfaces, the IHXRMFFDump interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXRMFFDump::Process

Begin the dump process.

```
STDMETHOD(Process) (
   THIS
) PURE;
```

IHXRMFFDump::SetEndTime

Sets the end time for the dump.

```
STDMETHOD(SetEndTime) (
   THIS_
   UINT32 ulEndTime
) PURE;
```

ulEndTime

The end time for the dump in milliseconds. If this value is set to zero (0), the end time occurs at the end of file (EOF).

IHXRMFFDump::SetInputFile

Sets the specified RealMedia file as the input file to be dumped.

```
STDMETHOD(SetInputFile) (
   THIS_
   const char* szFileName
) PURE;
```

szFileName
Pointer to the .rm file to be used as the input file.

IHXRMFFDump::SetOutputFile

Sets the specified text file as the output file.

```
STDMETHOD(SetOutputFile) (
   THIS_
   const char* szFileName
) PURE;
```

szFileName
Pointer to the .txt file to be used as the output file.

IHXRMFFDump::SetStartTime

Sets the start time for the dump.

```
STDMETHOD(SetStartTime) (
   THIS_
   UINT32 ulStartTime
) PURE;
```

ulStartTime
The start time for the dump in milliseconds.

IHXRMFileSink

Purpose:	Modifies header and packet information sent to a .rm file.
Implemented by:	RealMedia file editors
Header file:	rmflsnk.h

A sink interface that can be registered with the IHXRMEdit interface allowing the user to modify (encrypt) the RealMedia file headers and packets before they are written to the file.

The IHXRMFileSink interface contains the following methods:

- IHXRMFileSink::OnMediaPropertyHeader
- IHXRMFileSink::OnPacket

As with all COM interfaces, the IHXRMFileSink interface inherits the following IUnknown methods:

- IUnknown::AddRef
- IUnknown::QueryInterface
- IUnknown::Release

IHXRMFileSink::OnMediaPropertyHeader

Retrieves the header information. After you have registered your IHXRMFileSink, this method is called every time a header (Property, MediaProperty, or Content) is about to be written to the .rm file. You then have the opportunity to modify (encrypt) the type-specific data of the header or mime type before it is written to the file.

```
STDMETHOD(OnMediaPropertyHeader) (
   THIS_
   IHXValues* pValues
) PURE;
```

pValues
: Pointer to an IHXValues interface that manages the header information.

IHXRMFileSink::OnPacket

Retrieves the packet information. After you have registered your IHXRMFileSink, this method is called every time a packet is about to be written to the .rm file. You then have the opportunity to modify (encrypt) the data buffer of the packet before it is written to the file.

```
STDMETHOD(OnPacket) (
   THIS_
   IHXPacket* pMediaPacket,
   BOOL bIsKeyFrame
) PURE;
```

pMediaPacket
: Pointer to the IHXPacket interface that manages the data packet.

bIsKeyFrame
: If this parameter is set to TRUE, the packet is a keyframe.

IHXRMFileSinkControl

Purpose:	Adds and removes the file sink interface.
Implemented by:	None
Header file:	`rmflsnk.h`

This interface is obsolete and should not be used.

IHXRMMetaInformation

Purpose:	Adds meta information to an .rm file.
Implemented by:	None
Header file:	`rmmetain.h`

This interface is obsolete and should not be used.

APPENDIX

C

FUNCTION LIST

CreateFileObserver

Purpose:	Creates a file observer object and returns the `IHXTFileObserver` interface from the created object.
Implemented by:	Logging
Header file:	`ihxtfileobserver.h`

```
(STDAPICALLTYPE *FPCREATEFILEOBSERVER)(IHXTFileObserver** ppIFileObserver);
```

ppIFileObserver
Address of a pointer to an `IHXTFileObserver` interface returned from the created object.

HXTCreateJobFactory

Purpose:	Creates an encoding job class factory.
Implemented by:	Encoding
Header file:	`ihxtencodingjob.h`

```
STDAPI HXTCreateJobFactory(
   IHXTClassFactory** ppJobClassFactory
);
```

ppJobClassFactory
Address of a pointer to the `IHXTClassFactory` interface returned from the created object.

RMACreateRMEdit

Purpose:	Creates an instance of a RealMedia editor object.
Implemented by:	RealMedia editors
Header file:	ihxtedit.h

```
STDAPI RMACreateRMEdit(
   IUnknown** ppIUnknown
);
```

ppIUnknown
Address of a pointer to an IUnknown interface that identifies the instance of the RealMedia editor object.

RMACreateRMEvents

Purpose:	Creates an instance of a RealMedit events object.
Implemented by:	RealMedia editors
Header file:	ihxtevnts.h

```
STDAPI RMACreateRMEvents(
   IUnknown** ppIUnknown
);
```

ppIUnknown
Address of a pointer to an IUnknown inteface that identifies the instance of the RealMedia events object.

RMACreateRMFFDump

Purpose:	Creates an instance of a RealMedit file dump object.
Implemented by:	RealMedia editors
Header file:	ihxtfdump.h

```
STDAPI RMACreateRMFFDump(
   IUnknown** ppIUnknown
);
```

ppIUnknown
Address of a pointer to an IUnknown interface that identifies the instance of the RealMedia file dump object.

RMAGetLogSystemInterface

Purpose:	Creates a logging system object and returns the `IHXTLogSystem` interface from the created object.
Implemented by:	Logging
Header file:	`ihxtlogsystem.h`

```
(STDAPICALLTYPE *FPRMAGETLOGSYSTEMINTERFACE)(IHXTLogSystem**
ppLogSystem);
```

ppLogSystem
Address of a pointer to an instance of an `IHXTLogSystem` interface returned from the created object. If the logging system has not yet been created, it is created and initialized.

SetDLLAccessPath

Purpose:	Module entry point to the encoding system used to specify DLL locations.
Implemented by:	Encoding
Header file:	`ihxtencodingjob.h`

```
STDAPI SetDLLAccessPath(
   const char* pPathDescriptor
);
```

pPathDescription
Pointer to the path to the DLLs.

GLOSSARY

A

API
Application programming interface. A technique specified by an operating system or application whereby a programmer's application can make requests of that operating system or application.

ASM
Adaptive stream management. Rules that describe a streaming data type to Helix DNA Producer.

activate
Only in the scope of the input and output lists, enabling the radio button on an input and make that input the source for the encoded content, or to enable the check box of an output and begin encoding to that output, in addition to any other checked outputs.

agent layer
The part of a plug-in where the plug-in's property initialization occurs and media format information is handled.

audience
An item that defines a set of stream properties that is designed as a streaming configuration for a player connection at a specific bit rate. A single audience definition contains information necessary to use that audience for audio only, audio and video, and video only streams. For example, "56k Modem" is the name of one audience. Unlike previous producers, duress streams are no longer part of an audience, but are themselves audiences.

audience template file
An XML file containing the properties and settings for a single audience.

audio clipping
Occurs when audio exceeds the maximum range of the recording medium.

audio gain
The level of increase in audio signal strength, usually expressed in dB.

B

bit rate
The rate at which a presentation is streamed, usually expressed in kilobits per second (Kbps).

black level
The amount of light in the darkest area of a video display.

C

CBR
Constant bit rate. An encoding method in which all parts of the video play back at the same bit rate. Contrast to *VBR*.

COM
Component object model. A technology used by the Helix DNA Producer SDK for describing interfaces and exporting objects that implement those interfaces.

capture device
An input filter that takes audio and video data from a hardware device and sends it to the encoding engine. Common capture device hardware are sound cards and video capture cards.

class factory
A COM object whose purpose is to create other objects of a particular class ID, and return an interface pointer to that object.

clip information
Additional information regarding a presentation, such as the title, author, copyright, and so on for any media file.

clipping
Defining new in and out points for a static input file such that only the portion of the source between the markers is sent through the encoder.

codec
Coder/decoder. An algorithm for compressing files used with Helix DNA Producer. A variety of codecs exist for different streaming bit-rates and content types.

configuration agent
The part of a plug-in that gets and sets properties during the configuration process in the plug-in.

connection agent
The part of the plug-in used by the *filter graph manager* to negotiate the connections between filters in a data flow.

cropping
Setting a subset of the dimensions of a video source. This subset is what is sent through the encoder (for example, cropping a source image with the dimensions 640x480 to 640x360 letterbox).

D

DRM
Digital rights management. A set of technology and rules that allows content owners to set rights on how, when, and with what frequency end users can view content.

deinterlace
The process of creating a single video frame from two interlaced fields of a video frame. Deinterlacing removes interlacing artifacts for a still frame, or if the video is being displayed at a different frame rate from the original.

deserialize
The process of extracting information from an XML file and using that information to recreate an encoding object. See also *serialize*.

destination
Refers to either a RealMedia file or Helix Universal Server that is a location where the encoded data will be sent to or stored.

down-shift
The ability of a RealPlayer connected to a Helix Universal Server to decrease the bit rate of a stream being received, such as in conditions of constrained network bandwidth.

E

encoding engine
The part of the encoding system that processes the media sample inputs and produces a modified media sample output.

encoding job
A set of inputs, outputs, destinations, target audiences, metadata, and filters: references to everything necessary to perform a single encode, which is one or multiple inputs going to one or multiple files and/or servers.

encoding session
The time during which the process of encoding audio and video data from one format to another occurs.

encoding system
A set of components that integrate with the purpose of encoding media. The Helix DNA Producer encoding system consists of the encoding engine, the filter graph manager, and all of the plug-ins associated with an encoding session.

enumeration
The process of navigating through a list of objects.

event
An occurrence that provides notification throughout the encoding system of a state change or an error condition. See *RealMedia event*.

F

file observer
A part of the logging system that writes messages to a text file. The file observer can be configured to remove certain messages before they are saved in the text file. A file observer is the standard logging system observer shipped with the Helix DNA Producer SDK. Contrast with *log observer*.

file reader
An input filter that reads from a media file and provides audio and video media for the encoding engine.

file rolling
The process of creating additional files at a certain time or after a specified file size is reached to prevent the original file size from becoming too large. During a file rolling process, a new file is automatically created, the original file is renamed, and packets are then sent to the new file.

filter
Any part of the encoding system that passes media data, and either analyzes or modifies the data as it passes. Alternatively, any process that removes artifacts that appear in the encoded clips because of the methods used to create the source video.

filter graph manager
Part of the Helix DNA Producer encoding system that manages the connection of various media filters, and also manages data flow through the filters once they are connected.

filter layer
The part of a plug-in that generates or accepts media samples, modifies or analyzes them, then passes them on to the next component in the data pipeline.

H

helper class
A code wrapper that provides access to low-level interfaces and methods used by Helix DNA Producer. These helper classes allow you to create plug-ins without having to populate these low-level interfaces and methods.

I

image map
A clickable region in a RealVideo presentation that can launch a URL in a browser, cause the player to seek in the current presentation, or cause the player to start playback of a new presentation.

input
A source, plus all of its associated filter and metadata settings. The name is generally take from the source, and is displayed in the input list of the main window.

inverse telecine
The process of stripping out redundant frames from video that was produced by converting 24 frames per second (fps) cinematic film to 30 fps NTSC video (*telecine*). The inverse telecine process removes the redundant frames and returns the video to its original 24 fps.

L

log message
Information passed through the logging system that can be written to a file or passed to other log observers. Log messages are commonly categorized as errors, warnings, information, or diagnostic messages. Alternatively, the log messages can be categorized by functional area, such as file reader, capture, codec, audio prefilter, broadcast, and so on.

log observer
A custom component in the logging system that receives log messages and can act on them, writing them to a file, showing them on a display, and so on. Contrast to *file observer*.

logging system
A part of the encoding system that handles the input and output of informational messages.

M

mark-in
The starting marker of a clipping region.

mark-out
The ending marker of a clipping region.

media format
The properties associated with a specific type of audio or video media.

media profile
A grouping for all items inside an output, excluding destinations.

media sample
A logical part of media data, for example, a frame of raw video or a raw PCM audio frame.

media sink
An object into which you push media samples. The media sink interface takes raw data that it then passes to the rest of the encoding system.

metadata
See *clip information*.

N

NTSC
National Television System Committee. The name of one type of analog transmission of television signals, consisting of a standard image format of 525 lines with an aspect ratio of 4:3 displayed at 60 Hz for an effective frame rate of 30 fps.

O

output profile
One or more destinations, plus a set of one or more audiences, and a set of filters: the data that defines the settings for the encoded data, plus where that data will be stored and/or which server(s) it will be streamed to. This can also be thought of as a group of destinations and their associated settings (which are common to all destinations).

P

PAL
Phase Alternate Line. The name of one type of analog transmission of television signals, consisting of a standard image format of 625 lines with an aspect ratio of 4:3 displayed at 50 Hz for an effective frame rate of 25 fps.

plug-in
A type of software that adds a specific capability to a program already on your computer. For instance, your browser probably requires a plug-in to see certain types of animation.

plug-in layer
The part of a plug-in that describes the functionality of the plug-in to the rest of the encoding system.

postfilter
A filter applied to source content after encoding. For example, digital rights management (*DRM*) is generally applied to an output source after encoding.

prefilter
A filter applied to source content prior to encoding. The purpose of a prefilter is generally to improve the quality of the source material or make it more suitable for encoding.

preview
The act of receiving the video and/or audio media at certain points in the encode sequence, either before it is encoded or during the encoding process, but before the data is written to the final output.

property
A single name/value pair used for configuration of an object, or used as a means for an object to describe itself.

property bag
A collection of properties.

pull broadcast
A form of broadcasting where the player initiates the stream of data from the encoder to the server, then from the server to the player.

push broadcast
A form of broadcasting where the encoder immediately sends data to the server as soon as the broadcast is initiated.

R

RealMedia
A blanket term used to refer to the various "real" data types that Helix can stream.

RealMedia event
An occurrence within an `.rm` file that either changes the title, author, or copyright clip information displayed in the player, or opens a specified URL in a browser, or performs a custom task. Contrast to *event*.

remote administrator
The component of Helix DNA Producer that allows remote configuration and monitoring of a cluster of encoder boxes, based on a web browser client side interface.

remote agent
A component on the same machine as the encoder that can talk to multiple encoder instances and external components, such as the remote administrator.

resampling
The process of recalculating the sample rate of an audio source at a different rate from which it was originally recorded. If the sample rate is decreased, samples are removed from the audio source, making the audio source smaller, but possibly decreasing the available frequency range. If the sample rate is increased, samples are added to the audio source, but the available frequency range is usually unchanged.

resize
The process of changing the size of a video presentation by removing some video data. Helix DNA Producer removes video data by using either a quick method (fast resize) or a complex analysis (high-quality resize). A high-quality resize results in a superior image, but it also lengthens the encoding time.

S

select
To select an item from a list such that the item is highlighted, and any operations performed effect that item. For example, selecting an input would allow a user to copy or paste input settings, but would not activate (that is, encode from or to) that input.

serialize
The process of saving the configuration settings of an object to a file or to a buffer in memory by converting the settings to an XML document. See also *deserialize*.

server definition
Defines all of the information required for Helix DNA Producer to connect to a specified Helix Universal Server.

smart scaling
The ability to create streams of different dimensions inside a single SureStream file, and the ability of the RealPlayer to switch between those streams when appropriate for up shifting and down shifting.

source
Refers to a file or device that contains or otherwise provides references to the data to be encoded.

splitter
The process of taking a live broadcast from a server acting as a transmitter, and providing that broadcast to a number of downstream proxies or servers acting as receivers, before providing the content locally to clients.

statistics
Information provided by the encoder that describes either the current or lifetime status of various aspects of the audio and video samples during the encoding process.

stream
In the case of an single-rate audio file, the single audio track. In the case of an audio/video file, the combination of the audio and video tracks makes up the stream.

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.

T

telecine
The process of converting 24 frames per second (fps) cinematic film to 30 fps NTSC video by adding redundant video frames. The redundant frames produced by this process can be reversed on the video using *inverse telecine*.

time stamp
A property set on a media sample that is primarily used to identify when in the presentation the media sample will be played.

two-pass encoding
The process of first analyzing the entire source video before making a second pass to encode the stream. Two-pass encoding helps the most with variable bit rate (*VBR*) encoding.

U

up-shift
The ability of a RealPlayer connected to a Helix Universal Server to increase the bit rate of a stream being received.

V

VBR
Variable bit rate. An encoding method that varies the bit rates of different parts of a video, even though the video is being streamed a a constant rate. Contrast to *CBR*.

video noise
Any number of different types of distortion that result in the degradation of the video media.

X

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

C

K