Java™ Media Framework API Guide

November 19,1999
JMF 2.0 FCS

Contents

Preface

The Java™ Media Framework (JMF) is an application programming interface (API) for incorporating time-based media into Java applications and applets. This guide is intended for Java programmers who want to incorporate time-based media into their applications and for technology providers who are interested in extending JMF and providing JMF plug-ins to support additional media types and perform custom processing and rendering.

About JMF

The JMF 1.0 API (the Java Media Player API) enabled programmers to develop Java programs that presented time-based media. The JMF 2.0 API extends the framework to provide support for capturing and storing media data, controlling the type of processing that is performed during playback, and performing custom processing on media data streams. In addition, JMF 2.0 defines a plug-in API that enables advanced developers and technology providers to more easily customize and extend JMF functionality.

The following classes and interfaces are new in JMF 2.0:

`AudioFormat`	`BitRateControl`	`Buffer`
`BufferControl`	`BufferToImage`	`BufferTransferHandler`
`CaptureDevice`	`CaptureDeviceInfo`	`CaptureDeviceManager`
`CloneableDataSource`	`Codec`	`ConfigureCompleteEvent`
`ConnnectionErrorEvent`	`DataSink`	`DataSinkErrorEvent`
`DataSinkEvent`	`DataSinkListener`	`Demultiplexer`

Effect	EndOfStreamEvent	FileTypeDescriptor
Format	FormatChangeEvent	FormatControl
FrameGrabbingControl	FramePositioningControl	FrameProcessingControl
FrameRateControl	H261Control	H261Format
H263Control	H263Format	ImageToBuffer
IndexedColorFormat	InputSourceStream	KeyFrameControl
MonitorControl	MpegAudioControl	Multiplexer
NoStorageSpaceErrorEvent	PacketSizeControl	PlugIn
PlugInManager	PortControl	Processor
ProcessorModel	PullBufferDataSource	PullBufferStream
PushBufferDataSource	PushBufferStream	QualityControl
Renderer	RGBFormat	SilenceSuppressionControl
StreamWriterControl	Track	TrackControl
VideoFormat	VideoRenderer	YUVFormat

In addition, the `MediaPlayer` Java Bean has been included with the JMF API in `javax.media.bean.playerbean`. `MediaPlayer` can be instantiated directly and used to present one or more media streams.

Future versions of the JMF API will provide additional functionality and enhancements while maintaining compatibility with the current API.

Design Goals for the JMF API

JMF 2.0 supports media capture and addresses the needs of application developers who want additional control over media processing and rendering. It also provides a plug-in architecture that provides direct access to media data and enables JMF to be more easily customized and extended. JMF 2.0 is designed to:

- Be easy to program
- Support capturing media data
- Enable the development of media streaming and conferencing applications in Java

- Enable advanced developers and technology providers to implement custom solutions based on the existing API and easily integrate new features with the existing framework
- Provide access to raw media data
- Enable the development of custom, downloadable demultiplexers, codecs, effects processors, multiplexers, and renderers (JMF *plug-ins*)
- Maintain compatibility with JMF 1.0

About the JMF RTP APIs

The classes in `javax.media.rtp`, `javax.media.rtp.event`, and `javax.media.rtp.rtcp` provide support for RTP (Real-Time Transport Protocol). RTP enables the transmission and reception of real-time media streams across the network. RTP can be used for media-on-demand applications as well as interactive services such as Internet telephony.

JMF-compliant implementations are not required to support the RTP APIs in `javax.media.rtp`, `javax.media.rtp.event`, and `javax.media.rtp.rtcp`. The reference implementations of JMF provided by Sun Microsystems, Inc. and IBM Corporation fully support these APIs.

The first version of the JMF RTP APIs (referred to as the RTP Session Manager API) enabled developers to receive RTP streams and play them using JMF. In JMF 2.0, the RTP APIs also support the transmission of RTP streams.

The following RTP classes and interfaces are new in JMF 2.0:

`SendStream`	`SendStreamListener`	`InactiveSendStreamEvent`
`ActiveSendStreamEvent`	`SendPayloadChangeEvent`	`NewSendStreamEvent`
`GlobalTransmissionStats`	`TransmissionStats`	

The RTP packages have been reorganized and some classes, interfaces, and methods have been renamed to make the API easier to use. The package reorganization consists of the following changes:

- The RTP event classes that were in `javax.media.rtp.session` are now in `javax.media.rtp.event`.
- The RTCP-related classes that were in `javax.media.rtp.session` are now in `javax.media.rtp.rtcp`.

- The rest of the classes in `javax.media.rtp.session` are now in `javax.media.rtp` and the `javax.media.rtp.session` package has been removed.

The name changes consist primarily of the removal of the RTP and RTCP prefixes from class and interface names and the elimination of non-standard abbreviations. For example, `RTPRecvStreamListener` has been renamed to `ReceiveStreamListener`. For a complete list of the changes made to the RTP packages, see the JMF 2.0 Beta release notes.

In addition, changes were made to the RTP APIs to make them compatible with other changes in JMF 2.0:

- `javax.media.rtp.session.io` and `javax.media.rtp.session.depacketizer` have been removed. Custom RTP packetizers and depacketizers are now supported through the JMF 2.0 plug-in architecture. Existing depacketizers will need to be ported to the new plug-in architecture.
- `Buffer` is now the basic unit of transfer between the `SessionManager` and other JMF objects, in place of `DePacketizedUnit` and `DePacketizedObject`. RTP-formatted `Buffers` have a specific format for their data and header objects.
- `BaseEncodingInfo` has been replaced by the generic JMF `Format` object. An RTP-specific `Format` is differentiated from other formats by its encoding string. Encoding strings for RTP-specific `Formats` end in `_RTP`. Dynamic payload information can be provided by associating a dynamic payload number with a `Format` object.

Design Goals for the JMF RTP APIs

The RTP APIs in JMF 2.0 support the reception and transmission of RTP streams and address the needs of application developers who want to use RTP to implement media streaming and conferencing applications. These APIs are designed to:

- Enable the development of media streaming and conferencing applications in Java
- Support media data reception and transmission using RTP and RTCP
- Support custom packetizer and depacketizer plug-ins through the JMF 2.0 plug-in architecture.
- Be easy to program

Partners in the Development of the JMF API

The JMF 2.0 API is being jointly designed by Sun Microsystems, Inc. and IBM Corporation.

The JMF 1.0 API was jointly developed by Sun Microsystems Inc., Intel Corporation, and Silicon Graphics, Inc.

Contact Information

For the latest information about JMF, visit the Sun Microsystems, Inc. website at:

```
http://java.sun.com/products/java-media/jmf/
```

Additional information about JMF can be found on the IBM Corporation website at:

```
http://www.software.ibm.com/net.media/
```

About this Document

This document describes the architecture and use of the JMF 2.0 API. It replaces the Java Media Player Guide distributed in conjunction with the JMF 1.0 releases.

Except where noted, the information in this book is not implementation specific. For examples specific to the JMF reference implementation developed by Sun Microsystems and IBM corporation, see the sample code and solutions available from Sun's JMF website (`http://java.sun.com/products/java-media/jmf/index.html`).

Guide to Contents

This document is split into two parts:

- Part 1 describes the features provided by the JMF 2.0 API and illustrates how you can use JMF to incorporate time-based media in your Java applications and applets.
- Part 2 describes the support for real-time streaming provided by the JMF RTP APIs and illustrates how to send and receive streaming media across the network.

Part 1 is organized into six chapters:

- **"Working with Time-Based Media"**—sets the stage for JMF by introducing the key concepts of media content, presentation, processing, and recording.
- **"Understanding JMF"**—introduces the JMF 2.0 API and describes the high-level architecture of the framework.
- **"Presenting Time-Based Media with JMF"**—describes how to use JMF `Players` and `Processors` to present time-based media.
- **"Processing Time-Based Media with JMF"**—describes how to manipulate media data using a JMF `Processor`.
- **"Capturing Time-Based Media with JMF"**—describes how to record media data using JMF `DataSources` and `Processors`.
- **"Extending JMF"**—describes how to enhance JMF functionality by creating new processing plug-ins and implementing custom JMF classes.

Part 2 is organized into six chapters:

- **"Working with Real-Time Media Streams"**--provides an overview of streaming media and the Real-time Transport protocol (RTP).
- **"Understanding the JMF RTP API"**—describes the JMF RTP APIs.
- **"Receiving and Presenting RTP Media Streams"**—illustrates how to handle RTP Client operations.
- **"Transmitting RTP Media Streams"**—illustrates how to handle RTP Server operations.
- **"Importing and Exporting RTP Media Streams"**—shows how to read and write RTP data to a file.
- **"Creating Custom Packetizers and Depacketizers"**—describes how to use JMF plug-ins to support additional RTP packet formats and codecs.

At the end of this document, you'll find Appendices that contain complete sample code for some of the examples used in these chapters and a glossary of JMF-specific terms.

Change History

Version JMF 2.0 FCS

- Fixed references to `TrackControl` methods to reflect modified `TrackControl` API.
- Fixed minor sample code errors.
- Clarified behavior of cloneable data sources.
- Clarified order of events when writing to a file.

Version 0.9

Internal Review Draft

Version 0.8

JMF 2.0 Beta draft:

- Added an introduction to RTP, Working with Real-Time Media Streams, and updated the RTP chapters.
- Updated to reflect API changes since the Early Access release.
- Added an example of registering a plug-in with the `PlugInManager`.
- Added chapter, figure, table, and example numbers and changed the example code style.

Version 0.7

JMF 2.0 Early Access Release 1 draft:

- Updated and expanded RTP chapters in Part 2.
- Added Demultiplexer example to "Extending JMF".
- Updated to reflect API changes since the public review.

Version 0.6

Internal Review Draft

Version 0.5

JMF 2.0 API public review draft.

- Added new concepts chapter, **"Working with Time-Based Media"**.
- Reorganized architecture information in **"Understanding JMF"**.

- Incorporated RTP Guide as Part 2.

Version 0.4

JMF 2.0 API licensee review draft.

Comments

Please submit any comments or suggestions you have for improving this document to `jmf-comments@eng.sun.com`.

Part 1: Java™ Media Framework

1

Working with Time-Based Media

Any data that changes meaningfully with respect to time can be characterized as time-based media. Audio clips, MIDI sequences, movie clips, and animations are common forms of time-based media. Such media data can be obtained from a variety of sources, such as local or network files, cameras, microphones, and live broadcasts.

This chapter describes the key characteristics of time-based media and describes the use of time-based media in terms of a fundamental data processing model:

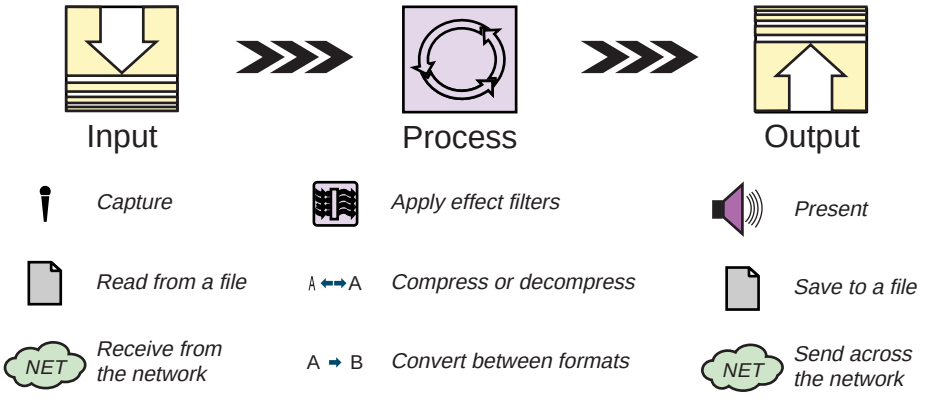


Figure 1-1: Media processing model.

Streaming Media

A key characteristic of time-based media is that it requires timely delivery and processing. Once the flow of media data begins, there are strict timing deadlines that must be met, both in terms of receiving and presenting the data. For this reason, time-based media is often referred to as *streaming media*—it is delivered in a steady stream that must be received and processed within a particular timeframe to produce acceptable results.

For example, when a movie is played, if the media data cannot be delivered quickly enough, there might be odd pauses and delays in playback. On the other hand, if the data cannot be received and processed quickly enough, the movie might appear jumpy as data is lost or frames are intentionally dropped in an attempt to maintain the proper playback rate.

Content Type

The format in which the media data is stored is referred to as its *content type.* QuickTime, MPEG, and WAV are all examples of content types. Content type is essentially synonymous with file type—content type is used because media data is often acquired from sources other than local files.

Media Streams

A *media stream* is the media data obtained from a local file, acquired over the network, or captured from a camera or microphone. Media streams often contain multiple channels of data called *tracks*. For example, a Quicktime file might contain both an audio track and a video track. Media streams that contain multiple tracks are often referred to as *multiplexed* or *complex* media streams. *Demultiplexing* is the process of extracting individual tracks from a complex media stream.

A track's *type* identifies the kind of data it contains, such as audio or video. The *format* of a track defines how the data for the track is structured.

A media stream can be identified by its location and the protocol used to access it. For example, a URL might be used to describe the location of a QuickTime file on a local or remote system. If the file is local, it can be accessed through the FILE protocol. On the other hand, if it's on a web server, the file can be accessed through the HTTP protocol. A *media locator* provides a way to identify the location of a media stream when a URL can't be used.

Media streams can be categorized according to how the data is delivered:

- Pull—data transfer is initiated and controlled from the client side. For example, Hypertext Transfer Protocol (HTTP) and FILE are pull protocols.
- Push—the server initiates data transfer and controls the flow of data. For example, Real-time Transport Protocol (RTP) is a push protocol used for streaming media. Similarly, the SGI MediaBase protocol is a push protocol used for video-on-demand (VOD).

Common Media Formats

The following tables identify some of the characteristics of common media formats. When selecting a format, it's important to take into account the characteristics of the format, the target environment, and the expectations of the intended audience. For example, if you're delivering media content via the web, you need to pay special attention to the bandwidth requirements.

The CPU Requirements column characterizes the processing power necessary for optimal presentation of the specified format. The Bandwidth Requirements column characterizes the transmission speeds necessary to send or receive data quickly enough for optimal presentation.

Format	Content Type	Quality	CPU Requirements	Bandwidth Requirements
Cinepak	AVI QuickTime	Medium	Low	High
MPEG-1	MPEG	High	High	High
H.261	AVI RTP	Low	Medium	Medium
H.263	QuickTime AVI RTP	Medium	Medium	Low
JPEG	QuickTime AVI RTP	High	High	High

Format	Content Type	Quality	CPU Requirements	Bandwidth Requirements
Indeo	QuickTime AVI	Medium	Medium	Medium

Table 1-1: Common video formats.

Format	Content Type	Quality	CPU Requirements	Bandwidth Requirements
PCM	AVI QuickTime WAV	High	Low	High
Mu-Law	AVI QuickTime WAV RTP	Low	Low	High
ADPCM (DVI, IMA4)	AVI QuickTime WAV RTP	Medium	Medium	Medium
MPEG-1	MPEG	High	High	High
MPEG Layer3	MPEG	High	High	Medium
GSM	WAV RTP	Low	Low	Low
G.723.1	WAV RTP	Medium	Medium	Low

Table 1-2: Common audio formats.

Some formats are designed with particular applications and requirements in mind. High-quality, high-bandwidth formats are generally targeted toward CD-ROM or local storage applications. H.261 and H.263 are generally used for video conferencing applications and are optimized for video where there's not a lot of action. Similarly, G.723 is typically used to produce low bit-rate speech for telephony applications.

Media Presentation

Most time-based media is audio or video data that can be presented through output devices such as speakers and monitors. Such devices are the most common *destination* for media data output. Media streams can also be sent to other destinations—for example, saved to a file or transmitted across the network. An output destination for media data is sometimes referred to as a *data sink*.

Presentation Controls

While a media stream is being presented, VCR-style presentation controls are often provided to enable the user to control playback. For example, a control panel for a movie player might offer buttons for stopping, starting, fast-forwarding, and rewinding the movie.

Latency

In many cases, particularly when presenting a media stream that resides on the network, the presentation of the media stream cannot begin immediately. The time it takes before presentation can begin is referred to as the *start latency.* Users might experience this as a delay between the time that they click the start button and the time when playback actually starts.

Multimedia presentations often combine several types of time-based media into a synchronized presentation. For example, background music might be played during an image slide-show, or animated text might be synchronized with an audio or video clip. When the presentation of multiple media streams is synchronized, it is essential to take into account the start latency of each stream—otherwise the playback of the different streams might actually begin at different times.

Presentation Quality

The quality of the presentation of a media stream depends on several factors, including:

- The compression scheme used
- The processing capability of the playback system
- The bandwidth available (for media streams acquired over the network)

Traditionally, the higher the quality, the larger the file size and the greater the processing power and bandwidth required. Bandwidth is usually represented as the number of bits that are transmitted in a certain period of time—the *bit rate*.

To achieve high-quality video presentations, the number of frames displayed in each period of time (the *frame rate*) should be as high as possible. Usually movies at a frame rate of 30 frames-per-second are considered indistinguishable from regular TV broadcasts or video tapes.

Media Processing

In most instances, the data in a media stream is manipulated before it is presented to the user. Generally, a series of processing operations occur before presentation:

1. If the stream is multiplexed, the individual tracks are extracted.
2. If the individual tracks are compressed, they are decoded.
3. If necessary, the tracks are converted to a different format.
4. Effect filters are applied to the decoded tracks (if desired).

The tracks are then delivered to the appropriate output device. If the media stream is to be stored instead of rendered to an output device, the processing stages might differ slightly. For example, if you wanted to capture audio and video from a video camera, process the data, and save it to a file:

1. The audio and video tracks would be captured.
2. Effect filters would be applied to the raw tracks (if desired).
3. The individual tracks would be encoded.
4. The compressed tracks would be multiplexed into a single media stream.
5. The multiplexed media stream would then be saved to a file.

Demultiplexers and Multiplexers

A demultiplexer extracts individual tracks of media data from a multiplexed media stream. A *mutliplexer* performs the opposite function, it takes individual tracks of media data and merges them into a single multiplexed media stream.

Codecs

A codec performs media-data compression and decompression. When a track is encoded, it is converted to a compressed format suitable for storage or transmission; when it is decoded it is converted to a non-compressed (raw) format suitable for presentation.

Each codec has certain input formats that it can handle and certain output formats that it can generate. In some situations, a series of codecs might be used to convert from one format to another.

Effect Filters

An effect filter modifies the track data in some way, often to create special effects such as blur or echo.

Effect filters are classified as either pre-processing effects or post-processing effects, depending on whether they are applied before or after the codec processes the track. Typically, effect filters are applied to uncompressed (raw) data.

Renderers

A renderer is an abstraction of a presentation device. For audio, the presentation device is typically the computer's hardware audio card that outputs sound to the speakers. For video, the presentation device is typically the computer monitor.

Compositing

Certain specialized devices support *compositing*. Compositing time-based media is the process of combining multiple tracks of data onto a single presentation medium. For example, overlaying text on a video presentation is one common form of compositing. Compositing can be done in either hardware or software. A device that performs compositing can be abstracted as a renderer that can receive multiple tracks of input data.

Media Capture

Time-based media can be captured from a live source for processing and playback. For example, audio can be captured from a microphone or a video capture card can be used to obtain video from a camera. Capturing can be thought of as the *input* phase of the standard media processing model.

A capture device might deliver multiple media streams. For example, a video camera might deliver both audio and video. These streams might be captured and manipulated separately or combined into a single, multiplexed stream that contains both an audio track and a video track.

Capture Devices

To capture time-based media you need specialized hardware—for example, to capture audio from a live source, you need a microphone and an appropriate audio card. Similarly, capturing a TV broadcast requires a TV tuner and an appropriate video capture card. Most systems provide a query mechanism to find out what capture devices are available.

Capture devices can be characterized as either push or pull sources. For example, a still camera is a pull source—the user controls when to capture an image. A microphone is a push source—the live source continuously provides a stream of audio.

The format of a captured media stream depends on the processing performed by the capture device. Some devices do very little processing and deliver raw, uncompressed data. Other capture devices might deliver the data in a compressed format.

Capture Controls

Controls are sometimes provided to enable the user to manage the capture process. For example, a capture control panel might enable the user to specify the data rate and encoding type for the captured stream and start and stop the capture process.

2

Understanding JMF

Java™ Media Framework (JMF) provides a unified architecture and messaging protocol for managing the acquisition, processing, and delivery of time-based media data. JMF is designed to support most standard media content types, such as AIFF, AU, AVI, GSM, MIDI, MPEG, QuickTime, RMF, and WAV.

By exploiting the advantages of the Java platform, JMF delivers the promise of "Write Once, Run Anywhere™" to developers who want to use media such as audio and video in their Java programs. JMF provides a common cross-platform Java API for accessing underlying media frameworks. JMF implementations can leverage the capabilities of the underlying operating system, while developers can easily create portable Java programs that feature time-based media by writing to the JMF API.

With JMF, you can easily create applets and applications that present, capture, manipulate, and store time-based media. The framework enables advanced developers and technology providers to perform custom processing of raw media data and seamlessly extend JMF to support additional content types and formats, optimize handling of supported formats, and create new presentation mechanisms.

High-Level Architecture

Devices such as tape decks and VCRs provide a familiar model for recording, processing, and presenting time-based media. When you play a movie using a VCR, you provide the media stream to the VCR by inserting a video tape. The VCR reads and interprets the data on the tape and sends appropriate signals to your television and speakers.

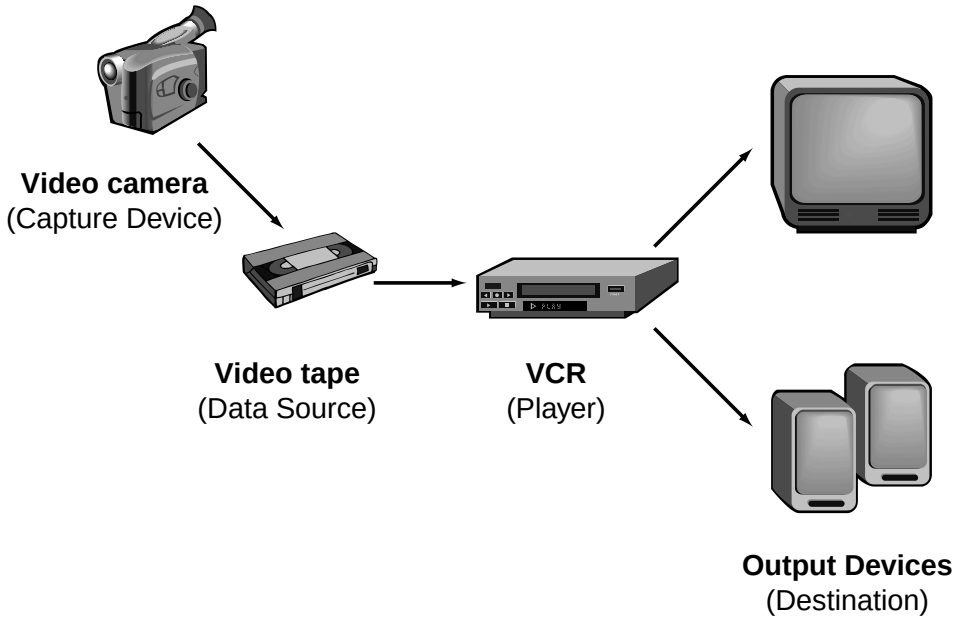


Figure 2-1: Recording, processing, and presenting time-based media.

JMF uses this same basic model. A *data source* encapsulates the media stream much like a video tape and a *player* provides processing and control mechanisms similar to a VCR. Playing and capturing audio and video with JMF requires the appropriate input and output devices such as microphones, cameras, speakers, and monitors.

Data sources and players are integral parts of JMF's high-level API for managing the capture, presentation, and processing of time-based media. JMF also provides a lower-level API that supports the seamless integration of custom processing components and extensions. This layering provides Java developers with an easy-to-use API for incorporating time-based media into Java programs while maintaining the flexibility and extensibility required to support advanced media applications and future media technologies.

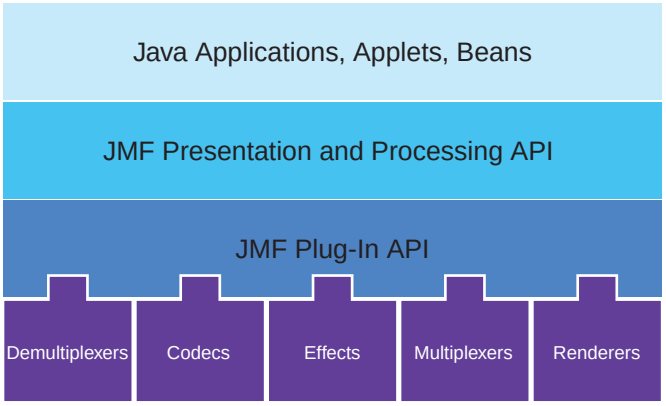


Figure 2-2: High-level JMF achitecture.

Time Model

JMF keeps time to nanosecond precision. A particular point in time is typically represented by a `Time` object, though some classes also support the specification of time in nanoseconds.

Classes that support the JMF time model implement `Clock` to keep track of time for a particular media stream. The `Clock` interface defines the basic timing and synchronization operations that are needed to control the presentation of media data.

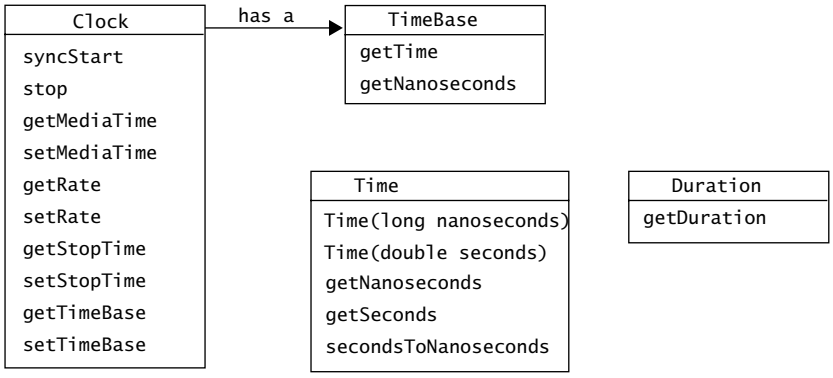


Figure 2-3: JMF time model.

A `Clock` uses a `TimeBase` to keep track of the passage of time while a media stream is being presented. A `TimeBase` provides a constantly ticking time source, much like a crystal oscillator in a watch. The only information that a `TimeBase` provides is its current time, which is referred to as the *time-base*

time. The time-base time cannot be stopped or reset. Time-base time is often based on the system clock.

A `Clock` object's *media time* represents the current position within a media stream—the beginning of the stream is media time zero, the end of the stream is the maximum media time for the stream. The *duration* of the media stream is the elapsed time from start to finish—the length of time that it takes to present the media stream. (Media objects implement the `Duration` interface if they can report a media stream's duration.)

To keep track of the current media time, a `Clock` uses:

- The time-base start-time—the time that its `TimeBase` reports when the presentation begins.
- The media start-time—the position in the media stream where presentation begins.
- The playback rate—how fast the `Clock` is running in relation to its `TimeBase`. The *rate* is a scale factor that is applied to the `TimeBase`. For example, a rate of 1.0 represents the normal playback rate for the media stream, while a rate of 2.0 indicates that the presentation will run at twice the normal rate. A negative rate indicates that the `Clock` is running in the opposite direction from its `TimeBase`—for example, a negative rate might be used to play a media stream backward.

When presentation begins, the media time is mapped to the time-base time and the advancement of the time-base time is used to measure the passage of time. During presentation, the current media time is calculated using the following formula:

```
MediaTime = MediaStartTime + Rate(TimeBaseTime - TimeBaseStartTime)
```

When the presentation stops, the media time stops, but the time-base time continues to advance. If the presentation is restarted, the media time is remapped to the current time-base time.

Managers

The JMF API consists mainly of interfaces that define the behavior and interaction of objects used to capture, process, and present time-based media. Implementations of these interfaces operate within the structure of the framework. By using intermediary objects called *managers*, JMF makes it easy to integrate new implementations of key interfaces that can be used seamlessly with existing classes.

JMF uses four managers:

- `Manager`—handles the construction of `Players`, `Processors`, `DataSources`, and `DataSinks`. This level of indirection allows new implementations to be integrated seamlessly with JMF. From the client perspective, these objects are always created the same way whether the requested object is constructed from a default implementation or a custom one.
- `PackageManager`—maintains a registry of packages that contain JMF classes, such as custom `Players`, `Processors`, `DataSources`, and `DataSinks`.
- `CaptureDeviceManager`—maintains a registry of available capture devices.
- `PlugInManager`—maintains a registry of available JMF plug-in processing components, such as `Multiplexers`, `Demultiplexers`, `Codecs`, `Effects`, and `Renderers`.

To write programs based on JMF, you'll need to use the `Manager` `create` methods to construct the `Players`, `Processors`, `DataSources`, and `DataSinks` for your application. If you're capturing media data from an input device, you'll use the `CaptureDeviceManager` to find out what devices are available and access information about them. If you're interested in controlling what processing is performed on the data, you might also query the `PlugInManager` to determine what plug-ins have been registered.

If you extend JMF functionality by implementing a new plug-in, you can register it with the `PlugInManager` to make it available to `Processors` that support the plug-in API. To use a custom `Player`, `Processor`, `DataSource`, or `DataSink` with `JMF`, you register your unique package prefix with the `PackageManager`.

Event Model

JMF uses a structured event reporting mechanism to keep JMF-based programs informed of the current state of the media system and enable JMF-based programs to respond to media-driven error conditions, such as out-of data and resource unavailable conditions. Whenever a JMF object needs to report on the current conditions, it posts a `MediaEvent`. `MediaEvent` is subclassed to identify many particular types of events. These objects follow the established Java Beans patterns for events.

For each type of JMF object that can post `MediaEvents`, JMF defines a corresponding listener interface. To receive notification when a `MediaEvent` is posted, you implement the appropriate listener interface and register your listener class with the object that posts the event by calling its `addListener` method.

`Controller` objects (such as `Players` and `Processors`) and certain `Control` objects such as `GainControl` post media events.

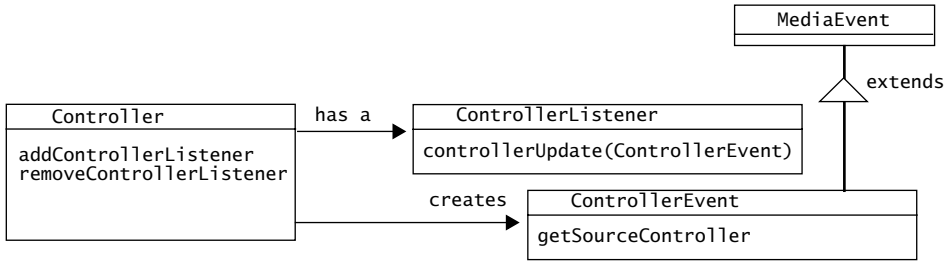


Figure 2-4: JMF event model.

`RTPSessionManager` objects also post events. For more information, see "RTP Events" on page 122.

Data Model

JMF media players usually use `DataSources` to manage the transfer of media-content. A `DataSource` encapsulates both the location of media and the protocol and software used to deliver the media. Once obtained, the source cannot be reused to deliver other media.

A `DataSource` is identified by either a JMF `MediaLocator` or a `URL` (universal resource locator). A `MediaLocator` is similar to a `URL` and can be constructed from a `URL`, but can be constructed even if the corresponding protocol handler is not installed on the system. (In Java, a `URL` can only be constructed if the corresponding protocol handler is installed on the system.)

A `DataSource` manages a set of `SourceStream` objects. A standard data source uses a byte array as the unit of transfer. A *buffer data source* uses a `Buffer` object as its unit of transfer. JMF defines several types of `DataSource` objects:

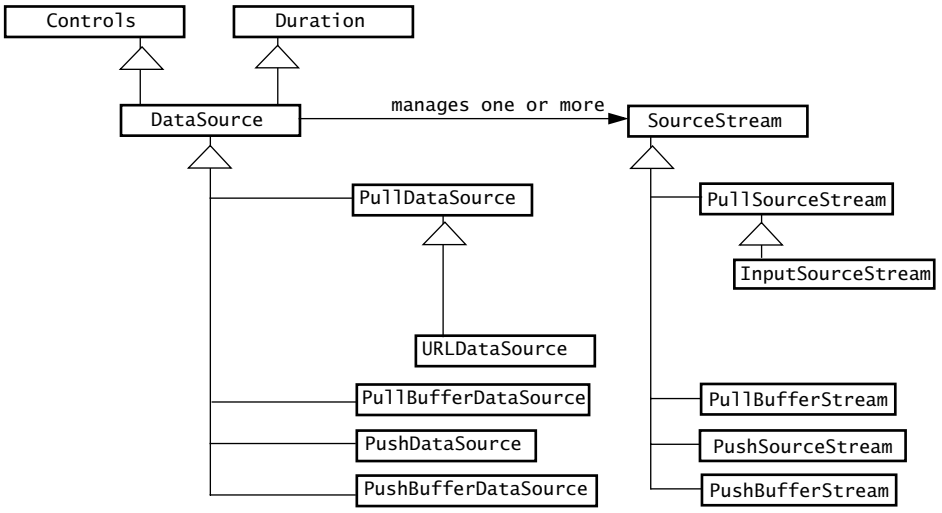


Figure 2-5: JMF data model.

Push and Pull Data Sources

Media data can be obtained from a variety of sources, such as local or network files and live broadcasts. JMF data sources can be categorized according to how data transfer is initiated:

- *Pull Data-Source*—the client initiates the data transfer and controls the flow of data from pull data-sources. Established protocols for this type of data include Hypertext Transfer Protocol (HTTP) and FILE. JMF defines two types of pull data sources: `PullDataSource` and `PullBufferDataSource`, which uses a `Buffer` object as its unit of transfer.
- *Push Data-Source*—the server initiates the data transfer and controls the flow of data from a push data-source. Push data-sources include broadcast media, multicast media, and video-on-demand (VOD). For broadcast data, one protocol is the Real-time Transport Protocol (RTP), under development by the Internet Engineering Task Force (IETF). The MediaBase protocol developed by SGI is one protocol used for VOD. JMF defines two types of push data sources: `PushDataSource` and `PushBufferDataSource`, which uses a `Buffer` object as its unit of transfer.

The degree of control that a client program can extend to the user depends on the type of data source being presented. For example, an MPEG file can

be repositioned and a client program could allow the user to replay the video clip or seek to a new position in the video. In contrast, broadcast media is under server control and cannot be repositioned. Some VOD protocols might support limited user control—for example, a client program might be able to allow the user to seek to a new position, but not fast forward or rewind.

Specialty DataSources

JMF defines two types of specialty data sources, cloneable data sources and merging data sources.

A cloneable data source can be used to create clones of either a pull or push `DataSource`. To create a cloneable `DataSource`, you call the `Manager` `createCloneableDataSource` method and pass in the `DataSource` you want to clone. Once a `DataSource` has been passed to `createCloneableDataSource`, you should only interact with the cloneable `DataSource` and its clones; the original `DataSource` should no longer be used directly.

Cloneable data sources implement the `SourceCloneable` interface, which defines one method, `createClone`. By calling `createClone`, you can create any number of clones of the `DataSource that was` used to construct the cloneable `DataSource`. The clones can be controlled through the cloneable `DataSource` used to create them— when `connect`, `disconnect`, `start`, or `stop` is called on the cloneable `DataSource`, the method calls are propagated to the clones.

The clones don't necessarily have the same properties as the cloneable data source used to create them or the original `DataSource`. For example, a cloneable data source created for a capture device might function as a master data source for its clones—in this case, unless the cloneable data source is used, the clones won't produce any data. If you hook up both the cloneable data source and one or more clones, the clones will produce data at the same rate as the master.

A `MergingDataSource` can be used to combine the `SourceStreams` from several `DataSources` into a single `DataSource`. This enables a set of `DataSources` to be managed from a single point of control—when `connect`, `disconnect`, `start`, or `stop` is called on the `MergingDataSource`, the method calls are propagated to the merged `DataSources`.

To construct a `MergingDataSource`, you call the `Manager` `createMergingDataSource` method and pass in an array that contains the data sources you want to merge. To be merged, all of the `DataSources` must be of the

same type; for example, you cannot merge a `PullDataSource` and a `PushDataSource`. The duration of the merged `DataSource` is the maximum of the merged `DataSource` objects' durations. The `ContentType` is `application/mixed-media`.

Data Formats

The exact media format of an object is represented by a `Format` object. The format itself carries no encoding-specific parameters or global timing information, it describes the format's encoding name and the type of data the format requires.

JMF extends `Format` to define audio- and video-specific formats.

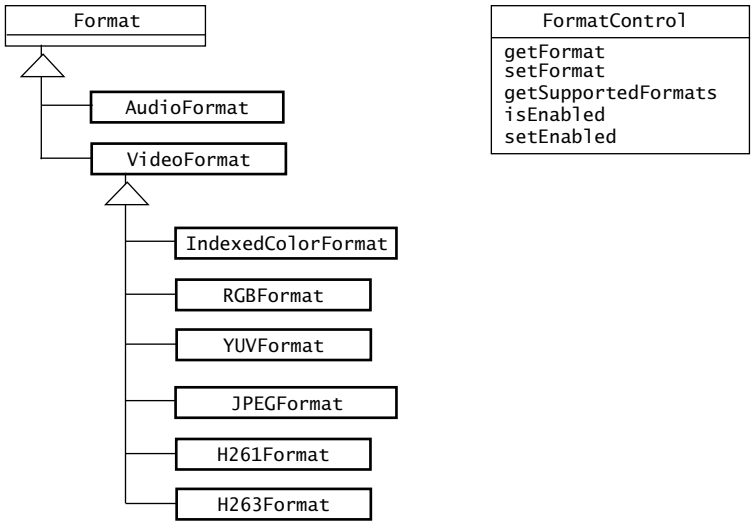


Figure 2-6: JMF media formats.

An `AudioFormat` describes the attributes specific to an audio format, such as sample rate, bits per sample, and number of channels. A `VideoFormat` encapsulates information relevant to video data. Several formats are derived from `VideoFormat` to describe the attributes of common video formats, including:

- `IndexedColorFormat`
- `RGBFormat`
- `YUVFormat`
- `JPEGFormat`
- `H261Format`
- `H263Format`

To receive notification of format changes from a `Controller`, you implement the `ControllerListener` interface and listen for `FormatChangeEvents`. (For more information, see "Responding to Media Events" on page 54.)

Controls

JMF `Control` provides a mechanism for setting and querying attributes of an object. A `Control` often provides access to a corresponding user interface component that enables user control over an object's attributes. Many JMF objects expose `Controls`, including `Controller` objects, `DataSource` objects, `DataSink` objects, and JMF plug-ins.

Any JMF object that wants to provide access to its corresponding `Control` objects can implement the `Controls` interface. `Controls` defines methods for retrieving associated `Control` objects. `DataSource` and `PlugIn` use the `Controls` interface to provide access to their `Control` objects.

Standard Controls

JMF defines the standard `Control` interfaces shown in Figure 2-8:, "JMF controls."

`CachingControl` enables download progress to be monitored and displayed. If a `Player` or `Processor` can report its download progress, it implements this interface so that a progress bar can be displayed to the user.

`GainControl` enables audio volume adjustments such as setting the level and muting the output of a `Player` or `Processor`. It also supports a listener mechanism for volume changes.

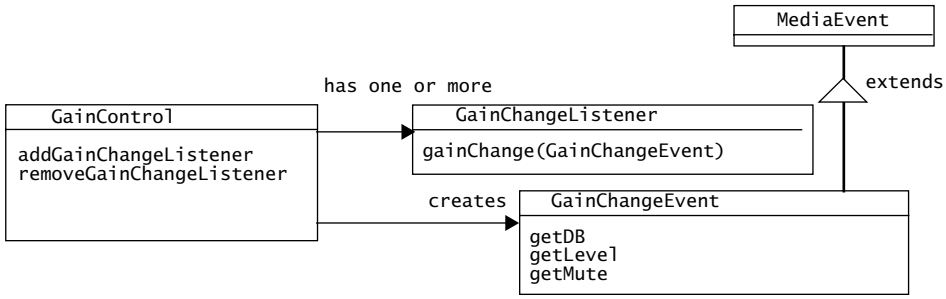


Figure 2-7: Gain control.

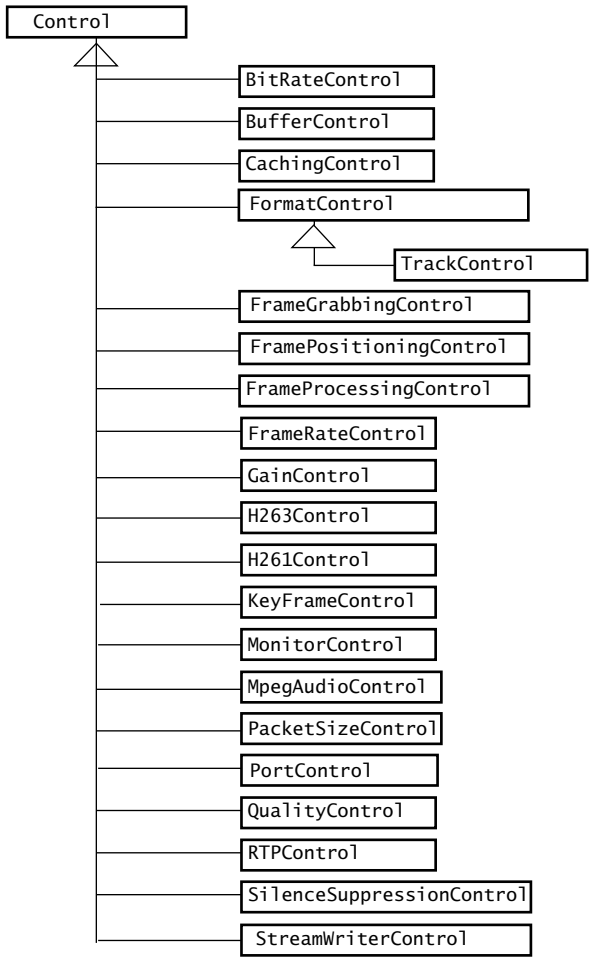


Figure 2-8: JMF controls.

`DataSink` or `Multiplexer` objects that read media from a `DataSource` and write it out to a destination such as a file can implement the `StreamWriterControl` interface. This `Control` enables the user to limit the size of the stream that is created.

`FramePositioningControl` and `FrameGrabbingControl` export frame-based capabilities for `Players` and `Processors`. `FramePositioningControl` enables precise frame positioning within a `Player` or `Processor` object's media stream. `FrameGrabbingControl` provides a mechanism for grabbing a still video frame from the video stream. The `FrameGrabbingControl` can also be supported at the `Renderer` level.

Objects that have a Format can implement the FormatControl interface to provide access to the Format. FormatControl also provides methods for querying and setting the format.

A TrackControl is a type of FormatControl that provides the mechanism for controlling what processing a Processor object performs on a particular track of media data. With the TrackControl methods, you can specify what format conversions are performed on individual tracks and select the Effect, Codec, or Renderer plug-ins that are used by the Processor. (For more information about processing media data, see "Processing Time-Based Media with JMF" on page 71.)

Two controls, PortControl and MonitorControl enable user control over the capture process. PortControl defines methods for controlling the output of a capture device. MonitorControl enables media data to be previewed as it is captured or encoded.

BufferControl enables user-level control over the buffering done by a particular object.

JMF also defines several codec controls to enable control over hardware or software encoders and decoders:

- BitRateControl—used to export the bit rate information for an incoming stream or to control the encoding bit rate. Enables specification of the bit rate in bits per second.
- FrameProcessingControl—enables the specification of frame processing parameters that allow the codec to perform minimal processing when it is falling behind on processing the incoming data.
- FrameRateControl—enables modification of the frame rate.
- H261Control—enables control over the H.261 video codec still-image transmission mode.
- H263Control—enables control over the H.263 video-codec parameters, including support for the unrestricted vector, arithmetic coding, advanced prediction, PB Frames, and error compensation extensions.
- KeyFrameControl—enables the specification of the desired interval between key frames. (The encoder can override the specified key-frame interval if necessary.)
- MpegAudioControl—exports an MPEG audio codec's capabilities and enables the specification of selected MPEG encoding parameters.
- QualityControl—enables specification of a preference in the trade-off

between quality and CPU usage in the processing performed by a codec. This quality hint can have different effects depending on the type of compression. A higher quality setting will result in better quality of the resulting bits, for example better image quality for video.

- `SilenceSuppressionControl`—enables specification of silence suppression parameters for audio codecs. When silence suppression mode is on, an audio encoder does not output any data if it detects silence at its input.

User Interface Components

A `Control` can provide access to a user interface `Component` that exposes its control behavior to the end user. To get the default user interface component for a particular `Control`, you call `getControlComponent`. This method returns an AWT `Component` that you can add to your applet's presentation space or application window.

A `Controller` might also provide access to user interface `Components`. For example, a `Player` provides access to both a visual component and a control panel component—to retrieve these components, you call the `Player` methods `getVisualComponent` and `getControlPanelComponent`.

If you don't want to use the default control components provided by a particular implementation, you can implement your own and use the event listener mechanism to determine when they need to be updated. For example, you might implement your own GUI components that support user interaction with a `Player`. Actions on your GUI components would trigger calls to the appropriate `Player` methods, such as `start` and `stop`. By registering your custom GUI components as `ControllerListeners` for the `Player`, you can also update your GUI in response to changes in the `Player` object's state.

Extensibility

Advanced developers and technology providers can extend JMF functionality in two ways:

- By implementing custom processing components (*plug-ins*) that can be interchanged with the standard processing components used by a JMF `Processor`

- By directly implementing the `Controller`, `Player`, `Processor`, `DataSource`, or `DataSink` interfaces

Implementing a JMF plug-in enables you to customize or extend the capabilities of a `Processor` without having to implement one from scratch. Once a plug-in is registered with JMF, it can be selected as a processing option for any `Processor` that supports the plug-in API. JMF plug-ins can be used to:

- Extend or replace a `Processor` object's processing capability piecewise by selecting the individual plug-ins to be used.
- Access the media data at specific points in the data flow. For example, different `Effect` plug-ins can be used for pre- and post-processing of the media data associated with a `Processor`.
- Process media data outside of a `Player` or `Processor`. For example, you might use a `Demultiplexer` plug-in to get individual audio tracks from a multiplexed media-stream so you could play the tracks through Java Sound.

In situations where an even greater degree of flexibility and control is required, custom implementations of the JMF `Controller`, `Player`, `Processor`, `DataSource`, or `DataSink` interfaces can be developed and used seamlessly with existing implementations. For example, if you have a hardware MPEG decoder, you might want to implement a `Player` that takes input from a `DataSource` and uses the decoder to perform the parsing, decoding, and rendering all in one step. Custom `Players` and `Processors` can also be implemented to integrate media engines such as Microsoft's Media Player, Real Network's RealPlayer, and IBM's HotMedia with JMF.

Note: JMF `Players` and `Processors` are not required to support plug-ins. Plug-ins won't work with JMF 1.0-based `Players` and some `Processor` implementations might choose not to support them. The reference implementation of JMF 2.0 provided by Sun Microsystems, Inc. and IBM Corporation fully supports the plug-in API.

Presentation

In JMF, the presentation process is modeled by the `Controller` interface. `Controller` defines the basic state and control mechanism for an object that controls, presents, or captures time-based media. It defines the phases

that a media controller goes through and provides a mechanism for controlling the transitions between those phases. A number of the operations that must be performed before media data can be presented can be time consuming, so JMF allows programmatic control over when they occur.

A `Controller` posts a variety of controller-specific `MediaEvents` to provide notification of changes in its status. To receive events from a `Controller` such as a `Player`, you implement the `ControllerListener` interface. For more information about the events posted by a `Controller`, see "Controller Events" on page 30.

The JMF API defines two types of `Controllers`: `Players` and `Processors`. A `Player` or `Processor` is constructed for a particular data source and is normally not re-used to present other media data.

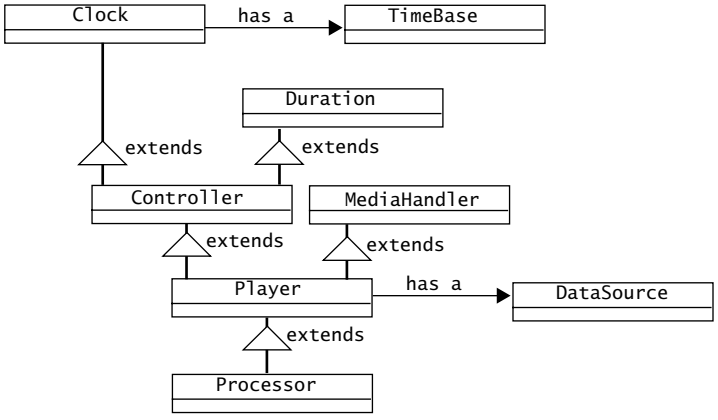


Figure 2-9: JMF controllers.

Players

A `Player` processes an input stream of media data and renders it at a precise time. A `DataSource` is used to deliver the input media-stream to the `Player`.The rendering destination depends on the type of media being presented.

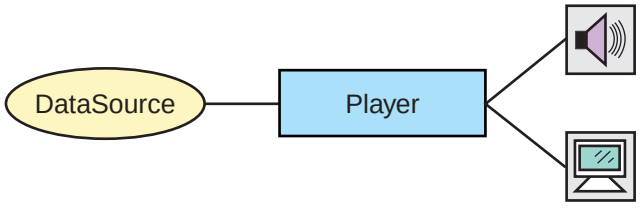


Figure 2-10: JMF player model.

A `Player` does not provide any control over the processing that it performs or how it renders the media data.

`Player` supports standardized user control and relaxes some of the operational restrictions imposed by `Clock` and `Controller`.

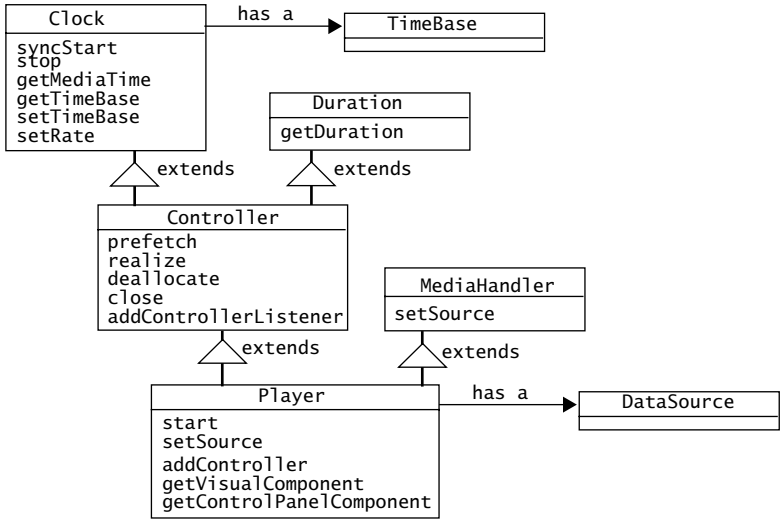


Figure 2-11: JMF players.

Player States

A `Player` can be in one of six states. The `Clock` interface defines the two primary states: *Stopped* and *Started*. To facilitate resource management, `Controller` breaks the *Stopped* state down into five standby states: *Unrealized*, *Realizing*, *Realized*, *Prefetching*, and *Prefetched*.

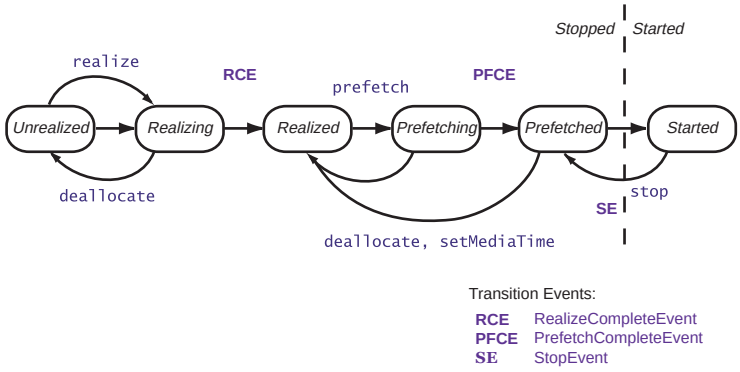


Figure 2-12: Player states.

In normal operation, a `Player` steps through each state until it reaches the *Started* state:

- A `Player` in the *Unrealized* state has been instantiated, but does not yet know anything about its media. When a media `Player` is first created, it is *Unrealized*.
- When `realize` is called, a `Player` moves from the *Unrealized* state into the *Realizing* state. A *Realizing* `Player` is in the process of determining its resource requirements. During realization, a `Player` acquires the resources that it only needs to acquire once. These might include rendering resources other than exclusive-use resources. (Exclusive-use resources are limited resources such as particular hardware devices that can only be used by one `Player` at a time; such resources are acquired during *Prefetching*.) A *Realizing* `Player` often downloads assets over the network.
- When a `Player` finishes *Realizing*, it moves into the *Realized* state. A *Realized* `Player` knows what resources it needs and information about the type of media it is to present. Because a *Realized* `Player` knows how to render its data, it can provide visual components and controls. Its connections to other objects in the system are in place, but it does not own any resources that would prevent another `Player` from starting.
- When `prefetch` is called, a `Player` moves from the *Realized* state into the *Prefetching* state. A *Prefetching* `Player` is preparing to present its media. During this phase, the `Player` preloads its media data, obtains exclusive-use resources, and does whatever else it needs to do to prepare itself to play. *Prefetching* might have to recur if a `Player` object's media presentation is repositioned, or if a change in the `Player` object's rate requires that additional buffers be acquired or alternate processing take place.
- When a `Player` finishes *Prefetching*, it moves into the *Prefetched* state. A *Prefetched* `Player` is ready to be started.
- Calling `start` puts a `Player` into the *Started* state. A *Started* `Player` object's time-base time and media time are mapped and its clock is running, though the `Player` might be waiting for a particular time to begin presenting its media data.

A `Player` posts `TransitionEvents` as it moves from one state to another. The `ControllerListener` interface provides a way for your program to determine what state a `Player` is in and to respond appropriately. For example, when your program calls an asynchronous method on a `Player`

or `Processor`, it needs to listen for the appropriate event to determine when the operation is complete.

Using this event reporting mechanism, you can manage a `Player` object's start latency by controlling when it begins *Realizing* and *Prefetching*. It also enables you to determine whether or not the `Player` is in an appropriate state before calling methods on the `Player`.

Methods Available in Each Player State

To prevent race conditions, not all methods can be called on a `Player` in every state. The following table identifies the restrictions imposed by JMF. If you call a method that is illegal in a `Player` object's current state, the `Player` throws an error or exception.

Method	Unrealized Player	Realized Player	Prefetched Player	Started Player
addController	NotRealizedError	legal	legal	ClockStartedError
deallocate	legal	legal	legal	ClockStartedError
getControlPanelComponent	NotRealizedError	legal	legal	legal
getGainControl	NotRealizedError	legal	legal	legal
getStartLatency	NotRealizedError	legal	legal	legal
getTimeBase	NotRealizedError	legal	legal	legal
getVisualComponent	NotRealizedError	legal	legal	legal
mapToTimeBase	ClockStoppedException	ClockStoppedException	ClockStoppedException	legal
removeController	NotRealizedError	legal	legal	ClockStartedError
setMediaTime	NotRealizedError	legal	legal	legal
setRate	NotRealizedError	legal	legal	legal
setStopTime	NotRealizedError	legal	legal	StopTimeSetError if previously set
setTimeBase	NotRealizedError	legal	legal	ClockStartedError
syncStart	NotPrefetchedError	NotPrefetchedError	legal	ClockStartedError

Table 2-1: Method restrictions for players.

Processors

`Processors` can also be used to present media data. A `Processor` is just a specialized type of `Player` that provides control over what processing is performed on the input media stream. A `Processor` supports all of the same presentation controls as a `Player`.

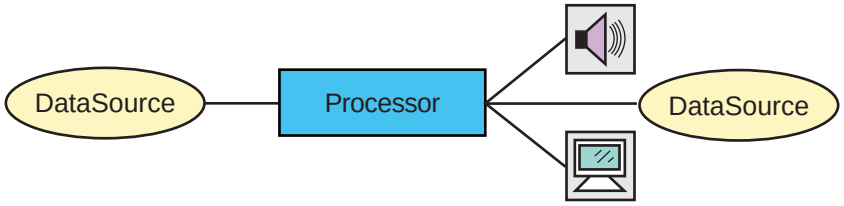


Figure 2-13: JMF processor model.

In addition to rendering media data to presentation devices, a `Processor` can output media data through a `DataSource` so that it can be presented by another `Player` or `Processor`, further manipulated by another `Processor`, or delivered to some other destination, such as a file.

For more information about `Processors`, see "Processing" on page 32.

Presentation Controls

In addition to the standard presentation controls defined by `Controller`, a `Player` or `Processor` might also provide a way to adjust the playback volume. If so, you can retrieve its `GainControl` by calling `getGainControl`. A `GainControl` object posts a `GainChangeEvent` whenever the gain is modified. By implementing the `GainChangeListener` interface, you can respond to gain changes. For example, you might want to update a custom gain control `Component`.

Additional custom `Control` types might be supported by a particular `Player` or `Processor` implementation to provide other control behaviors and expose custom user interface components. You access these controls through the `getControls` method.

For example, the `CachingControl` interface extends `Control` to provide a mechanism for displaying a download progress bar. If a `Player` can report its download progress, it implements this interface. To find out if a `Player` supports `CachingControl`, you can call `getControl(CachingControl)` or use `getControls` to get a list of all the supported `Controls`.

Standard User Interface Components

A `Player` or `Processor` generally provides two standard user interface components, a visual component and a control-panel component.You can access these `Components` directly through the `getVisualComponent` and `getControlPanelComponent` methods.

You can also implement custom user interface components, and use the event listener mechanism to determine when they need to be updated.

Controller Events

The `ControllerEvents` posted by a `Controller` such as a `Player` or `Processor` fall into three categories: change notifications, closed events, and transition events:

- Change notification events such as `RateChangeEvent`, `DurationUpdateEvent`, and `FormatChangeEvent` indicate that some attribute of the `Controller` has changed, often in response to a method call. For example, a `Player` posts a `RateChangeEvent` when its rate is changed by a call to `setRate`.
- `TransitionEvents` allow your program to respond to changes in a `Controller` object's state. A `Player` posts transition events whenever it moves from one state to another. (See "Player States" on page 26 for more information about the states and transitions.)
- `ControllerClosedEvents` are posted by a `Controller` when it shuts down. When a `Controller` posts a `ControllerClosedEvent`, it is no longer usable. A `ControllerErrorEvent` is a special case of `ControllerClosedEvent`. You can listen for `ControllerErrorEvents` so that your program can respond to `Controller` malfunctions to minimize the impact on the user.

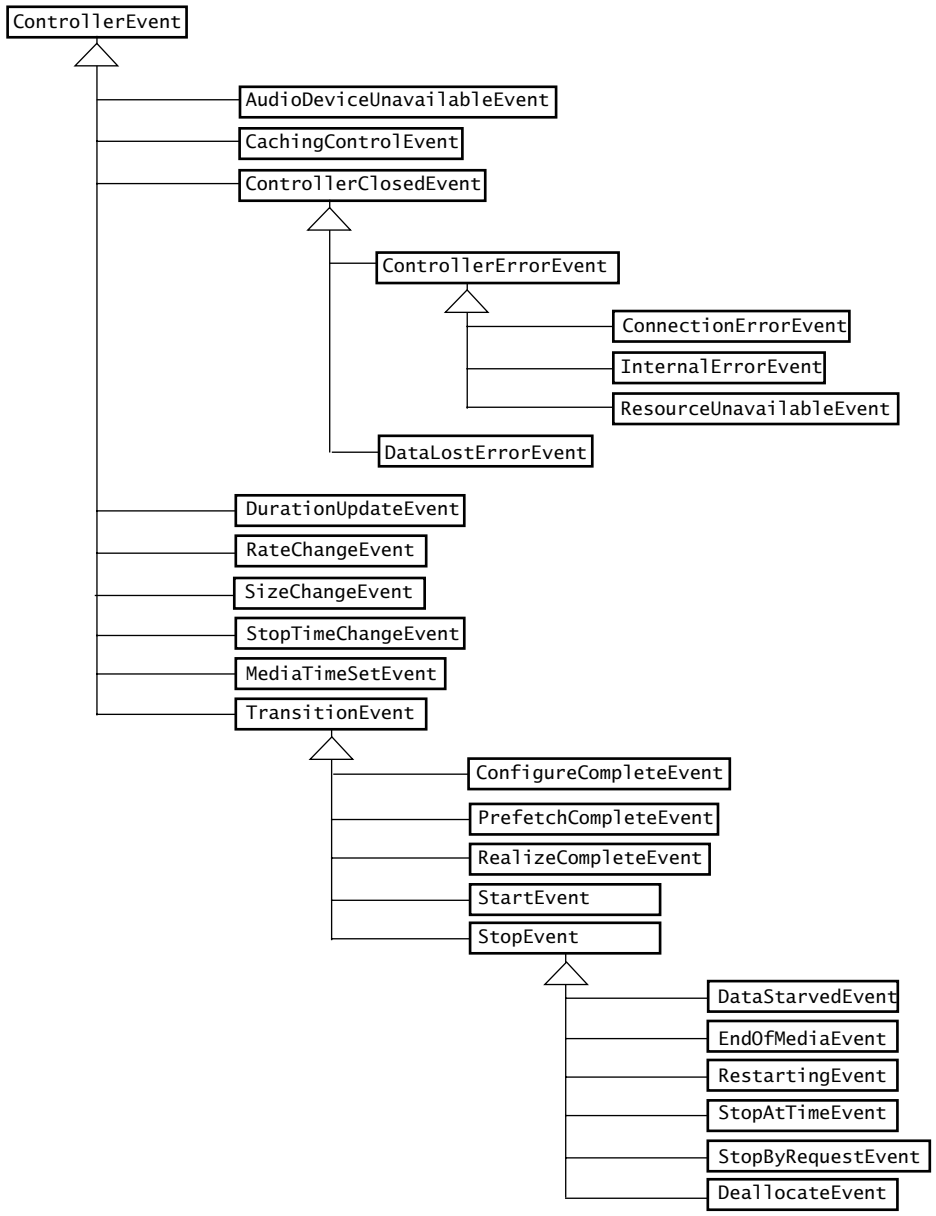


Figure 2-14: JMF events.

Processing

A `Processor` is a `Player` that takes a `DataSource` as input, performs some user-defined processing on the media data, and then outputs the processed media data.

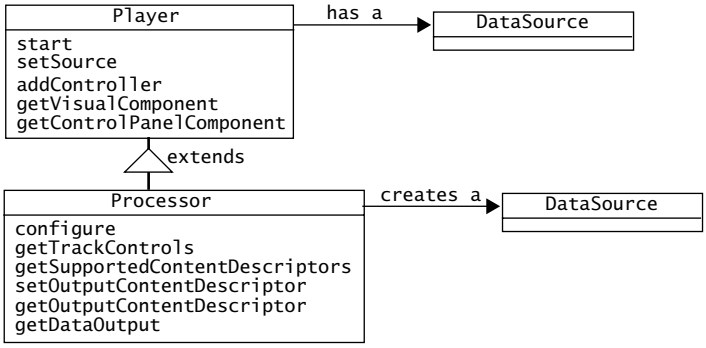


Figure 2-15: JMF processors.

A `Processor` can send the output data to a presentation device or to a `DataSource`. If the data is sent to a `DataSource`, that `DataSource` can be used as the input to another `Player` or `Processor`, or as the input to a `DataSink`.

While the processing performed by a `Player` is predefined by the implementor, a `Processor` allows the application developer to define the type of processing that is applied to the media data. This enables the application of effects, mixing, and compositing in real-time.

The processing of the media data is split into several stages:

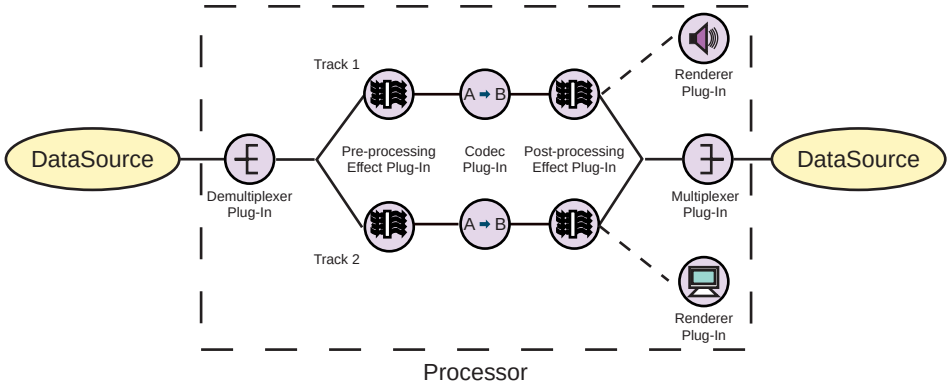


Figure 2-16: Processor stages.

- Demultiplexing is the process of parsing the input stream. If the stream contains multiple tracks, they are extracted and output

separately. For example, a QuickTime file might be demultiplexed into separate audio and video tracks. Demultiplexing is performed automatically whenever the input stream contains multiplexed data.

- Pre-Processing is the process of applying effect algorithms to the tracks extracted from the input stream.
- Transcoding is the process of converting each track of media data from one input format to another. When a data stream is converted from a compressed type to an uncompressed type, it is generally referred to as decoding. Conversely, converting from an uncompressed type to a compressed type is referred to as encoding.
- Post-Processing is the process of applying effect algorithms to decoded tracks.
- Multiplexing is the process of interleaving the transcoded media tracks into a single output stream. For example, separate audio and video tracks might be multiplexed into a single MPEG-1 data stream. You can specify the data type of the output stream with the `Processor` `setOutputContentDescriptor` method.
- Rendering is the process of presenting the media to the user.

The processing at each stage is performed by a separate processing component. These processing components are JMF *plug-in*s. If the `Processor` supports `TrackControls`, you can select which plug-ins you want to use to process a particular track. There are five types of JMF plug-ins:

- `Demultiplexer`—parses media streams such as WAV, MPEG or QuickTime. If the stream is multiplexed, the separate tracks are extracted.
- `Effect`—performs special effects processing on a track of media data.
- `Codec`—performs data encoding and decoding.
- `Multiplexer`—combines multiple tracks of input data into a single interleaved output stream and delivers the resulting stream as an output `DataSource`.
- `Renderer`—processes the media data in a track and delivers it to a destination such as a screen or speaker.

Processor States

A `Processor` has two additional standby states, *Configuring* and *Configured*, which occur before the `Processor` enters the *Realizing* state..

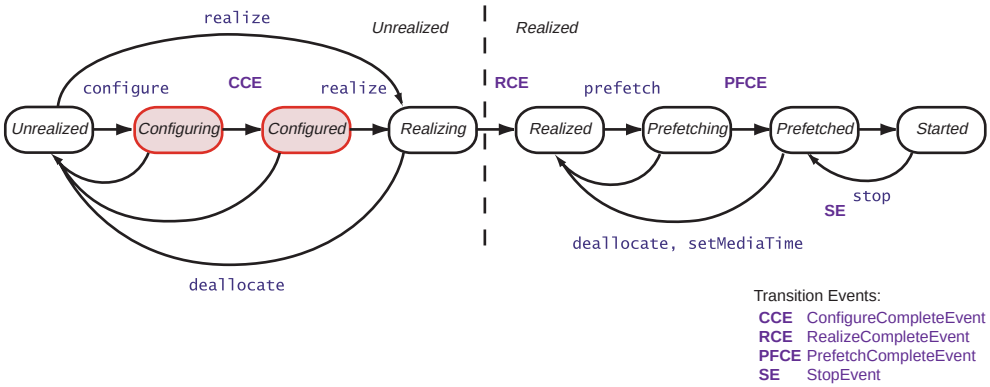


Figure 2-17: Processor states.

- A `Processor` enters the *Configuring* state when `configure` is called. While the `Processor` is in the *Configuring* state, it connects to the `DataSource`, demultiplexes the input stream, and accesses information about the format of the input data.
- The `Processor` moves into the *Configured* state when it is connected to the `DataSource` and data format has been determined. When the `Processor` reaches the *Configured* state, a `ConfigureCompleteEvent` is posted.
- When `Realize` is called, the `Processor` is transitioned to the `Realized` state. Once the `Processor` is *Realized* it is fully constructed.

While a `Processor` is in the *Configured* state, `getTrackControls` can be called to get the `TrackControl` objects for the individual tracks in the media stream. These `TrackControl` objects enable you specify the media processing operations that you want the `Processor` to perform.

Calling `realize` directly on an *Unrealized* `Processor` automatically transitions it through the *Configuring* and *Configured* states to the *Realized* state. When you do this, you cannot configure the processing options through the `TrackControls`—the default `Processor` settings are used.

Calls to the `TrackControl` methods once the `Processor` is in the *Realized* state will typically fail, though some `Processor` implementations might support them.

Methods Available in Each Processor State

Since a `Processor` is a type of `Player`, the restrictions on when methods can be called on a `Player` also apply to `Processors`. Some of the `Processor`-specific methods also are restricted to particular states. The following table shows the restrictions that apply to a `Processor`. If you call a method that is illegal in the current state, the `Processor` throws an error or exception.

Method	Unrealized Processor	Configuring Processor	Configured Processor	Realized Processor
addController	NotRealizedError	NotRealizedError	NotRealizedError	legal
deallocate	legal	legal	legal	legal
getControlPanelComponent	NotRealizedError	NotRealizedError	NotRealizedError	legal
getControls	legal	legal	legal	legal
getDataOutput	NotRealizedError	NotRealizedError	NotRealizedError	legal
getGainControl	NotRealizedError	NotRealizedError	NotRealizedError	legal
getOutputContentDescriptor	NotConfiguredError	NotConfiguredError	legal	legal
getStartLatency	NotRealizedError	NotRealizedError	NotRealizedError	legal
getSupportedContent-Descriptors	legal	legal	legal	legal
getTimeBase	NotRealizedError	NotRealizedError	NotRealizedError	legal
getTrackControls	NotConfiguredError	NotConfiguredError	legal	FormatChange-Exception
getVisualComponent	NotRealizedError	NotRealizedError	NotRealizedError	legal
mapToTimeBase	ClockStoppedException	ClockStoppedException	ClockStoppedException	ClockStopped-Exception
realize	legal	legal	legal	legal
removeController	NotRealizedError	NotRealizedError	NotRealizedError	legal
setOutputContentDescriptor	NotConfiguredError	NotConfiguredError	legal	FormatChange-Exception
setMediaTime	NotRealizedError	NotRealizedError	NotRealizedError	legal
setRate	NotRealizedError	NotRealizedError	NotRealizedError	legal
setStopTime	NotRealizedError	NotRealizedError	NotRealizedError	legal
setTimeBase	NotRealizedError	NotRealizedError	NotRealizedError	legal
syncStart	NotPrefetchedError	NotPrefetchedError	NotPrefetchedError	NotPrefetchedError

Table 2-2: Method restrictions for processors.

Processing Controls

You can control what processing operations the Processor performs on a track through the TrackControl for that track. You call Processor getTrackControls to get the TrackControl objects for all of the tracks in the media stream.

Through a TrackControl, you can explicitly select the Effect, Codec, and Renderer plug-ins you want to use for the track. To find out what options are available, you can query the PlugInManager to find out what plug-ins are installed.

To control the transcoding that's performed on a track by a particular Codec, you can get the Controls associated with the track by calling the TrackControl getControls method. This method returns the codec controls available for the track, such as BitRateControl and QualityControl. (For more information about the codec controls defined by JMF, see "Controls" on page 20.)

If you know the output data format that you want, you can use the setFormat method to specify the Format and let the Processor choose an appropriate codec and renderer. Alternatively, you can specify the output format when the Processor is created by using a ProcessorModel. A ProcessorModel defines the input and output requirements for a Processor. When a ProcessorModel is passed to the appropriate Manager create method, the Manager does its best to create a Processor that meets the specified requirements.

Data Output

The getDataOutput method returns a Processor object's output as a DataSource. This DataSource can be used as the input to another Player or Processor or as the input to a *data sink.* (For more information about data sinks, see "Media Data Storage and Transmission" on page 37.)

A Processor object's output DataSource can be of any type: PushDataSource, PushBufferDataSource, PullDataSource, or PullBufferDataSource.

Not all Processor objects output data—a Processor can render the processed data instead of outputting the data to a DataSource. A Processor that renders the media data is essentially a configurable Player.

Capture

A multimedia capturing device can act as a source for multimedia data delivery. For example, a microphone can capture raw audio input or a digital video capture board might deliver digital video from a camera. Such capture devices are abstracted as `DataSources`. For example, a device that provides timely delivery of data can be represented as a `PushDataSource`. Any type of `DataSource` can be used as a capture `DataSource`: `PushDataSource`, `PushBufferDataSource`, `PullDataSource`, or `PullBufferDataSource`.

Some devices deliver multiple data streams—for example, an audio/video conferencing board might deliver both an audio and a video stream. The corresponding `DataSource` can contain multiple `SourceStreams` that map to the data streams provided by the device.

Media Data Storage and Transmission

A `DataSink` is used to read media data from a `DataSource` and render the media to some destination—generally a destination other than a presentation device. A particular `DataSink` might write data to a file, write data across the network, or function as an RTP broadcaster. (For more information about using a `DataSink` as an RTP broadcaster, see "Transmitting RTP Data With a Data Sink" on page 149.)

Like `Players`, `DataSink` objects are constructed through the `Manager` using a `DataSource`. A `DataSink` can use a `StreamWriterControl` to provide additional control over how data is written to a file. See "Writing Media Data to a File" on page 74 for more information about how `DataSink` objects are used.

Storage Controls

A `DataSink` posts a `DataSinkEvent` to report on its status. A `DataSinkEvent` can be posted with a reason code, or the `DataSink` can post one of the following `DataSinkEvent` subtypes:

- `DataSinkErrorEvent`, which indicates that an error occurred while the `DataSink` was writing data.
- `EndOfStreamEvent`, which indicates that the entire stream has successfully been written.

To respond to events posted by a DataSink, you implement the DataSinkListener interface.

Extensibility

You can extend JMF by implementing custom plug-ins, media handlers, and data sources.

Implementing Plug-Ins

By implementing one of the JMF plug-in interfaces, you can directly access and manipulate the media data associated with a Processor:

- Implementing the Demultiplexer interface enables you to control how individual tracks are extracted from a multiplexed media stream.
- Implementing the Codec interface enables you to perform the processing required to decode compressed media data, convert media data from one format to another, and encode raw media data into a compressed format.
- Implementing the Effect interface enables you to perform custom processing on the media data.
- Implementing the Multiplexer interface enables you to specify how individual tracks are combined to form a single interleaved output stream for a Processor.
- Implementing the Renderer interface enables you to control how data is processed and rendered to an output device.

Note: The JMF Plug-In API is part of the official JMF API, but JMF Players and Processors are not required to support plug-ins. Plug-ins won't work with JMF 1.0-based Players and some Processor implementations might choose not to support them. The reference implementation of JMF 2.0 provided by Sun Microsystems, Inc. and IBM Corporation fully supports the plug-in API.

Custom Codec, Effect, and Renderer plug-ins are available to a Processor through the TrackControl interface. To make a plug-in available to a default Processor or a Processor created with a ProcessorModel, you need to register it with the PlugInManager. Once you've registered your plug-in, it is included in the list of plug-ins returned by the PlugInManager get-

`PlugInList` method and can be accessed by the `Manager` when it constructs a `Processor` object.

Implementing MediaHandlers and DataSources

If the JMF Plug-In API doesn't provide the degree of flexibility that you need, you can directly implement several of the key JMF interfaces: `Controller`, `Player`, `Processor`, `DataSource`, and `DataSink`. For example, you might want to implement a high-performance `Player` that is optimized to present a single media format or a `Controller` that manages a completely different type of time-based media.

The `Manager` mechanism used to construct `Player`, `Processor`, `DataSource`, and `DataSink` objects enables custom implementations of these JMF interfaces to be used seamlessly with JMF. When one of the `create` methods is called, the `Manager` uses a well-defined mechanism to locate and construct the requested object. Your custom class can be selected and constructed through this mechanism once you register a unique package prefix with the `PackageManager` and put your class in the appropriate place in the predefined package hierarchy.

MediaHandler Construction

`Players`, `Processors`, and `DataSinks` are all types of `MediaHandlers`—they all read data from a `DataSource`. A `MediaHandler` is always constructed for a particular `DataSource`, which can be either identified explicitly or with a `MediaLocator`. When one of the `create`*`MediaHandler`* methods is called, `Manager` uses the content-type name obtained from the `DataSource` to find and create an appropriate `MediaHandler` object.

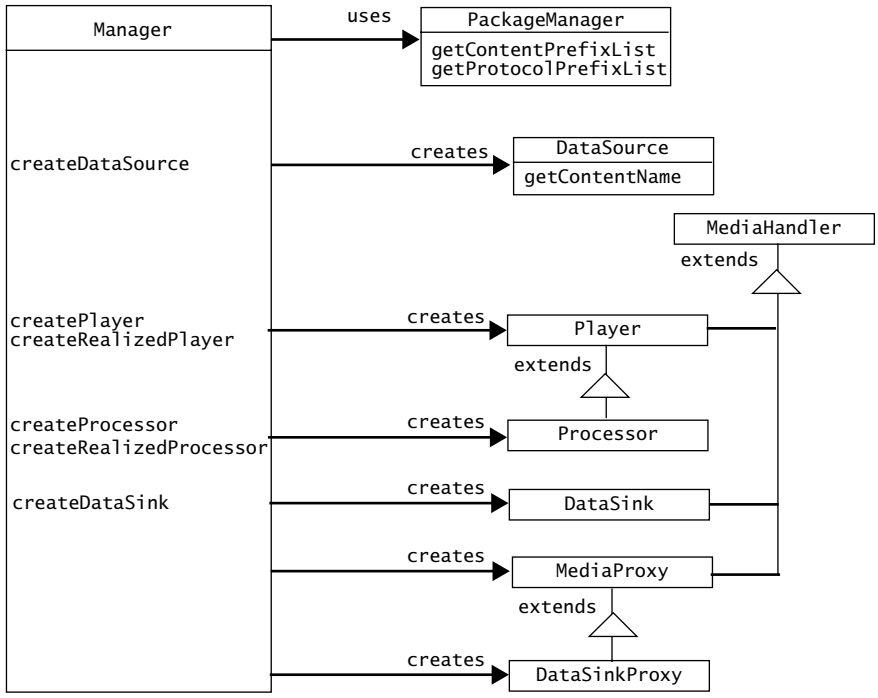


Figure 2-18: JMF media handlers.

JMF also supports another type of `MediaHandler`, `MediaProxy`. A `MediaProxy` processes content from one `DataSource` to create another. Typically, a `MediaProxy` reads a text configuration file that contains all of the information needed to make a connection to a server and obtain media data. To create a `Player` from a `MediaProxy`, `Manager`:

1. Constructs a `DataSource` for the protocol described by the `MediaLocator`
2. Uses the content-type of the `DataSource` to construct a `MediaProxy` to read the configuration file.
3. Gets a new `DataSource` from the `MediaProxy`.
4. Uses the content-type of the new `DataSource` to construct a `Player`.

The mechanism that `Manager` uses to locate and instantiate an appropriate `MediaHandler` for a particular `DataSource` is basically the same for all types of `MediaHandlers`:

- Using the list of installed content package-prefixes retrieved from `PackageManager, Manager` generates a search list of available `MediaHandler` classes.
- `Manager` steps through each class in the search list until it finds a class named `Handler` that can be constructed and to which it can attach the `DataSource`.

When constructing `Players` and `Processors`, `Manager` generates the search list of available handler classes from the list of installed *content package-prefixes* and the content-type name of the `DataSource`. To search for `Players`, `Manager` looks for classes of the form:

```
<content package-prefix>.media.content.<content-type>.Handler
```

To search for `Processors`, `Manager` looks for classes of the form:

```
<content package-prefix>.media.processor.<content-type>.Handler
```

If the located `MediaHandler` is a `MediaProxy`, `Manager` gets a new `DataSource` from the `MediaProxy` and repeats the search process.

If no appropriate `MediaHandler` can be found, the search process is repeated, substituting `unknown` for the content-type name. The `unknown` content type is supported by generic `Players` that are capable of handling a large variety of media types, often in a platform-dependent way.

Because a `DataSink` renders the data it reads from its `DataSource` to an output destination, when a `DataSink` is created the destination must also be taken into account. When constructing `DataSinks`, `Manager` uses the list of content package-prefixes and the protocol from the `MediaLocator` that identifies the destination. For each content package-prefix, `Manager` adds to the search list a class name of the form:

```
<content package-prefix>.media.datasink.protocol.Handler
```

If the located `MediaHandler` is a `DataSink`, `Manager` instantiates it, sets its `DataSource` and `MediaLocator`, and returns the resulting `DataSink` object. If the handler is a `DataSinkProxy`, `Manager` retrieves the content type of the proxy and generates a list of `DataSink` classes that support the protocol of the destination `Medialocator` and the content type returned by the proxy:

```
<content package-prefix>.media.datasink.protocol.<content-type>.Handler
```

The process continues until an appropriate `DataSink` is located or the `Manager` has iterated through all of the content package-prefixes.

DataSource Construction

`Manager` uses the same mechanism to construct `DataSources` that it uses to construct `MediaHandlers`, except that it generates the search list of `DataSource` class names from the list of installed *protocol package-prefixes*.

For each protocol package-prefix, `Manager` adds to the search list a class name of the form:

```
<protocol package-prefix>.media.protocol.<protocol>.DataSource
```

`Manager` steps through each class in the list until it finds a `DataSource` that it can instantiate and to which it can attach the `MediaLocator`.

3

Presenting Time-Based Media with JMF

To present time-based media such as audio or video with JMF, you use a `Player`. Playback can be controlled programmatically, or you can display a control-panel component that enables the user to control playback interactively. If you have several media streams that you want to play, you need to use a separate `Player` for each one. to play them in sync, you can use one of the `Player` objects to control the operation of the others.

A `Processor` is a special type of `Player` that can provide control over how the media data is processed before it is presented. Whether you're using a basic `Player` or a more advanced `Processor` to present media content, you use the same methods to manage playback. For information about how to control what processing is performed by a `Processor`, see "Processing Time-Based Media with JMF" on page 71.

The `MediaPlayer` bean is a Java Bean that encapsulates a JMF player to provide an easy way to present media from an applet or application. The `MediaPlayer` bean automatically constructs a new `Player` when a different media stream is selected, which makes it easier to play a series of media clips or allow the user to select which media clip to play. For information about using the `MediaPlayer` bean, see "Presenting Media with the MediaPlayer Bean" on page 66

Controlling a Player

To play a media stream, you need to construct a `Player` for the stream, configure the `Player` and prepare it to run, and then start the `Player` to begin playback.

Creating a Player

You create a Player indirectly through the media Manager. To display the Player, you get the Player object's components and add them to your applet's presentation space or application window.

When you need to create a new Player, you request it from the Manager by calling createPlayer or createProcessor. The Manager uses the media URL or MediaLocator that you specify to create an appropriate Player. A URL can only be successfully constructed if the appropriate corresponding URLStreamHandler is installed. MediaLocator doesn't have this restriction.

Blocking Until a Player is Realized

Many of the methods that can be called on a Player require the Player to be in the *Realized* state. One way to guarantee that a Player is *Realized* when you call these methods is to use the Manager createRealizedPlayer method to construct the Player. This method provides a convenient way to create and realize a Player in a single step. When this method is called, it blocks until the Player is *Realized*. Manager provides an equivalent createRealizeProcessor method for constructing a *Realized* Processor.

Note: Be aware that blocking until a Player or Processor is *Realized* can produce unsatisfactory results. For example, if createRealizedPlayer is called in an applet, Applet.start and Applet.stop will not be able to interrupt the construction process.

Using a ProcessorModel to Create a Processor

A Processor can also be created using a ProcessorModel. The ProcessorModel defines the input and output requirements for the Processor and the Manager does its best to create a Processor that meets these requirements. To create a Processor using a ProcessorModel, you call the Manager.createRealizedProcessor method. Example 3-1 creates a *Realized* Processor that can produce IMA4-encoded stereo audio tracks with a 44.1 kHz sample rate and a 16-bit sample size.

Example 3-1: Constructing a Processor with a ProcessorModel.

```
AudioFormat afs[] = new AudioFormat[1];
afs[0] = new AudioFormat("ima4", 44100, 16, 2);
Manager.createRealizedProcessor(new ProcessorModel(afs, null));
```

Since the `ProcessorModel` does not specify a source URL in this example, `Manager` implicitly finds a capture device that can capture audio and then creates a `Processor` that can encode that into IMA4.

Note that when you create a *Realized* `Processor` with a `ProcessorModel` you will not be able to specify processing options through the `Processor` object's `TrackControls`. For more information about specifying processing options for a Processor, see "Processing Time-Based Media with JMF" on page 71.

Displaying Media Interface Components

A `Player` generally has two types of user interface components, a visual component and a control-panel component. Some `Player` implementations can display additional components, such as volume controls and download-progress bars.

Displaying a Visual Component

A visual component is where a `Player` presents the visual representation of its media, if it has one. Even an audio `Player` might have a visual component, such as a waveform display or animated character.

To display a `Player` object's visual component, you:

1. Get the component by calling `getVisualComponent`.
2. Add it to the applet's presentation space or application window.

You can access the `Player` object's display properties, such as its *x* and *y* coordinates, through its visual component. The layout of the `Player` components is controlled through the AWT layout manager.

Displaying a Control Panel Component

A `Player` often has a control panel that allows the user to control the media presentation. For example, a `Player` might be associated with a set of buttons to start, stop, and pause the media stream, and with a slider control to adjust the volume.

Every Player provides a default control panel. To display the default control panel:

1. Call getControlPanelComponent to get the Component.
2. Add the returned Component to your applet's presentation space or application window.

If you prefer to define a custom user-interface, you can implement custom GUI Components and call the appropriate Player methods in response to user actions. If you register the custom components as ControllerListeners, you can also update them when the state of the Player changes.

Displaying a Gain-Control Component

Player implementations that support audio gain adjustments implement the GainControl interface. GainControl provides methods for adjusting the audio volume, such as setLevel and setMute. To display a GainControl Component if the Player provides one, you:

1. Call getGainControl to get the GainControl from the Player. If the Player returns null, it does not support the GainControl interface.
2. Call getControlComponent on the returned GainControl.
3. Add the returned Component to your applet's presentation space or application window.

Note that getControls does not return a Player object's GainControl. You can only access the GainControl by calling getGainControl.

Displaying Custom Control Components

Many Players have other properties that can be managed by the user. For example, a video Player might allow the user to adjust brightness and contrast, which are not managed through the Player interface. You can find out what custom controls a Player supports by calling the getControls method.

For example, you can call getControls to determine if a Player supports the CachingControl interface.

Example 3-2: Using `getControls` to find out what `Controls` are supported.

```
Control[] controls = player.getControls();
   for (int i = 0; i < controls.length; i++) {
      if (controls[i] instanceof CachingControl) {
         cachingControl = (CachingControl) controls[i];
      }
   }
```

Displaying a Download-Progress Component

The `CachingControl` interface is a special type of `Control` implemented by `Players` that can report their download progress. A `CachingControl` provides a default progress-bar component that is automatically updated as the download progresses. To use the default progress bar in an applet:

1. Implement the `ControllerListener` interface and listen for `CachingControlEvents` in `controllerUpdate`.
2. The first time you receive a `CachingControlEvent`:
   a. Call `getCachingControl` on the event to get the caching control.
   b. Call `getProgressBar` on the `CachingControl` to get the default progress bar component.
   c. Add the progress bar component to your applet's presentation space.
3. Each time you receive a `CachingControlEvent`, check to see if the download is complete. When `getContentProgress` returns the same value as `getContentLength`, remove the progress bar.

The `Player` posts a `CachingControlEvent` whenever the progress bar needs to be updated. If you implement your own progress bar component, you can listen for this event and update the download progress whenever `CachingControlEvent` is posted.

Setting the Playback Rate

The `Player` object's rate determines how media time changes with respect to time-base time; it defines how many units a `Player` object's media time advances for every unit of time-base time. The `Player` object's rate can be thought of as a temporal scale factor. For example, a rate of 2.0 indicates

that media time passes twice as fast as the time-base time when the Player is started.

In theory, a Player object's rate could be set to any real number, with negative rates interpreted as playing the media in reverse. However, some media formats have dependencies between frames that make it impossible or impractical to play them in reverse or at non-standard rates.

To set the rate, you call setRate and pass in the temporal scale factor as a float value. When setRate is called, the method returns the rate that is actually set, even if it has not changed. Players are only guaranteed to support a rate of 1.0.

Setting the Start Position

Setting a Player object's media time is equivalent to setting a read position within a media stream. For a media data source such as a file, the media time is bounded; the maximum media time is defined by the end of the media stream.

To set the media time you call setMediaTime and pass in a Time object that represents the time you want to set.

Frame Positioning

Some Players allow you to seek to a particular frame of a video. This enables you to easily set the start position to the beginning of particular frame without having to specify the exact media time that corresponds to that position. Players that support frame positioning implement the FramePositioningControl.

To set the frame position, you call the FramePositioningControl seek method. When you seek to a frame, the Player object's media time is set to the value that corresponds to the beginning of that frame and a MediaTimeSetEvent is posted.

Some Players can convert between media times and frame positions. You can use the FramePositioningControl mapFrameToTime and mapTimeToFrame methods to access this information, if it's available. (Players that support FramePositioningControl are not required to export this information.) Note that there is not a one-to-one correspondence between media times and frames —a frame has a duration, so several different media times might map to the same frame. (See "Getting the Media Time" on page 53 for more information.)

Preparing to Start

Most media `Players` cannot be started instantly. Before the `Player` can start, certain hardware and software conditions must be met. For example, if the `Player` has never been started, it might be necessary to allocate buffers in memory to store the media data. Or, if the media data resides on a network device, the `Player` might have to establish a network connection before it can download the data. Even if the `Player` has been started before, the buffers might contain data that is not valid for the current media position.

Realizing and Prefetching a Player

JMF breaks the process of preparing a `Player` to start into two phases, *Realizing* and *Prefetching*. *Realizing* and *Prefetching* a `Player` before you start it minimizes the time it takes the `Player` to begin presenting media when `start` is called and helps create a highly-responsive interactive experience for the user. Implementing the `ControllerListener` interface allows you to control when these operations occur.

Note: `Processor` introduces a third phase to the preparation process called *Configuring*. During this phase, `Processor` options can be selected to control how the `Processor` manipulates the media data. For more information, see "Selecting Track Processing Options" on page 72.

You call `realize` to move the `Player` into the *Realizing* state and begin the realization process. You call `prefetch` to move the `Player` into the *Prefetching* state and initiate the prefetching process. The `realize` and `prefetch` methods are asynchronous and return immediately. When the `Player` completes the requested operation, it posts a `RealizeCompleteEvent` or `PrefetchCompleteEvent`. "Player States" on page 26 describes the operations that a `Player` performs in each of these states.

A `Player` in the *Prefetched* state is prepared to start and its start-up latency cannot be further reduced. However, setting the media time through `setMediaTime` might return the `Player` to the *Realized* state and increase its start-up latency.

Keep in mind that a *Prefetched* `Player` ties up system resources. Because some resources, such as sound cards, might only be usable by one program at a time, having a `Player` in the *Prefetched* state might prevent other `Players` from starting.

Determining the Start Latency

To determine how much time is required to start a `Player`, you can call `getStartLatency`. For `Players` that have a variable start latency, the return value of `getStartLatency` represents the maximum possible start latency. For some media types, `getStartLatency` might return `LATENCY_UNKNOWN`.

The start-up latency reported by `getStartLatency` might differ depending on the `Player` object's current state. For example, after a `prefetch` operation, the value returned by `getStartLatency` is typically smaller. A `Controller` that can be added to a `Player` will return a useful value once it is *Prefetched*. (For more information, see "Using a Player to Synchronize Controllers" on page 57.)

Starting and Stopping the Presentation

The `Clock` and `Player` interfaces define the methods for starting and stopping presentation.

Starting the Presentation

You typically start the presentation of media data by calling `start`. The `start` method tells the `Player` to begin presenting media data as soon as possible. If necessary, `start` prepares the `Player` to start by performing the realize and prefetch operations. If `start` is called on a *Started* `Player`, the only effect is that a `StartEvent` is posted in acknowledgment of the method call.

Clock defines a `syncStart` method that can be used for synchronization. See "Synchronizing Multiple Media Streams" on page 56 for more information.

To start a `Player` at a specific point in a media stream:

1. Specify the point in the media stream at which you want to start by calling `setMediaTime`.
2. Call `start` on the `Player`.

Stopping the Presentation

There are four situations in which the presentation will stop:

- When the `stop` method is called

- When the specified stop time is reached
- When there's no more media data to present
- When the media data is being received too slowly for acceptable playback

When a `Player` is stopped, its media time is frozen if the source of the media can be controlled. If the `Player` is presenting streamed media, it might not be possible to freeze the media time. In this case, only the receipt of the media data is stopped—the data continues to be streamed and the media time continues to advance.

When a *Stopped* `Player` is restarted, if the media time was frozen, presentation resumes from the stop time. If media time could not be frozen when the `Player` was stopped, reception of the stream resumes and playback begins with the newly-received data.

To stop a `Player` immediately, you call the `stop` method. If you call `stop` on a *Stopped* `Player`, the only effect is that a `StopByRequestEvent` is posted in acknowledgment of the method call.

Stopping the Presentation at a Specified Time

You can call `setStopTime` to indicate when a `Player` should stop. The `Player` stops when its media time passes the specified stop time. If the `Player` object's rate is positive, the `Player` stops when the media time becomes greater than or equal to the stop time. If the `Player` object's rate is negative, the `Player` stops when the media time becomes less than or equal to the stop time. The `Player` stops immediately if its current media time is already beyond the specified stop time.

For example, assume that a `Player` object's media time is 5.0 and `setStopTime` is called to set the stop time to 6.0. If the `Player` object's rate is positive, media time is increasing and the `Player` will stop when the media time becomes greater than or equal to 6.0. However, if the `Player` object's rate is negative, it is playing in reverse and the `Player` will stop immediately because the media time is already beyond the stop time. (For more information about `Player` rates, see "Setting the Playback Rate" on page 47.)

You can always call `setStopTime` on a *Stopped* `Player`. However, you can only set the stop time on a *Started* `Player` if the stop time is not currently

set. If the *Started* `Player` already has a stop time, `setStopTime` throws an error.

You can call `getStopTime` to get the currently scheduled stop time. If the clock has no scheduled stop time, `getStopTime` returns `Clock.RESET`. To remove the stop time so that the `Player` continues until it reaches end-of-media, call `setStopTime(Clock.RESET)`.

Releasing Player Resources

The `deallocate` method tells a `Player` to release any exclusive resources and minimize its use of non-exclusive resources. Although buffering and memory management requirements for `Players` are not specified, most `Players` allocate buffers that are large by the standards of Java objects. A well-implemented `Player` releases as much internal memory as possible when `deallocate` is called.

The `deallocate` method can only be called on a *Stopped* `Player`. To avoid `ClockStartedErrors`, you should call `stop` before you call `deallocate`. Calling `deallocate` on a `Player` in the *Prefetching* or *Prefetched* state returns it to the *Realized* state. If `deallocate` is called while the `Player` is realizing, the `Player` posts a `DeallocateEvent` and returns to the *Unrealized* state. (Once a `Player` has been realized, it can never return to the *Unrealized* state.)

You generally call `deallocate` when the `Player` is not being used. For example, an applet should call `deallocate` as part of its `stop` method. By calling `deallocate`, the program can maintain references to the `Player`, while freeing other resources for use by the system as a whole. (JMF does not prevent a *Realized* `Player` that has formerly been *Prefetched* or *Started* from maintaining information that would allow it to be started up more quickly in the future.)

When you are finished with a `Player` (or any other `Controller`) and are not going to use it anymore, you should call `close`. The `close` method indicates that the `Controller` will no longer be used and can shut itself down. Calling `close` releases all of the resources that the `Controller` was using and causes it to cease all activity. When a `Controller` is closed, it posts a `ControllerClosedEvent`. A closed `Controller` cannot be reopened and invoking methods on a closed `Controller` might generate errors.

Querying a Player

A `Player` can provide information about its current parameters, including its rate, media time, and duration.

Getting the Playback Rate

To get a `Player` object's current rate, you call `getRate`. Calling `getRate` returns the playback rate as a float value.

Getting the Media Time

To get a `Player` object's current media time, you call `getMediaTime`. Calling `getMediaTime` returns the current media time as a `Time` object. If the `Player` is not presenting media data, this is the point from which media presentation will commence.

Note that there is not a one-to-one correspondence between media times and frames. Each frame is presented for a certain period of time, and the media time continues to advance during that period.

For example, imagine you have a slide show `Player` that displays each slide for 5 seconds—the `Player` essentially has a frame rate of 0.2 frames per second.

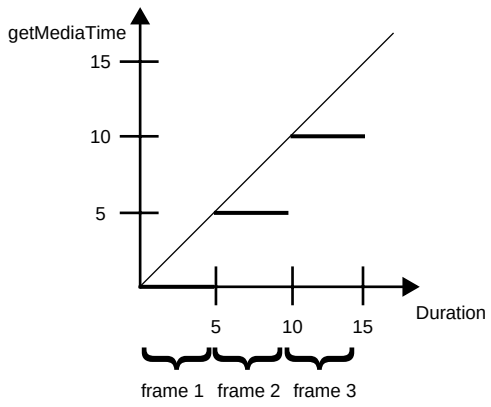


Figure 3-1: Frame duration and media time.

If you start the `Player` at time 0.0, while the first frame is displayed, the media time advances from 0.0 to 5.0. If you start at time 2.0, the first frame is displayed for 3 seconds, until time 5.0 is reached.

Getting the Time-Base Time

You can get a `Player` object's current time-base time by getting the `Player` object's `TimeBase` and calling `getTime`:

```
myCurrentTBTime = player1.getTimeBase().getTime();
```

When a `Player` is running, you can get the time-base time that corresponds to a particular media time by calling `mapToTimeBase`.

Getting the Duration of the Media Stream

Since programs often need to know how long a particular media stream will run, all `Controllers` implement the `Duration` interface. This interface defines a single method, `getDuration`. The duration represents the length of time that a media object would run, if played at the default rate of 1.0. A media stream's duration is only accessible through a `Player`.

If the duration can't be determined when `getDuration` is called, `DURATION_UNKNOWN` is returned. This can happen if the `Player` has not yet reached a state where the duration of the media source is available. At a later time, the duration might be available and a call to `getDuration` would return the duration value. If the media source does not have a defined duration, as in the case of a live broadcast, `getDuration` returns `DURATION_UNBOUNDED`.

Responding to Media Events

`ControllerListener` is an asynchronous interface for handling events generated by `Controller` objects. Using the `ControllerListener` interface enables you to manage the timing of potentially time-consuming `Player` operations such as prefetching.

Implementing the ControllerListener Interface

To implement the `ControllerListener` interface, you need to:

1. Implement the `ControllerListener` interface in a class.
2. Register that class as a listener by calling `addControllerListener` on the `Controller` that you want to receive events from.

When a `Controller` posts an event, it calls `controllerUpdate` on each registered listener.

Typically, `controllerUpdate` is implemented as a series of if-else statements.

Example 3-3:Implementing `controllerUpdate`.

```
if (event instanceof EventType){
...
} else if (event instanceof OtherEventType){
...
}
```

This filters out the events that you are not interested in. If you have registered as a listener with multiple `Controllers`, you also need to determine which `Controller` posted the event. `ControllerEvents` come "stamped" with a reference to their source that you can access by calling `getSource`.

When you receive events from a `Controller`, you might need to do some additional processing to ensure that the `Controller` is in the proper state before calling a control method. For example, before calling any of the methods that are restricted to *Stopped* `Players`, you should check the `Player` object's target state by calling `getTargetState`. If `start` has been called, the `Player` is considered to be in the *Started* state, though it might be posting transition events as it prepares the `Player` to present media.

Some types of `ControllerEvents` contain additional state information. For example, the `StartEvent` and `StopEvent` classes each define a method that allows you to retrieve the media time at which the event occurred.

Using ControllerAdapter

`ControllerAdapter` is a convenience class that implements `ControllerListener` and can be easily extended to respond to particular `Events`. To implement the `ControllerListener` interface using `ControllerAdapter`, you need to:

1. Subclass `ControllerAdapter` and override the event methods for the events that you're interested in.
2. Register your `ControllerAdapter` class as a listener for a particular `Controller` by calling `addControllerListener`.

When a Controller posts an event, it calls controllerUpdate on each registered listener. ControllerAdapter automatically dispatches the event to the appropriate event method, filtering out the events that you're not interested in.

For example, the following code extends a ControllerAdapter with a JDK 1.1 anonymous inner-class to create a self-contained Player that is automatically reset to the beginning of the media and deallocated when the Player reaches the end of the media.

Example 3-4: Using ControllerAdapter.

```
player.addControllerListener(new ControllerAdapter() {
   public void endOfMedia(EndOfMediaEvent e) {
      Controller controller = e.getSource();
      controller.stop();
      controller.setMediaTime(new Time(0));
      controller.deallocate();
   }
})
```

If you register a single ControllerAdapter as a listener for multiple Players, in your event method implementations you need to determine which Player generated the event. You can call getSource to determine where a ControllerEvent originated.

Synchronizing Multiple Media Streams

To synchronize the playback of multiple media streams, you can synchronize the Players by associating them with the same TimeBase. To do this, you use the getTimeBase and setTimeBase methods defined by the Clock interface. For example, you could synchronize player1 with player2 by setting player1 to use player2's time base:

```
player1.setTimeBase(player2.getTimeBase());
```

When you synchronize Players by associating them with the same TimeBase, you must still manage the control of each Player individually. Because managing synchronized Players in this way can be complicated, JMF provides a mechanism that allows a Player to assume control over any other Controller. The Player manages the states of these Controllers automatically, allowing you to interact with the entire group through a

single point of control. For more information, see See "Using a Player to Synchronize Controllers".

Using a Player to Synchronize Controllers

Synchronizing `Players` directly using `syncStart` requires that you carefully manage the states of all of the synchronized `Players`. You must control each one individually, listening for events and calling control methods on them as appropriate. Even with only a few `Players`, this quickly becomes a difficult task. Through the `Player` interface, JMF provides a simpler solution: a `Player` can be used to manage the operation of any `Controller`.

When you interact with a managing `Player`, your instructions are automatically passed along to the managed `Controllers` as appropriate. The managing `Player` takes care of the state management and synchronization for all of the other `Controllers`.

This mechanism is implemented through the `addController` and `removeController` methods. When you call `addController` on a `Player`, the `Controller` you specify is added to the list of `Controllers` managed by the `Player`. Conversely, when you call `removeController`, the specified `Controller` is removed from the list of managed `Controllers`.

Typically, when you need to synchronize `Players` or other `Controllers`, you should use this `addController` mechanism. It is simpler, faster, and less error-prone than attempting to manage synchronized `Players` individually.

When a `Player` assumes control of a `Controller`:

- The `Controller` assumes the `Player` object's time base.
- The `Player` object's duration becomes the longer of the `Controller` object's duration and its own. If multiple `Controllers` are placed under a `Player` object's control, the `Player` object's duration is set to longest duration.
- The `Player` object's start latency becomes the longer of the `Controller` object's start latency and its own. If multiple `Controllers` are placed under a `Player` object's control, the `Player` object's start latency is set to the longest latency.

A managing `Player` only posts completion events for asynchronous methods after each of its managed `Controllers` have posted the event. The

managing Player reposts other events generated by the Controllers as appropriate.

Adding a Controller

You use the addController method to add a Controller to the list of Controllers managed by a particular Player. To be added, a Controller must be in the *Realized* state; otherwise, a NotRealizedError is thrown. Two Players cannot be placed under control of each other. For example, if player1 is placed under the control of player2, player2 cannot be placed under the control of player1 without first removing player1 from player2's control.

Once a Controller has been added to a Player, do not call methods directly on the managed Controller. To control a managed Controller, you interact with the managing Player.

To have player2 assume control of player1, call:

```
player2.addController(player1);
```

Controlling Managed Controllers

To control the operation of a group of Controllers managed by a particular Player, you interact directly with the managing Player.

For example, to prepare all of the managed Controllers to start, call prefetch on the managing Player. Similarly, when you want to start them, call start on the managing Player. The managing Player makes sure that all of the Controllers are *Prefetched*, determines the maximum start latency among the Controllers, and calls syncStart to start them, specifying a time that takes the maximum start latency into account.

When you call a Controller method on the managing Player, the Player propagates the method call to the managed Controllers as appropriate. Before calling a Controller method on a managed Controller, the Player ensures that the Controller is in the proper state. The following table describes what happens to the managed Controllers when you call control methods on the managing Player.

Function	Stopped Player	Started Player
`setMediaTime`	Invokes `setMediaTime` on all managed `Controllers`.	Stops all managed `Controllers`, invokes `setMediaTime`, and restarts `Controllers`.
`setRate`	Invokes `setRate` on all managed `Controllers`. Returns the actual rate that was supported by all `Controllers` and set.	Stops all managed `Controllers`, invokes `setRate`, and restarts `Controllers`. Returns the actual rate that was supported by all `Controllers` and set.
`start`	Ensures all managed `Controllers` are *Prefetched* and invokes `syncStart` on each of them, taking into account their start latencies.	Depends on the `Player` implementation. `Player` might immediately post a `StartEvent`.
`realize`	The managing `Player` immediately posts a `RealizeCompleteEvent`. To be added, a `Controller` must already be realized.	The managing `Player` immediately posts a `RealizeCompleteEvent`. To be added, a `Controller` must already be realized.
`prefetch`	Invokes `prefetch` on all managed `Controllers`.	The managing `Player` immediately posts a `PrefetchCompleteEvent`, indicating that all managed `Controllers` are *Prefetched*.
`stop`	No effect.	Invokes `stop` on all managed `Controllers`.
`deallocate`	Invokes `deallocate` on all managed `Controllers`.	It is illegal to call `deallocate` on a *Started* `Player`.
`setStopTime`	Invokes `setStopTime` on all managed `Controllers`. (`Player` must be *Realized*.)	Invokes `setStopTime` on all managed `Controllers`. (Can only be set once on a *Started* `Player`.)
`syncStart`	Invokes `syncStart` on all managed `Controllers`.	It is illegal to call `syncStart` on a *Started* `Player`.
`close`	Invokes `close` on all managed `Controllers`.	It is illegal to call `close` on a *Started* `Player`.

Table 3-1: Calling control methods on a managing player.

Removing a Controller

You use the `removeController` method to remove a `Controller` from the list of controllers managed by a particular `Player`.

To have player2 release control of player1, call:

```
player2.removeController(player1);
```

Synchronizing Players Directly

In a few situations, you might want to manage the synchronization of multiple Player objects yourself so that you can control the rates or media times independently. If you do this, you must:

1. Register as a listener for each synchronized Player.
2. Determine which Player object's time base is going to be used to drive the other Player objects and set the time base for the synchronized Player objects. Not all Player objects can assume a new time base. For example, if one of the Player objects you want to synchronize has a push data-source, that Player object's time base must be used to drive the other Player objects.
3. Set the rate for all of the Players. If a Player cannot support the rate you specify, it returns the rate that was used. (There is no mechanism for querying the rates that a Player supports.)
4. Synchronize the states of all of the Player objects. (For example, stop all of the players.)
5. Synchronize the operation of the Player objects:
   - Set the media time for each Player.
   - Prefetch each Player.
   - Determine the maximum start latency among the synchronized Player objects.
   - Start the Player objects by calling syncStart with a time that takes into account the maximum latency.

You must listen for transition events for all of the Player objects and keep track of which ones have posted events. For example, when you prefetch the Player objects, you need to keep track of which ones have posted PrefetchComplete events so that you can be sure all of them are *Prefetched* before calling syncStart. Similarly, when you request that the synchronized Player objects stop at a particular time, you need to listen

for the stop event posted by each `Player` to determine when all of them have actually stopped.

In some situations, you need to be careful about responding to events posted by the synchronized `Player` objects. To be sure of the state of all of the `Player` objects, you might need to wait at certain stages for all of them to reach the same state before continuing.

For example, assume that you are using one `Player` to drive a group of synchronized `Player` objects. A user interacting with that `Player` sets the media time to 10, starts the `Player`, and then changes the media time to 20. You then:

1. Pass along the first `setMediaTime` call to all of the synchronized `Player` objects.
2. Call `prefetch` on each `Player` to prepare them to start.
3. Call `stop` on each `Player` when the second set media time request is received.
4. Call `setMediaTime` on each `Player` with the new time.
5. Restart the prefetching operation.
6. When all of the `Player` objects have been prefetched, start them by calling `syncStart`, taking into account their start latencies.

In this case, just listening for `PrefetchComplete` events from all of the `Player` objects before calling `syncStart` isn't sufficient. You can't tell whether those events were posted in response to the first or second prefetch operation. To avoid this problem, you can block when you call `stop` and wait for all of the `Player` objects to post stop events before continuing. This guarantees that the next `PrefetchComplete` events you receive are the ones that you are really interested in.

Example: Playing an MPEG Movie in an Applet

The sample program `PlayerApplet` demonstrates how to create a `Player` and present an MPEG movie from within a Java applet. This is a general example that could easily be adapted to present other types of media streams.

The Player object's visual presentation and its controls are displayed within the applet's presentation space in the browser window. If you create a Player in a Java application, you are responsible for creating the window to display the Player object's components.

Note: While PlayerApplet illustrates the basic usage of a Player, it does not perform the error handling necessary in a real applet or application. For a more complete sample suitable for use as a template, see "JMF Applet" on page 173.

Overview of PlayerApplet

The APPLET tag is used to invoke PlayerApplet in an HTML file. The WIDTH and HEIGHT fields of the HTML APPLET tag determine the dimensions of the applet's presentation space in the browser window. The PARAM tag identifies the media file to be played.

Example 3-5: Invoking PlayerApplet.

```
<APPLET CODE=ExampleMedia.PlayerApplet
WIDTH=320 HEIGHT=300>
<PARAM NAME=FILE VALUE="sample2.mpg">
</APPLET>
```

When a user opens a web page containing PlayerApplet, the applet loads automatically and runs in the specified presentation space, which contains the Player object's visual component and default controls. The Player starts and plays the MPEG movie once. The user can use the default Player controls to stop, restart, or replay the movie. If the page containing the applet is closed while the Player is playing the movie, the Player automatically stops and frees the resources it was using.

To accomplish this, PlayerApplet extends Applet and implements the ControllerListener interface. PlayerApplet defines five methods:

- init—creates a Player for the file that was passed in through the PARAM tag and registers PlayerApplet as a controller listener so that it can observe media events posted by the Player. (This causes the PlayerApplet controllerUpdate method to be called whenever the Player posts an event.)
- start—starts the Player when PlayerApplet is started.
- stop—stops and deallocates the Player when PlayerApplet is stopped.
- destroy—closes the Player when PlayerApplet is removed.

- `controllerUpdate`—responds to `Player` events to display the `Player` object's components.

Example 3-6: `PlayerApplet`.

```
import java.applet.*;
import java.awt.*;
import java.net.*;
import javax.media.*;

public class PlayerApplet extends Applet implements ControllerListener {
   Player player = null;
   public void init() {
      setLayout(new BorderLayout());
      String mediaFile = getParameter("FILE");
      try {
         URL mediaURL = new URL(getDocumentBase(), mediaFile);
         player = Manager.createPlayer(mediaURL);
         player.addControllerListener(this);
      }
      catch (Exception e) {
         System.err.println("Got exception "+e);
      }
   }
   public void start() {
      player.start();
   }
   public void stop() {
      player.stop();
      player.deallocate();
   }
   public void destroy() {
      player.close();
   }
   public synchronized void controllerUpdate(ControllerEvent event) {
      if (event instanceof RealizeCompleteEvent) {
         Component comp;
         if ((comp = player.getVisualComponent()) != null)
            add ("Center", comp);
         if ((comp = player.getControlPanelComponent()) != null)
            add ("South", comp);
         validate();
      }
   }
}
```

Initializing the Applet

When a Java applet starts, its `init` method is invoked automatically. You override `init` to prepare your applet to be started. `PlayerApplet` performs four tasks in `init`:

1. Retrieves the applet's FILE parameter.
2. Uses the FILE parameter to locate the media file and build a `URL` object that describes that media file.
3. Creates a `Player` for the media file by calling `Manager.createPlayer`.
4. Registers the applet as a controller listener with the new `Player` by calling `addControllerListener`. Registering as a listener causes the `PlayerApplet` `controllerUpdate` method to be called automatically whenever the `Player` posts a media event. The `Player` posts media events whenever its state changes. This mechanism allows you to control the `Player` object's transitions between states and ensure that the `Player` is in a state in which it can process your requests. (For more information, see "Player States" on page 26.)

Example 3-7: Initializing `PlayerApplet`.

```
public void init() {
   setLayout(new BorderLayout());
   // 1. Get the FILE parameter.
   String mediaFile = getParameter("FILE");
   try {
      // 2. Create a URL from the FILE parameter. The URL
      // class is defined in java.net.
      URL mediaURL = new URL(getDocumentBase(), mediaFile);
      // 3. Create a player with the URL object.
      player = Manager.createPlayer(mediaURL);
      // 4. Add PlayerApplet as a listener on the new player.
      player.addControllerListener(this);
   }
   catch (Exception e) {
      System.err.println("Got exception "+e);
   }
}
```

Controlling the Player

The `Applet` class defines `start` and `stop` methods that are called automatically when the page containing the applet is opened and closed. You override these methods to define what happens each time your applet starts and stops.

`PlayerApplet` implements `start` to start the `Player` whenever the applet is started.

Example 3-8: Starting the `Player` in `PlayerApplet`.

```
public void start() {
   player.start();
}
```

Similarly, `PlayerApplet` overrides `stop` to stop and deallocate the `Player`:

Example 3-9: Stopping the `Player` in `PlayerApplet`.

```
public void stop() {
   player.stop();
   player.deallocate();
}
```

Deallocating the `Player` releases any resources that would prevent another `Player` from being started. For example, if the `Player` uses a hardware device to present its media, `deallocate` frees that device so that other `Players` can use it.

When an applet exits, `destroy` is called to dispose of any resources created by the applet. `PlayerApplet` overrides `destroy` to close the `Player`. Closing a `Player` releases all of the resources that it's using and shuts it down permanently.

Example 3-10: Destroying the `Player` in `PlayerApplet`.

```
    public void destroy() {
       player.close();
    }
```

Responding to Media Events

PlayerApplet registers itself as a ControllerListener in its init method so that it receives media events from the Player. To respond to these events, PlayerApplet implements the controllerUpdate method, which is called automatically when the Player posts an event.

PlayerApplet responds to one type of event, RealizeCompleteEvent. When the Player posts a RealizeCompleteEvent, PlayerApplet displays the Player object's components.

Example 3-11: Responding to media events in PlayerApplet.

```
public synchronized void controllerUpdate(ControllerEvent event)
{
   if (event instanceof RealizeCompleteEvent) {
      Component comp;
      if ((comp = player.getVisualComponent()) != null)
         add ("Center", comp);
      if ((comp = player.getControlPanelComponent()) != null)
         add ("South", comp);
      validate();
   }
}
```

A Player object's user-interface components cannot be displayed until the Player is *Realized*; an *Unrealized* Player doesn't know enough about its media stream to provide access to its user-interface components. PlayerApplet waits for the Player to post a RealizeCompleteEvent and then displays the Player object's visual component and default control panel by adding them to the applet container. Calling validate triggers the layout manager to update the display to include the new components.

Presenting Media with the MediaPlayer Bean

Using the MediaPlayer Java Bean (javax.media.bean.playerbean.MediaPlayer) is the simplest way to present media streams in your applets and applications. MediaPlayer encapsulates a full-featured JMF Player in a Java Bean. You can either use the MediaPlayer bean's default controls or customize its control Components.

One key advantage to using the MediaPlayer bean is that it automatically constructs a new Player when a different media stream is selected for

playback. This makes it easy to play a series of media clips or enable the user to select the media clip that they want to play.

A `MediaPlayer` bean has several properties that you can set, including the media source:

Property	Type	Default	Description
Show control panel	Boolean	Yes	Controls whether or not the video control panel is visible.
Loop	Boolean	Yes	Controls whether or not the media clip loops continuously.

Media location	String	N/A	The location of the media clip to be played. It can be an URL or a relative address. For example:
			`file:///e:/video/media/Sample1.mov`
			`http://webServer/media/Sample1.mov`
			`media/Sample1.mov`

Show caching control	Boolean	No	Controls whether or not the download-progress bar is displayed.
Fixed Aspect Ratio	Boolean	Yes	Controls whether or not the media's original fixed aspect ratio is maintained.
Volume	int	3	Controls the audio volume.

Table 3-2: Media bean properties.

To play a media clip with the `MediaPlayer` bean:

1. Construct an instance of `MediaPlayer`:

```
MediaPlayer mp1 = new javax.media.bean.playerbean.MediaPlayer();
```

2. Set the location of the clip you want to play:

```
mp1.setMediaLocation(new java.lang.String("file:///E:/jvideo/media/
Sample1.mov"));
```

3. Start the `MediaPlayer`:

```
mp1.start();
```

You can stop playback by calling `stop` on the `MediaPlayer`:

```
mp1.stop();
```

By setting up the `MediaPlayer` in your Applet's `init` method and starting the `MediaPlayer` in your Applet's `start` method, you can automatically begin media presentation when the Applet is loaded. You should call `stop` in the Applet's `stop` method so that playback halts when the Applet is stopped.

Alternatively, you can display the `MediaPlayer` bean's default control panel or provide custom controls to allow the user to control the media presentation. If you provide custom controls, call the appropriate `MediaPlayer` control and properties methods when the user interacts with the controls. For example, if you provide a custom Start button in your Applet, listen for the mouse events and call `start` when the user clicks on the button.

Presenting RTP Media Streams

You can present streaming media with a JMF `Player` constructed through the `Manager` using a `MediaLocator` that has the parameters of an RTP session. For more information about streaming media and RTP, see "Working with Real-Time Media Streams" on page 109.

When you use a `MediaLocator` to construct a `Player` for an RTP session, only the first RTP stream that's detected in the session can be presented—`Manager` creates a `Player` for the first stream that's detected in the RTP session. For information about playing multiple RTP streams from the same session, see "Receiving and Presenting RTP Media Streams" on page 129.

Note: JMF-compliant implementations are not required to support the RTP APIs in `javax.media.rtp`, `javax.media.rtp.event`, and `javax.media.rtp.rtcp`. The reference implementations of JMF provided by Sun Microsystems, Inc. and IBM Corporation fully support these APIs.

Example 3-12: Creating a Player for an RTP session.

```
String url= "rtp://224.144.251.104:49150/audio/1";

MediaLocator mrl= new MediaLocator(url);

if (mrl == null) {
    System.err.println("Can't build MRL for RTP");
    return false;
}

// Create a player for this rtp session
try {
    player = Manager.createPlayer(mrl);
} catch (NoPlayerException e) {
    System.err.println("Error:" + e);
    return false;
} catch (MalformedURLException e) {
    System.err.println("Error:" + e);
    return false;
} catch (IOException e) {
    System.err.println("Error:" + e);
    return false;
}
```

When data is detected on the session, the `Player` posts a `RealizeCompleteEvent`. By listening for this event, you can determine whether or not any data has arrived and if the `Player` is capable of presenting any data. Once the `Player` posts this event, you can retrieve its visual and control components.

Listening for RTP Format Changes

When a `Player` posts a `FormatChangeEvent`, it can indicate that a payload change has occurred. `Player` objects constructed with a `MediaLocator` automatically process payload changes. In most cases, this processing involves constructing a new `Player` to handle the new format. Programs that present RTP media streams need to listen for `FormatChangeEvents` so that they can respond if a new `Player` is created.

When a `FormatChangeEvent` is posted, check whether or not the `Player` object's control and visual components have changed. If they have, a new `Player` has been constructed and you need to remove references to the old `Player` object's components and get the new `Player` object's components.

Example 3-13: Listening for RTP format changes.

```
public synchronized void controllerUpdate(ControllerEvent ce) {
    if (ce instanceof FormatChangeEvent) {
        Dimension vSize = new Dimension(320,0);
        Component oldVisualComp = visualComp;

        if ((visualComp = player.getVisualComponent()) != null) {
            if (oldVisualComp != visualComp) {
                if (oldVisualComp != null) {
                    oldVisualComp.remove(zoomMenu);
                }
                framePanel.remove(oldVisualComp);

                vSize = visualComp.getPreferredSize();
                vSize.width = (int)(vSize.width * defaultScale);
                vSize.height = (int)(vSize.height * defaultScale);

                framePanel.add(visualComp);

                visualComp.setBounds(0,
                                     0,
                                     vSize.width,
                                     vSize.height);
                addPopupMenu(visualComp);
            }
        }

        Component oldComp = controlComp;

        controlComp = player.getControlPanelComponent();

        if (controlComp != null)
        {
            if (oldComp != controlComp)
            {
                framePanel.remove(oldComp);
                framePanel.add(controlComp);

                if (controlComp != null) {
                    int prefHeight = controlComp
                                     .getPreferredSize()
                                     .height;

                    controlComp.setBounds(0,
                                          vSize.height,
                                          vSize.width,
                                          prefHeight);
                }
            }
        }
    }
}
```

4

Processing Time-Based Media with JMF

A `Processor` can be used as a programmable `Player` that enables you to control the decoding and rendering process. A `Processor` can also be used as a capture processor that enables you to control the encoding and multiplexing of the captured media data.

You can control what processing is performed by a `Processor` several different ways:

- Use a `ProcessorModel` to construct a `Processor` that has certain input and output characteristics.
- Use the `TrackControl` `setFormat` method to specify what format conversions are performed on individual tracks.
- Use the `Processor` `setOutputContentDescriptor` method to specify the multiplexed data format of the `Processor` object's output.
- Use the `TrackControl` `setCodecChain` method to select the `Effect` or `Codec` plug-ins that are used by the `Processor`.
- Use the `TrackControl` `setRenderer` method to select the `Renderer` plug-in used by the `Processor`.

Note: Some high-performance or light-weight `Processor` implementations might choose not to support the selection of processing options so that they can provide a highly-optimized JMF presentation solution. The reference implementation of JMF 2.0 provided by Sun Microsystems, Inc. and IBM Corporation fully supports the selection of processing options through `TrackControl` objects and `setOutputContentDescriptor`.

Configuring a Processor

In addition to the *Realizing* and *Prefetching* phases that any `Player` moves through as it prepares to start, a `Processor` also goes through a *Configuring* phase. You call `configure` to move an *Unrealized* `Processor` into the *Configuring* state.

While in the *Configuring* state, a `Processor` gathers the information it needs to construct `TrackControl` objects for each track. When it's finished, it moves into the *Configured* state and posts a `ConfigureCompleteEvent`. Once a `Processor` is *Configured*, you can set its output format and `TrackControl` options. When you're finished specifying the processing options, you call `realize` to move the `Processor` into the *Realizing* state and begin the realization process.

Once a `Processor` is *Realized*, further attempts to modify its processing options are not guaranteed to work. In most cases, a `FormatChangeException` will be thrown.

Selecting Track Processing Options

To select which plug-ins are used to process each track in the media stream, you:

1. Call the `PlugInManager.getPlugInList` method to determine what plug-ins are available. The `PlugInManager` returns a list of plug-ins that match the specified input and output formats and plug-in type.
2. Call `getTrackControls` on the `Processor` to get a `TrackControl` for each track in the media stream. The `Processor` must in the *Configured* state before you call `getTrackControls`.
3. Call the `TrackControl` `setCodecChain` or `setRenderer` methods to specify the plug-ins that you want to use for each track.

When you use `setCodecChain` to specify the codec and effect plug-ins for a `Processor`, the order in which the plug-ins actually appear in the processing chain is determined by the input and output formats each plug-in supports.

To control the transcoding that's performed on a track by a particular `Codec`, you can use the codec controls associated with the track. To get the codec controls, you call the `TrackControl` `getControls` method. This

returns all of the `Controls` associated with the track, including codec controls such as `H263Control`, `QualityControl`, and `MPEGAudioControl`. (For a list of the codec controls defined by JMF, see "Standard Controls" on page 20.)

Converting Media Data from One Format to Another

You can select the format for a particular track through the `TrackControl` for that track:

1. Call `getTrackControls` on the `Processor` to get a `TrackControl` for each track in the media stream. The `Processor` must be in the *Configured* state before you call `getTrackControls`.
2. Use the `TrackControl` `setFormat` method to specify the format to which you want to convert the selected track.

Specifying the Output Data Format

You can use the `Processor` `setContentDescriptor` method to specify the format of the data output by the `Processor`. You can get a list of supported data formats by calling `getSupportedContentDescriptors`.

You can also select the output format that you want by using a `ProcessorModel` to create the `Processor`. (See "Using a ProcessorModel to Create a Processor" on page 44 for more information.)

Specifying an output data format automatically selects the default processing options for this format, overriding the previous processing options selected through the `TrackControls`. Setting the output data format to `null` causes the media data to be rendered instead of output to the `Processor` object's output `DataSource`.

Specifying the Media Destination

You can specify a destination for the media stream by selecting a particular `Renderer` for a track through its `TrackControl`, or by using the output from a `Processor` as the input to a particular `DataSink`. You can also use the `Processor` output as the input to another `Player` or `Processor` that has a different destination.

Selecting a Renderer

To select the Renderer that you want to use, you:

1. Call getTrackControls on the Processor to get a TrackControl for each track in the media stream. The Processor must in the *Configured* state before you call getTrackControls.
2. Call the TrackControl setRenderer method to specify the Renderer plug-in.

Writing Media Data to a File

You can use a DataSink to read media data from Processor object's output DataSource and render the data to a file.

1. Get the output DataSource from the Processor by calling getDataOutput.
2. Construct a file writer DataSink by calling Manager.createDataSink. Pass in the output DataSource and a MediaLocator that specifies the location of the file to which you want to write.
3. Call open on the DataSink to open the file.
4. Call start on the DataSink to begin writing data.

The format of the data written to the specified file is controlled through the Processor. By default, a Processor outputs raw data. To change the content type of a Processor object's output DataSource, you use the setContentDescriptor method.

Example 4-1: Using a DataSink to write media data to a file.

```
DataSink sink;
MediaLocator dest = new MediaLocator(file://newfile.wav);
try{
     sink = Manager.createDataSink(p.getDataOutput(), dest);
     sink.open();
     sink.start();
} catch (Exception) {}
```

A Processor can enable user control over the maximum number of bytes that it can write to its destination by implementing the StreamWriterControl. You find out if a Processor provides a StreamWriterControl by call-

ing `getControl("javax.media.datasink.StreamWriterControl")` on the `Processor`.

Connecting a Processor to another Player

The output from a `Processor` can be used as the input to another `Player`. To get the output from a `Processor`, you call `getDataOutput`, which returns a `DataSource`. This `DataSource` can in turn be used to construct a `Player` or `Processor` through the `Manager`.

Using JMF Plug-Ins as Stand-alone Processing Modules

JMF Plug-ins can also be used outside of the JMF framework. You can instantiate the plug-in directly and call its processing method to perform the processing operation.

You might want to do this to encode or decode a media stream, or convert a stream from one format to another.

5

Capturing Time-Based Media with JMF

You can use JMF to capture media data from a capture device such as a microphone or video camera. Captured media data can be processed and rendered or stored for future use.

To capture media data, you:

1. Locate the capture device you want to use by querying the `CaptureDeviceManager`.
2. Get a `CaptureDeviceInfo` object for the device.
3. Get a `MediaLocator` from the `CaptureDeviceInfo` object and use it to create a `DataSource`.
4. Create a `Player` or `Processor` using the `DataSource`.
5. Start the `Player` or `Processor` to begin the capture process.

When you use a capture `DataSource` with a `Player`, you can only render the captured media data. To explicitly process or store the captured media data, you need to use a `Processor`.

Accessing Capture Devices

You access capture devices through the `CaptureDeviceManager`. The `CaptureDeviceManager` is the central registry for all of the capture devices available to JMF. You can get a list of the available capture devices by calling the `CaptureDeviceManager.getDeviceList` method.

Each device is represented by a CaptureDeviceInfo object. To get the CaptureDeviceInfo object for a particular device, you call CaptureDeviceManager.getDevice:

```
CaptureDeviceInfo deviceInfo = CaptureDeviceManager.getDevice("deviceName");
```

Capturing Media Data

To capture media data from a particular device, you need to get the device's MediaLocator from its CaptureDeviceInfo object. You can either use this MediaLocator to construct a Player or Processor directly, or use the MediaLocator to construct a DataSource that you can use as the input to a Player or Processor. To initiate the capture process, you start the Player or Processor.

Allowing the User to Control the Capture Process

A capture device generally has a set of implementation-specific attributes that can be used to control the device. Two control types are defined to enable programmatic control of capture devices: PortControl and MonitorControl. You access these controls by calling getControl on the capture DataSource and passing in the name of the control you want.

A PortControl provides a way to select the port from which data will be captured. A MonitorControl provides a means for displaying the device's capture monitor.

Like other Control objects, if there's a visual component that corresponds to the PortControl or MonitorControl, you can get it by calling getControlComponent. Adding the Component to your applet or application window will enable users to interact with the capture control.

You can also display the standard control-panel component and visual component associated with the Player or Processor you're using.

Example 5-1: Displaying GUI components for a processor.

```
Component controlPanel, visualComponent;
if ((controlPanel = p.getControlPanelComponent()) != null)
    add(controlPanel);
if ((visualComponent = p.getVisualComponent()) != null)
    add(visualComponent);
```

Storing Captured Media Data

If you want to save captured media data to a file, you need to use a `Processor` instead of a `Player`. You use a `DataSink` to read media data from `Processor` object's output data source and render the data to a file.

1. Get the output `DataSource` from the `Processor` by calling `getDataOutput`.
2. Construct a file writer `DataSink` by calling `Manager.createDataSink`. Pass in the output `DataSource` and a `MediaLocator` that specifies the location of the file to which you want to write.
3. Call `open` on the `DataSink` to open the file.
4. Call `start` on the `DataSink`.
5. Call `start` on the `Processor` to begin capturing data.
6. Wait for an `EndOfMediaEvent`, a particular media time, or a user event.
7. Call `stop` on the `Processor` to end the data capture.
8. Call `close` on the `Processor`.
9. When the `Processor` is closed and the `DataSink` posts an `EndOfStreamEvent`, call `close` on the `DataSink`.

Example 5-2: Saving captured media data to a file.

```
DataSink sink;
MediaLocator dest = new MediaLocator("file://newfile.wav");
try {
     sink = Manager.createDataSink(p.getDataOutput(), dest);
     sink.open();
     sink.start();
} catch (Exception) {}
```

Example: Capturing and Playing Live Audio Data

To capture live audio data from a microphone and present it, you need to:

1. Get the `CaptureDeviceInfo` object for the microphone.
2. Create a `Player` using the `MediaLocator` retrieved from the `CaptureDe-`

viceInfo object. (You can create the Player by calling createPlayer(MediaLocator) or create a DataSource with the MediaLocator and use createPlayer(DataSource) to construct the Player.)

Example 5-3: Capturing and playing audio from a microphone.

```
// Get the CaptureDeviceInfo for the live audio capture device
Vector deviceList = CaptureDeviceManager.getDeviceList(new
                        AudioFormat("linear", 44100, 16, 2));
if (deviceList.size() > 0)
     di = (CaptureDeviceInfo)deviceList.firstElement();
else
// Exit if we can't find a device that does linear, 44100Hz, 16 bit,
// stereo audio.
    System.exit(-1);

// Create a Player for the capture device:
try{
    Player p = Manager.createPlayer(di.getLocator());
} catch (IOException e) {
} catch (NoPlayerException e) {}
```

Example: Writing Captured Audio Data to a File

You can write captured media data to a file using a DataSink. To capture and store audio data, you need to:

1. Get a CaptureDeviceInfo object for the audio capture device.
2. Create a Processor using the MediaLocator retrieved from the CaptureDeviceInfo object.
3. Get the output DataSource from the Processor.
4. Create a MediaLocator for the file where you want to write the captured data.
5. Create a file writer DataSink using the output DataSource.
6. Start the file writer and the Processor.

This example uses a helper class, `StateHelper.java`, to manage the state of the `Processor`. The complete source for `StateHelper` is included in the appendix starting on page 179.

Example 5-4: Writing captured audio to a file with a `DataSink`. (1 of 2)

```
CaptureDeviceInfo di = null;
Processor p = null;
StateHelper sh = null;
Vector deviceList = CaptureDeviceManager.getDeviceList(new
          AudioFormat(AudioFormat.LINEAR, 44100, 16, 2));
if (deviceList.size() > 0)
    di = (CaptureDeviceInfo)deviceList.firstElement();
else
    // Exit if we can't find a device that does linear,
    // 44100Hz, 16 bit,
    // stereo audio.
    System.exit(-1);
try {
    p = Manager.createProcessor(di.getLocator());
    sh = new StateHelper(p);
} catch (IOException e) {
    System.exit(-1);
} catch (NoProcessorException e) {
    System.exit(-1);
}
// Configure the processor
if (!sh.configure(10000))
    System.exit(-1);
// Set the output content type and realize the processor
p.setContentDescriptor(new
           FileTypeDescriptor(FileTypeDescriptor.WAVE));
if (!sh.realize(10000))
    System.exit(-1);
// get the output of the processor
DataSource source = p.getDataOutput();
// create a File protocol MediaLocator with the location of the
// file to which the data is to be written
MediaLocator dest = new MediaLocator("file://foo.wav");
// create a datasink to do the file writing & open the sink to
// make sure we can write to it.
DataSink filewriter = null;
try {
    filewriter = Manager.createDataSink(source, dest);
    filewriter.open();
} catch (NoDataSinkException e) {
    System.exit(-1);
} catch (IOException e) {
    System.exit(-1);
} catch (SecurityException e) {
    System.exit(-1);
}
```

Example 5-4: Writing captured audio to a file with a `DataSink`. (2 of 2)

```
// if the Processor implements StreamWriterControl, we can
// call setStreamSizeLimit
// to set a limit on the size of the file that is written.
StreamWriterControl swc = (StreamWriterControl)
    p.getControl("javax.media.control.StreamWriterControl");
//set limit to 5MB
if (swc != null)
    swc.setStreamSizeLimit(5000000);

// now start the filewriter and processor
try {
    filewriter.start();
} catch (IOException e) {
    System.exit(-1);
}
// Capture for 5 seconds
sh.playToEndOfMedia(5000);
sh.close();
// Wait for an EndOfStream from the DataSink and close it...
filewriter.close();
```

Example: Encoding Captured Audio Data

You can configure a `Processor` to transcode captured media data before presenting, transmitting, or storing the data. To encode captured audio data in the IMA4 format before saving it to a file:

1. Get the `MediaLocator` for the capture device and construct a `Processor`.
2. Call `configure` on the `Processor`.
3. Once the `Processor` is in the *Configured* state, call `getTrackControls`.
4. Call `setFormat` on each track until you find one that can be converted to IMA4. (For `setFormat` to succeed, appropriate codec plug-ins must be available to perform the conversion.)
5. Realize the `Processor` and use it's output `DataSource` to construct a `DataSink` to write the data to a file.

Example 5-5: Encoding captured audio data.

```
// Configure the processor
if (!sh.configure(10000))
    System.exit(-1);
// Set the output content type
p.setContentDescriptor(new
            FileTypeDescriptor(FileTypeDescriptor.WAVE));

// Get the track control objects
TrackControl track[] = p.getTrackControls();
boolean encodingPossible = false;
// Go through the tracks and try to program one of them
// to output ima4 data.
for (int i = 0; i < track.length; i++) {
    try {
    track[i].setFormat(new AudioFormat(AudioFormat.IMA4_MS));
    encodingPossible = true;
    } catch (Exception e) {
    // cannot convert to ima4
    track[i].setEnabled(false);
    }
}

if (!encodingPossible) {
    sh.close();
    System.exit(-1);
}
// Realize the processor
if (!sh.realize(10000))
    System.exit(-1);
```

Example: Capturing and Saving Audio and Video Data

In this example, a `ProcessorModel` is used to create a `Processor` to capture live audio and video data, encode the data as IMA4 and Cinepak tracks, interleave the tracks, and save the interleaved media stream to a QuickTime file.

When you construct a `ProcessorModel` by specifying the track formats and output content type and then use that model to construct a `Processor`, the `Processor` is automatically connected to the capture device that meets the format requirements, if there is one.

Example 5-6: Creating a capture `Processor` with `ProcessorModel`.

```
Format formats[] = new Format[2];
formats[0] = new AudioFormat(AudioFormat.IMA4);
formats[1] = new VideoFormat(VideoFormat.CINEPAK);
FileTypeDescriptor outputType =
    new FileTypeDescriptor(FileTypeDescriptor.QUICKTIME);
Processor p = null;

try {
    p = Manager.createRealizedProcessor(new ProcessorModel(formats,
                                outputType));
} catch (IOException e) {
    System.exit(-1);
} catch (NoProcessorException e) {
    System.exit(-1);
} catch (CannotRealizeException e) {
    System.exit(-1);
}
// get the output of the processor
DataSource source = p.getDataOutput();
// create a File protocol MediaLocator with the location
// of the file to
// which bits are to be written
MediaLocator dest = new MediaLocator("file://foo.mov");
// create a datasink to do the file writing & open the
// sink to make sure
// we can write to it.
DataSink filewriter = null;
try {
    filewriter = Manager.createDataSink(source, dest);
    filewriter.open();
} catch (NoDataSinkException e) {
    System.exit(-1);
} catch (IOException e) {
    System.exit(-1);
} catch (SecurityException e) {
    System.exit(-1);
}
// now start the filewriter and processor
try {
    filewriter.start();
} catch (IOException e) {
    System.exit(-1);
}
p.start();
// stop and close the processor when done capturing...
// close the datasink when EndOfStream event is received...
```

6

Extending JMF

You can extend JMF by implementing one of the plug-in interfaces to perform custom processing on a `Track`, or by implementing completely new `DataSources` and `MediaHandlers`.

Note: JMF `Players` and `Processors` are not required to support plug-ins—plug-ins won't work with JMF 1.0-based Players and some 2.0-based implementations might choose not to support plug-ins. The reference implementation of JMF 2.0 provided by Sun Microsystems, Inc. and IBM Corporation fully supports the plug-in API.

Implementing JMF Plug-Ins

Custom JMF plug-ins can be used seamlessly with `Processors` that support the plug-in API. After you implement your plug-in, you need to install it and register it with the `PlugInManager` to make it available to plug-in compatible `Processors`.

Implementing a Demultiplexer Plug-In

A `Demultiplexer` parses media streams such as WAV, MPEG or QuickTime. If the stream is multiplexed, the separate tracks are extracted. You might want to implement a `Demultiplexer` plug-in to support a new file format or provide a higher-performance demultiplexer. If you implement a custom `DataSource`, you can implement a `Demultiplexer` plug-in that works with your custom `DataSource` to enable playback through an existing `Processor`.

A `Demultiplexer` is a single-input, multi-output processing component. It reads data from a push or pull `DataSource`, extracts the individual tracks, and outputs each track separately.

A Demultiplexer is a type of MediaHandler, it must implement the MediaHandler setSource method. This method is used by the Processor to locate a Demultiplexer that can handle its DataSource. The Processor goes through the list of registered Demultiplexers until it finds one that does not return an exception when setSource it called.

The main work performed by a Demultiplexer is done in the implementation of the getTracks method, which returns an array of the tracks extracted from the input DataSource.

A complete example of a GSM demultiplexer is provided in "Demultiplexer Plug-In" on page 183. When you implement a Demultiplexer, you need to:

1. Implement getSupportedInputContentDescriptors to advertise what input formats the demulitplexer supports. For example, the GSM demultiplexer needs to advertise that it supports GSM files.

Example 6-1: Implementing getSupportedInputContentDescriptors.

```
private static ContentDescriptor[] supportedFormat =
    new ContentDescriptor[] {new ContentDescriptor("audio.x_gsm")};

public ContentDescriptor [] getSupportedInputContentDescriptors() {
    return supportedFormat;
}
```

2. Implement the MediaHandler setSource method to check the DataSource and determine whether or not the Demultiplexer can handle that type of source. For example, the GSM demultiplexer supports PullDataSources:

Example 6-2: Implementing setSource for a Demultiplexer. (1 of 2)

```
public void setSource(DataSource source)
    throws IOException, IncompatibleSourceException {

    if (!(source instanceof PullDataSource)) {
        throw new IncompatibleSourceException("DataSource
            not supported: " + source);
    } else {
        streams = ((PullDataSource) source).getStreams();
    }
```

Example 6-2: Implementing `setSource` for a `Demultiplexer`. (2 of 2)

```
        if ( streams == null) {
          throw new IOException("Got a null stream from the DataSource");
        }

        if (streams.length == 0) {
            throw new IOException("Got a empty stream array
                from the DataSource");
        }

        this.source = source;
        this.streams = streams;

        positionable =  (streams[0] instanceof Seekable);
        seekable =  positionable && ((Seekable)
                    streams[0]).isRandomAccess();

        if (!supports(streams))
            throw new IncompatibleSourceException("DataSource not
                    supported: " + source);
    }
```

3. Implement `getTracks` to parse the header and extract the individual tracks from the stream if it is multiplexed. In the GSM demultiplexer a `readHeader` method is implemented to parse the header. The `getTracks` method returns an array of `GsmTracks`. (See "Demultiplexer Plug-In" on page 183 for the implementation of `GsmTracks`.)

Example 6-3: Implementing `getTracks` for a `Demultiplexer`. (1 of 2)

```
    public Track[] getTracks() throws IOException, BadHeaderException {

        if (tracks[0] != null)
            return tracks;
        stream = (PullSourceStream) streams[0];
        readHeader();
        bufferSize = bytesPerSecond;
        tracks[0] = new GsmTrack((AudioFormat) format,
                                 /*enabled=*/ true,
                                  new Time(0),
                                  numBuffers,
                                  bufferSize,
                                  minLocation,
                                  maxLocation
                                  );
        return tracks;
    }
```

Example 6-3: Implementing `getTracks` for a `Demultiplexer`. (2 of 2)

```
// ...

    private void readHeader()
        throws IOException, BadHeaderException {

        minLocation = getLocation(stream); // Should be zero

        long contentLength = stream.getContentLength();
        if ( contentLength != SourceStream.LENGTH_UNKNOWN ) {
            double durationSeconds = contentLength / bytesPerSecond;
            duration = new Time(durationSeconds);
            maxLocation = contentLength;
        } else {
            maxLocation = Long.MAX_VALUE;
        }
```

Implementing a Codec or Effect Plug-In

`Codec` plug-ins are used to decode compressed media data, convert media data from one format to another, or encode raw media data into a compressed format. You might want to implement a `Codec` to provide performance enhancements over existing solutions, support new compressed or uncompressed data formats, or convert data from a custom format to a standard format that can be easily processed and rendered.

A `Codec` is a single-input, single-output processing component. It reads data for an individual track, processes the data, and outputs the results.

A `Codec` plug-in can enable the user to control the processing it performs through `EncodingControl` or `DecodingControl` objects. These controls provide a way to adjust attributes such as the frame rate, bit rate, and compression ratio. Codec controls are accessed through the `getControls` method. If a particular `CodecControl` provides a user-interface component, its accessed by calling `getControlComponent`.

When you implement a `Codec`, you need to:

1. Implement `getSupportedInputFormats` and `getSupportedOutputFormats` to advertise what input and output formats the codec supports.
2. Enable the selection of those formats by implementing `setInputFormat` and `setOutputFormat`.
3. Implement `process` to actually perform the compression or decompression of the input `Track`.

Effect Plug-ins

An `Effect` plug-in is actually a specialized type of `Codec` that performs some processing on the input `Track` other than compression or decompression. For example, you might implement a gain effect that adjusts the volume of an audio track. Like a `Codec`, an `Effect` is a single-input, single-output processing component and the data manipulation that the `Effect` performs is implemented in the `process` method.

An `Effect` plug-in can be used as either a pre-processing effect or a post-processing effect. For example, if a `Processor` is being used to render a compressed media stream, the `Effect` would typically be used as a post-processing effect and applied after the stream has been decoded. Conversely, if the `Processor` was being used to output a compressed media stream, the `Effect` would typically be applied as a pre-processing effect before the stream is encoded.

When you implement an `Effect`, you need to:

1. Implement `getSupportedInputFormats` and `getSupportedOutputFormats` to advertise what input and output formats the effect supports.
2. Enable the selection of those formats by implementing `setInputFormat` and `setOutputFormat`.
3. Implement `process` to actually perform the effect processing.

Note that there's no mechanism for specifying what a particular `Effect` does—the name of an `Effect` plug-in class should provide some indication of its intended use.

Example: `GainEffect` *Plug-In*

In this example, the `Effect` interface is implemented to create an effect that adjusts the gain on the incoming audio data and outputs the modified data. By default, the `GainEffect` `process` method increases the gain by a factor of 2.

Example 6-4: Implementing a gain effect plug-in (1 of 5)

```
import javax.media.*;
import javax.media.format.*;
import javax.media.format.audio.*;

public class GainEffect implements Effect {

    /** The effect name **/
    private static String EffectName="GainEffect";

    /** chosen input Format **/
    protected AudioFormat inputFormat;

    /** chosen output Format **/
    protected AudioFormat outputFormat;

    /** supported input Formats **/
    protected Format[] supportedInputFormats=new Format[0];

    /** supported output Formats **/
    protected Format[] supportedOutputFormats=new Format[0];

    /** selected Gain **/
    protected float gain = 2.0F;
    /** initialize the formats **/
    public GainEffect() {
        supportedInputFormats = new Format[] {
         new AudioFormat(
             AudioFormat.LINEAR,
                Format.NOT_SPECIFIED,
                16,
                Format.NOT_SPECIFIED,
                AudioFormat.LITTLE_ENDIAN,
                AudioFormat.SIGNED,
                16,
                Format.NOT_SPECIFIED,
                Format.byteArray
         )
      };
         supportedOutputFormats = new Format[] {
         new AudioFormat(
             AudioFormat.LINEAR,
                Format.NOT_SPECIFIED,
                16,
                Format.NOT_SPECIFIED,
                AudioFormat.LITTLE_ENDIAN,
                AudioFormat.SIGNED,
                16,
                Format.NOT_SPECIFIED,
                Format.byteArray
         )
      };
```

Example 6-4: Implementing a gain effect plug-in (2 of 5)

```
    }
    /** get the resources needed by this effect **/
    public void open() throws ResourceUnavailableException {
    }

    /** free the resources allocated by this codec **/
    public void close() {
    }

    /** reset the codec **/
    public void reset() {
    }

    /** no controls for this simple effect **/
    public Object[] getControls() {
        return (Object[]) new Control[0];
    }

    /**
     * Return the control based on a control type for the effect.
     **/
    public Object getControl(String controlType) {
        try {
            Class cls = Class.forName(controlType);
            Object cs[] = getControls();
            for (int i = 0; i < cs.length; i++) {
                if (cls.isInstance(cs[i]))
                return cs[i];
            }
            return null;
        } catch (Exception e) { // no such controlType or such control
            return null;
        }
    }
    /************** format methods *************/
    /** set the input format **/
    public Format setInputFormat(Format input) {
        // the following code assumes valid Format
        inputFormat = (AudioFormat)input;
        return (Format)inputFormat;
    }
    /** set the output format **/
    public Format setOutputFormat(Format output) {
        // the following code assumes valid Format
        outputFormat = (AudioFormat)output;
        return (Format)outputFormat;
    }
    /** get the input format **/
    protected Format getInputFormat() {
        return inputFormat;
    }
```

Example 6-4: Implementing a gain effect plug-in (3 of 5)

```
/** get the output format **/
protected Format getOutputFormat() {
    return outputFormat;
}

/** supported input formats **/
public Format [] getSupportedInputFormats() {
    return supportedInputFormats;
}

/** output Formats for the selected input format **/
public Format [] getSupportedOutputFormats(Format in) {
    if (! (in instanceof AudioFormat) )
        return new Format[0];

    AudioFormat iaf=(AudioFormat) in;

    if (!iaf.matches(supportedInputFormats[0]))
        return new Format[0];

 AudioFormat oaf= new AudioFormat(
         AudioFormat.LINEAR,
            iaf.getSampleRate(),
            16,
            iaf.getChannels(),
            AudioFormat.LITTLE_ENDIAN,
            AudioFormat.SIGNED,
            16,
            Format.NOT_SPECIFIED,
            Format.byteArray
    );

    return new Format[] {oaf};
}

/** gain accessor method **/
public void setGain(float newGain){
    gain=newGain;
}
/** return effect name **/
public String getName() {
    return EffectName;
}

/** do the processing **/
public int process(Buffer inputBuffer, Buffer outputBuffer){

    // == prolog
    byte[] inData = (byte[])inputBuffer.getData();
    int inLength = inputBuffer.getLength();
    int inOffset = inputBuffer.getOffset();
```

Example 6-4: Implementing a gain effect plug-in (4 of 5)

```
        byte[] outData = validateByteArraySize(outputBuffer, inLength);
        int outOffset = outputBuffer.getOffset();

     int samplesNumber = inLength / 2 ;

        // == main

        for (int i=0; i< samplesNumber;i++) {
            int tempL = inData[inOffset ++];
            int tempH = inData[inOffset ++];
            int sample = tempH | (tempL & 255);

            sample = (int)(sample * gain);

            if (sample>32767) // saturate
                sample = 32767;
            else if (sample < -32768)
                sample = -32768;

            outData[outOffset ++]=(byte) (sample &  255);
            outData[outOffset ++]=(byte) (sample >> 8);

        }

        // == epilog
        updateOutput(outputBuffer,outputFormat, samplesNumber, 0);
        return BUFFER_PROCESSED_OK;
    }
    /**
     * Utility: validate that the Buffer object's data size is at least
     * newSize bytes.
     * @return array with sufficient capacity
     **/
    protected byte[] validateByteArraySize(Buffer buffer,int newSize) {
        Object objectArray=buffer.getData();
        byte[] typedArray;
      if (objectArray instanceof byte[]) { // is correct type AND not null
            typedArray=(byte[])objectArray;
           if (typedArray.length >= newSize ) { // is sufficient capacity
                return typedArray;
            }
        }
        System.out.println(getClass().getName()+
                      " : allocating byte["+newSize+"] ");
        typedArray = new byte[newSize];
        buffer.setData(typedArray);
        return typedArray;
    }
    /** utility: update the output buffer fields **/
    protected void updateOutput(Buffer outputBuffer,
                                Format format,int length, int offset) {
```

Example 6-4: Implementing a gain effect plug-in (5 of 5)

```
            outputBuffer.setFormat(format);
            outputBuffer.setLength(length);
            outputBuffer.setOffset(offset);
        }
    }
```

Implementing a Multiplexer Plug-In

A `Multiplexer` is essentially the opposite of a `Demultiplexer`: it takes individual tracks of media data and merges them into a single multiplexed media-stream such as an MPEG or QuickTime file. You might want to implement a `Multiplexer` plug-in to support a custom `DataSource` or provide a higher-performance. However, it's not always necessary to implement a separate `Multiplexer` plug-in—multiplexing can also be performed by a `DataSink`.

A `Multiplexer` is a multi-input, single-output processing component. It reads data from a set of tracks and outputs a `DataSource`.

The main work performed by a `Multiplexer` is done in the implementation of the `process` method. The `getDataSource` method returns the `DataSource` generated by the `Multiplexer`.

When you implement a `Multiplexer`, you need to:

1. Implement `getSupportedOutputContentDescriptors` to advertise what output formats the `Multiplexer` supports.
2. Enable the selection of the output format by implementing `setOutputContentDescriptor`.
3. Implement `process` to actually merge the individual tracks into an output stream of the selected format.

Unlike a `Codec`, there is no specific query mechanism. The `initializeTracks` method should return `false` if any of the specified track formats are not supported.

Implementing a Renderer Plug-In

A `Renderer` delivers media data in its final processed state. It is a single-input processing component with no output. `Renderer` plug-ins read data from a `DataSource` and typically present the media data to the user, but can also be used to provide access to the processed media data for use by another application or device. For example, you might implement a `Renderer` plug-in if you want to render a video to a location other than the screen.

If you're implementing a video renderer, you should implement the `VideoRenderer` interface, which extends `Renderer` to define video-specific attributes such as the `Component` where the video will be rendered.

The main work performed by a `Renderer` is done in the implementation of the `process` method. When you implement a `Renderer`, you need to:

1. Implement `getSupportedInputFormats` to advertise what input formats the `Renderer` supports.
2. Enable the selection of the input format by implementing `setInputFormat`.
3. Implement `process` to actually process the data and render it to the output device that this `Renderer` represents.

Example: AWTRenderer

This example implements the `Renderer` plug-in to create a `Renderer` for RGB images that uses `AWT Image`.

Example 6-5: Implementing a `Renderer` plug-in (1 of 7)

```
import javax.media.*;
import javax.media.renderer.VideoRenderer;
import javax.media.format.Format;
import javax.media.format.video.VideoFormat;
import javax.media.format.video.RGBFormat;
import java.awt.*;
import java.awt.image.*;
import java.awt.event.*;
import java.util.Vector;
```

Example 6-5: Implementing a `Renderer` plug-in (2 of 7)

```
/**********************************************************************
 * Renderer for RGB images using AWT Image.
**********************************************************************/
public class SampleAWTRenderer implements
javax.media.renderer.VideoRenderer {
    /**
     * Variables and Constants
     **/

    // The descriptive name of this renderer
    private static final String name = "Sample AWT Renderer";

    protected RGBFormat inputFormat;
    protected RGBFormat supportedRGB;
    protected Format [] supportedFormats;

    protected MemoryImageSource sourceImage;
    protected Image     destImage;
    protected Buffer    lastBuffer = null;

    protected int       inWidth = 0;
    protected int       inHeight = 0;
    protected Component component = null;
    protected Rectangle reqBounds = null;
    protected Rectangle bounds = new Rectangle();
    protected boolean started = false;

    /**
     * Constructor
     **/

    public SampleAWTRenderer() {
     // Prepare supported input formats and preferred format
     int rMask = 0x000000FF;
     int gMask = 0x0000FF00;
     int bMask = 0x00FF0000;

     supportedRGB = new RGBFormat(null,    // size
                    Format.NOT_SPECIFIED,  // maxDataLength
                    int[].class,    // buffer type
                    Format.NOT_SPECIFIED,  // frame rate
                    32,    // bitsPerPixel
                    RGBFormat.PACKED,    // packed
                    rMask, gMask, bMask,   // component masks
                    1,         // pixel stride
                    Format.NOT_SPECIFIED,  // line stride
                    Format.FALSE,          // flipped
                    Format.NOT_SPECIFIED   // endian
                    );
     supportedFormats = new VideoFormat[1];
     supportedFormats[0] = supportedRGB;
    }
```

Example 6-5: Implementing a `Renderer` plug-in (3 of 7)

```
/**
 * Controls implementation
 **/

// Returns an array of supported controls

public Object[] getControls() {
 // No controls
    return (Object[]) new Control[0];
}

/**
 * Return the control based on a control type for the PlugIn.
 */
public Object getControl(String controlType) {
   try {
      Class  cls = Class.forName(controlType);
      Object cs[] = getControls();
     for (int i = 0; i < cs.length; i++) {
         if (cls.isInstance(cs[i]))
            return cs[i];
      }
      return null;
   } catch (Exception e) {   // no such controlType or such control
     return null;
   }
}

/**
 * PlugIn implementation
 **/

public String getName() {
 return name;
}

// Opens the plugin
public void open() throws ResourceUnavailableException {
 sourceImage = null;
 destImage   = null;
 lastBuffer    = null;
}
/** Resets the state of the plug-in. Typically at end of media or
 * when media is repositioned.
 */
public void reset() {
 // Nothing to do
}

public void close() {
 // Nothing to do
}
```

Example 6-5: Implementing a `Renderer` plug-in (4 of 7)

```
/**
 * Renderer implementation
 **/

public void start() {
 started = true;
}

public void stop() {
 started = false;
}

// Lists the possible input formats supported by this plug-in.

public Format [] getSupportedInputFormats() {
 return supportedFormats;
}
// Set the data input format.

public Format setInputFormat(Format format) {
 if ( format != null && format instanceof RGBFormat &&
      format.matches(supportedRGB)) {

     inputFormat = (RGBFormat) format;
     Dimension size = inputFormat.getSize();
     inWidth = size.width;
     inHeight = size.height;
     return format;
 } else
     return null;
}

// Processes the data and renders it to a component

public synchronized int process(Buffer buffer) {
 if (component == null)
     return BUFFER_PROCESSED_FAILED;

 Format inf = buffer.getFormat();
 if (inf == null)
     return BUFFER_PROCESSED_FAILED;
 if (inf != inputFormat || !buffer.getFormat().equals(inputFormat))
 {
     if (setInputFormat(inf) != null)
         return BUFFER_PROCESSED_FAILED;
 }

 Object data = buffer.getData();
 if (!(data instanceof int[]))
     return BUFFER_PROCESSED_FAILED;
```

Example 6-5: Implementing a `Renderer` plug-in (5 of 7)

```
        if (lastBuffer != buffer) {
            lastBuffer = buffer;
            newImage(buffer);
        }

        sourceImage.newPixels(0, 0, inWidth, inHeight);

        Graphics g = component.getGraphics();
        if (g != null) {
            if (reqBounds == null) {
            bounds = component.getBounds();
            bounds.x = 0;
            bounds.y = 0;
            } else
                bounds = reqBounds;
            g.drawImage(destImage, bounds.x, bounds.y,
                bounds.width, bounds.height,
                0, 0, inWidth, inHeight, component);
        }

        return BUFFER_PROCESSED_OK;
    }

    /**
     * VideoRenderer implementation
     **/

    /**
     * Returns an AWT component that it will render to. Returns null
     * if it is not rendering to an AWT component.
     */
    public java.awt.Component getComponent() {
        if (component == null) {
            component = new Canvas() {
            public Dimension getPreferredSize() {
                return new Dimension(getInWidth(), getInHeight());
            }

            public void update(Graphics g) {
            }

            public void paint(Graphics g) {
                // Need to repaint image if the movie is in paused state
            }

            };
        }

        return component;
    }
```

Example 6-5: Implementing a `Renderer` plug-in (6 of 7)

```
/**
 * Requests the renderer to draw into a specified AWT component.
 * Returns false if the renderer cannot draw into the specified
 * component.
 */
public boolean setComponent(java.awt.Component comp) {
 component = comp;
 return true;
}

/**
 * Sets the region in the component where the video is to be
 * rendered to. Video is to be scaled if necessary. If
 * <code>rect</code> is null, then the video occupies the entire
 * component.
 */
public void setBounds(java.awt.Rectangle rect) {
 reqBounds = rect;
}

/**
 * Returns the region in the component where the video will be
 * rendered to. Returns null if the entire component is being used.
 */
public java.awt.Rectangle getBounds() {
 return reqBounds;
}

/**
 * Local methods
 **/

int getInWidth() {
 return inWidth;
}

int getInHeight() {
 return inHeight;
}

private void newImage(Buffer buffer) {
 Object data = buffer.getData();
 if (!(data instanceof int[]))
     return;
 RGBFormat format = (RGBFormat) buffer.getFormat();

 DirectColorModel dcm = new
       DirectColorModel(format.getBitsPerPixel(),
                        format.getRedMask(),
                        format.getGreenMask(),
                        format.getBlueMask());
```

Example 6-5: Implementing a `Renderer` plug-in (7 of 7)

```
        sourceImage = new MemoryImageSource(format.getLineStride(),
                          format.getSize().height,
                          dcm,
                          (int[])data, 0,
                          format.getLineStride());
        sourceImage.setAnimated(true);
        sourceImage.setFullBufferUpdates(true);
        if (component != null) {
            destImage = component.createImage(sourceImage);
            component.prepareImage(destImage, component);
        }
       }
  }
```

Registering a Custom Plug-In With the Plug-In Manager

To make a custom plug-in available to a `Processor` through the `TrackControl` interface, you need to register it with the `PlugInManager`. (The default plug-ins are registered automatically.)

To register a new plug-in, you use the `PlugInManager addPlugIn` method. You must call `commit` to make the addition permanent. For example, to register the `GainEffect` plug-in from the example on page 89:

Example 6-6: Registering a new plug-in. (1 of 2)

```
// Name of the new plugin
string GainPlugin = new String("COM.mybiz.media.GainEffect");

// Supported input Formats
Format[] supportedInputFormats = new Format[] {
         new AudioFormat(
             AudioFormat.LINEAR,
                Format.NOT_SPECIFIED,
                16,
                Format.NOT_SPECIFIED,
                AudioFormat.LITTLE_ENDIAN,
                AudioFormat.SIGNED,
                16,
                Format.NOT_SPECIFIED,
                Format.byteArray
```

Example 6-6: Registering a new plug-in. (2 of 2)

```
        )
};

// Supported output Formats
Format[] supportedOutputFormats = new Format[] {
        new AudioFormat(
            AudioFormat.LINEAR,
               Format.NOT_SPECIFIED,
               16,
               Format.NOT_SPECIFIED,
               AudioFormat.LITTLE_ENDIAN,
               AudioFormat.SIGNED,
               16,
               Format.NOT_SPECIFIED,
               Format.byteArray
        )
};

// Add the new plug-in to the plug-in registry
PlugInManager.addPlugIn(GainPlugin, supportedInputFormats,
                        supportedOutputFormats, EFFECT);

// Save the changes to the plug-in registry
PlugInManager.commit();
```

If you want to make your plug-in available to other users, you should create an Java applet or application that performs this registration process and distribute it with your plug-in.

You can remove a plug-in either temporarily or permanently with the `removePlugIn` method. To make the change permanent, you call `commit`.

Note: The reference implementation of JMF 2.0 provided by Sun Microsystems, Inc. and IBM Corporation provides a utility application, `JMFRegistry`, that you can use to register plug-ins interactively.

Implementing Custom Data Sources and Media Handlers

Custom `DataSources` and `MediaHandlers` such as `Players` and `Processors` can be used seamlessly with JMF to support new formats and integrate existing media engines with JMF.

Implementing a Protocol Data Source

A `DataSource` is an abstraction of a media protocol-handler. You can implement new types of `DataSources` to support additional protocols by

extending `PullDataSource`, `PullBufferDataSource`, `PushDataSource`, or `PushBufferDataSource`. If you implement a custom `DataSource`, you can implement `Demultiplexer` and `Multiplexer` plug-ins that work with your custom `DataSource` to enable playback through an existing `Processor`, or you can implement a completely custom `MediaHandler` for your `DataSource`.

A `DataSource` manages a collection of `SourceStreams` of the corresponding type. For example, a `PullDataSource` only supports pull data-streams; it manages a collection of `PullSourceStreams`. Similarly, a `PushDataSource` only supports push data-streams; it manages a collection of `PushSourceStreams`. When you implement a new `DataSource`, you also need to implement the corresponding source stream: `PullSourceStream`, `PullBufferStream`, `PushSourceStream`, or `PushBufferStream`.

If your `DataSource` supports changing the media position within the stream to a specified time, it should implement the `Positionable` interface. If the `DataSource` supports seeking to a particular point in the stream, the corresponding `SourceStream` should implement the `Seekable` interface.

So that the `Manager` can construct your custom `DataSource`, the name and package hierarchy for the `DataSource` must follow certain conventions. The fully qualified name of your custom `DataSource` should be:

```
<protocol package-prefix>.media.protocol.<protocol>.DataSource
```

The *protocol package-prefix* is a unique identifier for your code that you register with the JMF `PackageManager` (for example, `COM.mybiz`) as a protocol package-prefix. The protocol identifies the protocol for your new `DataSource`. For more information, see "Integrating a Custom Data Source with JMF" on page 103.

Example: Creating an FTP DataSource

The example in "Sample Data Source Implementation" on page 197 demonstrates how to support an additional protocol by implementing a custom `DataSource` and `SourceStream`. This `DataSource`, `FTPDataSource`, implements `PullDataSource`.

Integrating a Custom Data Source with JMF

To integrate a custom `DataSource` implementation with JMF you need to:

- Install the package containing the new DataSource class, <protocol-prefix>.media.protocol.<protocol>.DataSource.
- Add your package prefix to the protocol package-prefix list controlled by the PackageManager. The Manager queries the PackageManager for the list of protocol package-prefixes it uses to search for a DataSource.

For example, to integrate a new DataSource for the protocol type *xxx*, you would create and install a package called:

```
<protocol package-prefix>.media.protocol.xxx.DataSource
```

that contains the new DataSource class. You also need to add your package prefix (an identifier for your code, such as COM.mybiz) to the protocol package-prefix list managed by the PackageManager.

Example 6-7: Registering a protocol package-prefix.

```
Vector  packagePrefix = PackageManager.getProtocolPrefixList();
string myPackagePrefix = new String("COM.mybiz");
// Add new package prefix to end of the package prefix list.
packagePrefix.addElement(myPackagePrefix);
PackageManager.setProtocolPrefixList();
// Save the changes to the package prefix list.
PackageManager.commitProtocolPrefixList();
```

If you want to make your new DataSource available to other users, you should create an Java applet or application that performs this registration process and distribute it with your DataSource.

Implementing a Basic Controller

Controllers can be implemented to present time-based media other than audio or video data. For example, you might want to create a Controller that manages a slide-show presentation of still images.

Example: Creating a Timeline Controller

The sample in "Sample Controller Implementation" on page 207 illustrates how a simple time-line Controller can be implemented in JMF. This Controller, TimeLineController, takes array of time values (representing a time line) and it keeps track of which segment in the time line you are in.

`TimeLineController` uses a custom media event, `TimeLineEvent`, to indicate when the segment in the time line changes.

Implementing a DataSink

JMF provides a default `DataSink` that can be used to write data to a file. Other types of `DataSink` classes can be implemented to facilitate writing data to the network or to other destinations.

To create a custom `DataSink`, you implement the `DataSink` interface. A `DataSink` is a type of `MediaHandler`, so you must also implement the `MediaHandler setSource` method.

To use your `DataSink` with JMF, you need to add your package-prefix to the content package-prefix list maintained by the `PackageManager`. For more information, see "Integrating a Custom Media Handler with JMF".

Integrating a Custom Media Handler with JMF

To integrate a new `MediaHandler` with JMF, you need to:

- Implement the `MediaHandler setSource` method to check the `DataSource` and determine whether or not the `handler` can handle that type of source. When the client programmer calls the appropriate `Manager create` method, `setSource` is called as the `Manager` searches for an appropriate `MediaHandler`.
- Install the package containing the new class.
- Add your package prefix to the content package-prefix list controlled by the `PackageManager`. The `Manager` queries the `PackageManager` for the list of content package-prefixes it uses to search for a `MediaHandler`.

For example, to integrate a new `Player` for the content type mpeg.sys, you would create and install a package called:

```
<content package-prefix>.media.content.mpeg.sys.Handler
```

that contains the new `Player` class. The package prefix is an identifier for your code, such as `COM.mybiz`. You also need to add your package prefix to the content package-prefix list managed by the `PackageManager`.

Example 6-8: Registering a content package-prefix.

```
Vector  packagePrefix = PackageManager.getContentPrefixList();
string myPackagePrefix = new String("COM.mybiz");
// Add new package prefix to end of the package prefix list.
packagePrefix.addElement(myPackagePrefix);
PackageManager.setContentPrefixList();
// Save the changes to the package prefix list.
PackageManager.commitContentPrefixList();
```

If you want to make your new `MediaHandler` available to other users, you should create an Java applet or application that performs this registration process and distribute it with your `MediaHandler`.

Registering a Capture Device with JMF

The implementor of a device is responsible for defining a `CaptureDeviceInfo` object for the device. When the device is installed, it must be registered with the `CaptureDeviceManager` by calling `addDevice`.

Part 2: Real-Time Transport Protocol

7

Working with Real-Time Media Streams

To send or receive a live media broadcast or conduct a video conference over the Internet or Intranet, you need to be able to receive and transmit media streams in real-time. This chapter introduces streaming media concepts and describes the Real-time Transport Protocol JMF uses for receiving and transmitting media streams across the network.

Streaming Media

When media content is streamed to a client in real-time, the client can begin to play the stream without having to wait for the complete stream to download. In fact, the stream might not even have a predefined duration—downloading the entire stream before playing it would be impossible. The term *streaming media* is often used to refer to both this technique of delivering content over the network in real-time and the real-time media content that's delivered.

Streaming media is everywhere you look on the web—live radio and television broadcasts and webcast concerts and events are being offered by a rapidly growing number of web portals, and it's now possible to conduct audio and video conferences over the Internet. By enabling the delivery of dynamic, interactive media content across the network, streaming media is changing the way people communicate and access information.

Protocols for Streaming Media

Transmitting media data across the net in real-time requires high network throughput. It's easier to compensate for lost data than to compensate for

large delays in receiving the data. This is very different from accessing static data such as a file, where the most important thing is that all of the data arrive at its destination. Consequently, the protocols used for static data don't work well for streaming media.

The HTTP and FTP protocols are based on the Transmission Control Protocol (TCP). TCP is a transport-layer protocol[1] designed for reliable data communications on low-bandwidth, high-error-rate networks. When a packet is lost or corrupted, it's retransmitted. The overhead of guaranteeing reliable data transfer slows the overall transmission rate.

For this reason, underlying protocols other than TCP are typically used for streaming media. One that's commonly used is the User Datagram Protocol (UDP). UDP is an unreliable protocol; it does not guarantee that each packet will reach its destination. There's also no guarantee that the packets will arrive in the order that they were sent. The receiver has to be able to compensate for lost data, duplicate packets, and packets that arrive out of order.

Like TCP, UDP is a general transport-layer protocol—a lower-level networking protocol on top of which more application-specific protocols are built. The Internet standard for transporting real-time data such as audio and video is the Real-Time Transport Protocol (RTP).

RTP is defined in IETF RFC 1889, a product of the AVT working group of the Internet Engineering Task Force (IETF).

Real-Time Transport Protocol

RTP provides end-to-end network delivery services for the transmission of real-time data. RTP is network and transport-protocol independent, though it is often used over UDP.

1. In the seven layer ISO/OSI data communications model, the transport layer is level four. For more information about the ISO/OSI model, see *Understanding OSI*. Larmouth, John. International Thompson Computer Press, 1996. ISBN 1850321760.

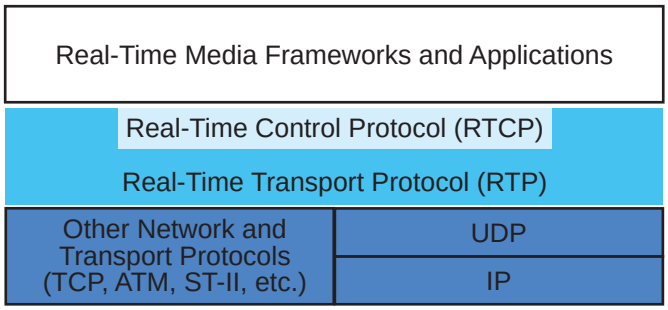


Figure 7-1: RTP architecture.

RTP can be used over both unicast and multicast network services. Over a *unicast* network service, separate copies of the data are sent from the source to each destination. Over a *multicast* network service, the data is sent from the source only once and the network is responsible for transmitting the data to multiple locations. Multicasting is more efficient for many multimedia applications, such as video conferences. The standard Internet Protocol (IP) supports multicasting.

RTP Services

RTP enables you to identify the type of data being transmitted, determine what order the packets of data should be presented in, and synchronize media streams from different sources.

RTP data packets are not guaranteed to arrive in the order that they were sent—in fact, they're not guaranteed to arrive at all. It's up to the receiver to reconstruct the sender's packet sequence and detect lost packets using the information provided in the packet header.

While RTP does not provide any mechanism to ensure timely delivery or provide other quality of service guarantees, it is augmented by a control protocol (RTCP) that enables you to monitor the quality of the data distribution. RTCP also provides control and identification mechanisms for RTP transmissions.

If quality of service is essential for a particular application, RTP can be used over a resource reservation protocol that provides connection-oriented services.

RTP Architecture

An RTP *session* is an association among a set of applications communicating with RTP. A session is identified by a network address and a pair of ports. One port is used for the media data and the other is used for control (RTCP) data.

A *participant* is a single machine, host, or user participating in the session. Participation in a session can consist of passive reception of data (receiver), active transmission of data (sender), or both.

Each media type is transmitted in a different session. For example, if both audio and video are used in a conference, one session is used to transmit the audio data and a separate session is used to transmit the video data. This enables participants to choose which media types they want to receive—for example, someone who has a low-bandwidth network connection might only want to receive the audio portion of a conference.

Data Packets

The media data for a session is transmitted as a series of packets. A series of data packets that originate from a particular source is referred to as an *RTP stream*. Each RTP data packet in a stream contains two parts, a structured header and the actual data (the packet's *payload*).

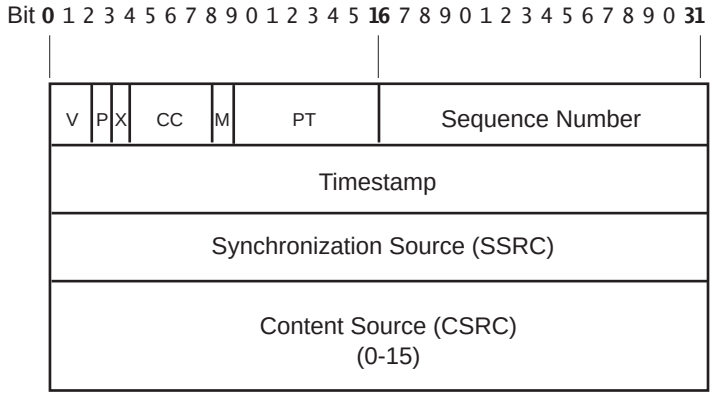


Figure 7-2: RTP data-packet header format.

The header of an RTP data packet contains:

- **The RTP version number (V)**: 2 bits. The version defined by the current specification is 2.

- **Padding** (P): 1 bit. If the padding bit is set, there are one or more bytes at the end of the packet that are not part of the payload. The very last byte in the packet indicates the number of bytes of padding. The padding is used by some encryption algorithms.
- **Extension** (X): 1 bit. If the extension bit is set, the fixed header is followed by one header extension. This extension mechanism enables implementations to add information to the RTP Header.
- **CSRC Count** (CC): 4 bits. The number of CSRC identifiers that follow the fixed header. If the CSRC count is zero, the synchronization source is the source of the payload.
- **Marker (M)**: 1 bit. A marker bit defined by the particular media profile.
- **Payload Type** (PT)**:** 7 bits. An index into a media profile table that describes the payload format. The payload mappings for audio and video are specified in RFC 1890.
- **Sequence Number**: 16 bits. A unique packet number that identifies this packet's position in the sequence of packets. The packet number is incremented by one for each packet sent.
- **Timestamp**: 32 bits. Reflects the sampling instant of the first byte in the payload. Several consecutive packets can have the same timestamp if they are logically generated at the same time—for example, if they are all part of the same video frame.
- **SSRC**: 32 bits. Identifies the synchronization source. If the CSRC count is zero, the payload source is the synchronization source. If the CSRC count is nonzero, the SSRC identifies the mixer.
- **CSRC**: 32 bits each. Identifies the contributing sources for the payload. The number of contributing sources is indicated by the CSRC count field; there can be up to 16 contributing sources. If there are multiple contributing sources, the payload is the mixed data from those sources.

Control Packets

In addition to the media data for a session, control data (RTCP) packets are sent periodically to all of the participants in the session. RTCP packets can contain information about the quality of service for the session participants, information about the source of the media being transmitted on the data port, and statistics pertaining to the data that has been transmitted so far.

There are several types of RTCP packets:

- Sender Report
- Receiver Report
- Source Description
- Bye
- Application-specific

RTCP packets are "stackable" and are sent as a compound packet that contains at least two packets, a report packet and a source description packet.

All participants in a session send RTCP packets. A participant that has recently sent data packets issues a *sender report*. The sender report (SR) contains the total number of packets and bytes sent as well as information that can be used to synchronize media streams from different sessions.

Session participants periodically issue *receiver reports* for all of the sources from which they are receiving data packets. A receiver report (RR) contains information about the number of packets lost, the highest sequence number received, and a timestamp that can be used to estimate the round-trip delay between a sender and the receiver.

The first packet in a compound RTCP packet has to be a report packet, even if no data has been sent or received—in which case, an empty receiver report is sent.

All compound RTCP packets must include a source description (SDES) element that contains the canonical name (CNAME) that identifies the source. Additional information might be included in the source description, such as the source's name, email address, phone number, geographic location, application name, or a message describing the current state of the source.

When a source is no longer active, it sends an RTCP BYE packet. The BYE notice can include the reason that the source is leaving the session.

RTCP APP packets provide a mechanism for applications to define and send custom information via the RTP control port.

RTP Applications

RTP applications are often divided into those that need to be able to receive data from the network (RTP Clients) and those that need to be able

to transmit data across the network (RTP Servers). Some applications do both—for example, conferencing applications capture and transmit data at the same time that they're receiving data from the network.

Receiving Media Streams From the Network

Being able to receive RTP streams is necessary for several types of applications. For example:

- Conferencing applications need to be able to receive a media stream from an RTP session and render it on the console.
- A telephone answering machine application needs to be able to receive a media stream from an RTP session and store it in a file.
- An application that records a conversation or conference must be able to receive a media stream from an RTP session and both render it on the console and store it in a file.

Transmitting Media Streams Across the Network

RTP server applications transmit captured or stored media streams across the network.

For example, in a conferencing application, a media stream might be captured from a video camera and sent out on one or more RTP sessions. The media streams might be encoded in multiple media formats and sent out on several RTP sessions for conferencing with heterogeneous receivers. Multiparty conferencing could be implemented without IP multicast by using multiple unicast RTP sessions.

References

The RTP specification is a product of the Audio Video Transport (AVT) working group of the Internet Engineering Task Force (IETF). For additional information about the IETF, see `http://www.ietf.org`. The AVT working group charter and proceedings are available at `http://www.ietf.org/html.charters/avt-charter.html`.

IETF RFC 1889, RTP: A Transport Protocol for Real Time Applications
Current revision: `http://www.ietf.org.internet-drafts/draft-ietf-avt-rtp-new-04.txt`

IETF RFC 1890: RTP Profile for Audio and Video Conferences with Minimal Control
Current revision: `http://www.ietf.org.internet-drafts/draft-ietf-avt-profile-new-06.txt`

Note: These RFCs are undergoing revisions in preparation for advancement from Proposed Standard to Draft Standard and the URLs listed here are for the Internet Drafts of the revisions available at the time of publication.

In addition to these RFCs, separate payload specification documents define how particular payloads are to be carried in RTP. For a list of all of the RTP-related specifications, see the AVT working group charter at: `http://www.ietf.org/html.charters/avt-charter.html`.

8

Understanding the JMF RTP API

JMF enables the playback and transmission of RTP streams through the APIs defined in the `javax.media.rtp`, `javax.media.rtp.event`, and `javax.media.rtp.rtcp` packages. JMF can be extended to support additional RTP-specific formats and dynamic payloads through the standard JMF plug-in mechanism.

Note: JMF-compliant implementations are not required to support the RTP APIs in `javax.media.rtp`, `javax.media.rtp.event`, and `javax.media.rtp.rtcp`. The reference implementations of JMF provided by Sun Microsystems, Inc. and IBM Corporation fully support these APIs.

You can play incoming RTP streams locally, save them to a file, or both.

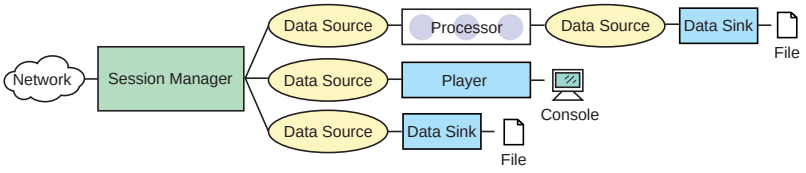


Figure 8-1: RTP reception.

For example, the RTP APIs could be used to implement a telephony application that answers calls and records messages like an answering machine.

Similarly, you can use the RTP APIs to transmit captured or stored media streams across the network. Outgoing RTP streams can originate from a file or a capture device. The outgoing streams can also be played locally, saved to a file, or both.

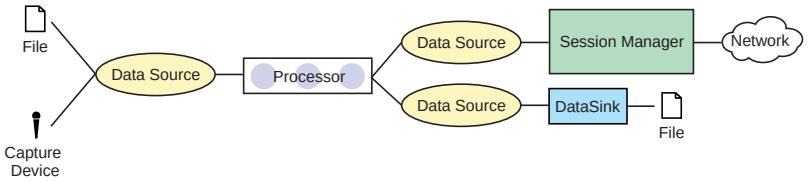


Figure 8-2: RTP transmission.

For example, you could implement a video conferencing application that captures live audio and video and transmits it across the network using a separate RTP session for each media type.

Similarly, you might record a conference for later broadcast or use a prerecorded audio stream as "hold music" in a conferencing application.

RTP Architecture

The JMF RTP APIs are designed to work seamlessly with the capture, presentation, and processing capabilities of JMF. Players and processors are used to present and manipulate RTP media streams just like any other media content. You can transmit media streams that have been captured from a local capture device using a capture `DataSource` or that have been stored to a file using a `DataSink`. Similarly, JMF can be extended to support additional RTP formats and payloads through the standard plug-in mechanism.

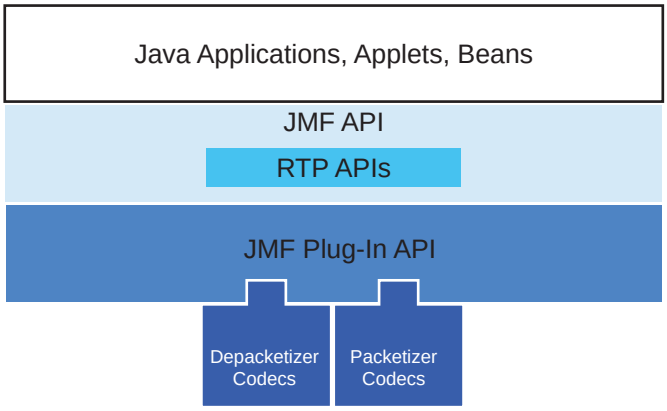


Figure 8-3: High-level JMF RTP architecture.

Session Manager

In JMF, a `SessionManager` is used to coordinate an RTP session. The session manager keeps track of the session participants and the streams that are being transmitted.

The session manager maintains the state of the session as viewed from the local participant. In effect, a session manager is a local representation of a distributed entity, the RTP session. The session manager also handles the *RTCP* control channel, and supports RTCP for both senders and receivers.

The `SessionManager` interface defines methods that enable an application to initialize and start participating in a session, remove individual streams created by the application, and close the entire session.

Session Statistics

The session manager maintains statistics on all of the RTP and RTCP packets sent and received in the session. Statistics are tracked for the entire session on a per-stream basis. The session manager provides access to global reception and transmission statistics:

- **`GlobalReceptionStats`**: Maintains global reception statistics for the session.
- **`GlobalTransmissionStats`**: Maintains cumulative transmission statistics for all local senders.

Statistics for a particular recipient or outgoing stream are available from the stream:

- **`ReceptionStats`**: Maintains source reception statistics for an individual participant.
- **`TransmissionStats`**: Maintains transmission statistics for an individual send stream.

Session Participants

The session manager keeps track of all of the participants in a session. Each participant is represented by an instance of a class that implements the `Participant` interface. `SessionManagers` create a `Participant` whenever an RTCP packet arrives that contains a source description (SDES) with a canonical name`(CNAME)` that has not been seen before in the session (or has timed-out since its last use). Participants can be passive (sending

control packets only) or active (also sending one or more RTP data streams).

There is exactly one *local participant* that represents the local client/server participant. A local participant indicates that it will begin sending RTCP control messages or data and maintain state on incoming data and control messages by starting a session.

A participant can own more than one stream, each of which is identified by the synchronization source identifier (SSRC) used by the source of the stream.

Session Streams

The `SessionManager` maintains an `RTPStream` object for each stream of RTP data packets in the session. There are two types of RTP streams:

- `ReceiveStream` represents a stream that's being received from a remote participant.
- `SendStream` represents a stream of data coming from the `Processor` or input `DataSource` that is being sent over the network.

A `ReceiveStream` is constructed automatically whenever the session manager detects a new source of RTP data. To create a new `SendStream`, you call the `SessionManager` `createSendStream` method.

RTP Events

Several RTP-specific events are defined in `javax.media.rtp.event`. These events are used to report on the state of the RTP session and streams.

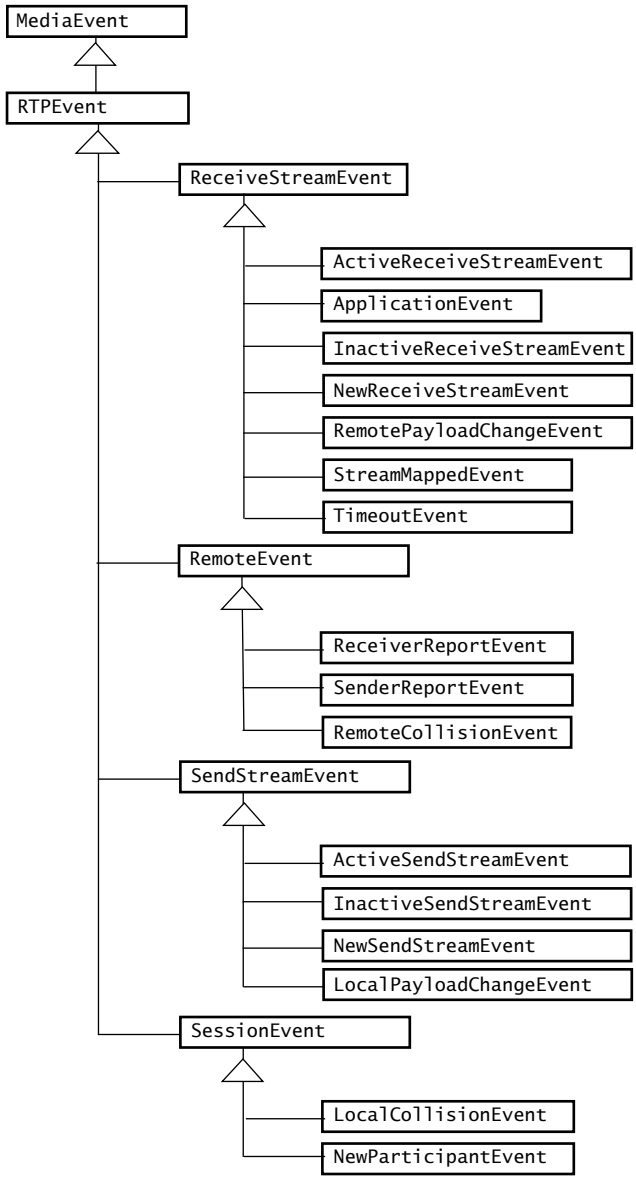


Figure 8-4: RTP events.

To receive notification of RTP events, you implement the appropriate RTP listener and register it with the session manager:

- **`SessionListener`**: Receives notification of changes in the state of the session.

- **`SendStreamListener`**: Receives notification of changes in the state of an RTP stream that's being transmitted.
- **`ReceiveStreamListener`**: Receives notification of changes in the state of an RTP stream that's being received.
- **`RemoteListener`**: Receives notification of events or RTP control messages received from a remote participant.

Session Listener

You can implement `SessionListener` to receive notification about events that pertain to the RTP session as a whole, such as the addition of new participants.

There are two types of session-wide events:

- **`NewParticipantEvent`**: Indicates that a new participant has joined the session.
- **`LocalCollisionEvent`**: Indicates that the participant's synchronization source is already in use.

Send Stream Listener

You can implement `SendStreamListener` to receive notification whenever:

- New send streams are created by the local participant.
- The transfer of data from the `DataSource` used to create the send stream has started or stopped.
- The send stream's format or payload changes.

There are five types of events associated with a `SendStream`:

- **`NewSendStreamEvent`**: Indicates that a new send stream has been created by the local participant.
- **`ActiveSendStreamEvent`**: Indicates that the transfer of data from the `DataSource` used to create the send stream has started.
- **`InactiveSendStreamEvent`**: Indicates that the transfer of data from the DataSource used to create the send stream has stopped.
- **`LocalPayloadChangeEvent`**: Indicates that the stream's format or payload has changed.

- **`StreamClosedEvent`**: Indicates that the stream has been closed.

Receive Stream Listener

You can implement `ReceiveStreamListener` to receive notification whenever:

- New receive streams are created.
- The transfer of data starts or stops.
- The data transfer times out.
- A previously orphaned `ReceiveStream` has been associated with a `Participant`.
- An RTCP APP packet is received.
- The receive stream's format or payload changes.

You can also use this interface to get a handle on the stream and access the RTP `DataSource` so that you can create a `MediaHandler`.

There are seven types of events associated with a `ReceiveStream`:

- **`NewReceiveStreamEvent`**: Indicates that the session manager has created a new receive stream for a newly-detected source.
- **`ActiveReceiveStreamEvent:`** Indicates that the transfer of data has started.
- **`InactiveReceiveStreamEvent`**: Indicates that the transfer of data has stopped.
- **`TimeoutEvent`**: Indicates that the data transfer has timed out.
- **`RemotePayloadChangeEvent`**: Indicates that the format or payload of the receive stream has changed.
- **`StreamMappedEvent`**: Indicates that a previously orphaned receive stream has been associated with a participant.
- **`ApplicationEvent`**: Indicates that an RTCP APP packet has been received.

Remote Listener

You can implement `RemoteListener` to receive notification of events or RTP control messages received from a remote participants. You might want to implement `RemoteListener` in an application used to monitor the

session—it enables you to receive RTCP reports and monitor the quality of the session reception without having to receive data or information on each stream.

There are three types of events associated with a remote participant:

- **`ReceiverReportEvent`**: Indicates that an RTP receiver report has been received.
- **`SenderReportEvent`**: Indicates that an RTP sender report has been received.
- **`RemoteCollisionEvent`**: Indicates that two remote participants are using the same synchronization source ID (SSRC).

RTP Data

The streams within an RTP session are represented by `RTPStream` objects. There are two types of `RTPStreams`: `ReceiveStream` and `SendStream`. Each RTP stream has a buffer data source associated with it. For `ReceiveStreams`, this `DataSource` is always a `PushBufferDataSource`.

The session manager automatically constructs new receive streams as it detects additional streams arriving from remote participants. You construct new send streams by calling `createSendStream` on the session manager.

Data Handlers

The JMF RTP APIs are designed to be transport-protocol independent. A custom RTP data handler can be created to enable JMF to work over a specific transport protocol. The data handler is a `DataSource` that can be used as the media source for a `Player`.

The abstract class `RTPPushDataSource` defines the basic elements of a JMF RTP data handler. A data handler has both an input data stream (`PushSourceStream`) and an output data stream (`OuputDataStream`). A data handler can be used for either the data channel or the control channel of an RTP session. If it is used for the data channel, the data handler implements the `DataChannel` interface.

An `RTPSocket` is an `RTPPushDataSource` has both a data and control channel. Each channel has an input and output stream to stream data to and from the underlying network. An `RTPSocket` can export `RTPControls` to add dynamic payload information to the session manager.

Because a custom `RTPSocket` can be used to construct a `Player` through the `Manager`, JMF defines the name and location for custom `RTPSocket` implementations:

```
<protocol package-prefix>.media.protocol.rtpraw.DataSource
```

RTP Data Formats

All RTP-specific data uses an RTP-specific format encoding as defined in the `AudioFormat` and `VideoFormat` classes. For example, gsm RTP encapsulated packets have the encoding set to `AudioFormat.GSM_RTP`, while jpeg-encoded video formats have the encoding set to `VideoFormat.JPEG_RTP`.

`AudioFormat` defines four standard RTP-specific encoding strings:

```
public static final String ULAW_RTP = "JAUDIO_G711_ULAW/rtp";
public static final String DVI_RTP = "dvi/rtp";
public static final String G723_RTP = "g723/rtp";
public static final String GSM_RTP = "gsm/rtp";
```

`VideoFormat` defines three standard RTP-specific encoding strings:

```
public static final String JPEG_RTP = "jpeg/rtp";
public static final String H261_RTP = "h261/rtp";
public static final String H263_RTP = "h263/rtp";
```

RTP Controls

The RTP API defines one RTP-specific control, `RTPControl`. `RTPControl` is typically implemented by RTP-specific `DataSources`. It provides a mechanism to add a mapping between a dynamic payload and a `Format`. `RTPControl` also provides methods for accessing session statistics and getting the current payload `Format`.

`SessionManager` also extends the `Controls` interface, enabling a session manager to export additional `Controls` through the `getControl` and `getControls` methods. For example, the session manager can export a `BufferControl` to enable you to specify the buffer length and threshold.

Reception

The presentation of an incoming RTP stream is handled by a `Player`. To receive and present a single stream from an RTP session, you can use a

MediaLocator that describes the session to construct a Player. A media locator for an RTP session is of the form:

```
rtp://address:port[:ssrc]/content-type/[ttl]
```

The Player is constructed and connected to the first stream in the session.

If there are multiple streams in the session that you want to present, you need to use a session manager. You can receive notification from the session manager whenever a stream is added to the session and construct a Player for each new stream. Using a session manager also enables you to directly monitor and control the session.

Transmission

A session manager can also be used to initialize and control a session so that you can stream data across the network. The data to be streamed is acquired from a Processor.

For example, to create a send stream to transmit data from a live capture source, you would:

1. Create, initialize, and start a SessionManager for the session.
2. Construct a Processor using the appropriate capture DataSource.
3. Set the output format of the Processor to an RTP-specific format. An appropriate RTP packetizer codec must be available for the data format you want to transmit.
4. Retrieve the output DataSource from the Processor.
5. Call createSendStream on the session manager and pass in the DataSource.

You control the transmission through the SendStream start and stop methods.

When it is first started, the SessionManager behaves as a receiver (sends out RTCP receiver reports). As soon as a SendStream is created, it begins to send out RTCP sender reports and behaves as a sender host as long as one or more send streams exist. If all SendStreams are closed (not just stopped), the session manager reverts to being a passive receiver.

Extensibility

Like the other parts of JMF, the RTP capabilities can be enhanced and extended. The RTP APIs support a basic set of RTP formats and payloads. Advanced developers and technology providers can implement JMF plug-ins to support dynamic payloads and additional RTP formats.

Implementing Custom Packetizers and Depacketizers

To implement a custom packetizer or depacketizer, you implement the JMF `Codec` interface. (For general information about JMF plug-ins, see "Implementing JMF Plug-Ins" on page 85.)

9

Receiving and Presenting RTP Media Streams

JMF `Players` and `Processors` provide the presentation, capture, and data conversion mechanisms for RTP streams.

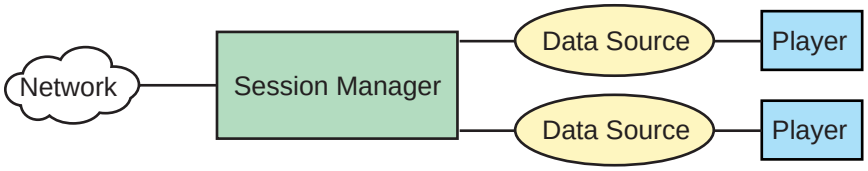


Figure 9-1: RTP reception data flow.

A separate player is used for each stream received by the session manager. You construct a `Player` for an RTP stream through the standard `Manager createPlayer` mechanism. You can either:

- Use a `MediaLocator` that has the parameters of the RTP session and construct a `Player` by calling `Manager.createPlayer(MediaLocator)`
- Construct a `Player` for a particular `ReceiveStream` by retrieving the `DataSource` from the stream and passing it to `Manager.createPlayer(DataSource)`.

If you use a `MediaLocator` to construct a `Player`, you can only present the first RTP stream that's detected in the session. If you want to play back multiple RTP streams in a session, you need to use the `SessionManager` directly and construct a `Player` for each `ReceiveStream`.

Creating a Player for an RTP Session

When you use a `MediaLocator` to construct a `Player` for an RTP session, the `Manager` creates a `Player` for the first stream detected in the session. This `Player` posts a `RealizeCompleteEvent` once data has been detected in the session.

By listening for the `RealizeCompleteEvent`, you can determine whether or not any data has arrived and if the `Player` is capable of presenting any data. Once the `Player` posts this event, you can retrieve its visual and control components.

Note: Because a `Player` for an RTP media stream doesn't finish realizing until data is detected in the session, you shouldn't try to use `Manager.createRealizedPlayer` to construct a `Player` for an RTP media stream. No `Player` would be returned until data arrives and if no data is detected, attempting to create a *Realized* `Player` would block indefinitely.

A `Player` can export one RTP-specific control, `RTPControl`, which provides overall session statistics and can be used for registering dynamic payloads with the `SessionManager`.

Example 9-1: Creating a `Player` for an RTP session (1 of 2)

```
String url= "rtp://224.144.251.104:49150/audio/1";

 MediaLocator mrl= new MediaLocator(url);

 if (mrl == null) {
     System.err.println("Can't build MRL for RTP");
     return false;
 }

 // Create a player for this rtp session
 try {
     player = Manager.createPlayer(mrl);
 } catch (NoPlayerException e) {
     System.err.println("Error:" + e);
     return false;
 } catch (MalformedURLException e) {
     System.err.println("Error:" + e);
     return false;
 } catch (IOException e) {
     System.err.println("Error:" + e);
     return false;
 }

 if (player != null) {
     if (this.player == null) {
```

Example 9-1: Creating a `Player` for an RTP session (2 of 2)

```
                this.player = player;
                player.addControllerListener(this);
                player.realize();
            }
        }
```

Listening for Format Changes

When a `Player` posts a `FormatChangeEvent`, it might indicate that a payload change has occurred. `Players` constructed with a `MediaLocator` automatically process payload changes. In most cases, this processing involves constructing a new `Player` to handle the new format. Applications that present RTP media streams need to listen for `FormatChangeEvents` so that they can respond if a new `Player` is created.

When a `FormatChangeEvent` is posted, check whether or not the `Player` object's control and visual components have changed. If they have, a new `Player` has been constructed and you need to remove references to the old `Player` object's components and get the new `Player` object's components.

Example 9-2: Listening for RTP format changes (1 of 2)

```
    public synchronized void controllerUpdate(ControllerEvent ce) {
        if (ce instanceof FormatChangeEvent) {
            Dimension vSize = new Dimension(320,0);
            Component oldVisualComp = visualComp;

            if ((visualComp = player.getVisualComponent()) != null) {
                if (oldVisualComp != visualComp) {
                    if (oldVisualComp != null) {
                        oldVisualComp.remove(zoomMenu);
                    }

                    framePanel.remove(oldVisualComp);

                    vSize = visualComp.getPreferredSize();
                    vSize.width = (int)(vSize.width * defaultScale);
                    vSize.height = (int)(vSize.height * defaultScale);

                    framePanel.add(visualComp);

                    visualComp.setBounds(0,
                                         0,
                                         vSize.width,
                                         vSize.height);
                    addPopupMenu(visualComp);
                }
```

Example 9-2: Listening for RTP format changes (2 of 2)

```
            }

            Component oldComp = controlComp;

            controlComp = player.getControlPanelComponent();

            if (controlComp != null)
            {
                if (oldComp != controlComp)
                {
                    framePanel.remove(oldComp);
                    framePanel.add(controlComp);

                    if (controlComp != null) {
                        int prefHeight = controlComp
                                         .getPreferredSize()
                                         .height;

                        controlComp.setBounds(0,
                                              vSize.height,
                                              vSize.width,
                                              prefHeight);
                    }
                }
            }
        }
    }
```

Creating an RTP Player for Each New Receive Stream

To play all of the `ReceiveStreams` in a session, you need to create a separate `Player` for each stream. When a new stream is created, the session manager posts a `NewReceiveStreamEvent`. Generally, you register as a `ReceiveStreamListener` and construct a `Player` for each new `ReceiveStream`. To construct the `Player`, you retrieve the `DataSource` from the `ReceiveStream` and pass it to `Manager.createPlayer`.

To create a `Player` for each new receive stream in a session:

1. Set up the RTP session:
    a. Create a `SessionManager`. For example, construct an instance of `com.sun.media.rtp.RTPSessionMgr`. (`RTPSessionMgr` is an implementation of `SessionManager` provided with the JMF reference implementation.)

b. Call `RTPSessionMgr addReceiveStreamListener` to register as a listener.

c. Initialize the RTP session by calling `RTPSessionMgr initSession`.

d. Start the RTP session by calling `RTPSessionMgr startSession`.

Example 9-3: Setting up an RTP session (1 of 2)

```
public SessionManager createManager(String address,
                                    int port,
                                    int ttl,
                                    boolean listener,
                                    boolean sendlistener)
{
    mgr = (SessionManager)new com.sun.media.rtp.RTPSessionMgr();

    if (mgr == null) return null;

    mgr.addFormat(new AudioFormat(AudioFormat.DVI_RTP,
                                  44100,
                                  4,
                                  1),
                  18);

    if (listener) mgr.addReceiveStreamListener(this);
    if (sendlistener) new RTPSendStreamWindow(mgr);

    // ask session mgr to generate the local participant's CNAME
    String cname = mgr.generateCNAME();
    String username = null;

    try {
        username = System.getProperty("user.name");
    } catch (SecurityException e){
        username = "jmf-user";
    }

    // create our local Session Address
    SessionAddress localaddr = new SessionAddress();

    try{
        InetAddress destaddr = InetAddress.getByName(address);

        SessionAddress sessaddr = new SessionAddress(destaddr,
                                                     port,
                                                     destaddr,
                                                     port + 1);
        SourceDescription[] userdesclist= new SourceDescription[]
        {
            new SourceDescription(SourceDescription
                                  .SOURCE_DESC_EMAIL,
                                  "jmf-user@sun.com",
```

Example 9-3: Setting up an RTP session (2 of 2)

```
                                    1,
                                    false),

                new SourceDescription(SourceDescription
                                    .SOURCE_DESC_CNAME,
                                    cname,
                                    1,
                                    false),

                new SourceDescription(SourceDescription
                                    .SOURCE_DESC_TOOL,
                                    "JMF RTP Player v2.0",
                                    1,
                                    false)
            };

            mgr.initSession(localaddr,
                            userdesclist,
                            0.05,
                            0.25);

            mgr.startSession(sessaddr,ttl,null);
        } catch (Exception e) {
            System.err.println(e.getMessage());
            return null;
        }

        return mgr;
    }
```

2. In your `ReceiveStreamListener update` method, watch for `NewReceiveStreamEvent`, which indicates that a new data stream has been detected.

3. When a `NewReceiveStreamEvent` is detected, retrieve the `ReceiveStream` from the `NewReceiveStreamEvent` by calling `getReceiveStream`.

4. Retrieve the RTP `DataSource` from the `ReceiveStream` by calling `getDataSource`. This is a `PushBufferDataSource` with an RTP-specific `Format`. For example, the encoding for a DVI audio player will be DVI_RTP.

5. Pass the `DataSource` to `Manager.createPlayer` to construct a `Player`. For the `Player` to be successfully constructed, the necessary plug-ins for decoding and depacketizing the RTP-formatted data must be available. (For more information, see "Creating Custom Packetizers and Depacketizers" on page 167).

Example 9-4: Listening for `NewReceiveStreamEvents`

```
public void update( ReceiveStreamEvent event)
{
    Player newplayer = null;
    RTPPlayerWindow playerWindow = null;

    // find the sourceRTPSM for this event
    SessionManager source = (SessionManager)event.getSource();

    // create a new player if a new recvstream is detected
    if (event instanceof NewReceiveStreamEvent)
    {
        String cname = "Java Media Player";
        ReceiveStream stream = null;

        try
        {
            // get a handle over the ReceiveStream
            stream =((NewReceiveStreamEvent)event)
                    .getReceiveStream();

            Participant part = stream.getParticipant();

            if (part != null) cname = part.getCNAME();

            // get a handle over the ReceiveStream datasource
            DataSource dsource = stream.getDataSource();

            // create a player by passing datasource to the
            // Media Manager
            newplayer = Manager.createPlayer(dsource);
            System.out.println("created player " + newplayer);
        } catch (Exception e) {
            System.err.println("NewReceiveStreamEvent exception "
                               + e.getMessage());
            return;
        }

        if (newplayer == null) return;

        playerlist.addElement(newplayer);
        newplayer.addControllerListener(this);

        // send this player to player GUI
        playerWindow = new RTPPlayerWindow( newplayer, cname);
    }
}
```

See `RTPUtil` in "RTPUtil" on page 223 for a complete example.

Handling RTP Payload Changes

If the payload of a stream in the RTP session changes, the `ReceiveStream` posts a `RemotePayloadChangeEvent`. Generally, when the payload changes, the existing `Player` will not be able to handle the new format and JMF will throw an error if you attempt to present the new payload. To avoid this, your `ReceiveStreamListener` needs to watch for `RemotePayloadChangeEvents`. When a `RemotePayloadChangeEvent` is detected, you need to:

1. Close the existing `Player`.
2. Remove all listeners for the removed `Player`.
3. Create a new `Player` with the same RTP `DataSource`.
4. Get the visual and control `Components` for the new `Player`.
5. Add the necessary listeners to the new `Player`.

Example 9-5: Handling RTP payload changes (1 of 2)

```
public void update(ReceiveStreamEvent event) {
    if (event instanceof RemotePayloadChangeEvent) {
        // payload has changed. we need to close the old player
        // and create a new player

        if (newplayer != null) {
            // stop player and wait for stop event
            newplayer.stop();

            // block until StopEvent received...

            // remove controllerlistener
            newplayer.removeControllerListener(listener);

            // remove any visual and control components
            // attached to this application
            // close the player and wait for close event
            newplayer.close();

            // block until ControllerClosedEvent received...

            try {
                // when the player was closed, its datasource was
                // disconnected. Now we must reconnect the data-
                // source before a player can be created for it.
```

Example 9-5: Handling RTP payload changes (2 of 2)

```
                    // This is the same datasource received from
                    // NewReceiveStreamEvent and used to create the
                    // initial rtp player

                    rtpsource.connect();
                    newplayer = Manager.createPlayer(rtpsource);

                    if (newplayer == null) {
                        System.err.println("Could not create player");
                        return;
                    }

                    newplayer.addControllerListener(listener);
                    newplayer.realize();

                    // when the new player is realized, retrieve its
                    // visual and control components
                } catch (Exception e) {
                    System.err.println("could not create player");
                }
            }
        }
    }
```

Controlling Buffering of Incoming RTP Streams

You can control the RTP receiver buffer through the `BufferControl` exported by the `SessionManager`. This control enables you to set two parameters, buffer length and threshold.

The buffer length is the size of the buffer maintained by the receiver. The threshold is the minimum amount of data that is to be buffered by the control before pushing data out or allowing data to be pulled out (jitter buffer). Data will only be available from this object when this minimum threshold has been reached. If the amount of data buffered falls below this threshold, data will again be buffered until the threshold is reached.

The buffer length and threshold values are specified in milliseconds. The number of audio packets or video frames buffered depends on the format of the incoming stream. Each receive stream maintains its own default and maximum values for both the buffer length and minimum threshold. (The default and maximum buffer lengths are implementation dependent.)

To get the `BufferControl` for a session, you call `getControl` on the `SessionManager`. You can retrieve a GUI `Component` for the `BufferControl` by calling `getControlComponent`.

Presenting RTP Streams with RTPSocket

RTP is transport-protocol independent. By using RTPSocket, you can stream RTP from any underlying network. The format of the RTP socket is designed to have both a data and a control channel. Each channel has an input and output stream to stream data into and out of the underlying network.

SessionManager expects to receive individual RTP packets from the RTPSocket. Users are responsible for streaming individual RTP packets to the RTPSocket.

To play an RTP stream from the RTPSocket, you pass the socket to Manager.createPlayer to construct the Player. Alternatively, you could construct a Player by calling createPlayer(MediaLocator) and passing in a MediaLocator with a new protocol that is a variant of RTP, "rtpraw". For example:

```
Manager.createPlayer(new MediaLocator("rtpraw://"));
```

According to the JMF Player creation mechanism, Manager will attempt to construct the DataSource defined in:

```
<protocol package-prefix>.media.protocol.rtpraw.DataSource
```

This must be the RTPSocket. The content of the RTPsocket should be set to rtpraw. Manager will then attempt to create a player of type <content-prefix>.media.content.rptraw.Handler and set the RTPSocket on it.

Note: The RTPSocket created at <protocol package-prefix>.media.protocol.rtpraw.DataSource is your own implementation of RTPSocket. The JMF API does not define a default implementation of RTPSocket. The implementation of RTPSocket is dependent on the underlying transport protocol that you are using. Your RTPSocket class must be located at <protocol package-prefix>.media.protocol.rtpraw.DataSource and its control and data channel streams must be set as shown in the following example.

RTPControl interfaces for the RTPSocket can be used to add dynamic payload information to the RTP session manager.

The following example implements an RTP over UDP player that can receive RTP UDP packets and stream them to the Player or session manager, which is not aware of the underlying network/transport protocol.

This sample uses the interfaces defined in `javax.media.rtp.RTPSocket` and its related classes.

Example 9-6: `RTPSocketPlayer` (1 of 6)

```
import java.io.*;
import java.net.*;
import java.util.*;

import javax.media.*;
import javax.media.format.*;
import javax.media.protocol.*;
import javax.media.rtp.*;
import javax.media.rtp.event.*;
import javax.media.rtp.rtcp.*;

public class RTPSocketPlayer implements ControllerListener {
    // ENTER THE FOLLOWING SESSION PARAMETERS FOR YOUR RTP SESSION

    // RTP Session address, multicast, unicast or broadcast address
    String address = "224.144.251.245";

    // RTP Session port
    int port = 49150;

    // Media Type i.e. one of audio or video
    String media = "audio";

    // DO NOT MODIFY ANYTHING BELOW THIS LINE

    // The main rtpsocket abstraction which we will create and send
    // to the Manager for appropriate handler creation
    RTPSocket rtpsocket = null;

    // The control RTPPushDataSource of the above RTPSocket
    RTPPushDataSource rtcpsource = null;

    // The GUI to handle the player
    // PlayerWindow playerWindow;

    // The handler created for the RTP session,
    // as returned by the Manager
    Player player;

    // maximum size of buffer for UDP receive from the sockets
    private  int maxsize = 2000;

    UDPHandler rtp = null;
    UDPHandler rtcp = null;

    public RTPSocketPlayer() {
        // create the RTPSocket
        rtpsocket = new RTPSocket();
```

Example 9-6: `RTPSocketPlayer` (2 of 6)

```
        // set its content type :
        // rtpraw/video for a video session
        // rtpraw/audio for an audio session
        String content = "rtpraw/" + media;
        rtpsocket.setContentType(content);

        // set the RTP Session address and port of the RTP data
        rtp = new UDPHandler(address, port);

        // set the above UDP Handler to be the
        // sourcestream of the rtpsocket
        rtpsocket.setOutputStream(rtp);

        // set the RTP Session address and port of the RTCP data
        rtcp = new UDPHandler(address, port +1);

        // get a handle over the RTCP Datasource so that we can
        // set the sourcestream and deststream of this source
        // to the rtcp udp handler we created above.
        rtcpsource = rtpsocket.getControlChannel();

        // Since we intend to send RTCP packets from the
        // network to the session manager and vice-versa, we need
        // to set the RTCP UDP handler as both the input and output
        // stream of the rtcpsource.
        rtcpsource.setOutputStream(rtcp);
        rtcpsource.setInputStream(rtcp);

        // connect the RTP socket data source before
        // creating the player
        try {
            rtpsocket.connect();
            player = Manager.createPlayer(rtpsocket);
            rtpsocket.start();
        } catch (NoPlayerException e) {
            System.err.println(e.getMessage());
            e.printStackTrace();
            return;
        }
        catch (IOException e) {
            System.err.println(e.getMessage());
            e.printStackTrace();
            return;
        }

        if (player != null) {
            player.addControllerListener(this);
            // send this player to out playerwindow
            // playerWindow = new PlayerWindow(player);
        }
    }
```

Example 9-6: `RTPSocketPlayer` (3 of 6)

```
    public synchronized void controllerUpdate(ControllerEvent ce) {
        if ((ce instanceof DeallocateEvent) ||
            (ce instanceof ControllerErrorEvent)) {

            // stop udp handlers
            if (rtp != null) rtp.close();

            if (rtcp != null) rtcp.close();
        }
    }

    // method used by inner class UDPHandler to open a datagram or
    // multicast socket as the case maybe

    private DatagramSocket InitSocket(String sockaddress,
                                      int     sockport)
    {
        InetAddress addr = null;
        DatagramSocket sock = null;

        try {
            addr = InetAddress.getByName(sockaddress);

            if (addr.isMulticastAddress()) {
                MulticastSocket msock;

                msock = new MulticastSocket(sockport);

                msock.joinGroup(addr);

                sock = (DatagramSocket)msock;
            }
            else {
                sock = new DatagramSocket(sockport,addr);
            }

            return sock;
        }
        catch (SocketException e) {
            e.printStackTrace();
            return null;
        }
        catch (UnknownHostException e) {
            e.printStackTrace();
            return null;
        }
        catch (IOException e) {
            e.printStackTrace();
            return null;
        }
    }
```

Example 9-6: RTPSocketPlayer (4 of 6)

```
// INNER CLASS UDP Handler which will receive UDP RTP Packets and
// stream them to the handler of the sources stream. IN case of
// RTCP, it will also accept RTCP packets and send them on the
// underlying network.

public class UDPHandler extends Thread implements PushSourceStream,
                                                   OutputDataStream
{
    DatagramSocket        mysock;
    DatagramPacket        dp;
    SourceTransferHandler outputHandler;
    String                myAddress;
    int                   myport;
    boolean               closed = false;


    // in the constructor we open the socket and create the main
    // UDPHandler thread.

    public UDPHandler(String haddress, int hport) {
        myAddress = haddress;
        myport = hport;
        mysock = InitSocket(myAddress,myport);
        setDaemon(true);
        start();
    }

    // the main thread receives RTP data packets from the
    // network and transfer's this data to the output handler of
    // this stream.

    public void run() {
        int len;

        while(true) {
            if (closed) {
                cleanup();
                return;
            }
            try {
                do {
                    dp = new DatagramPacket( new byte[maxsize],
                                             maxsize);

                    mysock.receive(dp);

                    if (closed){
                        cleanup();
                        return;
                    }

                    len = dp.getLength();
```

Example 9-6: `RTPSocketPlayer` (5 of 6)

```
                        if (len > (maxsize >> 1)) maxsize = len << 1;
                    }
                    while (len >= dp.getData().length);
                }catch (Exception e){
                    cleanup();
                    return;
                }

                if (outputHandler != null) {
                    outputHandler.transferData(this);
                }
            }
        }

        public void close() {
            closed = true;
        }

        private void cleanup() {
            mysock.close();
            stop();
        }

        // methods of PushSourceStream
        public Object[] getControls() {
            return new Object[0];
        }

        public Object getControl(String controlName) {
            return null;
        }

        public ContentDescriptor getContentDescriptor() {
            return null;
        }

        public long getContentLength() {
            return SourceStream.LENGTH_UNKNOWN;
        }

        public boolean endOfStream() {
            return false;
        }

        // method by which data is transferred from the underlying
        // network to the session manager.

        public int read(byte buffer[],
                        int offset,
                        int length)
        {
```

Example 9-6: `RTPSocketPlayer` (6 of 6)

```
            System.arraycopy(dp.getData(),
                             0,
                             buffer,
                             offset,
                             dp.getLength());

            return dp.getData().length;
        }

        public int getMinimumTransferSize(){
            return dp.getLength();
        }

        public void setTransferHandler(SourceTransferHandler
                                       transferHandler)
        {
            this.outputHandler = transferHandler;
        }

        // methods of OutputDataStream used by the session manager to
        // transfer data to the underlying network.

        public int write(byte[] buffer,
                         int offset,
                         int length)
        {
            InetAddress addr = null;

            try {
                addr = InetAddress.getByName(myAddress);
            } catch (UnknownHostException e) {
                e.printStackTrace();
            }

            DatagramPacket dp = new DatagramPacket( buffer,
                                                    length,
                                                    addr,
                                                    myport);
            try {
                mysock.send(dp);
            } catch (IOException e){
                e.printStackTrace();
            }

            return dp.getLength();
        }
    }

    public static void main(String[] args) {
        new RTPSocketPlayer();
    }
}
```

10

Transmitting RTP Media Streams

To transmit an RTP stream, you use a `Processor` to produce an RTP-encoded `DataSource` and construct either a `SessionManager` or `DataSink` to control the transmission.

The input to the `Processor` can be either stored or live captured data. For stored data, you can use a `MediaLocator` to identify the file when you create the `Processor`. For captured data, a capture `DataSource` is used as the input to the `Processor`, as described in "Capturing Media Data" on page 78.

There are two ways to transmit RTP streams:

- Use a `MediaLocator` that has the parameters of the RTP session to construct an RTP `DataSink` by calling `Manager.createDataSink`.
- Use a session manager to create send streams for the content and control the transmission.

If you use a `MediaLocator` to construct an RTP `DataSink`, you can only transmit the first stream in the `DataSource`. If you want to transmit multiple RTP streams in a session or need to monitor session statistics, you need to use the `SessionManager` directly.

Regardless of how you choose to transmit the RTP stream, you need to:

1. Create a `Processor` with a `DataSource` that represents the data you want to transmit.
2. Configure the `Processor` to output RTP-encoded data.
3. Get the output from the `Processor` as a `DataSource`.

Configuring the Processor

To configure the Processor to generate RTP-encoded data, you set RTP-specific formats for each track and specify the output content descriptor you want.

The track formats are set by getting the TrackControl for each track and calling setFormat to specify an RTP-specific format. An RTP-specific format is selected by setting the encoding string of the format to an RTP-specific string such as "AudioFormat.GSM_RTP". The Processor attempts to load a plug-in that supports this format. If no appropriate plug-in is installed, that particular RTP format cannot be supported and an UnSupportedFormatException is thrown.

The output format is set with the setOutputContentDescriptor method. If no special multiplexing is required, the output content descriptor can be set to "ContentDescriptor.RAW". Audio and video streams should not be interleaved. If the Processor's tracks are of different media types, each media stream is transmitted in a separate RTP session.

Retrieving the Processor Output

Once the format of a Processor's track has been set and the Processor has been realized, the output DataSource of the Processor can be retrieved. You retrieve the output of the Processor as a DataSource by calling getDataOutput. The returned DataSource can be either a PushBufferDataSource or a PullBufferDataSource, depending on the source of the data.

The output DataSource is connected to the SessionManager using the createSendStream method. The session manager must be initialized before you can create the send stream.

If the DataSource contains multiple SourceStreams, each SourceStream is sent out as a separate RTP stream, either in the same session or a different session. If the DataSource contains both audio and video streams, separate RTP sessions must be created for audio and video. You can also clone the DataSource and send the clones out as different RTP streams in either the same session or different sessions.

Controlling the Packet Delay

The packet delay, also known as the packetization interval, is the time represented by each RTP packet as it is transmitted over the network. The packetization interval determines the minimum end-to-end delay; longer

packets introduce less header overhead but higher delay and make packet loss more noticeable. For non-interactive applications such as lectures, or for links with severe bandwidth constraints, a higher packetization delay might be appropriate.

A receiver should accept packets representing between 0 and 200 ms of audio data. (For framed audio encodings, a receiver should accept packets with 200 ms divided by the frame duration, rounded up.) This restriction allows reasonable buffer sizing for the receiver. Each packetizer codec has a default packetization interval appropriate for its encoding.

If the codec allows modification of this interval, it exports a corresponding `PacketSizeControl`. The packetization interval can be changed or set by through the `setPacketSize` method.

For video streams, a single video frame is transmitted in multiple RTP packets. The size of each packet is limited by the Maximum Transmission Unit (MTU) of the underlying network. This parameter is also set using the `setPacketSize` method of the packetizer codec's `PacketSizeControl`.

Transmitting RTP Data With a Data Sink

The simplest way to transmit RTP data is to construct an RTP `DataSink` using the `Manager.createDataSink` method. You pass in the output `DataSource` from the `Processor` and a `MediaLocator` that describes the RTP session to which the `DataSource` is to be streamed. (The `MediaLocator` provides the address and port of the RTP session.)

To control the transmission, you call `start` and `stop` on the `DataSink`. Only the first stream in the `DataSource` is transmitted.

In Example 10-1, live audio is captured and then transmitted using a `DataSink`.

Example 10-1: Transmitting RTP Data using a `DataSink` (1 of 2)

```
// First find a capture device that will capture linear audio
// data at 8bit 8Khz

AudioFormat format= new AudioFormat(AudioFormat.LINEAR,
                                    8000,
                                    8,
                                    1);

Vector devices= CaptureDeviceManager.getDeviceList( format);

CaptureDeviceInfo di= null;

if (devices.size() > 0) {
     di = (CaptureDeviceInfo) devices.elementAt( 0);
}
else {
    // exit if we could not find the relevant capturedevice.
    System.exit(-1);
}

// Create a processor for this capturedevice & exit if we
// cannot create it
try {
    Processor p = Manager.createProcessor(di.getLocator());
} catch (IOException e) {
    System.exit(-1);
} catch (NoProcessorException e) {
    System.exit(-1);
}

// configure the processor
processor.configure();

// block until it has been configured

processor.setContentDescriptor(
    new ContentDescriptor( ContentDescriptor.RAW));

TrackControl track[] = processor.getTrackControls();

boolean encodingOk = false;

// Go through the tracks and try to program one of them to
// output gsm data.

 for (int i = 0; i < track.length; i++) {
    if (!encodingOk && track[i] instanceof FormatControl) {
```

Example 10-1: Transmitting RTP Data using a `DataSink` (2 of 2)

```
                if (((FormatControl)track[i]).
                    setFormat( new AudioFormat(AudioFormat.GSM_RTP,
                                               8000,
                                               8,
                                               1)) == null) {

                   track[i].setEnabled(false);
                }
                else {
                    encodingOk = true;
                }
            } else {
                // we could not set this track to gsm, so disable it
                track[i].setEnabled(false);
            }
        }

        // At this point, we have determined where we can send out
        // gsm data or not.
        // realize the processor
        if (encodingOk) {
            processor.realize();
            // block until realized.
            // get the output datasource of the processor and exit
            // if we fail
            DataSource ds = null;

            try {
                ds = processor.getDataOutput();
            } catch (NotRealizedError e) {
                System.exit(-1);
            }

            // hand this datasource to manager for creating an RTP
            // datasink our RTP datasimnk will multicast the audio
            try {
                String url= "rtp://224.144.251.104:49150/audio/1";

                MediaLocator m = new MediaLocator(url);

                DataSink d = Manager.createDataSink(ds, m);

                d.open();
                d.start();
            } catch (Exception e) {
                System.exit(-1);
            }
        }
```

Transmitting RTP Data with the Session Manager

The basic process for transmitting RTP data with the session manager is:

1. Create a JMF `Processor` and set each track format to an RTP-specific format.
2. Retrieve the output `DataSource` from the `Processor`.
3. Call `createSendStream` on a previously created and initialized `SessionManager`, passing in the `DataSource` and a stream index. The session manager creates a `SendStream` for the specified `SourceStream`.
4. Start the session manager by calling `SessionManager startSession`.
5. Control the transmission through the `SendStream` methods. A `SendStreamListener` can be registered to listen to events on the `SendStream`.

Creating a Send Stream

Before the session manager can transmit data, it needs to know where to get the data to transmit. When you construct a new `SendStream`, you hand the `SessionManager` the `DataSource` from which it will acquire the data. Since a `DataSource` can contain multiple streams, you also need to specify the index of the stream to be sent in this session. You can create multiple send streams by passing different `DataSources` to `createSendStream` or by specifying different stream indexes.

The session manager queries the format of the `SourceStream` to determine if it has a registered payload type for this format. If the format of the data is not an RTP format or a payload type cannot be located for the RTP format, an `UnSupportedFormatException` is thrown with the appropriate message. Dynamic payloads can be associated with an RTP format using the `SessionManager addFormat` method

Using Cloneable Data Sources

Many RTP usage scenarios involve sending a stream over multiple RTP sessions or encoding a stream into multiple formats and sending them over multiple RTP sessions. When a stream encoded in a single format has to be sent over multiple RTP sessions, you need to clone the `DataSource` output from the `Processor` from which data is being captured. This is done by creating a cloneable `DataSource` through the `Manager` and calling `getClone` on the cloneable `DataSource`. A new `Processor` can be created

from each cloned `DataSource`, its tracks encoded in the desired format, and the stream sent out over an RTP session.

Using Merging Data Sources

If you want to mix multiple media streams of the same type (such as audio) into a single stream going out from one source, you need to use an RTP mixer. If the streams to be mixed originate from multiple `DataSources`, you can create a `MergingDataSource` from the separate `DataSources` and hand it to the `SessionManager` to create the stream.

Controlling a Send Stream

You use the `RTPStream` `start` and `stop` methods to control a `SendStream`. Starting a `SendStream` begins data transfer over the network and stopping a `SendStream` indicates halts the data transmission. To begin an RTP transmission, each `SendStream` needs to be started.

Starting or stopping a send stream triggers the corresponding action on its `DataSource`. However, if the `DataSource` is started independently while the `SendStream` is stopped, data will be dropped (`PushBufferDataSource`) or not pulled (`PullBufferDataSource`) by the session manager. During this time, no data will be transmitted over the network.

Sending Captured Audio Out in a Single Session

Example 10-2 captures mono audio data and sends it out on an RTP session.

Example 10-2: Sending captured audio out on a single session (1 of 3)

```
// First, we'll need a DataSource that captures live audio:

AudioFormat format = new AudioFormat(AudioFormat.ULAW,
                                     8000,
                                     8,
                                     1);

Vector devices= CaptureDeviceManager.getDeviceList( format);

CaptureDeviceInfo di= null;
if (devices.size() > 0) {
     di = (CaptureDeviceInfo) devices.elementAt( 0);
}
```

Example 10-2: Sending captured audio out on a single session (2 of 3)

```
        else {
            // exit if we could not find the relevant capture device.
            System.exit(-1);
        }
        // Create a processor for this capture device & exit if we
        // cannot create it
        try {
            Processor p = Manager.createProcessor(di.getLocator());
        } catch (IOException e) {
            System.exit(-1);
        } catch (NoProcessorException e) {
            System.exit(-1);
        }

        // at this point, we have succesfully created the processor.
        // Realize it and block until it is configured.

        processor.configure();

        // block until it has been configured

        processor.setContentDescriptor(
            new ContentDescriptor( ContentDescriptor.RAW));

        TrackControl track[] = processor.getTrackControls();

        boolean encodingOk = false;

        // Go through the tracks and try to program one of them to
        // output ULAW_RTP data.
        for (int i = 0; i < track.length; i++) {
            if (!encodingOk && track[i] instanceof FormatControl) {

                if (((FormatControl)track[i]).
                    setFormat( new AudioFormat(AudioFormat.ULAW_RTP,
                                               8000,
                                               8,
                                               1)) == null) {

                    track[i].setEnabled(false);
                }
                else {
                    encodingOk = true;
                }
            }
            else {
                // we could not set this track to gsm, so disable it
                track[i].setEnabled(false);
            }
        }
```

Example 10-2: Sending captured audio out on a single session (3 of 3)

```
// Realize it and block until it is realized.
processor.realize();

// block until realized.
// get the output datasource of the processor  and exit
// if we fail

DataSource ds = null;

try {
    ds = processor.getDataOutput();
} catch (NotRealizedError e){
    System.exit(-1);
}

// Create a SessionManager and hand over the
// datasource for SendStream creation.

SessionManager rtpsm = new com.sun.media.rtp.RTPSessionMgr();

// The session manager then needs to be initialized and started:
// rtpsm.initSession(...);
// rtpsm.startSession(...);

try {
    rtpsm.createSendStream(ds, 0);
} catch (IOException e){
    e.printStackTrace();
} catch( UnsupportedFormatException e) {
    e.printStackTrace();
}
```

Sending Captured Audio Out in Multiple Sessions

Example 10-3 and Example 10-4 both encode the captured audio and send it out in multiple RTP sessions. In Example 10-3, the data is encoded in gsm; in Example 10-4, the data is encoded in several different formats.

Example 10-3: Sending RTP data out in multiple sessions (1 of 4)

```
// First find a capture device that will capture linear audio
// data at 8bit 8Khz

AudioFormat format= new AudioFormat(AudioFormat.LINEAR,
                                    8000,
                                    8,
                                    1);
```

Example 10-3: Sending RTP data out in multiple sessions (2 of 4)

```
Vector devices= CaptureDeviceManager.getDeviceList( format);

CaptureDeviceInfo di= null;

if (devices.size() > 0) {
     di = (CaptureDeviceInfo) devices.elementAt( 0);
}
else {
    // exit if we could not find the relevant capturedevice.
    System.exit(-1);
}

// Now create a processor for this capturedevice & exit if we
// cannot create it
try {
    Processor p = Manager.createProcessor(di.getLocator());
} catch (IOException e) {
    System.exit(-1);
} catch (NoProcessorException e) {
    System.exit(-1);
}

// configure the processor
processor.configure();

// block until it has been configured

processor.setContentDescriptor(
    new ContentDescriptor( ContentDescriptor.RAW));

TrackControl track[] = processor.getTrackControls();

boolean encodingOk = false;

// Go through the tracks and try to program one of them to
// output gsm data.

for (int i = 0; i < track.length; i++) {
    if (!encodingOk && track[i] instanceof FormatControl) {

        if (((FormatControl)track[i]).
            setFormat( new AudioFormat(AudioFormat.GSM_RTP,
                                       8000,
                                       8,
                                       1)) == null) {

            track[i].setEnabled(false);
        }
        else {
            encodingOk = true;
        }
    }
```

Example 10-3: Sending RTP data out in multiple sessions (3 of 4)

```
            else {
                // we could not set this track to gsm, so disable it
                track[i].setEnabled(false);
            }
        }

        // At this point, we have determined where we can send out
        // gsm data or not.
        // realize the processor

        if (encodingOk) {
            processor.realize();

            // block until realized.

            // get the output datasource of the processor  and exit
            // if we fail

            DataSource origDataSource = null;

            try {
                origDataSource = processor.getDataOutput();
            } catch (NotRealizedError e) {
                System.exit(-1);
            }

            // We want to send the stream of this datasource over two
            // RTP sessions.

            // So we need to clone the output datasource of the
            // processor and hand the clone over to the second
            // SessionManager

            DataSource cloneableDataSource = null;
            DataSource clonedDataSource = null;

            cloneableDataSource
              = Manager.createCloneableDataSource(origDataSource);

            clonedDataSource
              = ((SourceCloneable)cloneableDataSource).createClone();

            // Now create the first SessionManager and hand over the
            // first datasource for SendStream creation.

            SessionManager rtpsm1
              = new com.sun.media.rtp.RTPSessionMgr();

            // The session manager then needs to be
            // initialized and started:
            // rtpsm1.initSession(...);
            // rtpsm1.startSession(...);
```

Example 10-3: Sending RTP data out in multiple sessions (4 of 4)

```
            try {
            rtpsm1.createSendStream(cloneableDataSource, // Datasource 1
                                    0);

            } catch (IOException e) {
                e.printStackTrace();
            } catch( UnsupportedFormatException e) {
                e.printStackTrace();
            }

            try {
                cloneableDataSource.connect();
                cloneableDataSource.start();
            } catch (IOException e) {
                e.printStackTrace();
            }

            // create the second RTPSessionMgr and hand over the
            // cloned datasource
            if (clonedDataSource != null) {
                SessionManager rtpsm2
                  = new com.sun.media.rtp.RTPSessionMgr();

                // rtpsm2.initSession(...);
                // rtpsm2.startSession(...);

                try {
                    rtpsm2.createSendStream(clonedDataSource,0);
                } catch (IOException e) {
                    e.printStackTrace();
                } catch( UnsupportedFormatException e) {
                    e.printStackTrace();
                }
            }
        }
        else {
            // we failed to set the encoding to gsm. So deallocate
            // and close the processor before we leave.

            processor.deallocate();
            processor.close();
        }
```

Example 10-4 encodes captured audio in several formats and then sends it out in multiple RTP sessions. It assumes that there is one stream in the input `DataSource`.

The input `DataSource` is cloned and a second processor is created from the clone. The tracks in the two `Processors` are individually set to gsm and dvi and the output `DataSources` are sent to two different RTP session man-

agers. If the number of tracks is greater than 1, this example attempts to set the encoding of one track to gsm and the other to dvi. The same `DataSource` is handed to two separate RTP session managers with the index of the first stream set to 0 and the index of the second stream set to 1 (for heterogeneous receivers).

Example 10-4: Encoding and sending data in multiple formats (1 of 3)

```
// Find a capture device that will capture linear 8bit 8Khz
// audio

AudioFormat format = new AudioFormat(AudioFormat.LINEAR,
                                     8000,
                                     8,
                                     1);

Vector devices= CaptureDeviceManager.getDeviceList( format);

CaptureDeviceInfo di= null;

if (devices.size() > 0) {
     di = (CaptureDeviceInfo) devices.elementAt( 0);
}
else {
    // exit if we could not find the relevant capture device.
    System.exit(-1);
}

// Since we have located a capturedevice, create a data
// source for it.

DataSource origDataSource= null;

try {
    origDataSource = Manager.createDataSource(di.getLocator());
} catch (IOException e) {
    System.exit(-1);
} catch (NoDataSourceException e) {
   System.exit(-1);
}

SourceStream streams[] = ((PushDataSource)origDataSource)
                            .getStreams();

DataSource cloneableDataSource = null;
DataSource clonedDataSource = null;

if (streams.length == 1) {
    cloneableDataSource
      = Manager.createCloneableDataSource(origDataSource);
```

Example 10-4: Encoding and sending data in multiple formats (2 of 3)

```
            clonedDataSource
              = ((SourceCloneable)cloneableDataSource).createClone();
        }
        else {
            // DataSource has more than 1 stream and we should try to
            // set the encodings of these streams to dvi and gsm
        }

        // at this point, we have a cloneable data source and its clone,
        // Create one processor from each of these datasources.

       Processor p1 = null;

        try {
            p1 = Manager.createProcessor(cloneableDataSource);
        } catch (IOException e) {
            System.exit(-1);
        } catch (NoProcessorException e) {
            System.exit(-1);
        }

        p1.configure();

        // block until configured.

        TrackControl track[] = p1.getTrackControls();
        boolean encodingOk = false;

        // Go through the tracks and try to program one of them
        // to output gsm data
        for (int i = 0; i < track.length; i++) {
            if (!encodingOk && track[i] instanceof FormatControl) {
                if (((FormatControl)track[i]).
                    setFormat( new AudioFormat(AudioFormat.GSM_RTP,
                                               8000,
                                               8,
                                               1)) == null) {

                    track[i].setEnabled(false);
                }
                else {
                    encodingOk = true;
                }
            }
            else {
                track[i].setEnabled(false);
            }
        }
```

Example 10-4: Encoding and sending data in multiple formats (3 of 3)

```
        if (encodingOk) {
            processor.realize();
            // block until realized.
            // ...
            // get the output datasource of the processor
            DataSource ds = null;

            try {
                ds = processor.getDataOutput();
            } catch (NotRealizedError e) {
                System.exit(-1);
            }

            // Now create the first SessionManager and hand over the
            // first datasource for SendStream creation .

            SessionManager rtpsm1
              = new com.sun.media.rtp.RTPSessionMgr();

            // rtpsm1.initSession(...);
            // rtpsm1.startSession(...);

            try {
                rtpsm1.createSendStream(ds, // first datasource
                                        0); // first sourcestream of
                                            // first datasource
            } catch (IOException e) {
                e.printStackTrace();
            } catch( UnsupportedFormatException e) {
                e.printStackTrace();
            }
        }

        // Now repeat the above with the cloned data source and
        // set the encoding to dvi. i.e create a processor with
        // inputdatasource clonedDataSource
        // and set encoding of one of its tracks to dvi.
        // create SessionManager giving it the output datasource of
        // this processor.
```

Transmitting RTP Streams with RTPSocket

You can also use `RTPSocket` to transmit RTP media streams. To use `RTPSocket` for transmission, you create an RTP `DataSink` with `createDataSink` by passing in a a `MediaLocator` with a new protocol that is a variant of RTP, "`Ratibor`". Manager attempts to construct a `DataSink` from:

```
<protocol package-prefix>.media.datasink.rtpraw.Handler
```

The session manager prepares individual RTP packets that are ready to be transmitted across the network and sends them to the RTPSocket created from:

```
<protocol package-prefix>.media.protocol.rtpraw.DataSource
```

The RTPSocket created at <protocol-prefix>.media.protocol.rtpraw.DataSource is your own implementation of RTPSocket. The JMF API does not define a default implementation of RTPSocket. The implementation of RTPSocket is dependent on the underlying transport protocol that you are using. Your RTPSocket class must be located at <protocol-prefix>.media.protocol.rtpraw.DataSource.

You're responsible for transmitting the RTP packets out on the underlying network

In the following example, an RTPSocket is used to transmitting captured audio:

Example 10-5: Transmitting RTP data with RTPSocket (1 of 3)

```
// Find a capture device that will capture linear audio
// data at 8bit 8Khz

AudioFormat format = new AudioFormat(AudioFormat.LINEAR,
                                     8000,
                                     8,
                                     1);

Vector devices= CaptureDeviceManager.getDeviceList( format);

CaptureDeviceInfo di= null;
if (devices.size() > 0) {
     di = (CaptureDeviceInfo) devices.elementAt( 0);
}
else {
    // exit if we could not find the relevant capture device.
    System.exit(-1);
}

// Create a processor for this capturedevice & exit if we
// cannot create it

try {
    processor = Manager.createProcessor(di.getLocator());
} catch (IOException e) {
    System.exit(-1);
} catch (NoProcessorException e) {
    System.exit(-1);
}
```

Example 10-5: Transmitting RTP data with `RTPSocket` (2 of 3)

```
        // configure the processor
        processor.configure();

        // block until it has been configured

        processor.setContentDescriptor(
            new ContentDescriptor( ContentDescriptor.RAW));

        TrackControl track[] = processor.getTrackControls();
        boolean encodingOk = false;

        // Go through the tracks and try to program one of them to
        // output gsm data.
        for (int i = 0; i < track.length; i++) {
            if (!encodingOk && track[i] instanceof FormatControl) {

                if (((FormatControl)track[i]).
                    setFormat( new AudioFormat(AudioFormat.GSM_RTP,
                                               8000,
                                               8,
                                               1)) == null) {

                    track[i].setEnabled(false);
                }
                else {
                    encodingOk = true;
                }
            }
            else {
                // we could not set this track to gsm, so disable it
                track[i].setEnabled(false);
            }
        }

        // At this point, we have determined where we can send out
        // gsm data or not.
        // realize the processor
        if (encodingOk) {
            processor.realize();
            // block until realized.
            // get the output datasource of the processor and exit
            // if we fail
            DataSource ds = null;
            try {
                ds = processor.getDataOutput();
            } catch (NotRealizedError e) {
                System.exit(-1);
            }

            // hand this datasource to manager for creating an RTP
            // datasink
            // our RTP datasimnk will multicast the audio
```

Example 10-5: Transmitting RTP data with `RTPSocket` (3 of 3)

```
            try {
                MediaLocator m = new MediaLocator("rtpraw://");
                // here, manager will look for a datasink in
                // <protocol.prefix>.media.protocol.rtpraw.DataSink
                // the datasink will create an RTPSocket at
                // <protocol.prefix>.media.protocol.rtpraw.DataSource
                // and sink all RTP data to this socket.

                DataSink d = Manager.createDataSink(ds, m);

                d.open();
                d.start();
            } catch (Exception e) {
                System.exit(-1);
            }
        }
```

11

Importing and Exporting RTP Media Streams

Many applications need to be able to read and write RTP streams. For example, conferencing application might record a conference and broadcast it at a later time, or telephony applications might transmit stored audio streams for announcement messages or hold music.

You can save RTP streams received from the network to a file using an RTP file writer `DataSink`. Similarly, you can read saved files and either present them locally or transmit them across the network.

Reading RTP Media Streams from a File

To read data from a file and present or transmit it, you can use a `MediaLocator` that identifies the file to construct a `DataSource`, or use the `MediaLocator` to directly construct your `Processor`. The file types that can be used for RTP transmissions depend on what codec plug-ins you have available to transcode and packetize the data into an RTP-specific format.

Example 11-1: Reading RTP streams from a file (1 of 3)

```
// Create a Processor for the selected file. Exit if the
// Processor cannot be created.
try {
    String url= "file:/home/foo/foo.au";

    processor
      = Manager.createProcessor( new MediaLocator(url));
} catch (IOException e) {
    System.exit(-1);
```

Example 11-1: Reading RTP streams from a file (2 of 3)

```
        } catch (NoProcessorException e) {
            System.exit(-1);
        }

        // configure the processor
        processor.configure();

        // Block until the Processor has been configured

        TrackControl track[] = processor.getTrackControls();

        boolean encodingOk = false;

        // Go through the tracks and try to program one of them to
        // output ulaw data.
        for (int i = 0; i < track.length; i++) {
            if (!encodingOk && track[i] instanceof FormatControl) {

                if (((FormatControl)track[i]).
                    setFormat( new AudioFormat(AudioFormat.ULAW_RTP,
                                               8000,
                                               8,
                                               1)) == null) {

                    track[i].setEnabled(false);
                }
                else {
                    encodingOk = true;
                }
            }
            else {
                // we could not set this track to ulaw, so disable it
                track[i].setEnabled(false);
            }
        }

        // At this point, we have determined where we can send out
        // ulaw data or not.
        // realize the processor

        if (encodingOk) {
            processor.realize();

            // block until realized.
            // get the output datasource of the processor and exit
            // if we fail
            DataSource ds = null;

            try {
                ds = processor.getDataOutput();
            } catch (NotRealizedError e) {
                System.exit(-1);
            }
```

Example 11-1: Reading RTP streams from a file (3 of 3)

```
            // hand this datasource to manager for creating an RTP
            // datasink.
            // our RTP datasink will multicast the audio

            try {
                String url= "rtp://224.144.251.104:49150/audio/1";

                MediaLocator m = new MediaLocator(url);

                DataSink d = Manager.createDataSink(ds, m);

                d.open();
                d.start();
            } catch (Exception e) {
                System.exit(-1);
            }
        }
```

Exporting RTP Media Streams

RTP streams received from the network can be stored as well as presented. To write the data to a file, you retrieve the `DataSource` from the `ReceiveStream` and use it to create a file writing `DataSink` through the `Manager`.

If you want to transcode the data before storing it, you can use the `DataSource` retrieved from the `ReceiveStream` to construct a `Processor`. You then:

1. Set the track formats to perform the desired encoding.
2. Get the output `DataSource` from the `Processor`.
3. Construct an RTP file writer with the `DataSource`.

In the following example, whenever a new stream is created in the session:

1. The stream is retrieved from `NewReceiveStreamEvent`.
2. The `DataSource` is acquired from the `ReceiveStream`.
3. The `DataSource` is passed to the `Manager.createDataSink` method along with a `MediaLocator` that identifies the file where we want to store the data.

This example handles a single track. To write a file that contains both audio and video tracks, you need to retrieve the audio and video streams from the separate session managers and create a merging `DataSource` that carries both of the streams. Then you hand the merged `DataSource` to `Manager.createDataSink`.

Example 11-2: Writing an RTP stream to a file

```
public void update(ReceiveStreamEvent event) {
    // find the source session manager for this event
    SessionManager source = (SessionManager)event.getSource();

    // create a filewriter datasink if a new ReceiveStream
    // is detected
    if (event instanceof NewReceiveStreamEvent) {
        String cname = "Java Media Player";
        ReceiveStream stream = null;

        try {
            // get the ReceiveStream
            stream =((NewReceiveStreamEvent)event)
                    .getReceiveStream();

            Participant part = stream.getParticipant();

            // get the ReceiveStream datasource
            DataSource dsource = stream.getDataSource();

            // hand this datasource over to a file datasink
            MediaLocator f = new MediaLocator("file://foo.au");

            Manager.createDataSink(dsource, f);
        } catch (Exception e) {
            System.err.println("newReceiveStreamEvent exception "
                               + e.getMessage());
            return;
        }
    }
}
```

12

Creating Custom Packetizers and Depacketizers

Note: *The RTP 1.0 API supported custom packetizers and depacketizers through RTP-specific APIs. These APIs have been replaced by the generic JMF plug-in API and any custom packetizers or depacketizers created for RTP 1.0 will need to be ported to the new architecture.*

RTP *packetizers* are responsible for taking entire video frames or multiple audio samples and distributing them into packets of a particular size that can be streamed over the underlying network. Video frames are divided into smaller chunks, while audio samples are typically grouped together. RTP *depacketizers* reverse the process and reconstruct complete video frames or extract individual audio samples from a stream of RTP packets. The RTP session manager itself does not perform any packetization or depacketization. These operations are performed by the `Processor` using specialized `Codec` plug-ins.

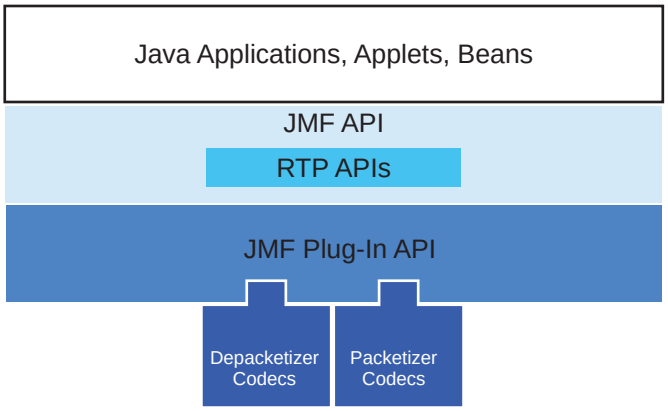


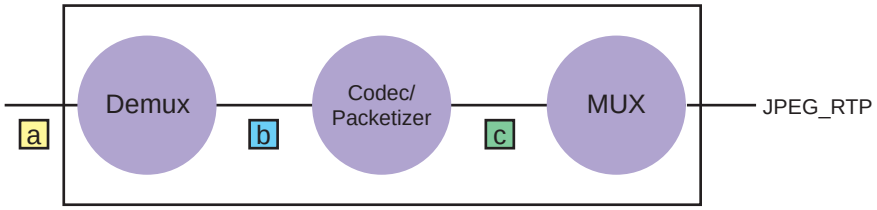
Figure 12-1: JMF RTP architecture.

To determine what RTP packetizer and depacketizer plug-ins are available, you can query the `PlugInManager` by calling `getPlugInList(CODEC)`. The input and output formats of a particular plug-in can be determined through the `getSupportedInputFormats` and `getSupportedOutputFormats` methods.

To receive or transmit any format not supported by one of the standard plug-ins, you need to implement a custom plug-in to perform the necessary conversions. The formats of the data streamed by the `DataSource` created by the session manager are well-defined to facilitate packetization and depacketization of the formatted data.

For a custom plug-in to work, there must be either a standard or custom plug-in available that can handle the output format it generates. In some cases, if the necessary encoder or decoder is available, you might only need to write a packetizer or depacketizer. In other cases, you might need to provide both the encoder/decoder and packetizer/depacketizer.

Custom packetizers and depacketizers can be combined with custom encoders and decoders, or you can implement independent packetizer and depacketizer plug-ins. For example, a depacketizer-only plug-in might advertise DVI_RTP as its input format and DVI as its output format.



JPEG RTP Transmission

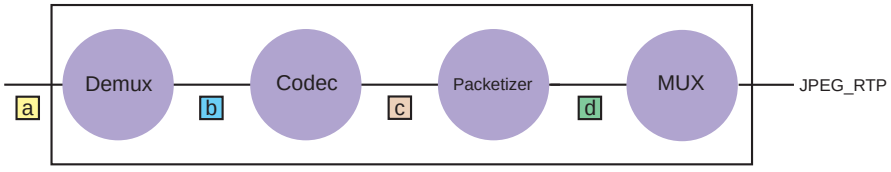
a Raw RGB Video

b Raw RGB Video output from Demultiplexer and input to Codec/Packetizer
(Demultiplexer doesn't change format)

c Packetized JPEG_RTP encoded data output from Codec/Packetizer
and input to Multiplexer
Buffer Format = JPEG_RTP
Buffer Header = RTPHeader (javax.media.rtp.RTPHeader)
Buffer Data = JPEG Payload header + JPEG Payload

Figure 12-2: Data flow with a custom depacketizer plug-in.

A combined depacketizer-decoder plug-in that decompressed DVI to linear audio would advertise DVI_RTP as its input format and AUDIO_LINEAR as its output format.



DVI RTP Player

a Raw RGB Video

b Raw RGB Video output from Demultiplexer and input to Encoder
(Demultiplexer doesn't change format)

c JPEG encoded data output from Encoder and input to Packetizer

d Packetized JPEG_RTP encoded data output from Packetizer and input to Multiplexer
Buffer Format = JPEG_RTP
Buffer Header = RTPHeader (javax.media.rtp.RTPHeader)
Buffer Data = JPEG Payload header + JPEG Payload

Figure 12-3: Data flow with combined depacketizer/decoder plug-in.

RTP Data Handling[1]

Data is transferred between the session manager and a `Player` or `Processor` using the `Buffer` object. Therefore, all `DataSources` created by the `Processor` with an RTP-specific format are buffer `DataSources`. Similarly, all `DataSources` created by the session manager and handed over to the `Manager` for `Player` creation are buffer `DataSources`.

All RTP-specific data uses an RTP-specific format encoding as defined in the `AudioFormat` and `VideoFormat` classes. For example, gsm RTP encapsulated packets have the encoding set to `AudioFormat.GSM_RTP`, while jpeg-encoded video formats have the encoding set to `VideoFormat.JPEG_RTP`.

`AudioFormat` defines four standard RTP-specific encoding strings:

```
public static final String ULAW_RTP = "JAUDIO_G711_ULAW/rtp";
public static final String DVI_RTP = "dvi/rtp";
public static final String G723_RTP = "g723/rtp";
public static final String GSM_RTP = "gsm/rtp";
```

`VideoFormat` defines three standard RTP-specific encoding strings:

```
public static final String JPEG_RTP = "jpeg/rtp";
public static final String H261_RTP = "h261/rtp";
public static final String H263_RTP = "h263/rtp";
```

`Buffers` that have an RTP-specific encoding might have a non-null header defined in `javax.media.rtp.RTPHeader`. Payload-specific headers are not part of the `RTPHeader`. Instead, payload headers are part of the data object in the `Buffers` transferred between the `Player` or `Processor` and the session manager. The packet's actual RTP header is also included as part of the `Buffer` object's data. The `Buffer` object's *offset* points to the end of this header.

For packets received from the network by the `SessionManager`, all available fields from the RTP Header (as defined in RFC 1890) are translated to appropriate fields in the `Buffer` object: timestamp and sequence number. The marker bit from the RTP header is sent over as flags on the `Buffer` object, which you can access by calling the `Buffer getFlags` method. The flag used to indicate the marker bit is `Buffer.FLAG_RTP_MARKER`. If there is

1. See the IETF RTP payload specifications for more information about how particular payloads are to be carried in RTP.

an extension header, it is sent over in the header of be `Buffer`, which is a `RTPHeader` object. The format of the `Buffer` is set to `AudioFormat.GSM_RTP.`

All source streams streamed out on RTP `DataSources` have their content descriptor set to an empty content descriptor of `""` and their format set to the appropriate RTP-specific format and encoding. To be able to intercept or depacketize this data, plug-in codecs must advertise this format as one of their input formats.

For packets being sent over the network, the `Processor's` format must be set to one of the RTP-specific formats (encodings). The plug-in codec must advertise this format as one of its supported output formats. All `Buffer` objects passed to the `SessionManager` through the `DataSource` sent to `createSendStream` must have an RTP-specific format. The header of the `Buffer` is as described in `javax.media.rtp.RTPHeader`.

Dynamic RTP Payloads

The `SessionManager` has a provision for entering information on dynamic RTP payloads. For more information about how dynamic payloads are used in RTP, refer to IETF RFC 1890, the RTP Audio-Video profile[2] that accompanies the RTP specification.

The dynamic RTP-payload information typically contains a mapping from a predetermined RTP payload ID to a specific encoding. In the JMF RTP API, this information is passed via the `Format` object. To enable playback or transmission of dynamic RTP payloads, you must associate a specific `Format` with an RTP payload number. This information can be sent to the session manager in two ways:

- Through the `RTPControl addFormat` method—every RTP `DataSource` exports an `RTPControl` that can be retrieved through the `DataSource getControl` method. A handle for the `DataSource` is typically obtained by calling the `Processor getDataOutput` method or the `Manager createDataSource(MediaLocator)` method. The `RTPControl's addFormat` method can be used to enter the encoding Information. See `javax.media.rtp.RTPControl` for more information.
- Through the `SessionManager addFormat` method—if you use the JMF RTP API but do not use the `Manager` to create players or send streams,

2. This document is being revised in preparation for advancement from Proposed Standard to Draft standard. At the time of publication, the most recent draft was http://www.ietf.org/internet-drafts/draft-ietf-avt-profile-new-06.txt.

the dynamic payload information can be entered using the `addFormat` method of the `SessionManager` interface. For playback, this must be done prior to calling `startSession` since the session manager must be configured with dynamic payload information before data arrives. For transmission, this must be done prior to calling `createSendStream` since the session manager must be configured with dynamic payload information before attempting to send data out.

Registering Custom Packetizers and Depacketizers

Whenever custom packetizers or depacketizers are used, a new payload number must be associated with the RTP format in the session manager's registry. For RTP transmission, you need to call `addFormat` on the `SessionManager` to register new formats. For RTP reception, you can either:

- Call `addFormat` on the `RTPControl` associated with the `DataSource`.
- Call `addFormat` on the `SessionManager`.

A

JMF Applet

This Java Applet demonstrates proper error checking in a Java Media program. Like `PlayerApplet`, it creates a simple media player with a media event listener.

When this applet is started, it immediately begins to play the media clip. When the end of media is reached, the clip replays from the beginning.

Example A-1: `TypicalPlayerApplet` with error handling. (1 of 5)

```
import java.applet.Applet;
import java.awt.*;
import java.lang.String;
import java.net.URL;
import java.net.MalformedURLException;
import java.io.IOException;
import javax.media.*;

/**
 * This is a Java Applet that demonstrates how to create a simple
 * media player with a media event listener.  It will play the
 * media clip right away and continuously loop.
 *
 * <!-- Sample HTML
 * <applet code=TypicalPlayerApplet width=320 height=300>
 * <param name=file value="Astrnmy.avi">
 * </applet>
 * -->
 */

public class TypicalPlayerApplet extends Applet implements
ControllerListener
{
   // media player
   Player player = null;
```

Example A-1: `TypicalPlayerApplet` with error handling. (2 of 5)

```
// component in which video is playing
Component visualComponent  = null;
// controls gain, position, start, stop
Component controlComponent = null;
// displays progress during download
Component progressBar      = null;

/**
 * Read the applet file parameter and create the media
 * player.
 */

public void init()
{
   setLayout(new BorderLayout());
   // input file name from html param
   String mediaFile = null;
   // URL for our media file
   URL url = null;
   // URL for doc containing applet
   URL codeBase = getDocumentBase();

   // Get the media filename info.
   // The applet tag should contain the path to the
   // source media file, relative to the html page.

   if ((mediaFile = getParameter("FILE")) == null)
      Fatal("Invalid media file parameter");
   try
   {
      // Create an url from the file name and the url to the
      // document containing this applet.

      if ((url = new URL(codeBase, mediaFile)) == null)
         Fatal("Can't build URL for " + mediaFile);

      // Create an instance of a player for this media
      if ((player = Manager.createPlayer(url)) == null)
         Fatal("Could not create player for "+url);

      // Add ourselves as a listener for player's events
         player.addControllerListener(this);
   }
   catch (MalformedURLException u)
   {
      Fatal("Invalid media file URL!");
   }
```

Example A-1: `TypicalPlayerApplet` with error handling. (3 of 5)

```
        catch(IOException i)
        {
           Fatal("IO exception creating player for "+url);
        }

        // This applet assumes that its start() calls
        // player.start().This causes the player to become
        // Realized. Once Realized, the Applet will get
        //  the visual and control panel components and add
        // them to the Applet.  These components are not added
        // during init() because they are long operations that
        // would make us appear unresposive to the user.
     }

     /**
      * Start media file playback.  This function is called the
      * first time that the Applet runs and every
      * time the user re-enters the page.
      */

     public void start()
     {
        // Call start() to prefetch and start the player.

        if (player != null) player.start();
     }

     /**
      * Stop media file playback and release resources before
      * leaving the page.
      */

     public void stop()
     {
        if (player != null)
        {
           player.stop();
           player.deallocate();
        }
     }

     /**
      * This controllerUpdate function must be defined in order
      * to implement a ControllerListener interface.  This
      * function will be called whenever there is a media event.
      */

     public synchronized void controllerUpdate(ControllerEvent event)
     {
        // If we're getting messages from a dead player,
```

Example A-1: `TypicalPlayerApplet` with error handling. (4 of 5)

```
        // just leave

        if (player == null) return;

        // When the player is Realized, get the visual
        // and control components and add them to the Applet

        if (event instanceof RealizeCompleteEvent)
        {
           if ((visualComponent = player.getVisualComponent()) != null)
              add("Center", visualComponent);
           if ((controlComponent = player.getControlPanelComponent()) != null)
              add("South",controlComponent);
              // force the applet to draw the components
              validate();
        }
        else if (event instanceof CachingControlEvent)
        {

           // Put a progress bar up when downloading starts,
           // take it down when downloading ends.

           CachingControlEvent  e = (CachingControlEvent) event;
           CachingControl      cc = e.getCachingControl();
           long cc_progress       = e.getContentProgress();
           long cc_length         = cc.getContentLength();

           // Add the bar if not already there ...

           if (progressBar == null)
              if ((progressBar = cc.getProgressBarComponent()) != null)
              {
                 add("North", progressBar);
                 validate();
              }

           // Remove bar when finished ownloading
           if (progressBar != null)
              if (cc_progress == cc_length)
              {
                 remove (progressBar);
                 progressBar = null;
                 validate();
              }
        }
        else if (event instanceof EndOfMediaEvent)
        {
```

Example A-1: `TypicalPlayerApplet` with error handling. (5 of 5)

```
            // We've reached the end of the media; rewind and
            // start over

            player.setMediaTime(new Time(0));
            player.start();
        }
        else if (event instanceof ControllerErrorEvent)
        {
            // Tell TypicalPlayerApplet.start() to call it a day

            player = null;
            Fatal (((ControllerErrorEvent)event).getMessage());
        }
    }

    void Fatal (String s)
    {
        // Applications will make various choices about what
        // to do here.  We print a message and then exit

        System.err.println("FATAL ERROR: " + s);
        throw new Error(s);  // Invoke the uncaught exception
                             // handler System.exit() is another
                             // choice
    }
}
```

B

StateHelper

`StateHelper` is a helper class that implements the `ControllerListener` interface and can be used to manage the state of a `Processor`. This helper class is used in examples 5-4, 5-5, and 5-6 in"Capturing Time-Based Media with JMF" on page 77.

Example B-1: `StateHelper` (1 of 3)

```
import javax.media.*;


public class StateHelper implements javax.media.ControllerListener {

    Player player = null;
    boolean configured = false;
    boolean realized = false;
    boolean prefetched = false;
    boolean eom = false;
    boolean failed = false;
    boolean closed = false;

    public StateHelper(Player p) {
     player = p;
     p.addControllerListener(this);
    }

    public boolean configure(int timeOutMillis) {
     long startTime = System.currentTimeMillis();
     synchronized (this) {
         if (player instanceof Processor)
        ((Processor)player).configure();
         else
        return false;
```

Example B-1: `StateHelper` (2 of 3)

```
            while (!configured && !failed) {
            try {
                wait(timeOutMillis);
            } catch (InterruptedException ie) {
            }
            if (System.currentTimeMillis() - startTime > timeOutMillis)
                break;
            }
        }
        return configured;
    }

    public boolean realize(int timeOutMillis) {
        long startTime = System.currentTimeMillis();
        synchronized (this) {
            player.realize();
            while (!realized && !failed) {
            try {
                wait(timeOutMillis);
            } catch (InterruptedException ie) {
            }
            if (System.currentTimeMillis() - startTime > timeOutMillis)
                break;
            }
        }
        return realized;
    }

    public boolean prefetch(int timeOutMillis) {
        long startTime = System.currentTimeMillis();
        synchronized (this) {
            player.prefetch();
            while (!prefetched && !failed) {
            try {
                wait(timeOutMillis);
            } catch (InterruptedException ie) {
            }
            if (System.currentTimeMillis() - startTime > timeOutMillis)
                break;
            }
        }
        return prefetched && !failed;
    }
    public boolean playToEndOfMedia(int timeOutMillis) {
        long startTime = System.currentTimeMillis();
        eom = false;
        synchronized (this) {
            player.start();
```

Example B-1: `StateHelper` (3 of 3)

```
            while (!eom && !failed) {
            try {
                wait(timeOutMillis);
            } catch (InterruptedException ie) {
            }
            if (System.currentTimeMillis() - startTime > timeOutMillis)
                break;
            }
        }
        return eom && !failed;
    }

    public void close() {
        synchronized (this) {
            player.close();
            while (!closed) {
            try {
                wait(100);
            } catch (InterruptedException ie) {
            }
            }
        }
        player.removeControllerListener(this);
    }

    public synchronized void controllerUpdate(ControllerEvent ce) {
        if (ce instanceof RealizeCompleteEvent) {
            realized = true;
        } else if (ce instanceof ConfigureCompleteEvent) {
            configured = true;
        } else if (ce instanceof PrefetchCompleteEvent) {
            prefetched = true;
        } else if (ce instanceof EndOfMediaEvent) {
            eom = true;
        } else if (ce instanceof ControllerErrorEvent) {
            failed = true;
        } else if (ce instanceof ControllerClosedEvent) {
            closed = true;
        } else {
            return;
        }
        notifyAll();
    }
}
```

C

Demultiplexer Plug-In

This sample demonstrates how to implement a `Demultiplexer` plug-in to extract individual tracks from a media file. This example processes GSM files.

Example C-1: GSM `Demultiplexer` plug-in. (1 of 13)

```
import java.io.IOException;
import javax.media.*;
import javax.media.protocol.*;
import javax.media.format.Format;
import javax.media.format.audio.AudioFormat;

/**
 * Demultiplexer for GSM file format
 */

/**
 * GSM
 * 8000 samples per sec.
 * 160 samples represent 20 milliseconds and GSM represents them
 * in 33 bytes. So frameSize is 33 bytes and there are 50 frames
 * in one second. One second is 1650 bytes.
 */

public class SampleDeMux implements Demultiplexer {
    private Time duration = Duration.DURATION_UNKNOWN;
    private Format format = null;
   private Track[] tracks = new Track[1]; // Only 1 track is there for Gsm
    private int numBuffers = 4;
    private int bufferSize;
    private int dataSize;
    private int encoding;
    private String encodingString;
    private int sampleRate;
    private int samplesPerBlock;
    private int bytesPerSecond = 1650; // 33 * 50
    private int blockSize = 33;
```

Example C-1: GSM `Demultiplexer` plug-in. (2 of 13)

```
    private int maxFrame = Integer.MAX_VALUE;
    private long minLocation;
    private long maxLocation;
    private PullSourceStream stream = null;
    private long currentLocation = 0;

    protected DataSource source;
    protected SourceStream[] streams;
    protected boolean seekable = false;
    protected boolean positionable = false;
    private Object sync = new Object(); // synchronizing variable

    private static ContentDescriptor[] supportedFormat =
        new ContentDescriptor[] {new ContentDescriptor("audio.x_gsm")};

    public ContentDescriptor [] getSupportedInputContentDescriptors() {
        return supportedFormat;
    }

    public void setSource(DataSource source)
        throws IOException, IncompatibleSourceException {

        if (!(source instanceof PullDataSource)) {
            throw new IncompatibleSourceException("DataSource
                not supported: " + source);
        } else {
            streams = ((PullDataSource) source).getStreams();
        }

        if ( streams == null) {
          throw new IOException("Got a null stream from the DataSource");
        }

        if (streams.length == 0) {
            throw new IOException("Got a empty stream array
                from the DataSource");
        }
        this.source = source;
        this.streams = streams;

        positionable =  (streams[0] instanceof Seekable);
        seekable =  positionable && ((Seekable)
                    streams[0]).isRandomAccess();

        if (!supports(streams))
            throw new IncompatibleSourceException("DataSource not
                    supported: " + source);
    }

    /**
     * A Demultiplexer may support pull only or push only or both
     * pull and push streams.
```

Example C-1: GSM `Demultiplexer` plug-in. (3 of 13)

```
        * Some Demultiplexer may have other requirements.
        * For e.g a quicktime Demultiplexer imposes an additional
        * requirement that
        * isSeekable() and isRandomAccess() be true
        */
       protected boolean supports(SourceStream[] streams) {
           return ( (streams[0] != null) &&
                    (streams[0] instanceof PullSourceStream) );
       }

       public boolean isPositionable() {
           return positionable;
       }

       public boolean isRandomAccess() {
           return seekable;
       }

       /**
        * Opens the plug-in software or hardware component and acquires
        * necessary resources. If all the needed resources could not be
       * acquired, it throws a ResourceUnavailableException. Data should not
        * be passed into the plug-in without first calling this method.
        */
       public void open() {
           // throws ResourceUnavailableException;
       }

       /**
        * Closes the plug-in component and releases resources. No more data
        * will be accepted by the plug-in after a call to this method. The
        * plug-in can be reinstated after being closed by calling
        * <code>open</code>.
        */
       public void close() {
           if (source != null) {
               try {
                   source.stop();
                   source.disconnect();
               } catch (IOException e) {
                   // Internal error?
               }
               source = null;
           }
       }

       /**
        * This get called when the player/processor is started.
        */
       public void start() throws IOException {
            if (source != null)
                source.start();
       }
```

Example C-1: GSM `Demultiplexer` plug-in. (4 of 13)

```
    /**
     * This get called when the player/processor is stopped.
     */
    public void stop() {
        if (source != null) {
            try {
                source.stop();
            } catch (IOException e) {
                // Internal errors?
            }
        }
    }

    /**
     * Resets the state of the plug-in. Typically at end of media
     * or when media is repositioned.
     */
    public void reset() {
    }

    public Track[] getTracks() throws IOException, BadHeaderException {

        if (tracks[0] != null)
            return tracks;
        stream = (PullSourceStream) streams[0];
        readHeader();
        bufferSize = bytesPerSecond;
        tracks[0] = new GsmTrack((AudioFormat) format,
                                 /*enabled=*/ true,
                                  new Time(0),
                                  numBuffers,
                                  bufferSize,
                                  minLocation,
                                  maxLocation
                                  );
        return tracks;
    }

    public Object[] getControls() {
        return new Object[0];
    }

    public Object getControl(String controlType) {
        return null;
    }
    private void /* for now void */ readHeader()
        throws IOException, BadHeaderException {

        minLocation = getLocation(stream); // Should be zero

        long contentLength = stream.getContentLength();
        if ( contentLength != SourceStream.LENGTH_UNKNOWN ) {
            double durationSeconds = contentLength / bytesPerSecond;
```

Example C-1: GSM `Demultiplexer` plug-in. (5 of 13)

```
            duration = new Time(durationSeconds);
            maxLocation = contentLength;

        } else {
            maxLocation = Long.MAX_VALUE;
        }

        boolean signed = true;
        boolean bigEndian = false;
        format = new AudioFormat(AudioFormat.GSM,
                                 8000,  // sampleRate,
                                 16,    // sampleSizeInBits,
                                 1,     // channels,
                                 bigEndian ? AudioFormat.BIG_ENDIAN :
                                 AudioFormat.LITTLE_ENDIAN,
                                 signed ? AudioFormat.SIGNED :
                                 AudioFormat.UNSIGNED,
                                 (blockSize * 8), // frameSizeInBits
                                 Format.NOT_SPECIFIED,
                                 Format.byteArray);
    }

    // Contains 1 audio track
    public String getTrackLayout() {
        return "A";
    }

    public Time setPosition(Time where, int rounding) {
        if (! seekable ) {
            return getMediaTime();
        }

        long time = where.getNanoseconds();
        long newPos;

        if (time < 0)
            time = 0;

        double newPosd = time * bytesPerSecond / 1000000000.0;
        double remainder = (newPosd % blockSize);

        newPos = (long) (newPosd - remainder);

        if (remainder > 0) {
            switch (rounding) {
            case Positionable.RoundUp:
                newPos += blockSize;
                break;
            case Positionable.RoundNearest:
                if (remainder > (blockSize / 2.0))
                    newPos += blockSize;
```

Example C-1: GSM `Demultiplexer` plug-in. (6 of 13)

```
                break;
            }
        }

        if ( newPos > maxLocation )
            newPos = maxLocation;

        newPos += minLocation;
        ((BasicTrack) tracks[0]).setSeekLocation(newPos);
        return where;
    }

    public Time getMediaTime() {
        long location;
        long seekLocation = ((BasicTrack) tracks[0]).getSeekLocation();
        if (seekLocation != -1)
            location = seekLocation - minLocation;
        else
            location = getLocation(stream) - minLocation;

        return new Time( location / (double) bytesPerSecond );
    }

    public Time getDuration() {
        if ( duration.equals(Duration.DURATION_UNKNOWN) &&
             ( tracks[0] != null ) ) {
            long mediaSizeAtEOM = ((BasicTrack)
                                   tracks[0]).getMediaSizeAtEOM();
            if (mediaSizeAtEOM > 0) {
               double durationSeconds = mediaSizeAtEOM / bytesPerSecond;
               duration = new Time(durationSeconds);
            }
        }
        return duration;
    }

    /**
     * Returns a descriptive name for the plug-in.
     * This is a user readable string.
     */
    public String getName() {
        return "Parser for raw GSM";
    }

    /**
     * Read numBytes from offset 0
     */
    public int readBytes(PullSourceStream pss, byte[] array,
                         int numBytes) throws IOException {

        return readBytes(pss, array, 0, numBytes);
    }
```

Example C-1: GSM `Demultiplexer` plug-in. (7 of 13)

```
    public int readBytes(PullSourceStream pss, byte[] array,
                         int offset,
                         int numBytes) throws IOException {
        if (array == null) {
            throw new NullPointerException();
        } else if ((offset < 0) || (offset > array.length) ||
                   (numBytes < 0) ||
                   ((offset + numBytes) > array.length) ||
                   ((offset + numBytes) < 0)) {
            throw new IndexOutOfBoundsException();
        } else if (numBytes == 0) {
            return 0;
        }

        int remainingLength = numBytes;
        int actualRead = 0;

        remainingLength = numBytes;
        while (remainingLength > 0) {

            actualRead = pss.read(array, offset, remainingLength);
            if (actualRead == -1) {// End of stream
                if (offset == 0) {
                    throw new IOException("SampleDeMux: readBytes():
                          Reached end of stream while trying to read " +
                              numBytes + " bytes");
                } else {
                    return offset;
                }
            } else if (actualRead ==
             com.sun.media.protocol.BasicSourceStream.LENGTH_DISCARD) {
                return
               com.sun.media.protocol.BasicSourceStream.LENGTH_DISCARD;
            } else if (actualRead < 0) {
                throw new IOException("SampleDeMux: readBytes()
                          read returned " + actualRead);
            }
            remainingLength -= actualRead;
            offset += actualRead;
            synchronized(sync) {
                currentLocation += actualRead;
            }
        }
        return numBytes;
    }

    protected final long getLocation(PullSourceStream pss) {
        synchronized(sync) {
            if ( (pss instanceof Seekable) )
                return ((Seekable)pss).tell();
```

Example C-1: GSM `Demultiplexer` plug-in. (8 of 13)

```
            else
                return currentLocation;
        }
    }

    ////////////////////////////////////////////////////////
    // Inner classes begin
    abstract private class BasicTrack implements Track {

        private Format format;
        private boolean enabled = true;
        protected Time duration;
        private Time startTime;
        private int numBuffers;
        private int dataSize;
        private PullSourceStream stream;
        private long minLocation;
        private long maxLocation;
        private long maxStartLocation;
        private SampleDeMux parser;
        private long sequenceNumber = 0;
        private TrackListener listener;
        private long seekLocation = -1L;
        private long mediaSizeAtEOM = -1L; // update when EOM
                                           // implied by IOException occurs

        BasicTrack(SampleDeMux parser,
                   Format format, boolean enabled,
                   Time duration, Time startTime,
                   int numBuffers, int dataSize,
                   PullSourceStream stream) {
            this(parser, format,  enabled,  duration,  startTime,
                 numBuffers, dataSize, stream,
                 0L, Long.MAX_VALUE);
        }


        /**
         * Note to implementors who want to use this class.
         * If the maxLocation is not known, then
         * specify Long.MAX_VALUE for this parameter
         */
        public BasicTrack(SampleDeMux parser,
                          Format format, boolean enabled,
                          Time duration, Time startTime,
                          int numBuffers, int dataSize,
                          PullSourceStream stream,
                          long minLocation, long maxLocation) {

            this.parser = parser;
```

Example C-1: GSM `Demultiplexer` plug-in. (9 of 13)

```
            this.format = format;
            this.enabled = enabled;
            this.duration = duration;
            this.startTime = startTime;
            this.numBuffers = numBuffers;
            this.dataSize = dataSize;
            this.stream = stream;
            this.minLocation = minLocation;
            this.maxLocation = maxLocation;
            maxStartLocation = maxLocation - dataSize;
        }

        public Format getFormat() {
            return format;
        }

        public void setEnabled(boolean t) {
            enabled = t;
        }

        public boolean isEnabled() {
            return enabled;
        }

        public Time getDuration() {
            return duration;
        }


        public Time getStartTime() {
            return startTime;
        }


        public int getNumberOfBuffers() {
            return numBuffers;
        }


        public void setTrackListener(TrackListener l) {
            listener = l;
        }

        public synchronized void setSeekLocation(long location) {
            seekLocation = location;
        }
        public synchronized long getSeekLocation() {
            return seekLocation;
        }

        public void readFrame(Buffer buffer) {
            if (buffer == null)
                return;
```

Example C-1: GSM `Demultiplexer` plug-in. (10 of 13)

```
            if (!enabled) {
                buffer.setDiscard(true);
                return;
            }

            buffer.setFormat(format);
            Object obj = buffer.getData();
            byte[] data;
            long location;
            boolean needToSeek;

            synchronized(this) {
                if (seekLocation != -1) {
                    location = seekLocation;
                    seekLocation = -1;
                    needToSeek = true;
                } else {
                    location = parser.getLocation(stream);
                    needToSeek = false;
                }
            }

            int needDataSize;

            if (location < minLocation) {
                buffer.setDiscard(true);
                return;
            } else if (location >= maxLocation) {
                buffer.setLength(0);
                buffer.setEOM(true);
                return;
            } else if (location > maxStartLocation) {
                needDataSize = dataSize - (int) (location -
                                maxStartLocation);
            } else {
                needDataSize = dataSize;
            }

            if  ( (obj == null) ||
                  (! (obj instanceof byte[]) ) ||
                  ( ((byte[])obj).length < needDataSize) ) {
                data = new byte[needDataSize];
                buffer.setData(data);
            } else {
                data = (byte[]) obj;
            }
            try {
                if (needToSeek) {
                    long pos =
                ((javax.media.protocol.Seekable)stream).seek(location);
```

Example C-1: GSM `Demultiplexer` plug-in. (11 of 13)

```
                    if ( pos ==
              com.sun.media.protocol.BasicSourceStream.LENGTH_DISCARD) {
                        buffer.setDiscard(true);
                        return;
                    }
                }
                int actualBytesRead = parser.readBytes(stream,
                                      data, needDataSize);
                buffer.setOffset(0);
                buffer.setLength(actualBytesRead);
                buffer.setSequenceNumber(++sequenceNumber);
            buffer.setTimeStamp(parser.getMediaTime().getNanoseconds());
            } catch (IOException e) {
                if (maxLocation != Long.MAX_VALUE) {
                    // Known maxLocation. So, this is a case of
                    // deliberately reading past EOM
                    System.err.println("readFrame: EOM " + e);
                    buffer.setLength(0); // Need this??
                    buffer.setEOM(true);
                } else {
                   // Unknown maxLocation, due to unknown content length
                  // EOM reached before the required bytes could be read.
                    long length = parser.streams[0].getContentLength();
                    if ( length != SourceStream.LENGTH_UNKNOWN ) {
                     // If content-length is known, discard this buffer,
                    // updatemaxLocation, maxStartLocation and
                    // mediaSizeAtEOM.  The next readFrame will read
                    // the remaining data till EOM.
                        maxLocation = length;
                        maxStartLocation = maxLocation - dataSize;
                        mediaSizeAtEOM = maxLocation - minLocation;
                        buffer.setLength(0); // Need this??
                        buffer.setDiscard(true);
                    } else {
                        // Content Length is still unknown after an
                        // IOException.
                   // We can still discard this buffer and keep discarding
                    // until content length is known. But this may go into
                     // into an infinite loop, if there are real IO errors
                        // So, return EOM
                        maxLocation = parser.getLocation(stream);
                        maxStartLocation = maxLocation - dataSize;
                        mediaSizeAtEOM = maxLocation - minLocation;
                        buffer.setLength(0); // Need this??
                        buffer.setEOM(true);
                    }
                }
            }
        }

        public void readKeyFrame(Buffer buffer) {
            readFrame(buffer);
        }
```

Example C-1: GSM `Demultiplexer` plug-in. (12 of 13)

```
        public boolean willReadFrameBlock() {
            return false;
        }


        public long getMediaSizeAtEOM() {
        return mediaSizeAtEOM; // updated when EOM implied by
                               // IOException occurs
        }
    }

    private class GsmTrack extends BasicTrack {
        private double sampleRate;
        private float timePerFrame = 0.020F; // 20 milliseconds

        GsmTrack(AudioFormat format, boolean enabled, Time startTime,
                 int numBuffers, int bufferSize,
                 long minLocation, long maxLocation) {
            super(SampleDeMux.this,
                  format, enabled, SampleDeMux.this.duration,
                  startTime, numBuffers, bufferSize,
                  SampleDeMux.this.stream, minLocation, maxLocation);

            double sampleRate = format.getSampleRate();
            int channels = format.getChannels();
            int sampleSizeInBits = format.getSampleSizeInBits();

            float bytesPerSecond;
            float bytesPerFrame;
            float samplesPerFrame;

            long durationNano = this.duration.getNanoseconds();
            if (!( (durationNano ==
                    Duration.DURATION_UNKNOWN.getNanoseconds()) ||
                    (durationNano ==
                     Duration.DURATION_UNBOUNDED.getNanoseconds()) )) {
                maxFrame = mapTimeToFrame(this.duration.getSeconds());
            }
        }

        GsmTrack(AudioFormat format, boolean enabled, Time startTime,
                 int numBuffers, int bufferSize) {
            this(format, enabled,
                 startTime, numBuffers, bufferSize,
                 0L, Long.MAX_VALUE);

        }

        // Frame numbers start from 0
        private int mapTimeToFrame(double time) {
            double frameNumber = time / timePerFrame;
            return (int) frameNumber;
        }
```

Example C-1: GSM `Demultiplexer` plug-in. (13 of 13)

```
        // Frame numbers start from 0
        // 0-1 ==> 0, 1-2 ==> 1
        public int mapTimeToFrame(Time t) {
            double time = t.getSeconds();
            int frameNumber = mapTimeToFrame(time);

            if ( frameNumber > maxFrame)
               frameNumber = maxFrame; // Do we clamp it or return error
           System.out.println("mapTimeToFrame: " + (int) time + " ==> " +
                              frameNumber + " ( " + frameNumber + " )");
            return frameNumber;
        }
        public Time mapFrameToTime(int frameNumber) {
            if (frameNumber > maxFrame)
               frameNumber = maxFrame; // Do we clamp it or return error
            double time = timePerFrame * frameNumber;
          System.out.println("mapFrameToTime: " + frameNumber + " ==> " +
                                    time);
            return new Time(time);
        }
    }
}
```

D

Sample Data Source Implementation

This sample demonstrates how to implement a new `DataSource` to support an additional protocol, the FTP protocol. There are two classes:

- `DataSource` extends `PullDataSource` and implements `intel.media.protocol.PullProtocolHandler.`
- `FTPSourceStream` implements `PullSourceStream`.

Example D-1: FTP Data Source. (1 of 8)

```
package COM.intel.media.protocol.ftp;

import javax.media.protocol.PullDataSource;
import javax.media.protocol.SourceStream;
import javax.media.protocol.PullSourceStream;
import javax.media.Time;
import javax.media.Duration;
import java.io.*;
import java.net.*;
import java.util.Vector;

public class DataSource extends PullDataSource
{
   public static final int FTP_PORT = 21;
   public static final int FTP_SUCCESS = 1;
   public static final int FTP_TRY_AGAIN = 2;
   public static final int FTP_ERROR = 3;

   // used to send commands to server
   protected Socket controlSocket;
```

Example D-1: FTP Data Source. (2 of 8)

```
// used to receive file
protected Socket dataSocket;
// wraps controlSocket's output stream
protected PrintStream controlOut;

// wraps controlSocket's input stream
protected InputStream controlIn;

// hold (possibly multi-line) server response
protected Vector response = new Vector(1);

// reply code from previous command
protected int previousReplyCode;

// are we waiting for command reply?
protected boolean replyPending;

// user login name
 protected String user = "anonymous";

// user login password
protected String password = "anonymous";

// FTP server name
protected String hostString;

// file to retrieve
protected String fileString;

public void connect() throws IOException
{
   initCheck();  // make sure the locator is set
   if (controlSocket != null)
   {
      disconnect();
   }
   // extract FTP server name and target filename from locator
   parseLocator();
   controlSocket = new Socket(hostString, FTP_PORT);
   controlOut = new PrintStream(new BufferedOutputStream(
      controlSocket.getOutputStream()), true);
   controlIn = new
      BufferedInputStream(controlSocket.getInputStream());

  if (readReply() == FTP_ERROR)
   {
       throw new IOException("connection failed");
   }

   if (issueCommand("USER " + user) == FTP_ERROR)
   {
      controlSocket.close();
```

Example D-1: FTP Data Source. (3 of 8)

```
            throw new IOException("USER command failed");
        }

        if (issueCommand("PASS " + password) == FTP_ERROR)
        {
            controlSocket.close();
            throw new IOException("PASS command failed");
        }
    }

    public void disconnect()
    {
        if (controlSocket == null)
        {
            return;
        }

        try
        {
            issueCommand("QUIT");
            controlSocket.close();
        }

        catch (IOException e)
        {
            // do nothing, we just want to shutdown
        }

        controlSocket = null;
        controlIn = null;
        controlOut = null;
    }

     public void start() throws IOException
     {
          ServerSocket serverSocket;
          InetAddress myAddress = InetAddress.getLocalHost();
          byte[] address = myAddress.getAddress();

          String portCommand = "PORT ";
          serverSocket = new ServerSocket(0, 1);

          // append each byte of our address (comma-separated)

          for (int i = 0; i < address.length; i++)
          {
              portCommand = portCommand + (address[i] & 0xFF) + ",";
          }

        // append our server socket's port as two comma-separated
        // hex bytes
```

Example D-1: FTP Data Source. (4 of 8)

```
            portCommand = portCommand +
               ((serverSocket.getLocalPort() >>> 8)
               & 0xFF) + "," + (serverSocket.getLocalPort() & 0xFF);

            // issue PORT command
            if (issueCommand(portCommand) == FTP_ERROR)
            {
               serverSocket.close();
               throw new IOException("PORT");
            }

            // issue RETRieve command
            if (issueCommand("RETR " + fileString) == FTP_ERROR)
            {
               serverSocket.close();
               throw new IOException("RETR");
            }

            dataSocket = serverSocket.accept();
            serverSocket.close();
      }
      public void stop()
      {
         try
         {
         // issue ABORt command
         issueCommand("ABOR");
         dataSocket.close();
         }
         catch(IOException e) {}
      }

      public String getContentType()
      {
       // We don't get MIME info from FTP server.  This
       // implementation makes an attempt guess the type using
       // the File name and returns "unknown" in the default case.
       // A more robust mechanisms should
       // be supported for real-world applications.

         String locatorString = getLocator().toExternalForm();
         int dotPos = locatorString.lastIndexOf(".");
         String extension = locatorString.substring(dotPos + 1);
         String typeString = "unknown";

         if (extension.equals("avi"))
            typeString = "video.x-msvideo";
         else if (extension.equals("mpg") ||
            extension.equals("mpeg"))
            typeString = "video.mpeg";
         else if (extension.equals("mov"))
            typeString = "video.quicktime";
```

Example D-1: FTP Data Source. (5 of 8)

```
        else if (extension.equals("wav"))
           typeString = "audio.x-wav";
        else if (extension.equals("au"))
           typeString = "audio.basic";
        return typeString;
     }

     public PullSourceStream[] getStreams()
     {
        PullSourceStream[] streams = new PullSourceStream[1];
        try
        {
           streams[0] = new FTPSourceStream(dataSocket.getInputStream());
        }

        catch(IOException e)
        {
           System.out.println("error getting streams");
        }
        return streams;
     }



     public Time getDuration()
     {
        return Duration.DURATION_UNKNOWN;
     }

     public void setUser(String user)

     {
        this.user = user;
     }

     public String getUser()
     {
        return user;
     }

     public void setPassword(String password)
     {
        this.password = password;
     }

     public String getPassword()
     {
        return password;
     }
```

Example D-1: FTP Data Source. (6 of 8)

```
    private int readReply() throws IOException
    {
       previousReplyCode = readResponse();
       System.out.println(previousReplyCode);
       switch (previousReplyCode / 100)
       {
          case 1:
             replyPending = true;
             // fall through
          case 2:
          case 3:
             return FTP_SUCCESS;
          case 5:
             if (previousReplyCode == 530)
             {
                if (user == null)
                {
                   throw new IOException("Not logged in");
                }
                return FTP_ERROR;
             }
             if (previousReplyCode == 550)
             {
                throw new FileNotFoundException();
             }
       }
       return FTP_ERROR;
     }

    /**
     * Pulls the response from the server and returns the code as a
     * number. Returns -1 on failure.
     */

    private int readResponse() throws IOException
    {
       StringBuffer buff = new StringBuffer(32);
       String responseStr;
       int   c;
       int   continuingCode = -1;
       int   code = 0;

       response.setSize(0);

       while (true)
       {
          while ((c = controlIn.read()) != -1)
          {
             if (c == '\r')
             {
                if ((c = controlIn.read()) != '\n')
                {
                   buff.append('\r');
```

Example D-1: FTP Data Source. (7 of 8)

```
                }
            }
            buff.append((char)c);


            if (c == '\n')
            {
                break;
            }
        }
        responseStr = buff.toString();
        buff.setLength(0);
        try
        {
            code = Integer.parseInt(responseStr.substring(0, 3));
        }
        catch (NumberFormatException e)
        {
            code = -1;
        }
        catch (StringIndexOutOfBoundsException e)
        {
            /* this line doesn't contain a response code, so
             * we just completely ignore it
             */
            continue;
        }
        response.addElement(responseStr);
        if (continuingCode != -1)
        {
            /* we've seen a XXX- sequence */
            if (code != continuingCode ||
                (responseStr.length() >= 4 &&
                responseStr.charAt(3) == '-'))
            {
                continue;
            }
            else
            {
                /* seen the end of code sequence */
                continuingCode = -1;
                break;
            }
        }
        else if (responseStr.length() >= 4 &&
            responseStr.charAt(3) == '-')
        {
            continuingCode = code;
            continue;
        }
        else
        {
```

Example D-1: FTP Data Source. (8 of 8)

```
            break;
         }
      }

      previousReplyCode = code;
      return code;
   }

   private int issueCommand(String cmd) throws IOException
   {
      int reply;
      if (replyPending)
      {
         if (readReply() == FTP_ERROR)
         {
            System.out.print("Error reading pending reply\n");
         }
      }
      replyPending = false;
      do
      {
         System.out.println(cmd);
         controlOut.print(cmd + "\r\n");
         reply = readReply();
      } while (reply == FTP_TRY_AGAIN);
      return reply;
   }
   /**
    * Parses the mediaLocator field into host and file strings
    */

   protected void parseLocator()
   {
      initCheck();
      String rest = getLocator().getRemainder();
      System.out.println("Begin parsing of: " + rest);
      int p1, p2 = 0;
      p1 = rest.indexOf("//");
      p2 = rest.indexOf("/", p1+2);
      hostString = rest.substring(p1 + 2, p2);
      fileString = rest.substring(p2);
      System.out.println("host: " + hostString + "   file: "
         + fileString);
   }
}
```

Source Stream

Example D-2:

```
 package intel.media.protocol.ftp;

 import java.io.*;
import javax.media.protocol.ContentDescriptor;
import javax.media.protocol.PullSourceStream;
import javax.media.protocol.SourceStream;

public class FTPSourceStream implements PullSourceStream
{
   protected InputStream dataIn;
   protected boolean eofMarker;
   protected ContentDescriptor cd;

   public FTPSourceStream(InputStream in)
   {
      this.dataIn = in;
      eofMarker = false;
      cd = new ContentDescriptor("unknown");
   }

   // SourceSteam methods

   public ContentDescriptor getContentDescriptor()
   {
      return cd;
   }

   public void close() throws IOException
   {
      dataIn.close();
   }

   public boolean endOfStream()
   {
      return eofMarker;
   }

   // PullSourceStream methods

   public int available() throws IOException
   {
      return dataIn.available();
   }
```

Example D-2:

```
    public int read(byte[] buffer, int offset, int length) throws
IOException
    {
       int n = dataIn.read(buffer, offset, length);
       if (n == -1)
       {
          eofMarker = true;
       }
       return n;
    }

    public boolean willReadBlock() throws IOException
    {
       if(eofMarker)
       {
          return true;
       }
       else
       {
          return dataIn.available() == 0;
       }
    }

    public long getContentLength()
    {
       return SourceStream.LENGTH_UNKNOWN;
    }
}
```

E

Sample Controller Implementation

This sample illustrates how a simple time-line `Controller` can be implemented in JMF. This sample is provided as a reference for developers who are implementing their own `Controllers`. Please note that it has not been tested or optimized for production use.

This sample consists of four classes:

- `TimeLineController.java`

  The `Controller`. You give it an array of time values (representing a time line) and it keeps track of which segment in the time line you are in.

- `TimeLineEvent.java`

  An event posted by the `TimeLineController` when the segment in the time line changes.

- `EventPostingBase.java`

  A base class used by `TimeLineController` that handles the `Controller` methods `addControllerListener` and `removeControllerListener`. It also provides a `postEvent` method that can be used by the subclass to post events.

- `ListenerList.java`

  A class used to maintain a list of `ControllerListener` objects that the `TimeLineController` needs to post events to.

This implementation also uses two additional classes whose implementations are not shown here.

- `EventPoster`

A class that spins a thread to post events to a `ControllerListener`.

- `BasicClock`

  A simple `Clock` implementation that implements all of the `Clock` methods.

TimeLineController

Example E-1: `TimeLineController.java` (1 of 11)

```
import javax.media.*;
import com.sun.media.MediaClock;

// This Controller uses two custom classes:
//    The base class is EventPostingBase.  It has three methods:
//      public void addControllerListener (ControllerListener
//         observer);
//      public void removeControllerListener (ControllerListener
//         observer);
//      protected void postEvent (ControllerEvent event);
//

//   This Controller posts TimeLineEvents.  TimeLineEvent has
//   two methods:
//      public TimeLineEvent (Controller who, int
//         segmentEntered);
//      public final int getSegment ();

public class TimeLineController extends EventPostingBase
    implements Controller, Runnable
{
    Clock ourClock;

    // This simple controller really only has two states:
    // Prefetched and Started.

    int ourState;
    long timeLine[];
    int currentSegment = -1;
    long duration;
    Thread myThread;

    // Create a TimeLineController giving it a sorted time line.
    // The TimeLineController will post events indicating when
    // it has passed to different parts of the time line.

    public TimeLineController (long timeLine[])
    {
        this.timeLine = timeLine;
        ourClock = new MediaClock ();
```

Example E-1: `TimeLineController.java` (2 of 11)

```
            duration = timeLine[timeLine.length-1];
            myThread = null;
            // We always start off ready to go!
            ourState = Controller.Prefetched;
        }

        // Binary search for which segment we are now in.  Segment
        // 0 is considered to start at 0 and end at timeLine[0].
        // Segment timeLine.length is considered to start at
        // timeLine[timeLine.length-1] and end at infinity.  At the
        // points of 0 and timeLine[timeLine.length-1] the
        // Controller will stop (and post an EndOfMedia event).

        int computeSegment (long time)
        {
            int max = timeLine.length;
            int min = 0;

            for (;;)
            {
                if (min == max) return min;
                int current = min + ((max - min) >> 1);

                if (time < timeLine[current])
                {
                    max = current;
                }

                else
                {
                    min = current + 1;
                }
            }
        }
        // These are all simple...

        public float setRate (float factor)
        {
            // We don't support a rate of 0.0.  Not worth the extra math
            // to handle something the user should do with the stop()
            // method!

            if (factor == 0.0f)
            {
                factor = 1.0f;
            }

            float newRate = ourClock.setRate (factor);
```

Example E-1: `TimeLineController.java` (3 of 11)

```
        postEvent (new RateChangeEvent (this, newRate));
        return newRate;
    }

    public void setTimeBase (TimeBase master)
        throws IncompatibleTimeBaseException
    {
        ourClock.setTimeBase (master);
    }

    public Time getStopTime ()
    {
        return ourClock.getStopTime ();
    }

    public Time getSyncTime ()
    {
        return ourClock.getSyncTime ();
    }

    public Time mapToTimeBase (Time t) throws ClockStoppedException
    {
        return ourClock.mapToTimeBase (t);
    }

    public Time getMediaTime ()
    {
        return ourClock.getMediaTime ();
    }

    public TimeBase getTimeBase ()
    {
        return ourClock.getTimeBase ();
    }
    public float getRate ()
    {
        return ourClock.getRate ();
    }

    // From Controller

    public int getState ()
    {
        return ourState;
    }
```

Example E-1: `TimeLineController.java` (4 of 11)

```
    public int getTargetState ()
    {
        return ourState;
    }

    public void realize ()
    {
        postEvent (new RealizeCompleteEvent (this, ourState,
            ourState, ourState));
    }

    public void prefetch ()
    {
      postEvent (new PrefetchCompleteEvent (this, ourState,
            ourState, ourState));
    }

    public void deallocate ()
    {
      postEvent (new DeallocateEvent (this, ourState,
            ourState, ourState, ourClock.getMediaTime ()));
    }

    public Time getStartLatency ()
    {
        // We can start immediately, of course!

        return new Time(0);
    }

    public Control[] getControls ()
    {
        return new Control[0];
    }
    public Time getDuration ()
    {
        return new Time(duration);
    }

    // This one takes a little work as we need to compute if we
    // changed segments.

    public void setMediaTime (Time now)
    {
        ourClock.setMediaTime (now);
        postEvent (new MediaTimeSetEvent (this, now));
```

Example E-1: `TimeLineController.java` (5 of 11)

```
        checkSegmentChange (now.getNanoseconds());
    }

    // We now need to spin a thread to compute/observe the
    // passage of time.

    public synchronized void syncStart (Time tbTime)
    {
        long startTime = ourClock.getMediaTime().getNanoseconds();

        // We may actually have to stop immediately with an
        // EndOfMediaEvent. We compute that now.  If we are already
        // past end of media, then we
        // first post the StartEvent then we post a EndOfMediaEvent

        boolean endOfMedia;
        float rate = ourClock.getRate ();

        if ((startTime > duration && rate >= 0.0f) ||
            (startTime < 0 && rate <= 0.0f))
        {
            endOfMedia = true;
        }

        else
        {
            endOfMedia = false;
        }

        // We face the same possible problem with being past the stop
        // time.  If so, we stop immediately.

        boolean pastStopTime;
        long stopTime = ourClock.getStopTime().getNanoseconds();
        if ((stopTime != Long.MAX_VALUE) &&
            ((startTime >= stopTime && rate >= 0.0f) ||
            (startTime <= stopTime && rate <= 0.0f)))
        {
            pastStopTime = true;
        }

        else
        {
            pastStopTime = false;
        }

        if (!endOfMedia && !pastStopTime)
        {
```

Example E-1: `TimeLineController.java` (6 of 11)

```
            ourClock.syncStart (tbTime);
            ourState = Controller.Started;
        }

        postEvent (new StartEvent (this, Controller.Prefetched,
            Controller.Started, Controller.Started,
            new Time(startTime), tbTime));

        if (endOfMedia)
        {
            postEvent (new EndOfMediaEvent (this,
                Controller.Started,
                Controller.Prefetched, Controller.Prefetched,
                new Time(startTime)));
        }

        else if (pastStopTime)
        {
            postEvent (new StopAtTimeEvent (this, Controller.Started,
                Controller.Prefetched, Controller.Prefetched,
                new Time(startTime)));
        }

        else
        {
            myThread = new Thread (this, "TimeLineController");

            // Set thread to appopriate priority...
            myThread.start ();
        }
    }



    // Nothing really special here except that we need to notify
    // the thread that we may have.

    public synchronized void setStopTime (Time stopTime)
    {
        ourClock.setStopTime (stopTime);
        postEvent (new StopTimeChangeEvent (this, stopTime));
        notifyAll ();
    }

    // This one is also pretty easy.  We stop and tell the running
    // thread to exit.
```

Example E-1: `TimeLineController.java` (7 of 11)

```
public synchronized void stop ()
{
    int previousState = ourState;
    ourClock.stop ();
    ourState = Controller.Prefetched;
    postEvent (new StopByRequestEvent (this, previousState,
        Controller.Prefetched, Controller.Prefetched,
        ourClock.getMediaTime ()));
    notifyAll ();

    // Wait for thread to shut down.

    while (myThread != null)
    {
        try
        {
            wait ();
        }
        catch (InterruptedException e)
        {
            // NOT REACHED
        }
    }
}

protected void checkSegmentChange (long timeNow)
{
    int segment = computeSegment (timeNow);
    if (segment != currentSegment)
    {
        currentSegment = segment;
        postEvent (new TimeLineEvent (this, currentSegment));
    }
}

// Most of the real work goes here.  We have to decide when
// to post events like EndOfMediaEvent and StopAtTimeEvent
// and TimeLineEvent.

public synchronized void run ()
{
    long timeToNextSegment = 0;
    long mediaTimeToWait = 0;

    float ourRate = 1.0f;
```

Example E-1: `TimeLineController.java` (8 of 11)

```
            for (;;)
            {
                // First, have we changed segments?  If so, post an event!

                long timeNow = ourClock.getMediaTime ().getNanoseconds ();
                checkSegmentChange (timeNow);

                // Second, have we already been stopped?  If so, stop
                // the thread.

                if (ourState == Controller.Prefetched)
                {
                    myThread = null;

                   // If someone is waiting for the thread to die, let them
                    // know.

                    notifyAll ();
                    break;
                }

                // Current rate.  Our setRate() method prevents the value
                // 0 so we don't check for that here.

                ourRate = ourClock.getRate ();

                // How long in clock time do we need to wait before doing
                // something?

                long endOfMediaTime;

                // Next, are we past end of media?

                if (ourRate > 0.0f)
                {
                    mediaTimeToWait = duration - timeNow;
                    endOfMediaTime = duration;
                }
                else
                {
                    mediaTimeToWait = timeNow;
                    endOfMediaTime = 0;
                }

                // If we are at (or past) time to stop due to EndOfMedia,
                // we do that now!
```

Example E-1: TimeLineController.java (9 of 11)

```
            if (mediaTimeToWait <= 0)
            {
                ourClock.stop ();
                ourClock.setMediaTime (new Time(endOfMediaTime));
                ourState = Controller.Prefetched;
               postEvent (new EndOfMediaEvent (this, Controller.Started,
                    Controller.Prefetched, Controller.Prefetched,
                    new Time(endOfMediaTime)));
                continue;
            }

            // How long until we hit our stop time?

            long stopTime = ourClock.getStopTime ().getNanoseconds();
            if (stopTime != Long.MAX_VALUE)
            {
                long timeToStop;
                if (ourRate > 0.0f)
                {
                    timeToStop = stopTime - timeNow;
                }
                else
                {
                    timeToStop = timeNow - stopTime;
                }

                // If we are at (or past) time to stop due to the stop
                // time, we stop now!
                if (timeToStop <= 0)
                {
                    ourClock.stop ();
                    ourClock.setMediaTime (new Time(stopTime));
                    ourState = Controller.Prefetched;
                    postEvent (new StopAtTimeEvent (this,
                        Controller.Started,
                        Controller.Prefetched, Controller.Prefetched,
                        new Time(stopTime)));
                    continue;
                }
                else if (timeToStop < mediaTimeToWait)
                {
                    mediaTimeToWait = timeToStop;
                }
            }
```

Example E-1: `TimeLineController.java` (10 of 11)

```
            // How long until we pass into the next time line segment?

            if (ourRate > 0.0f)
            {
                timeToNextSegment = timeLine[currentSegment] - timeNow;
            }

            else if (currentSegment == 0)
            {
                timeToNextSegment = timeNow;
            }

            else
            {
               timeToNextSegment = timeNow - timeLine[currentSegment-1];
            }
        }

        if (timeToNextSegment < mediaTimeToWait)
        {
            mediaTimeToWait = timeToNextSegment;
        }

        // Do the ugly math to compute what value to pass to
        // wait():

        long waitTime;
        if (ourRate > 0)
        {
            waitTime = (long) ((float) mediaTimeToWait / ourRate)  /
                 1000000;
        }
        else
        {
            waitTime = (long) ((float) mediaTimeToWait / -ourRate) /
                1000000;
        }
        // Add one because we just rounded down and we don't
        // really want to waste CPU being woken up early.

        waitTime++;

        if (waitTime > 0)
        {
            // Bug in some systems deals poorly with really large
            // numbers. We will cap our wait() to 1000 seconds
            // which point we will probably decide to wait again.
```

Example E-1: TimeLineController.java (11 of 11)

```
            if (waitTime > 1000000) waitTime = 1000000;
            try
            {
                wait (waitTime);
            }
            catch (InterruptedException e)
            {
            // NOT REACHED
            }
        }
    }

    public void close()
    {

    }

    public Control getControl(String type)
    {
        return null;
    }

    public long getMediaNanoseconds()
    {
        return 0;
    }
}
```

TimeLineEvent

Example E-2: `TimeLineEvent.java`

```
import javax.media.*;

// TimeLineEvent is posted by TimeLineController when we have
// switched segments in the time line.

public class TimeLineEvent extends ControllerEvent
{
    protected int segment;

    public TimeLineEvent (Controller source, int currentSegment)
    {
      super (source);
      segment = currentSegment;
    }

    public final int getSegment ()
    {
      return segment;
    }
}
```

EventPostingBase

Example E-3: `EventPostingBase.java` (1 of 3)

```
import javax.media.*;

// import COM.yourbiz.media.EventPoster;

// The implementation of the EventPoster class is not included as part
// of this example. EventPoster supports two methods:
// public EventPoster ();
// public void postEvent (ControllerListener who, ControllerEvent
//        what);

public class EventPostingBase
{
    protected ListenerList olist;
    protected Object olistLock;
    protected EventPoster eventPoster;
    // We sync around a new object so that we don't mess with
    // the super class synchronization.
```

Example E-3: `EventPostingBase.java` (2 of 3)

```
EventPostingBase ()
{
    olistLock = new Object ();
}

public void addControllerListener (ControllerListener observer)
{
    synchronized (olistLock)
    {
        if (eventPoster == null)
        {
            eventPoster = new EventPoster ();
        }

        ListenerList iter;
        for (iter = olist; iter != null; iter = iter.next)
        {
            if (iter.observer == observer) return;
        }

        iter = new ListenerList ();
        iter.next = olist;
        iter.observer = observer;
        olist = iter;
    }
}

public void removeControllerListener (ControllerListener  observer)
{
    synchronized (olistLock)
    {
        if (olist == null)
        {
            return;
        }
        else if (olist.observer == observer)
        {
            olist = olist.next;
        }
        else
        {
            ListenerList iter;
            for (iter = olist; iter.next != null; iter = iter.next)
            {
                if (iter.next.observer == observer)
                {
                    iter.next = iter.next.next;
```

Example E-3: `EventPostingBase.java` (3 of 3)

```
                        return;
                    }
                }
            }
        }
    }


    protected void postEvent (ControllerEvent event)
    {
        synchronized (olistLock)
        {
            ListenerList iter;
            for (iter = olist; iter != null; iter = iter.next)
            {
                eventPoster.postEvent (iter.observer, event);
            }
        }
    }
}
```

ListenerList

Example E-4: `ListenerList.java`

```
// A list of controller listeners that we are supposed to send
// events to.

class ListenerList
{
    ControllerListener observer;
    ListenerList next;
}
```

EventPoster

Example E-5: `EventPoster.java`

```
class EventPoster
{
    void postEvent(Object object, ControllerEvent evt)
   {
   // Post event.
    }
}
```

F

RTPUtil

`RTPUtil` demonstrates how to create separate RTP players for each stream in a session so that you can play the streams. To do this, you need to listen for `NewRecvStreamEvents` and retrieve the `DataSource` from each new stream. (See "Creating an RTP Player for Each New Receive Stream" on page 132 for more information about this example.)

Example F-1: `RTPUtil` (1 of 5)

```
import javax.media.rtp.*;
import javax.media.rtp.rtcp.*;
import javax.media.rtp.event.*;
import javax.media.*;
import javax.media.protocol.*;
import java.net.InetAddress;
import javax.media.format.AudioFormat;
// for PlayerWindow
import java.awt.*;
import com.sun.media.ui.*;

import java.util.Vector;

public class RTPUtil implements ReceiveStreamListener,
                                ControllerListener
{

    Vector playerlist = new Vector();
    SessionManager mgr = null;
    boolean terminatedbyClose = false;

    public SessionManager createManager(String address,
                                        String sport,
                                        String sttl,
                                        boolean listener,
                                        boolean sendlistener)
    {
```

Example F-1: `RTPUtil` (2 of 5)

```
        return createManager(address,
                             new Integer(sport).intValue(),
                             new Integer(sttl).intValue(),
                             listener,
                             sendlistener);
    }
    public SessionManager createManager(String address,
                                        int port,
                                        int ttl,
                                        boolean listener,
                                        boolean sendlistener)
    {
        mgr = (SessionManager)new com.sun.media.rtp.RTPSessionMgr();

        if (mgr == null) return null;

        mgr.addFormat(new AudioFormat(AudioFormat.DVI_RTP,
                                      44100,
                                      4,
                                      1),
                      18);
        if (listener) mgr.addReceiveStreamListener(this);
        if (sendlistener) new RTPSendStreamWindow(mgr);

        // ask RTPSM to generate the local participants CNAME
        String cname = mgr.generateCNAME();
        String username = null;

        try {
            username = System.getProperty("user.name");
        } catch (SecurityException e){
            username = "jmf-user";
        }

        // create our local Session Address
        SessionAddress localaddr = new SessionAddress();

        try{
            InetAddress destaddr = InetAddress.getByName(address);

            SessionAddress sessaddr = new SessionAddress(destaddr,
                                                         port,
                                                         destaddr,
                                                         port + 1);

            SourceDescription[] userdesclist= new SourceDescription[]
            {
                new SourceDescription(SourceDescription
                                      .SOURCE_DESC_EMAIL,
                                      "jmf-user@sun.com",
                                      1,
                                      false),
```

Example F-1: `RTPUtil` (3 of 5)

```
                new SourceDescription(SourceDescription
                                      .SOURCE_DESC_CNAME,
                                      cname,
                                      1,
                                      false),

                new SourceDescription(SourceDescription
                                      .SOURCE_DESC_TOOL,
                                      "JMF RTP Player v2.0",
                                      1,
                                      false)
            };

            mgr.initSession(localaddr,
                            userdesclist,
                            0.05,
                            0.25);

            mgr.startSession(sessaddr,ttl,null);
        } catch (Exception e) {
            System.err.println(e.getMessage());
            return null;
        }

        return mgr;
    }
    public void update( ReceiveStreamEvent event)
    {
        Player newplayer = null;
        RTPPlayerWindow playerWindow = null;

        // find the sourceRTPSM for this event
        SessionManager source = (SessionManager)event.getSource();

        // create a new player if a new recvstream is detected
        if (event instanceof NewReceiveStreamEvent)
        {
            String cname = "Java Media Player";
            ReceiveStream stream = null;

            try
            {
                // get a handle over the ReceiveStream
                stream =((NewReceiveStreamEvent)event)
                        .getReceiveStream();

                Participant part = stream.getParticipant();

                if (part != null) cname = part.getCNAME();

                // get a handle over the ReceiveStream datasource
                DataSource dsource = stream.getDataSource();
```

Example F-1: `RTPUtil` (4 of 5)

```
                // create a player by passing datasource to the
                // Media Manager
                newplayer = Manager.createPlayer(dsource);
                System.out.println("created player " + newplayer);
            } catch (Exception e) {
                System.err.println("NewReceiveStreamEvent exception "
                                   + e.getMessage());
                return;
            }

            if (newplayer == null) return;

            playerlist.addElement(newplayer);
            newplayer.addControllerListener(this);

            // send this player to player GUI
            playerWindow = new RTPPlayerWindow( newplayer, cname);
        }
    }
    public void controllerUpdate( ControllerEvent evt)
    {
        // get a handle over controller, remove it from the player
        // list.
        // if player list is empty, close the sesssion manager.

        if ((evt instanceof ControllerClosedEvent) ||
            (evt instanceof ControllerErrorEvent) ||
            (evt instanceof DeallocateEvent)){
            Player p = (Player)evt.getSourceController();

            if (!terminatedbyClose){
                if (playerlist.contains(p))
                    playerlist.removeElement(p);
                if ((playerlist.size() == 0) && (mgr != null))
                    mgr.closeSession("All players are closed");
            }
        }
    }

    public void closeManager()
    {
        terminatedbyClose = true;

        // first close all the players
        for (int i = 0; i < playerlist.size(); i++) {
            ((Player)playerlist.elementAt(i)).close();
        }
        if (mgr != null) {
            mgr.closeSession("RTP Session Terminated");
            mgr = null;
        }
    }
```

Example F-1: `RTPUtil` (5 of 5)

```
    class RTPPlayerWindow extends PlayerWindow
    {
        public RTPPlayerWindow( Player player, String title)
        {
            super(player);
            setTitle(title);
        }
        public void Name(String title){
            setTitle(title);
        }
    }
}
```

Glossary

broadcast
Transmit a data stream that multiple clients can receive if they choose to.

Buffer
The container for a chunk of media data.

CachingControl
A media control that is used to monitor and display download progress. information.

capture device
A microphone, video capture board, or other source that generates or provides time-based media data. A capture device is represented by a `DataSource`.

CaptureDeviceControl
A media control that enables the user to control a capture device.

CaptureDeviceInfo
An information object that maintains information about a capture device, such as its name, the formats it supports, and the `MediaLocator` needed to construct a `DataSource` for the device.

CaptureDeviceManager
The manager that provides access to the capture devices available to JMF.

Clock
A media object that defines a transformation on a `TimeBase`.

close
Release all of the resources associated with a `Controller`.

codec
A compression/decompression engine used to convert media data between compressed and raw formats. The JMF plug-in architecture enables technology providers to supply codecs that can be seamlessly integrated into JMF's media processing.

compositing
Combining multiple sources of media data to form a single finished product.

configured
A `Processor` state that indicates that the `Processor` has been connected to its data source and the data format has been determined.

configuring
A `Processor` state that indicates that `configure` has been called and the `Processor` is connecting to the `DataSource`, demultiplexing the input stream, and accessing information about the data format.

content name
A string that identifies a content type.

content package-prefix
A package prefix in the list of package prefixes that the `PackageManager` maintains for content extensions such as new `DataSource` implementations.

content package-prefix list
The list of content package prefixes maintained by the `PackageManager`.

content type
A multiplexed media data format such as MPEG-1, MPEG-2, QuickTime, AVI, WAV, AU, or MIDI. Content types are usually identified by MIME types.

Control
A JMF construct that can provide access to a user interface component to supports user interaction. JMF controls implement the `Control` interface.

control-panel component
The user interface component that enables the user to control the media presentation.

Controller
The key construct in the JMF Player/Processor API. The `Controller` interface defines the basic state and control mechanism for an object that controls, presents, or captures time-based media.

ControllerAdapter
An event adapter that receives `ControllerEvents` and dispatches them to an appropriate stub-method. Classes that extend this adapter can easily replace only the message handlers they are interested in

ControllerClosedEvent
An event posted by a `Controller` when it shuts down. A `ControllerErrorEvent` is a special type of `ControllerClosedEvent`.

ControllerEvent
The ControllerEvent class is the base class for events posted by a Controller object. To receive `ControllerEvents`, you implement the `ControllerListener` interface.

ControllerListener
An object that implements the ControllerListener interface to receive notification whenever a `Controller` posts a `ControllerEvent`. See also *ControllerAdapter.*

data
The actual media data contained by a `Buffer` object.

DataSink
An object that implements the `DataSink` interface to read media content from a `DataSource` and render the media to a destination.

DataSource
An object that implements the DataSource interface to encapsulate the location of media and the protocol and software used to deliver the media.

deallocate
Release any exclusive resources and minimize the use of non-exclusive resources.

decode
Convert a data stream from a compressed type to an uncompressed type.

demultiplex
Extract individual tracks from a multiplexed media stream.

Demultiplexer
A JMF plug-in that parses the input stream. If the stream contains interleaved tracks, they are extracted and output as separate tracks.

duration
The length of time it takes to play the media at the default rate.

Effect
A JMF plug-in that applies and effect algorithm to a track and outputs the modified track in the same format.

encode
Convert a data stream from an uncompressed type to a compressed type.

end of media (eom)
The end of a media stream.

StreamWriterControl
A `Control` implemented by data sinks and multiplexers that generate output data. This `Control` enables users to specify a limit on the amount of data generated.

format
A structure for describing a media data type.

frame
One unit of data in a track. For example, one image in a video track.

frame rate
The number of frames that are displayed per second.

GainChangeEvent
An event posted by a `GainControl` whenever the volume changes.

GainChangeListener
An object that implements the `GainChangeListener` interface to receive `GainChangeEvents` from a `GainControl`.

GainControl
A JMF `Control` that enables programmatic or interactive control over the playback volume.

JMF (Java Media Framework)
An application programming interface (API) for incorporating media data types into Java applications and applets.

key frame
A frame of video that contains the data for the entire frame rather than just the differences from the previous frame.

latency
See *start latency.*

managed controller
A `Controller` that is synchronized with other `Controllers` through a managing `Player`. The managing `Player` drives the operation of each managed `Controller`—while a `Controller` is being managed, you should not directly manipulate its state.

Manager
The JMF access point for obtaining system dependent resources such as `Players`, `Processors`, `DataSources` and the system `TimeBase`.

managing player
A `Player` that is driving the operation of other `Controllers` in order to synchronize them. The `addController` method is used to place `Controllers` under the control of a managing `Player`.

maximum start latency
The maximum length of time before a `Player` will be ready to present media data.

media capture
The process of acquiring media data from a source such as a microphone or a video capture card that's connected to a camera.

media data
The media information contained in a media stream.

media event
An event posted by a `GainControl`, `DataSink`, or a `Controller` to notify listeners that the status of the object posting the event has changed.

media processing
Manipulating media data to apply effect algorithms, convert the data to a different format, or present it.

media stream
A data stream that contains time-based media information.

media time
The current position in a media stream.

MediaHandler
An object that implements the `MediaHandler` interface, which defines how the media source that the handler uses to obtain content is selected. There are currently three supported types of `MediaHandlers`: `Player` (including `Processor`), `MediaProxy`, and `DataSink`.

MediaLocator
An object that describes the media that a `Player` presents. A `MediaLocator` is similar to a `URL` and can be constructed from a `URL`. In the Java programming language, a `URL` can only be constructed if the corresponding protocol handler is installed on the system. `MediaLocator` doesn't have this restriction.

MediaProxy
An object that processes content from one `DataSource` to create another. Typically, a `MediaProxy` reads a text configuration file that contains all of the information needed to make a connection to a server and obtain media data.

MIME type
A standardized content type description based on the Multipurpose Internet Mail Extensions (MIME) specification.

MonitorControl
A `Control` that provides a way to display the capture monitor for a particular capture device.

multicast
Transmit a data stream to a select group of participants. See also *broadcast*, *unicast*.

multiplex
Merge separate tracks into one multiplexed media stream.

multiplexed media stream
A media stream that contains multiple channels of media data.

Multiplexer
A JMF plug-in that combines multiple tracks of input data into a single interleaved output stream and delivers the resulting stream as an output `DataSource`.

output data source
A `DataSource` that encapsulates a `Processor` object's output.

package prefix
An identifier for your code that you register with the JMF `PackageManager`. For example, `COM.yourbiz`. The PackageManager maintains separate lists of package prefixes for content and protocol extensions. See also *content package-prefix*, *protocol package-prefix*.

participant
In RTP, an application participating in an RTP session

payload
In RTP, the data transported by RTP in a packet, such as audio samples or compressed video data.

Player
An object that implements the `Player` interface to processes a stream of data as time passes, reading data from a `DataSource` and rendering it at a precise time.

Player state
One of the six states that a `Player` can be in: *Unrealized*, *Realizing*, *Realized*, *Prefetching*, *Prefetched,* and *Started.* In normal operation, a `Player` steps through each state until it reaches the *Started* state.

playback
The process of presenting time-based media to the user.

plug-in
A media processing component that implements the JMF `PlugIn` interface.

PlugInManager
A manager object that maintains a registry of installed plug-ins and is used to search the available plug-ins.

Positionable
An object that supports changing the media position within the stream and implements the `Positionable` interface.

post-process

Apply an effect algorithm after the media stream has been decoded.

pre-process
Apply an effect algorithm before the media stream is encoded.

prefetch
Prepare a `Player` to present its media. During this phase, the `Player` preloads its media data, obtains exclusive-use resources, and anything else it needs to do to prepare itself to play.

prefetched
A `Player` state in which the `Player` is ready to be started.

prefetching
A `Player` state in which the `Player` is in the process of preparing itself to play.

Processor
A special type of JMF `Player` that can provide control over how the media data is processed before it is presented.

Processor state
One of the eight states that a `Processor` can be in. A `Processor` has two more *Stopped* states than a `Player`: *Configuring* and *Configured*. See also *Player state*.

ProcessorModel
An object that defines the input and output requirements for a `Processor`. When a `Processor` is created using a `ProcessorModel`, the `Manager` does its best to create a `Processor` that meets these requirements.

progress bar component
The user interface component that can be retrieved from a `CachingControl` to display download progress to the user.

protocol
A data delivery mechanism such as HTTP, RTP, FILE.

protocol package-prefix
A package prefix in the list of package prefixes that the `PackageManager` maintains for protocol extensions such as new `MediaHandlers`.

protocol package-prefix list
The list of protocol package prefixes maintained by the `PackageManager`.

pull
Initiate the data transfer and control the flow of data from the client side.

PullBufferDataSource
A pull `DataSource` that uses a `Buffer` object as its unit of transfer.

PullDataSource
A `DataSource` that enables the client to initiate the data transfer and control the flow of data.

PullBufferStream
A `SourceStream` managed by a `PullBufferDataSource`.

PullSourceStream
A `SourceStream` managed by a `PullDataSource`.

push
Initiate the data transfer and control the flow of data from the server side.

PushBufferDataSource
A push `DataSource` that uses a `Buffer` object as its unit of transfer.

PushDataSource
A `DataSource` that enables the server to initiate the data transfer and control the flow of data.

PushBufferStream
A `SourceStream` managed by a `PushBufferDataSource`.

PushSourceStream
A `SourceStream` managed by a `PushDataSource`.

rate
A temporal scale factor that determines how media time changes with respect to time-base time. A `Player` object's rate defines how many media time units advance for every unit of time-base time.

raw media format
A format that can be directly rendered by standard media rendering devices without the need for decompression. For audio, a PCM sample representation is one example of a raw media format.

realize
Determine resource requirements and acquire the resources that the `Player` only needs to acquire once.

realized
: The `Player` state in which the `Player` knows what resources it needs and information about the type of media it is to present. A *Realized* `Player` knows how to render its data and can provide visual components and controls. Its connections to other objects in the system are in place, but it doesn't own any resources that would prevent another `Player` from starting.

realizing
: The `Player` state in which the `Player` is determining what resources it needs and gathering information about the type of media it is to present.

render
: Deliver media data to some destination, such as a monitor or speaker.

Renderer
: A JMF plug-in that delivers media data to some destination, such as a monitor or speaker.

RTCP
: RTP Control Protocol.

RTP
: Real-time Transfer Protocol.

session
: In RTP, the association among a set of participants communicating with RTP. A session is defined by a network address plus a port pair for RTP and RTCP.

source
: A provider of a stream of media data.

SourceStream
: A single stream of media data.

SSRC
: See synchronization source.

start
: Activate a `Player`. A *Started* `Player's` time-base time and media time are mapped and its clock is running, though the `Player` might be waiting for a particular time to begin presenting its media data.

start latency
The time it takes before a `Player` can begin presenting media data.

started
One of the two fundamental Clock states. (The other is *Stopped*.) `Controller` breaks the `Started` state down into several resource allocation phases: *Unrealized*, *Realizing*, *Realized*, *Prefetching*, and *Prefetched*.

status change events
`Controller` events such as `RateChangeEvent`, `SizeChangeEvent` `StopTimeChangeEvent` that indicate that the status of a `Controller` has changed.

stop
Halt a `Player's` presentation of media data.

stop time
The media time at which a `Player` should halt.

synchronization source
The source of a stream of RTP packets, identified by a 32-bit numeric SSRC identifier carried in the RTP header.

synchronize
Coordinate two or more Controller's so that they can present media data together. Synchronized Controller's use the same `TimeBase`.

target state
The state that a `Player` is heading toward. For example, when a `Player` is in the *Realizing* state, its target state is *Realized*.

time-based media
Media such as audio, video, MIDI, and animations that change with respect to time.

TimeBase
An object that defines the flow of time for a `Controller`. A `TimeBase` is a constantly ticking source of time, much like a crystal.

time-base time
The current time returned by a `TimeBase`.

track
A channel in a multiplexed media stream that contains media or control data. For example, a multiplexed media stream might contain an audio track and a video track.

TrackControl
A `Control` used to query, control and manipulate the data of individual media tracks.

track format
The format associated with a particular track.

transcode
Convert a data stream from an uncompressed type to a compressed type or vice-versa.

transition events
`ControllerEvents` posted by a `Controller` as its state changes.

unicast
Transmit a data stream to a single recipient.

unrealized
The initial state of a `Player`. A `Player` in the *Unrealized* state has been instantiated, but does not yet know anything about its media.

URL
Universal Resource Locator.

user-interface component
An instance of a class that implements the `Component` interface. JMF `Players` have two types of default user-interface components, a `ControlPanelComponent` and a `VisualComponent`.

visual component
The user interface component that displays the media or information about the media.

VOD
Video on Demand.

Index