

QuickTime Streaming

This documentation provides

- **A brief overview of QuickTime streaming**
- **A section on using QuickTime streaming that includes code samples**

Read both of these sections if your application plays, edits, or creates streaming QuickTime movies, or if you want to send streaming QuickTime movies from a server.

- **A list of common streaming error codes**

If your application plays streaming QuickTime movies, it should deal gracefully with the errors on this list.

- **A description of the hint track file structure**

Read the description of the hint track file structure only if you are writing an RTP server that will send QuickTime movies, or if you are writing a media packetizer.

- **A section on writing media packetizers and reassemblers**

This section is only for developers who want to optimize the delivery and error-recovery of a particular media by writing a packetizer and a reassembler. You might want to do this if you have developed your own codec, for example.

- **Definitions of media packetizer, reassembler, and packet builder functions**

The definitions of these functions are provided for developers who are writing media packetizers and reassemblers. These functions are not called by applications.

About Streaming

Streaming involves sending movies from a **server** to a **client** over a network such as the Internet. The server breaks the movie into **packets** that can be sent over the network. At the receiving end, the packets are reassembled by the client and the movie is played as it comes in. A series of related packets is called a **stream**.

Streaming is different from simple file transfer, in that the client plays the movie as it comes in over the network, rather than waiting for the entire movie to

download before it can be played. In fact, a streaming client may never actually download a streaming movie; it may simply play the movie's packets as they come in, then discard them.

QuickTime movies can be streamed using a variety of protocols, including

- HTTP (Hyper Text Transport Protocol)
- FTP (File Transfer Protocol)

and, new in QuickTime 4

- RTP (Realtime Transport Protocol).

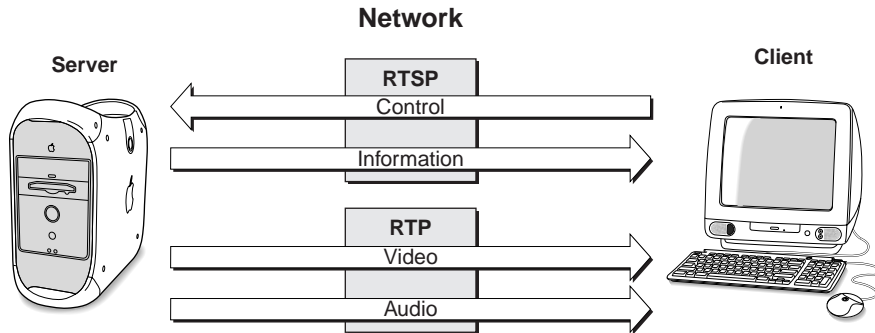
HTTP and FTP are essentially file transfer protocols. Any QuickTime movie saved using the QuickStart option can be streamed using these protocols because the QuickTime client software is able to start playing the movie before the entire file has arrived.

RTP is used for **real time streaming**. The movie packets are sent in real time, so that a one-minute movie is sent over the network in one minute. The packets are time-stamped, so they can be displayed in time-synchronized order. Because packets are sent in real time, RTP streaming works with **live content** in addition to previously-recorded movies. It can even carry a mixture of the two.

Real-time streams can be sent one-to-one (**unicast**) or one-to-many (**multicast**).

Unicast Streaming

In a unicast, the client contacts the server to request a movie using **RTSP** (Real Time Streaming Protocol). The server then replies to the client over RTSP with information describing the movie as a **streaming session**. A streaming session consists of one or more streams of data, such as a video stream and an audio stream. The server tells the client how many streams to expect and gives details on each stream, such as the media type and codec. The actual streams are then sent to the client over RTP. When a QuickTime movie is streamed over RTP, each track in the movie is sent as a separate stream.

Figure 26-1 Unicast streaming using RTSP

A stream can contain live content, such as a stock ticker or a radio broadcast, or stored content, such as a video track from a QuickTime movie. When a client is receiving unicast streams from stored content, the client's movie controller includes a "thumb" that allows the user to jump to any point in the movie. This gives the client random access to long movies without having to download an entire movie or store it locally. The client simply asks the server to begin streaming the movie from a new point.

Figure 26-2 Movie controller with "thumb"

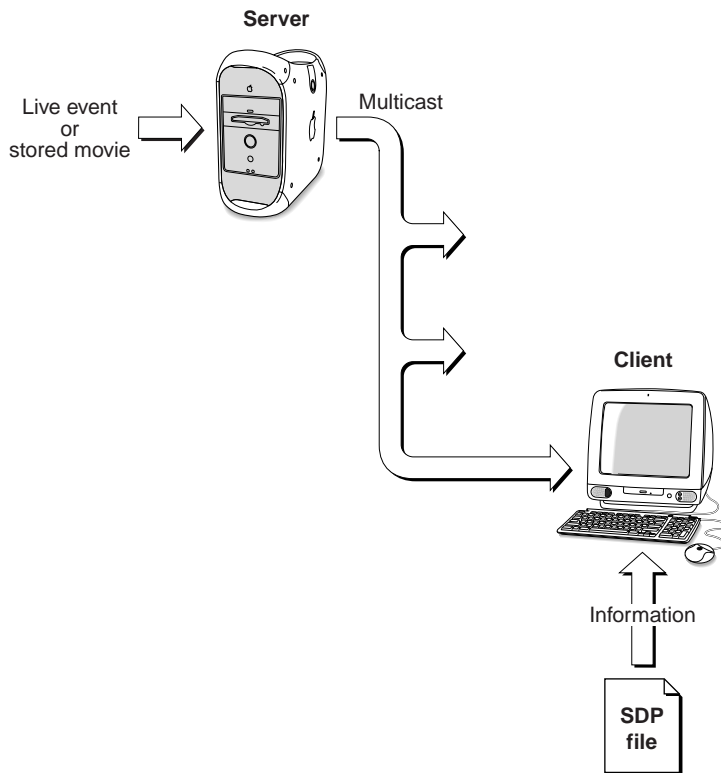
Note: RSTP uses TCP/IP transport, but RTP uses low-level UDP/IP transport. If the client is behind a firewall that doesn't pass UDP, the movie will set up without error, but no media will be displayed. RTSP Proxy servers for several operating systems are available from Apple to help resolve firewall problems.

Multicast Streaming

In a multicast, one copy of each stream is sent over each branch of a network. This reduces the amount of network traffic required to send the streams to large numbers of clients. A client receives the streams by "joining" the multicast.

The client finds out how to join the multicast by opening an **SDP** (Session Description Protocol) file. The SDP file contains the information needed to join the multicast, such as group address and port number, as well as the stream description information that would come over RTSP for a unicast. SDP files are commonly posted on web servers to announce upcoming multicasts.

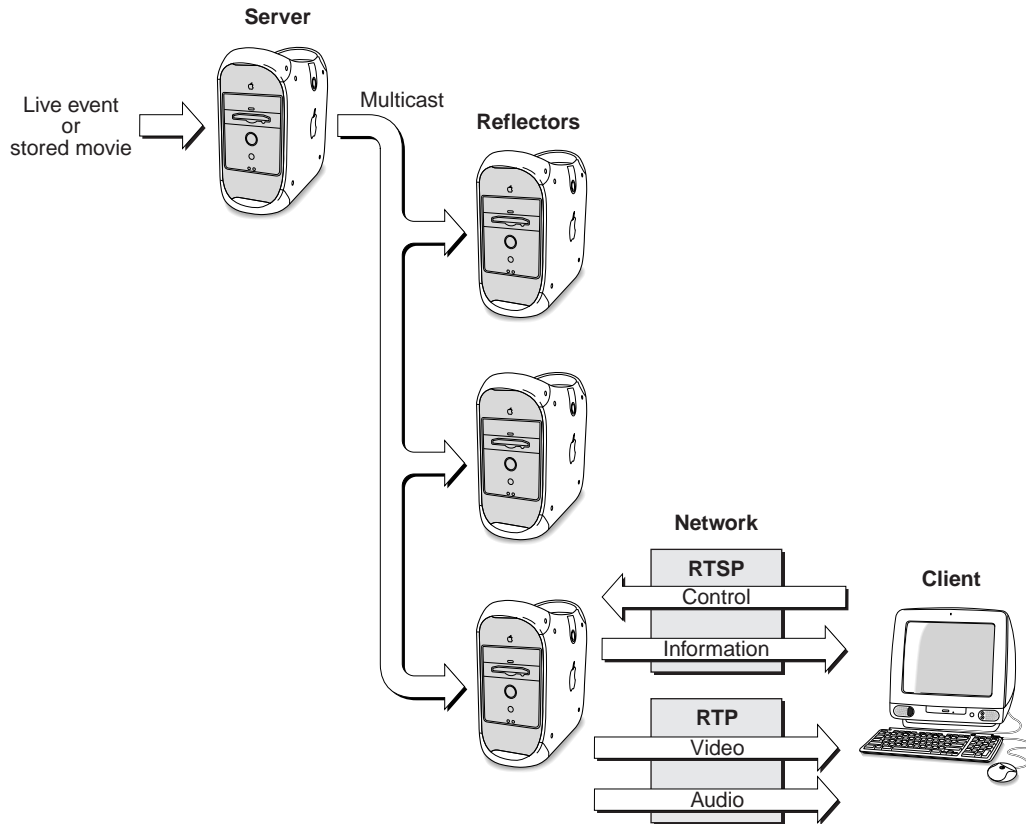
Figure 26-3 QuickTime 4 allows your computer to join a multicast by opening an SDP file.



Not all routers support multicasting. QuickTime clients behind routers that don't implement multicasting can still receive a multicast by requesting the streams from a **reflector**. A reflector is an RTSP server that joins a multicast, then converts the multicast into a series of unicasts, passing the streams to clients who request them (see diagram below). The original server may be sending live

content, such as a concert or a news broadcast, or a previously-recorded movie. The reflector is always sending “live” data, passing the streams in real time.

Figure 26-4 You can also receive a multicast through a reflector



When a QuickTime client is viewing a multicast or a live unicast, the user’s movie controller has no “thumb” control. The user can stop or resume the display, and may have audio controls if the movie includes a sound stream, but there is no way to skip forward or back in a live transmission or a multicast.

Figure 26-5 Movie controller without “thumb”

Streaming QuickTime Over RTP or HTTP

QuickTime 4 supports streaming over RTP and HTTP. The main advantages of streaming over RTP are:

- RTP can be used for live transmission and multicast.
- Real-time streaming allows the user to view long movies or continuous transmissions without having to store more than a few seconds of data locally.
- Using RTP transmission under RTSP control, a user can skip to any point in a movie on a server without downloading the intervening material.
- You can stream a single track over RTP, whereas HTTP streams only whole movies. RTP streams can be incorporated in a movie using **streaming tracks**. A streaming track is a track in a QuickTime movie that contains the URL of the streaming content.
- A QuickTime movie that contains streaming tracks can also include non-streaming tracks whose media exist on the client's computer. This allows a live transmission, or data stored on the Internet, to be incorporated into a movie along with material stored on the client's hard drive or distributed over CD-ROM.
- RTP uses UDP/IP protocol, which doesn't attempt to retransmit lost packets. This allows multicasts as well as live streams, both cases where retransmission would not be practical.

The main advantages of streaming over HTTP are:

- HTTP uses TCP/IP protocol to ensure that all movie packets are delivered, retransmitting if necessary.
- HTTP does not attempt to stream in real time. To stream in real time, the bandwidth of the network must be greater than the data rate of the movie. If there is not enough bandwidth to transmit the movie in real time, streaming by HTTP allows the client to store the data locally and play the movie after enough has arrived.

- Most firewalls and network configuration schemes will pass HTTP without modification.
- Any QuickTime movie can be streamed using HTTP. QuickTime 4 supports RTP streaming of video, audio, text, and MIDI. To stream a movie with other media types, such as sprites, you should use HTTP.

Features In QuickTime 4

QuickTime Streaming extends the QuickTime software architecture to support the creation, transmission, and reception of multimedia streams. This allows QuickTime programmers to create applications that receive multimedia in real time, and to create authoring and editing tools that work with streaming content. Existing applications that play QuickTime movies can play real-time streaming movies with little or no code change.

Earlier versions of QuickTime supported unicast streaming of whole movie files using HTTP. QuickTime 4 adds support for Realtime Transport Protocol (RTP), which can be used for multicasts and for transmission of live content, as well as unicast of stored movies. QuickTime 4 also adds the ability to stream individual tracks via RTP, allowing developers to incorporate live or remotely-stored content in a local movie.

This release of QuickTime supports streaming using RTP transport for media and RTSP protocol for control.

This release of QuickTime includes client software that can receive multicasts directly from routers. This software is standards-based and is interoperable with products such as VIC, VAT, or Cisco's IP/TV.

This release of QuickTime understands SDP files, which are used to "tune into" multicast streaming sessions.

This release of QuickTime supports RTP streaming of audio, video, MIDI, text (including HREF tracks), and tweens. RTP streaming of other media types is not supported in this release, but movies with other media types can now incorporate streaming content.

This release of QuickTime supports **IETF standard payload types** for RTP:

- Video
 - H.261
 - H.263+

- ☐ DVI
- ☐ JPEG
- Audio
 - ☐ Qualcomm QCELP
 - ☐ GSM (receive only)
 - ☐ Raw audio
 - ☐ μ -law
 - ☐ a-law

This release also supports **QuickTime in RTP packing** (IETF draft) for all QuickTime encodings of

- video
- audio
- text
- MIDI
- tween

Special packing is included for **optimized transmission** of some QuickTime codecs, including

- Sorenson video
- Qualcomm Purevoice audio
- QDesign music

If you have your own codec, you can design special packing for it by writing a packetizer and a reassembler, as described later in this chapter.

This release of QuickTime extends the QuickTime File Format to include **hint tracks**, which simplify the process of packetizing QuickTime movies into RTP streams. Hint tracks allow QuickTime movies to be served from RTP servers without requiring the servers to have QuickTime software installed or to know about QuickTime media types or codecs.

This release of QuickTime adds the ability to export QuickTime movies to hinted movies that can be streamed over RTP.

This release of QuickTime does not support RTP streaming of track references. Some QuickTime features, such as effects, chapter lists, and various applications of tweens, make use of track references. These features can be included in client

movies that incorporate RTP streaming content, as described in the section [Creating Streaming Movies](#).

Using QuickTime Streaming

This section discusses three areas of primary interest to applications developers:

- Receiving streaming movies
- Serving streaming movies
- Creating streaming movies

Code samples for receiving and creating streaming movies are included.

Receiving Streaming Movies

An application receives streaming content by opening a movie and playing it.

Opening A Streaming Movie

In general, opening a streaming movie is like opening any QuickTime movie. You can open a streaming movie by opening

- a movie file that contains streaming tracks
- an SDP file
- a URL

You can open a movie file that contains streaming tracks, or open an SDP file, by calling `NewMovieFromFile` in the usual way.

You open a movie from a URL by calling `NewMovieFromDataRef` with a URL data reference. The URL for a real-time streaming movie will use `RTSP://` protocol. The following is a code sample for opening a movie from an RTSP URL:

```
char url[] = "rtsp://www.mycompany.com/mymovie.mov";
Handle urlDataRef;

urlDataRef = NewHandle(strlen(url) + 1);
if ( ( err = MemError() ) != noErr) goto bail;
```

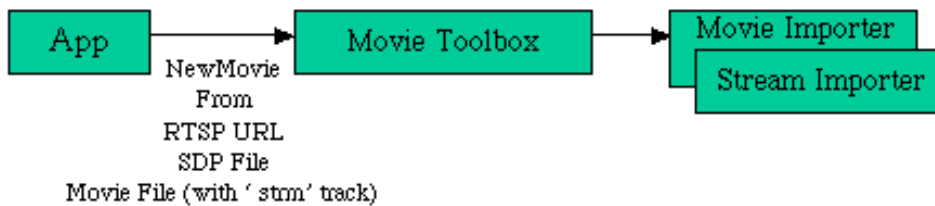
```
BlockMoveData(url, *urlDataRef, strlen(url) + 1);

err = NewMovieFromDataRef(&movieInfo->theMovie, newMovieActive,
    nil, urlDataRef, URLDataHandlerSubType);
DisposeHandle(urlDataRef);
```

It's also possible to open a movie file from an HTTP:// or FTP:// URL, and for that movie file to contain streaming tracks.

If you open a streaming movie from a URL or an SDP file, the Movie Toolbox will call the appropriate movie importer. If you open a movie file that contains streaming tracks, stream importers will be called.

Figure 26-6 Opening a streaming movie



Opening a streaming movie typically takes more time than opening a movie with purely local content. Each track in the movie on the server is transmitted as an RTP stream, so the client computer must establish a network connection with the server for each track, and often must establish a connection for RTSP control as well. This takes time, particularly if a dial-up connection must be established.

It also means the Movie Toolbox can return connection status messages or networking errors in the process of opening a movie.

Playing A Streaming Movie

Applications that can already play QuickTime movies need to take the following steps to play real-time streaming movies reliably:

- Open movies with high-level Movie Toolbox calls or a movie importer.

- Do not assume that the track structure of the movie you play reflects the track structure of the original movie; a streaming track can contain sound, video, text, MIDI, or all of these.
- Use a movie controller to play the movie, or use the new `PrePrerollMovie` function to set up any streams before playing the movie.
- Display status messages returned by the movie controller.
- Be prepared to deal with network errors when your application plays a movie (see “Common Streaming Error Codes”).
- Be prepared for movie characteristics to change dynamically; the height, width, duration, and whether or not the movie has audio or video can all change as the movie plays.
- Do not assume that the movie will begin immediately.

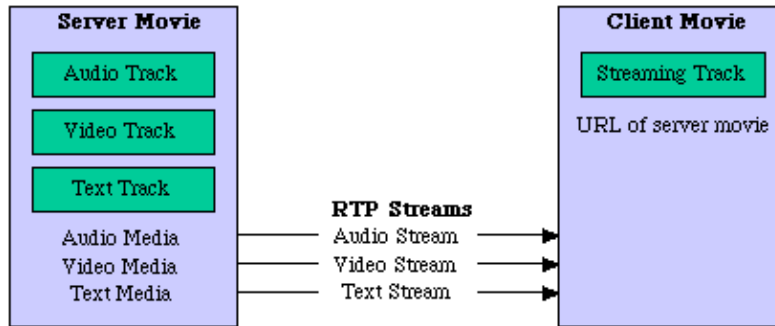
The following sections describe some of these steps in more detail.

Track Structure

Unlike other QuickTime movies, streaming movies consist of two distinct movie files—one on the server and the other on the client machine—often with different track structures. You sometimes need to distinguish between the **server movie** and the **client movie**.

The media of each track in the server movie is transmitted as a separate RTP stream. On the client side, multiple RTP streams can be combined into a single streaming track.

If you open a movie from a URL, for example, the movie importer may create a client movie that contains only a streaming (`'strm'`) track. Unlike most other QuickTime track types, a streaming track can contain multiple media streams of different types. A streaming track in a client movie may contain the URL of a server movie with audio, video, text, and/or MIDI tracks.

Figure 26-7 Server movie and client movie

The client movie file never contains audio or video media from the server movie; they are displayed and discarded. If a client movie is saved, the saved movie contains information such as the URL of the server movie and the currently-displayed movie time. The client movie can also contain local tracks and local media, but the media samples in the server movie remain on the server except when playing.

Pre-Prerolling

Before any movie is played, QuickTime needs to allocate buffers and open appropriate media handlers. This process is called **prerolling** the movie. Before a streaming movie can be played, additional steps need to be taken, such as establishing RTP streams between the client and the server. This setup process is called **pre-prerolling**. Pre-prerolling is performed automatically when a streaming movie is played using a movie controller. If your application uses movie controllers to play movies, you do not need to take any special steps to pre-preroll a streaming movie.

If you are playing movies using lower-level commands, you will need to use the new `PrePrerollMovie` function to set up the network connections for a streaming movie before you can preroll or play the movie. The `PrePrerollMovie` function does nothing unless a movie contains streaming content, so it's safe to call it for all movies. A code sample follows.

```
PrePrerollMovie(myMovie, 0, GetMoviePreferredRate(myMovie),
NewMoviePrePrerollCompleteProc(MyMoviePrePrerollCompleteProc),
(void *)0L);
```

`PrePrerollMovie` operates either synchronously or asynchronously, depending on whether you specify a completion procedure when you call it.

If called asynchronously, it returns almost immediately, even if the movie contains streaming content. This allows your application to perform other tasks while awaiting the completion of the pre-prerolling. You need to call `MoviesTask` periodically to grant time for the task of pre-prerolling when using it asynchronously.

When the pre-prerolling is finished, `PrePrerollMovie` calls the completion procedure whose address you pass in the `NewMoviePrePrerollCompleteProc` parameter. In the simplest case, the completion procedure will want to preroll and start the movie.

If no completion procedure is specified, `PrePrerollMovie` returns when the pre-preroll process is complete.

Reacting to Changes in Movie Characteristics

One feature of streamed movies is that their characteristics may change dynamically during playback. For example, when you open a movie from a URL you may not know the actual height and width of the movie, its duration, how many streams it contains, whether it has a sound track, or whether it is an audio-only movie.

QuickTime will assign default values to these characteristics if they are unknown. QuickTime can notify your application when the movie characteristics become known or are changed.

In most cases, your application will want to adjust the size of the window or pane that contains the movie to reflect such changes. You make these adjustments by implementing a movie controller action filter proc.

To do this, you first need to indicate to the Movie Toolbox that you want to be informed of any changes. You do this by setting a movie playback hint:

```
SetMoviePlayHints(myMovie, hintsAllowDynamicResize,  
                  hintsAllowDynamicResize);
```

Changes in the movie size are announced to your filter proc by the `mcActionControllerSizeChanged` selector. Changes in other movie characteristics are announced through the `mcActionMovieEdited` selector.

Size Changes

Whenever the size of the movie changes, the associated movie controller sends the `mcActionControllerSizeChanged` action to your movie controller action filter procedure. You can intercept that action and respond to it as follows:

```
pascal Boolean MyMCActionFilterProc (MovieController theMC, short
theAction, void *theParams, long theRefCon)

{
    Movie    myMovie = MCGetMovie(theMC);

    switch (theAction) {
        // handle window resizing
        case mcActionControllerSizeChanged:
            MyResizeWindow(myMovie);
            break;

        default:
            break;
    }

    return(false);
}
```

Duration Changes

The duration of a streaming movie or a streaming track may not be initially known. A track or movie whose duration is not known is assigned an **indefinite duration**: `x7FFFFFFF`. If you do not treat this as a special case, a streaming movie will appear to your application as a very long movie indeed.

Once the pre-preroll process is complete, QuickTime should know the actual duration of the streams. If you set an action filter proc and call `SetMoviePlayHints`, your application will be called with the `mcActionMovieEdited` selector when QuickTime determines the actual duration.

If the movie contains live content, it may not have a specific duration. In this case, the duration remains indefinite: `x7FFFFFFF`. You might want to set a timeout in your code that detects the fact that QuickTime has not adjusted the duration from `x7FFFFFFF` after a few seconds of movie play. `QuickTimePlayer` uses such a timeout and displays a “Live Transmission” message where the slider would be for a movie with a known duration.

Sound and Video Changes

Whether a streaming movie has sound ('ears') or is sound-only (no video) may not be initially known or may change dynamically. If you set an action filter proc and call `SetMoviePlayHints`, your application will be called with the `mcActionMovieEdited` selector when the sound or video characteristics change.

Other Playback Considerations

Streaming movies only play back at a rate of 1. Other playback rates, such as playing backward, are not supported.

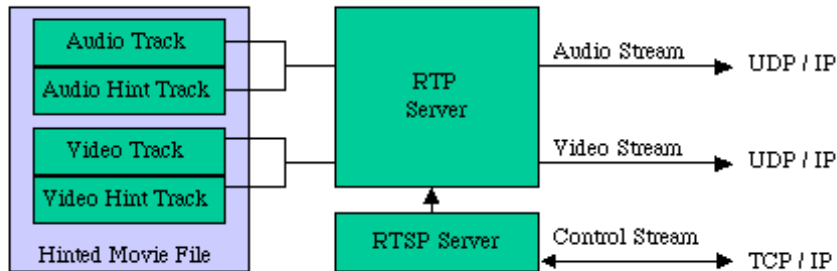
Serving Streaming Movies

To serve QuickTime movies over RTP, your movie server must be equipped with RTP server software that understands the QuickTime file format, including the structure of **hint tracks**, which are described in the section "Hint Track Structure". Your server also needs an RTSP controller application.

Your server does not need to have QuickTime software installed to serve streaming movies.

If your server is merely acting as a reflector for multicasts, no special software is required to reflect QuickTime movies. Forward the RTP streams on request in the usual way.

Your server uses the hint tracks in a streaming QuickTime movie to packetize the movie's media into RTP streams. Hint tracks are added to QuickTime movies to prepare them for streaming over RTP. One hint track is added to the movie for each track whose media will be streamed, as illustrated below.

Figure 26-8 Streaming a hinted movie

If your server is sending a unicast of a hinted movie, the QuickTime movie controller will provide the client with the ability to pause the movie and to skip backward and forward in movie time. This will be communicated to your server over RTSP.

The RTP server does not need to know about QuickTime media types or codecs. The hint tracks within the movie file provide the information needed to turn QuickTime media into RTP packets. Each hint track contains the data needed to build packet headers for a specific track's media. The hint track also supplies a pointer to the media data that goes into each packet.

The RTP server needs to be able to parse a QuickTime movie file sufficiently to find each hint track, then to find the track and sample data that the hint track points to. The hint track contains any precalculated values that may be needed, making it easier for the server to create the RTP packets.

Hint tracks offload a great deal of computation from the RTP server. Consequently, you may find that an RTP server is able to send data more efficiently if it is contained in a QuickTime movie, even if the RTP server already understands the media type, codec, and RTP packing being used.

For example, the H.263+ video codec is an IETF-standard RTP format which your server may already understand, but creating an H.263+ stream from video data requires complex calculations to properly determine packet boundaries. This information is precalculated and stored in the hint track when H.263+ video is contained in a QuickTime movie, allowing your server to packetize the data quickly and efficiently.

If you are writing QuickTime extensions to an RTP server application, you will need to read the *QuickTime File Format*, as well as the Hint Track Format section of this document.

Creating Streaming Movies

Streaming movies come in two forms: server movies and client movies.

You create a server movie that can be streamed over RTP by adding hint tracks. Hint tracks tell the server how to packetize the movie. You add hint tracks by exporting a movie to a hinted movie using QuickTime's standard movie export mechanism.

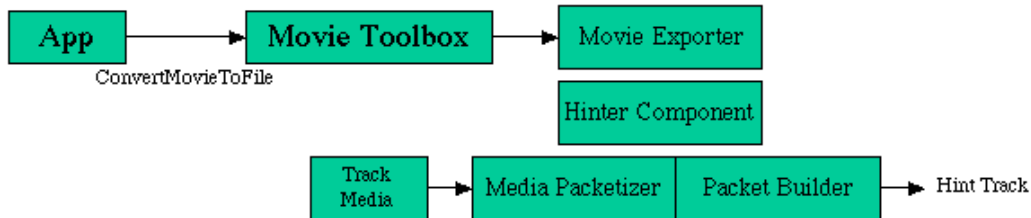
You create a client movie that includes streaming content by adding one or more streaming tracks. A streaming track tells the client where to get the streaming media. In its simplest form, a client movie consists of just a streaming track containing the URL of a movie on a server. A client movie can contain multiple streaming tracks.

A client movie can contain non-streaming tracks with local media content as well as streaming tracks. Streaming tracks can be composited with local tracks. For example, a streaming track could be used as the source for an effect track that is local to the client movie.

Server Movies

You create a streaming movie for an RTP server by adding hint tracks to an existing movie. You do this by calling `ConvertMovieToFile`, which invokes a movie exporter component. This displays a standard dialog box that lets the user specify "Export Movie to Hinted Movie", set the parameters for hinting the movie, select compressors, and specify a file name and directory for the hinted movie.

The hinting is performed by media packetizer components. QuickTime selects an appropriate media packetizer for each track and routes each packetizer's output through an Apple-provided packet builder to create a hint track. One hint track is created for each streamable track in the movie.

Figure 26-9 Exporting a hinted movie

Hint tracks are quite small compared with audio or video tracks. A movie that contains hint tracks can be played from a local disk or streamed over HTTP, like any other QuickTime movie. The hint tracks are only used when streaming the movie over RTP.

As long as your application supports movie exporter components, it should be able to create hinted movies. If you want to bypass the standard dialog for user input, selecting “Export Movie to Hinted Movie” programmatically, you will need to modify your code by adding the appropriate selectors.

A hinted movie does not need to be self-contained (flattened). It can reference sample media contained in other files. But observe these cautions:

- Use file names that are transportable between the system that the movies are created on and the RTP server.
- The relative path from the movie file to the data files must not change after the movie is saved.

The simplest way to ensure both of these is to use only lowercase letters in file names, without spaces, and to keep the media data files and the movie file in the same folder or directory.

Hint tracks should be created as the last step in making a streaming movie. Any editing of the movie that adds or deletes sample data, including the flattening of a movie with edit lists, invalidates the hint track. If your application edits a hinted movie in a way that invalidates the hint tracks, delete the hint tracks and re-export the movie.

Hint tracks are marked as inactive so they do not interfere with local playback. If you call `FlattenMovie` with the `flattenActiveTracksOnly` flag, the hint tracks are deleted from the flattened movie.

This release of QuickTime supports RTP streaming of video, audio, text (including HREF Tracks), and MIDI. If your movie contains other media types, or features that rely on track references, you cannot currently export the entire movie to a hinted movie. You can either stream such movies over HTTP, or you can put some of the tracks into a client movie and stream the rest, as described in the section Compositing Streaming and Non-Streaming Tracks.

Client Movies

You incorporate streaming content in a client movie by adding at least one streaming track. The simplest form of client movie has only one track: a streaming track with the URL of a movie on a server.

A streaming track has a media type of `kQTSSStreamMediaType` ('strm') and contains a single media sample: typically either an `RTSP://` URL of a streaming movie or the SDP text describing a multicast.

The following code example shows how to create a streaming track, insert a sample description, and add a URL media sample.

Listing 26-5 Creating a streaming ('strm') track with an RTSP:// URL

```
Handle dataRef;
long dataLength;
char url[] = "rtsp://myserver.bigcompany.com/mystreaming.mov";

dataRef = NewHandle(strlen(url) + 1);
BlockMoveData(url, *urlDataRef, strlen(url) + 1);

dataLength = GetHandleSize(dataRef);

newMedia = NewTrackMedia(newTrack, kQTSSStreamMediaType,
                        kQTSMediaTimeScale, handleDataRef, HandleDataHandlerSubType);

err = BeginMediaEdits(newMedia);

qtsDesc =
(QTSSampleDescriptionHandle)NewHandleClear(sizeof(QTSSampleDescription));

(**qtsDesc).descSize = sizeof(QTSSampleDescription);
(**qtsDesc).dataFormat = 'rtsp';
```

```
(**qtsDesc).dataRefIndex = 1;
(**qtsDesc).version = kQTSSampleDescriptionVersion;

duration = kQTSInfiniteDuration;

err = AddMediaSample(newMedia, dataRef, 0, dataLength, duration,
    (SampleDescriptionHandle)qtsDesc, 1, 0, nil);

err = EndMediaEdits(newMedia);
err = InsertMediaIntoTrack(newTrack, 0, 0, GetMediaDuration(newMedia),
    kQTSNormalForwardRate);
```

A movie that contains streaming tracks can be played from a local disk or served via HTTP. You can create a client movie with a streaming track that contains the URL of a server movie, then embed the client movie in a website or distribute it on a CD-ROM. When the movie is played, the user's computer will establish a network connection to the specified server for each streaming track.

A streaming track in a client movie can point to a server movie containing audio, video, text, and MIDI tracks. Any or all of the tracks in the server movie can appear as a single streaming track in the client movie.

To sum up:

- A client-side streaming movie contains at least one streaming 'strm' track.
- A streaming track contains a single media sample, typically an RTSP URL that points to streaming content. It can also contain SDP information for a multicast.
- The streaming content can be a live stream or a stored movie on a streaming server.
- The movie on the server can contain any number of tracks; multiple tracks in the server movie may be represented in the client movie as a single streaming track.

Compositing Streaming and Non-Streaming Tracks

A client movie can contain non-streaming tracks with locally-stored media, in addition to one or more streaming tracks. Use this technique to add a live stream to a locally-stored movie, or to distribute a movie on CD-ROM that references the latest version of some media stored on a server.

You can composite a streaming track with local tracks. A streaming track can be positioned in time and space like any other track. Bear in mind that the streaming track may contain more than one video, audio, text, or other streams, however.

If your application creates movies, you will want to provide options for creating a server movie and a client movie from the same data, as well as providing options for putting some tracks in the server movie and some in the client movie.

Ordinarily, you create a streaming movie by exporting a movie to a hinted movie, putting that hinted movie on an RTP server, and creating a client movie that contains the URL of the movie on the server.

Sometimes, however, you will want to export part of an existing movie to a hinted server movie while incorporating other parts in the client movie. You might do this to stream the audio and video parts of a movie over RTP, while streaming wired sprites or chapter lists over HTTP.

For example, to create a streaming movie with a chapter list, extract the text track containing the chapter list from the movie and export the remainder as a hinted movie. Create a client movie with just a streaming track containing the URL of the hinted movie. Add the extracted text track to the client movie and make it a chapter list by adding a track reference to the streaming track. The client movie now contains:

- a local text track containing the chapter list
- a streaming track that points to the rest of the movie
- a track reference of type 'chap' in the streaming track that references the chapter list track

The server movie contains the original movie, minus the chapter list, plus hint tracks.

You would use a similar technique to stream a movie with QuickTime video effects, storing the effects tracks in the client movie and applying the track references to the streaming tracks that act as sources to the effects.

To sum up:

- A client movie can contain tracks with locally-stored media as well as streaming tracks.
- Streaming tracks can be composited with local media tracks, bearing in mind that one streaming track can contain multiple streams. The compositing is done in the client movie.

- Tracks that cannot be streamed over RTP can be incorporated into client movies and streamed over HTTP.

Writing Media Packetizers and Reassemblers

Media packetizers are components that understand how to break QuickTime media into packets for RTP transmission. Packet reassemblers are components that understand how to put those packets back together to reconstitute the media. Packetizers and reassemblers are written in pairs; one to packetize for transmission, the other to reassemble during reception. Media packetizers are used during live transmissions and during the creation of hint tracks. Packet reassemblers are used when receiving streaming content.

RTP transmission is a best-effort delivery system. Over the Internet, it is inherently lossy. Packets may be dropped and not retransmitted; they may arrive out of order or after substantial delay. For streaming to work under these conditions, the sender and receiver must deal intelligently with unpredictable loss. In QuickTime, this intelligence resides primarily in the packetizers and reassemblers.

For example, a packetizer may break a video frame into a grid of independently coherent rectangles. If a packet is lost, only one rectangle needs to be discarded; the rest of the video frame can be reconstructed. The reassembler in this case might substitute the old rectangle for that part of the grid when a packet is lost, or slowly fade that rectangle to black, or use some other recovery scheme.

QuickTime includes a generic packetizer/reassembler pair that works with most QuickTime content. It also includes specialized packetizer/reassembler pairs optimized for specific media types and compression formats, such as Sorenson video, Qualcomm Purevoice audio, and QDesign music.

You can write your own packetizer/reassembler pairs and add them to QuickTime. You might want to do this if you have written your own codec, for example, or if you have a particularly clever scheme for packing media or recovering from packet loss.

Passing Non-Media Data

Media packetizers and packet reassemblers are generally concerned with media sample data, such as video frames or audio samples. QuickTime movies

commonly contain track-level information, such as a transformation matrix, or audio volume, that is applied to the sample data but is not embedded in it.

Packetizers and reassemblers can optionally support the passing of this type of data. If your packetizer supports passing of a particular type of information, such as audio volume or transformation matrix, it should indicate this in its public resource (as described in “The ‘pcki’ Public Resource”).

In the course of normal operations, a media packetizer may receive a `SetInfo` command that specifies this type of information. A packetizer that can send a transformation matrix, for example, could receive such a call whenever the matrix changes.

It is up to the packetizer to package this information in a way that will be recognized by the reassembler and to send it along.

When a reassembler receives this type of information, it signals the fact by calling `RTPRssmSendStreamHandlerChanged`. It may then receive a series of `GetInfo` calls to determine what information it has. It passes the information in response to these calls, and QuickTime does the rest.

Writing a Media Packetizer

A media packetizer takes media data and determines how to packetize it for transmission over RTP. A packetizer can be specific to a particular media type and compression format, such as a packetizer for Sorenson video, or it can be more generalized, such as a packetizer for any uncompressed audio, or even a packetizer for any QuickTime media.

QuickTime Streaming calls a media packetizer during the course of hinting a movie or during live transmission. The packetizer is passed sample data, which it breaks into packets and passes to a packet builder. The packet builder may then transmit the packets over RTP or use them to create a hint track. The media data is presented to the packetizer in the same format for a live transmission or for hinting, and the packetizer produces the same output in both cases.

The packetizer component must implement several functions, and must also provide a public component resource that describes the type of media, compression, and track characteristics that the packetizer supports. This resource also provides information on the packetizer’s relative speed and the format’s ability to handle loss.

This section describes

- the sequence of events your packetizer should expect
- the packetizer component type and subtype
- the 'pcki' public resource
- media preflight
- initialization
- set up and information functions
- sample processing functions
- flush and reset functions

Function definitions and additional discussion of all media packetizer functions can be found in the Media Packetizer Functions section.

Sequence of Events

When QuickTime is asked to provide a packetizer, it selects the packetizer based on the media type, data format, and other characteristics, such as whether a matrix transformation is in use. It selects the packetizer best able to handle the media and track characteristics by examining the packetizer's public resource.

Once a packetizer has been selected, it is opened. It is then asked to preflight the media, to verify that it can actually packetize the desired media data. If the packetizer indicates that it can handle the media, it is initialized. The packetizer then receives a series of set-up calls required to prepare it for operation. These calls deliver information such as the media time scale and the packet builder to use for output.

Once set-up is complete, the packetizer receives a series of calls with sample data. If the packetizer can process the data immediately, it does so; otherwise it returns a flag that indicates it is still processing the data. In the latter case, the packetizer's `Idle` function is called periodically until the packetizer has completed its processing. When it is ready, the packetizer creates packets by making calls to a packet builder component.

The packetizer is then called again with more sample data. This continues until all the media data has been packetized.

At any time in this process, the packetizer can be asked to return information, or to flush its input buffer, or to reset itself and prepare for a new sequence.

Packetizer Component Type and Subtype

Packetizers have a component type of `kRTPMediaPacketizerType` ('rtpm'). The subtype can be any four-character combination. Note, however, that all-lowercase types are reserved by Apple.

The 'pcki' Public Resource

A packetizer must provide a public resource of type 'pcki'. The public resource contains information about the capabilities of a given packetizer. This information lists the media types and compression formats the packetizer can work with. It also lists the track characteristics the packetizer can work with, such as layers or transformation matrices. In addition, it provides information about the packetizer's performance characteristics, such as its speed or ability to recover from packet loss.

QuickTime selects the packetizer best suited to the media, compression format, and track characteristics. If there are multiple packetizers that can work with a given track, performance is considered.

For example, one packetizer might be able to pack H.261-compressed video, but be unable to handle any transformation matrix other than identity. Another packetizer might also do H.261 and be able to handle any kind of transformation matrix. If a matrix was being used to scale and translate the media, QuickTime would select the more versatile packetizer. If there were no matrix applied to the track, QuickTime might select the faster packetizer for a live transmission, and the packetizer that dealt best with loss when creating a hint track.

The format of the public resource is defined in `QTStreamingComponents.r` as follows:

```
type 'pcki' {
    hex longintmediaType;
    hex longintdataFormat;
    hex longintvendor;
    hex longintcapabilityFlags;
    byte    canPackMatrixType;
    byte = 0;
    byte = 0;
    byte = 0;
    longint = $$CountOf(characteristicArray); /* Array size*/
    array characteristicArray {
```

```

        hex longinttag;
        hex longint value;
    };
    hex longintpayloadFlags;
    byte    payloadID;        /* if static payload */
    byte = 0;
    byte = 0;
    byte = 0;
    cstring;
};

```

The first three fields describe the media type, data format (compression format), and the manufacturer. The media type and data format are matched against the given track and media information when deciding whether to use a given packetizer. If your packetizer supports multiple compression formats, set `dataFormat` to 0. If it supports multiple media types, set `mediaType` to 0. The vendor field is for informational purposes only.

`capabilityFlags` can be a combination of:

```

enum {
    kMediaPacketizerCanPackEditRate = 1 << 0,
    kMediaPacketizerCanPackLayer = 1 << 1,
    kMediaPacketizerCanPackVolume = 1 << 2,
    kMediaPacketizerCanPackBalance = 1 << 3,
    kMediaPacketizerCanPackGraphicsMode = 1 << 4,
    kMediaPacketizerCanPackEmptyEdit = 1 << 5
};

```

These flags describe a packetizer's ability to deal with track characteristics. By indicating a capability, a packetizer says that it can transmit track information (such as a matrix) to the reassembler on the client side, so that it can be used in the client movie. The specific flags are these:

- `kMediaPacketizerCanPackEditRate` — **changing edit rates.** This is usually set for video, since each video frame typically has its own timestamp. Because of this, there isn't really a constant play rate. Sound, on the other hand, does have a play rate. If your packet format doesn't allow specification of non-standard rates, do not set this flag.
- `kMediaPacketizerCanPackLayer` — **only for visual formats.** Indicates that you can communicate the layering information for this stream. If your packetizer

supports this feature, the layering information from the QuickTime track structure may be passed to you in an `RTMPSetInfo` call.

- `kMediaPacketizerCanPackVolume` — **sound only**. Indicates you can communicate volume information from the QuickTime track structure.
- `kMediaPacketizerCanPackBalance` — **sound only**. Indicates you can communicate sound balance information from the QuickTime track structure.
- `kMediaPacketizerCanPackGraphicsMode` — **video only**. Indicates you can transmit the graphics mode and `opColor` for graphics modes other than `ditherCopy` and `srcCopy`.
- `kMediaPacketizerCanPackEmptyEdit` — **empty edits**. For sound formats, this is generally set, since the Sound Stream Handler will synthesize silence for any missing packets, which is exactly the intended effect for an empty edit in a sound track. For visual tracks this requires either sending down a duration, so that the Video Stream Handler knows when the frame is supposed to end, or some other method of describing empty edits in the packet format.
- `canPackMatrixType` — **applies only to visual tracks**. It is one of:

<code>canPackIdentityMatrixType</code>	<code>0x00</code>
<code>canPackTranslateMatrixType</code>	<code>0x01</code>
<code>canPackScaleMatrixType</code>	<code>0x02</code>
<code>canPackScaleTranslateMatrixType</code>	<code>0x03</code>
<code>canPackLinearMatrixType</code>	<code>0x04</code>
<code>canPackLinearTranslateMatrixType</code>	<code>0x05</code>
<code>canPackPerspectiveMatrixType</code>	<code>0x06</code>

Note that these are the same values as those defined in `ImageCompression.h` as return values for `GetMatrixType`. Indicates you can communicate the specified matrix type.

There are two performance characteristics currently defined:

```
#define kMediaPacketizerSpeedTag 'sped'/* 0-255, 255 is fastest */
#define kMediaPacketizerLossRecoveryTag 'loss'/* 0-255, 0 can't handle
any loss, 128 can handle 50% packet loss */
```

The speed tag is relative to other packetizers for the same media type and compression format. A value of 128 is a reasonable default.

The `payloadFlags` field is set to either `kRTPMPPayloadTypeDynamicFlag` for a dynamic payload type or `kRTPMPPayloadTypeStaticFlag` for a static payload type.

The payload ID field of the 'pcki' resource is set to the IETF-defined RTP payload value if a static payload type is used.

The C string contains the RTP payload type text if a dynamic payload type is used.

An code example of a resource for a packetizer that supports the 'OVAL' Video compression format follows:

```
resource 'thnr' (128) {
    {
        'pcki', 1, 0,
        'pcki', 128, cmpResourceNoFlags,
    };
};
resource 'pcki' (128) {
    'vide',          // media type
    'OVAL',          // data format type
    'ABCD',          // manufacturer type
    kMediaPacketizerCanPackEditRate,
    canPackIdentityMatrixType,
    {
        kMediaPacketizerSpeedTag, 128,
        kMediaPacketizerLossRecoveryTag, 50
    },
    kRTPMPPayloadTypeDynamicFlag,
    0,
    "OVAL-49545"
};
```

This packetizer is of medium speed, can handle some loss, can pack any arbitrary play rate, but can't pack any non-identity matrix. Its RTP payload type is a dynamic identifier, identified by the string "OVAL-49545".

Media Preflight

Once QuickTime has selected and opened a packetizer component, it will call the packetizer's `RTPMPPreflightMedia` function to verify that the packetizer can

handle the specific media and sample description. A packetizer can reject media as a result of this call, even if the media fits the profile described in the packetizer's public resource.

For example, the 'pcki' resource for the Qualcomm PureVoice audio packetizer indicates support for audio media and PureVoice compression, but the packetizer only supports 8 kHz sample rates over RTP. The

`RTPMPPreflightMedia` call to this packetizer returns `noErr` for 8 kHz audio, and an error for any other sample rate.

The media preflight function is defined as follows:

```
pascal ComponentResult RTPMPPreflightMedia(RTPMediaPacketizer rtpm,
                                           OSType inMediaType,
                                           SampleDescriptionHandle inSampleDescription);
```

This call is made to make sure your packetizer can handle a given media type and sample description. Return `noErr` if you can support it. Return `qtsUnsupportedFeatureErr` if you cannot.

Initialization

If your packetizer returns `noErr` during the media preflight, it will be initialized before it is asked to handle any data. The initialization function is defined as follows:

```
pascal ComponentResult RTPMPInitialize(RTPMediaPacketizer rtpm,
                                       SInt32 inFlags);

inFlags can be 0 or:
enum {
    kRTPMPRealtimeModeFlag= 0x00000001
};
```

The `kRTPMPRealtimeModeFlag` flag indicates that your packetizer is being used for live transmission, rather than for hinting. You might use this information if your packetizer has a tradeoff between speed and fidelity.

Set Up and Information Functions

Once your packetizer is initialized, it will receive several calls to set the information it needs prior to packetizing data, such as the media timescale. You

may also be called with requests to return the settings you have been given. The main set up functions are listed below:

```
RTPMPSetTimeScale  
RTPMPGetTimeScale  
RTPMPSetPacketBuilder  
RTPMPGetPacketBuilder  
RTPMPSetMediaType  
RTPMPGetMediaType  
RTPMPSetMaxPacketSize  
RTPMPGetMaxPacketSize  
RTPMPSetMaxPacketDuration  
RTPMPGetMaxPacketDuration
```

The **Set** functions listed above are used to set the time scale, packet builder (which receives your output), media type, maximum packet size, and maximum packet duration. These functions will be called before you are asked to begin packetizing data, and will not be called after you have begun packetizing data.

The **Get** functions listed above can be called at any time. When your packetizer is called with one of these **Get** functions, return the data that was passed to you in the corresponding **Set** function.

```
RTPMPSetTimeBase  
RTPMPGetTimeBase
```

The time base functions may be called for live transmission. The **Set** function sets the QuickTime time base that is in use. Your packetizer can query this time base to find out what time it is in the live stream. Your packetizer should not rely on receiving this call.

RTPMPHasCharacteristic may be called to determine whether your media packetizer has a particular characteristic, such as whether it supports a user settings dialog. Return `qtsBadSelectorErr` if your packetizer does not have the given characteristic.

```
ComponentResult RTPMPHasCharacteristic (  
    RTPMediaPacketizer rtpm,  
    OSType inSelector,  
    Boolean *outHasIt);
```

`rtpm` The component instance of your media packetizer

`inSelector` **A selector for the characteristic. Defined selectors are:**

- `kRTPMPNoSampleDataRequiredCharacteristic` - **if set, the media packetizer does not require the actual sample data to perform packetization. The caller can pass in `nil` data pointers instead of pointers to the actual media data (see `RTPMPSetSampleData`).**
- `kRTPMPHasUserSettingsDialogCharacteristic` - **if set, the media packetizer supports the calls `RTPMPDoUserDialog`, `RTPMPGetSettingsIntoAtomContainerAtAtom`, and `RTPMPSetSettingsFromAtomContainerAtAtom`.**
- `kRTPMPPrefersReliableTransportCharacteristic` - **if set, the packetizer would prefer its data to be sent reliably (such as Text or Music tracks, etc.)**
- `kRTPMPRequiresOutOfBandDimensionsCharacteristic` - **if set, the original visual dimensions of the media data cannot be determined simply by looking at the media packets, and must be transmitted via some other method.**

`outHasIt` **Return a boolean which is `true` if your media packetizer has this characteristic, `false` otherwise.**

If your packetizer supports a user dialog or additional settings, you may also receive dialog or settings calls as part of the set up process:

```
RTPMPDoUserDialog
RTPMPSetSettingsFromAtomContainerAtAtom
RTPMPGetSettingsIntoAtomContainerAtAtom
RTPMPGetSettingsAsText
```

The `RTPMPDoUserDialog` function will invoke your packetizer's modal dialog to obtain user settings.

The `RTPMPGetSettingsIntoAtomContainerAtAtom` function expects you to return those settings into an atom container at the specified offset.

The `RTPMPSetSettingsFromAtomContainerAtAtom` function is used to set the user settings programmatically, bypassing the user dialog.

The `RTPMPGetSettingsAsText` function expects you to return your user settings as text.

In addition to the set up calls listed above, you may receive RTPMPSetInfo function calls. These are used to send a variety of information to your packetizer, as indicated by the selector. If you don't support a given selector, return qtsBadSelectorErr.

RTPMPSetInfo selectors can be:

```
kQTSSourceTrackIDInfo      ('otid'), /* UInt32* */
kQTSSourceLayerInfo        ('olr'), /* UInt16* */
kQTSSourceLanguageInfo     ('oling'), /* UInt16* */
kQTSSourceTrackFlagsInfo   ('otfl'), /* Sint32* */
kQTSSourceDimensionsInfo   ('odim'), /* QTSDimensionParams* */
kQTSSourceVolumesInfo      ('ovol'), /* QTSVolumesParams* */
kQTSSourceMatrixInfo       ('omat'), /* MatrixRecord* */
kQTSSourceClipRectInfo     ('oclp'), /* Rect* */
kQTSSourceGraphicsModeInfo ('ogrm'), /* QTSGraphicsModeParams* */
kQTSSourceScaleInfo        ('oscl'), /* Point* */
kQTSSourceBoundingRectInfo ('orct'), /* Rect* */
kQTSSourceUserDataInfo     ('oudt'), /* UserData */
kQTSSourceInputMapInfo     ('oimp'), /* QTAtomContainer */
```

Your packetizer can be called with these selectors even if it indicated that it couldn't handle the given characteristic in its 'pcki' resource. In that case, return qtsBadSelectorErr.

Your packetizer may receive the related RTPMPGetInfo function at any time:

```
pascal ComponentResult RTPMPGetInfo(RTPMediaPacketizer rtpm,
                                     OSType inSelector, void *ioParams);
```

Your packetizer is expected to return the requested information in ioParams. The type of ioParams is dependent on inSelector. If you don't support the selector, return qtsBadSelectorErr. The selectors can be any of the selectors for RTPMPSetInfo or any of the additional selectors listed below:

```
/* info selectors - get only */
kRTPMPPayloadTypeInfo      ('rtpp'), /* RTPMPPayloadTypeParams* */
kRTPMPRTPTTimeScaleInfo    ('rtpt'), /* TimeScale* */
kRTPMPRequiredSampleDescriptionInfo ('sdsc'), /*
SampleDescriptionHandle* */
kRTPMPMinPayloadSize       ('mins'), /* UInt32*, doesn't include rtp
header; default 0 */
```



```
kRTPMPMinPacketDuration      ('mind'), /* UInt32* in milliseconds;
default is no min */
kRTPMPSuggestedRepeatPktCountInfo ('srpc'), /* UInt32* */
kRTPMPSuggestedRepeatPktSpacingInfo ('srps'), /* UInt32* in milliseconds
*/
kRTPMPMaxPartialSampleSizeInfo ('mpss'), /* UInt32* in bytes */
kRTPMPPreferredBufferDelayInfo ('prbd') /* UInt32* in milliseconds */
```

The `kRTPMPPayloadTypeInfo` selector requires you to fill out an `RTPMPPayloadTypeParams` structure.

```
/* flags for RTPMPPayloadTypeParams */
enum {
    kRTPMPPayloadTypeStaticFlag= 0x00000001,
    kRTPMPPayloadTypeDynamicFlag = 0x00000002
};

struct RTPMPPayloadTypeParams {
    UInt32  flags;
    UInt32  payloadNumber;
    short   nameLength;
    /* in: size of payloadName buffer (counting null terminator)
    /* -- this will be reset to needed length and paramErr returned if too
    small */
    char *                                     payloadName; /* caller must provide
buffer */
};

typedef struct RTPMPPayloadTypeParams RTPMPPayloadTypeParams;
```

If you have a dynamic RTP payload type, you need to copy the payload type string to the buffer pointed to by `payloadName` (a null-terminated string). When `RTPMPGetInfo` is called, `RTPMPPayloadTypeParams.nameLength` will be the available size of the input buffer (specified by `payloadName`). If this size is too small for the payload identifier, set `nameLength` to the size of the buffer you need (including the null terminator) and return `paramErr`. This will cause QuickTime to reallocate the buffer and call your packetizer again.

`kRTPMPRTPTimescaleInfo`

return the RTP timescale used by this packetizer if the time scale must be a specific value; otherwise, return an error for this selector.

`kRTPMPRequiredSampleDescriptionInfo` **return a handle to a sample description specifying that only data with the given sample description is supported; if no such sample description exists, return an error for this selector.**

`kRTPMPMinPayloadSize`

Return the minimum payload size, in bytes, of packets allowed by this packetizer (UInt32).

`kRTPMPMinPacketDuration`

Return the minimum packet duration allowed by this packetizer (UInt32, in milliseconds).

`kRTPMPSuggestedRepeatPktCountInfo`

Return the suggested number of repeat packets to send for this media (UInt32). This will typically be 0 for audio or video, nonzero for text or MIDI.

`kRTPMPSuggestedRepeatedPktSpacingInfo`

Return the suggested temporal distance between repeat packets (UInt32, in milliseconds).

`kRTPMPMaxPartialSampleSizeInfo`

Return the size of the largest partial sample your packetizer can accept (UInt 32 in bytes). If your packetizer returns true for `HasCharacteristic (kRTPMPPartialSamplesRequiredCharacteristic)`, this call will be made to ask what the largest size (in bytes) is that the packetizer can handle receiving in a `SetSampleData` call.

Very few packetizers will need to set this -- it is intended only for media formats where a single media sample can be huge. MPEG is an example -- one media sample covers the entire span of the MPEG movie.

`kRTPMPPreferredBufferDelayInfo`

Return the preferred buffer delay for your packetizer (UInt32, in milliseconds). Most packetizers will not need to set this. This would only be set if the packet duration was sufficiently long that you needed a longer buffer delay to work.

Sample Processing Functions

Once setup is complete, QuickTime will begin calling your packetizer's `RTPMPSampleData` function. This is where most of a typical packetizer's work is done. Your packetizer is presented with a block of media sample data, you break it into packets according to your own algorithm, and you pass the results to the selected packet builder. If your packetizer works synchronously, you packetize the data and return 0 in `outFlags`. If you need more sample data to complete a packet, return `kRTPMPWantsMoreDataFlag`.

If your packetizer works asynchronously, you set `outFlags` to `kRTPMPStillProcessingData` and your packetizer continues its work in the `RTPMPIdle` function, which will be called periodically. `RTPMPIdle` should also set `outFlags` to `kRTPMPStillProcessingData` if it still has work to do. It sets `outFlags` to 0 if all the sample data has been processed.

If your packetizer works synchronously, `RTPMPIdle` always sets `outFlags` to 0.

Note that the caller owns the `RTPMPSampleDataParams` struct. Your media packetizer must copy any fields of the struct it wants to keep. The caller will maintain only the media data in the struct until you call the release proc.

Your media packetizer must call the release proc when done with the media data.

Do the processing work in the `RTPMPSampleData` function if it does not take up too much CPU time; otherwise, do it in your idle function.

```
pascal ComponentResult RTPMPSampleData(RTPMediaPacketizer rtpm,
                                       const RTPMPSampleDataParams *inSampleData, SInt32
                                       *outFlags);

/* flags for RTPMPSampleDataParams */
enum {
    kRTPMPSyncSampleFlag= 0x00000001
};

struct RTPMPSampleDataParams {
    UInt32          version;
    UInt32          timeStamp;
    UInt32          duration;           /* 0 if not specified */
    UInt32          playOffset;        /* 0 if not specified */
    Fixed           playRate;
    SInt32          flags;
    UInt32          sampleDescSeed;
```

```

    Handle                sampleDescription;
    RTPMPSampleRef         sampleRef;
    UInt32                 dataLength;
    const UInt8 *          data;
    RTPMPDataReleaseUPP    releaseProc;
    void *                 refCon;
};

```

```
typedef struct RTPMPSampleDataParams RTPMPSampleDataParams;
```

version	Version of the data structure. Currently always 0.
timeStamp	RTP time stamp for the presentation of the sample data. This time stamp has already been adjusted by edits, edit rates, etc.
duration	Duration (in RTP time scale) of the sample.
playOffset	Offset within the media sample itself. This is only used for media formats where a single media sample can span across multiple time units. QuickTime Music is an example of this, where a single sample spans the entire track. For most video and audio formats, this will be 0.
playRate	1.0 (0x00010000) is normal. Higher numbers indicate faster play rates. Note that timeStamp is already adjusted by the rate. This field is generally of interest only to audio packetizers.
flags	kRTPMPSyncSampleFlag is set if the sample is a sync sample (key frame).
sampleDescSeed	If the sample description changes, this number will change.
sampleDescription	The sample description for the given media sample
sampleRef	Private field.
dataLength	Size of media data
data	Pointer to media data
releaseProc	If set, you need to call it when you are finished with the sample data.
refCon	Pass to releaseProc

`outFlags` **Set to `kRTPMPStillProcessingData` if you are not done with the sample. This will cause your idle routine to be called.**

If you can't do all the processing of the sample data in response to the `RTPMPSetSampleData` function, sets `outFlags` to `kRTPMPStillProcessingData` and do the work in your `RTPMPIdle` function:

```
// do work here if you need to - give up time periodically
// if you're doing time consuming operations
pascal ComponentResult RTPMPIdle(RTPMediaPacketizer rtpm,
                                SInt32 inFlags, SInt32 *outFlags);
```

This call is made periodically if you sets `outFlags` to `kRTPMPStillProcessingData` in your `RTPMPSetSampleData` routine. If you need more time, sets `outFlags` to `kRTPMPStillProcessingData` in `RTPMPIdle` as well. This will cause `RTPMPIdle` to be called again. When you are finished packetizing the sample data you were passed in the last `RTPMPSetSampleData` call, sets `outFlags` to 0 .

If you do work in the `RTPMPIdle` function, the idle function needs to call the release proc when you are done with the sample data.

Calling the Packet Builder

The actual work your packetizer does in the `RTPMPSetSampleData` or `RTPMPIdle` function consists of creating packets. This is done by making calls to a packet builder component. The packet builder functions are defined in `QTStreamingComponents.h` and described in “Packet Builder Functions”.

The first thing you will need to do is to begin a new packet group. To start a new group of packets, call `RTPPBBeginPacketGroup`. Packets in a single group have the same RTP time stamp (and go into the same hint sample, if the data is being used to create a hint track).

Next, you will want to add one or more packets to the group. Call `RTPPBBeginPacket` to begin a single network packet.

Call `RTPPBAddPacketLiteralData` to add literal data to the packet, such as header information.

Call `RTPPBAddPacketSampleData` to add a sample reference to the packet, such as the sample data specified in `inSampleDataParams` in a call to your packetizer's `RTPMPSetSampleData` function.

You may want to packetize the same data repeatedly. There are utility functions to make this easier. Begin by passing in `RTPPacketRepeatedDataRef` to `RTPPBAddPacketLiteralData` or `RTPPBAddPacketSampleData`, depending on whether you want to repeat literal data or sample data. The packet builder will store a copy of the data and return a data reference. You then call `RTPPBAddPacketRepeatedData` with the data reference whenever you want to add copies of the data. Call `RTPPBReleaseRepeatedData` when you are done with the data. The packet builder will maintain a copy of the data until you release it.

End a packet by calling `RTPPBEndPacket`.

End the group of packets by calling `RTPPBEndPacketGroup`.

Note that it is legal to have more than one packet group open at a time, or to have more than one packet open at a time. Packet groups are sent in the order they are closed (`RTPPBEndPacketGroup`). Packets within a group are also sent in the order they are closed (`RTPPBEndPacket`).

Flush and Reset Routines

Your packetizer may be called with commands to flush its input buffer or reset for a new session at any time. This normally happens at the end of a sequence or when a transmission is interrupted.

The flush command is given when your packetizer needs to finish any pending work.

```
pascal ComponentResult RTPMPFlush(RTPMediaPacketizer rtpm,
                                   SInt32 inFlags, SInt32 *outFlags);
```

If your packetizer has not yet written all the data from the last `RTPMPSetSampleData` call to the selected packet builder, do it now. `inFlags` is currently always 0. Write any pending data, flush your input buffer, and set `outFlags` to 0.

The reset command is given when your packetizer needs to stop packetizing, reset its state, and prepare for new orders.

```
pascal ComponentResult RTPMPReset(RTPMediaPacketizer rtpm,
                                   SInt32 inFlags);
```

`inFlags` is currently always 0.

Flush your input buffer. Do not send any buffered data, because in all probability there is no connection for you to send on. Reset your packetizer's state; it should be the same as if it had just been opened and initialized.

Writing a Packet Reassembler

A packet reassembler extracts meaningful chunks of data, such as video frames, from streams of RTP packets. A reassembler can be specific to a particular media type and compression format, such as a reassembler for Sorenson video, or it can be more generalized, such as a reassembler for any uncompressed audio, or even a reassembler for any QuickTime media.

Streaming media over RTP generally involves some packet loss. It is the responsibility of the reassembler to perform any loss recovery that goes beyond discarding data chunks that contain lost packets.

QuickTime includes a base reassembler which performs most of the routine work of packet reassembly. Your packet reassembler sets flags that control the behavior of the base reassembler. Your reassembler can also implement several functions to override the base reassembler, essentially taking over from it at almost any point.

The reassembler component must implement several functions, and must also provide a public component resource that describes the type of media, compression, and track characteristics that the reassembler supports. This resource also provides information on the reassembler's relative speed and the format's ability to handle loss.

This section describes

- the reassembler component type and subtype
- the 'rsmi' public resource
- the base reassembler
- opening your component
- initialization
- set up and information functions
- handling packets yourself (overriding the base reassembler)
- handling chunks yourself (overriding the base reassembler)
- reset and clear cache functions

■ closing your component

Function definitions and additional discussion of all packet reassembler functions can be found in the **Packet Reassembler Functions** section. The **QuickTime Streaming Reference** section also defines the `RTPRssmPacket` and `RTPRssmInitParams` data structures.

Reassembler Component Type and Subtype

Packetizers have a component type of `kRTPReassemblerType ('rtpr')`. The subtype can be any four-character combination. Note, however, that all-lowercase types are reserved by Apple.

The 'rsmi' Public Resource

A reassembler must provide a public resource of type 'rsmi'. The public resource contains information about the capabilities of a given reassembler. This information lists the RTP payload types the reassembler can work with. In addition, it provides information about the reassembler's performance characteristics, specifically its speed and ability to recover from lost packets.

If more than one reassembler is available for a given RTP payload type, QuickTime will choose the one with the best performance characteristics, such as speed or ability to deal with packet loss.

The format of the public resource is defined in `QTStreamingComponents.r` as follows:

```
type 'rsmi' {
    array infoArray {
        align long;
        longint = $$CountOf(characteristicArray); /* Array size */
        array characteristicArray {
            hex longinttag;
            hex longint value;
        };
    };

    hex longintpayloadFlags;
    /* kRTPPayloadTypeStaticFlag or kRTPPayloadTypeDynamicFlag */
    byte payloadID; /* if static payload */
    byte = 0;
```



```
        byte = 0;
        byte = 0;
        cstring;                                /* if dynamic payload */
    };

#define kRTPPayloadSpeedTag 'sped'/* 0-255, 255 is fastest */

#define kRTPPayloadLossRecoveryTag 'loss'
    /* 0-255, 0 can't handle any loss, 128 can handle 50% packet loss */

#define kRTPPayloadTypeStaticFlag 0x00000001

#define kRTPPayloadTypeDynamicFlag 0x00000002
```

The payload flags field is set to `kRTPMPPayloadTypeDynamicFlag` if the reassembler handles a dynamic payload type, or `kRTPMPPayloadTypeStaticFlag` if it handles a static type.

The payload ID field of the 'rsmi' resource is set to the IETF-defined RTP payload value if a static payload type is used.

The C string contains the RTP payload type text for dynamic types.

A declaration in a .r file might look like this:

```
resource kRTPReassemblerInfoResType (128) {
    {
        {
            kRTPPayloadSpeedTag, 128,
            kRTPPayloadLossRecoveryTag, 50
        },
        kRTPPayloadTypeDynamicFlag, 0, "x-oval"
    }
};
```

This resource indicates that the reassembler is of average speed and that it can handle 50% packet loss while still providing some meaningful data. It handles the dynamic RTP payload type identified by "x-oval".

The speed tag is relative to other reassemblers of the same type. 128 is a reasonable default.

The Base Reassembler

Your packet reassembler relies on a base reassembler component to do most of the routine reassembly work. Your reassembler can modify some of the base reassembler's default behaviors simply by setting flags. With a few exceptions, your reassembler implements reassembler functions only when it needs to override the normal behavior of the base reassembler.

The base reassembler's main function is to assemble incoming packets into "chunks", then pass the chunks to a stream handler. A chunk is the amount of data useful to a particular stream handler, such as a video frame or twenty milliseconds of audio.

Your reassembler must specify the type of stream handler needed, but your reassembler doesn't work with stream handlers directly. The base reassembler does this for you.

The default behavior of the base reassembler is as follows:

- The base reassembler fills out an `RTPRssmPacket` struct for each packet it receives.
- The base reassembler puts the packets into a packet list, which corresponds to a data chunk.
- By default, the packet list contains all the packets with the same RTP transmission time, which is also the sample time. The base reassembler will start a new packet list when it receives a packet with a different RTP transmission time or with the RTP marker bit set.

Your reassembler can change this default by telling the base reassembler that every packet is a chunk.

- Once a packet list is complete, the base reassembler creates a chunk from it.
- By default, the chunk is made by concatenating the payload of each packet in the packet list.
- By default, the payload is assumed to be the entire contents of the packet following the RTP header and the payload header.

The base reassembler assumes a zero-length payload header (no payload header) unless your reassembler sets a nonzero payload header length as the default.

- By default, the base reassembler discards chunks with missing packets.

The base reassembler can be set to inform your reassembler that a chunk has missing packets.

By implementing certain functions, you can cause the base reassembler to call your reassembler at one of several points to override or modify the default behavior:

Implement this function

RTPRssmAdjustPacketParams

RTPRssmSendPacketList

RTPRssmComputeChunkSize

RTPRssmCopyDataToChunk

Your reassembler will be called...

When each packet is received, after the base reassembler fills out the `RTPRssmPacket` struct

When the packet list is complete

When the chunk size is calculated

When the data is copied into the chunk record

This is discussed further in the sections "Handling Packets Yourself" and "Handling Chunks Yourself".

Opening Your Reassembler

QuickTime may open your packet reassembler to check its version or to get information. It may then close your reassembler without ever initializing it or using it to process packets.

Your reassembler component must be able to perform the standard component functions, such as `_Version`, and its specific `RTPRssmGetInfo` function, without being initialized.

When opened, your packet reassembler must open a base reassembler component. Your reassembler should delegate any functions it does not implement. It should also support the `_Target` call.

A typical reassembler's `_Open` and `_Target` functions might look like this:

```
pascal ComponentResult RTP0valRssm_Open(RTP0valRssmGlobalsPtr globals,
                                         ComponentInstance self)
{
    #pragma unused (inGlobals)
    ComponentResult err;

    globals =
    (RTP0valRssmGlobalsPtr)NewPtrClear(sizeof(RTP0valRssmGlobalsRecord));
    if ( (err = MemError()) != noErr )
```

```
        goto exit;

    SetComponentInstanceStorage(self, (Handle)globals);
    globals->self = self;

    err = OpenADefaultComponent(kRTPReassemblerType,
                                &globals->delegateComponent);

    if ( err == noErr )
        err = RTPOval_Target(globals, self);

exit:

    return err;
}

pascal ComponentResult RTPOval _Target(RTPOvalRssmGlobalsPtr inGlobals,
                                        ComponentInstance inParentComponent)
{
    inGlobals->parent = inParentComponent;
    return ComponentSetTarget(inGlobals->delegateComponent,
                              inParentComponent);
}
```

Initialization

Your reassembler must implement the `RTPRssmInitialize` function. When initialized, your reassembler is passed an `RTPRssmInitParams` struct containing three initialization parameters:

```
struct RTPRssmInitParams {
    UInt32      reserved;
    UInt8       payloadType;
    UInt8       pad[3];
    TimeBase    timeBase;
    TimeScale    controlTimeScale;
};
typedef struct RTPRssmInitParams RTPRssmInitParams;
```

<code>payloadType</code>	1 byte identifier for the payload type. Same number as sent in the RTP header. Useful if your rssm handles more than one format.
<code>timeBase</code>	The timebase for the presentation
<code>controlTimescale</code>	The timescale used for actions such as start, stop, etc. This is independent of the timescale used for the media. For example, the control timescale could be 600 while the media timescale could be 90000

During initialization, your reassembler should open a stream handler of the appropriate type. For example, if your reassembler works with H.261 packets, it should open a video stream handler. Stream handler types are specified in the same manner as track types (Video is `videoMediaType`, and so on). There are currently stream handlers for audio, video, text, and MIDI.

Your reassembler opens a stream handler by calling `RTPRssmNewStreamHandler`. You must specify the stream handler type when you open it. If your reassembler handles multiple media types, it can open a stream handler later, after it learns what kind of media is in the stream.

You should initialize the sample description of the stream handler when you open it, if possible. It can be set or changed later if necessary. The stream handler will be unable to process any data until its sample description is set.

Your reassembler should also initialize the timescale of the stream handler at this time. If your reassembler needs to get the timescale from the stream, it can monitor incoming packets and set the timescale when it is known. (No chunks will be sent to the stream handler until its timescale is set).

During initialization, your reassembler should call `RTPRssmSetCapabilities` with any initial flags to control the base reassembler's default behaviors, such as `kRTPRssmEverySampleAChunkFlag` or `kRTPRssmTrackLostPacketsFlag`.

You should also call `SetPayloadHeaderLength` at this time if you know the payload header length for your packets.

A typical reassembler initialization function might look like this:

```
EXTERN_API( ComponentResult )
RTPRssmOVAL_Initialize(
    RTPHOVALGlobalsPtr inGlobals,
    RTPRssmInitParams *inInitParams )
{
    #pragma unused(inInitParams)
```

```
ComponentResult err = noErr;
ImageDescriptionHandle imageDesc;
SInt32 flags = 0;

inGlobals->fTimeScale = kOVALRTPTimeScale;

flags = kRTPRssmQueueAndUseMarkerBitFlag + kRTPRssmTrackLostPacketsFlag;
err = RTPRssmSetCapabilities(inGlobals->delegateComponent, flags, -1L );
if (err == noErr) {
    imageDesc = __GetMyImageDesc( inGlobals );

    if( imageDesc )
    {
        err = RTPRssmNewStreamHandler(inGlobals->delegateComponent,
            VideoMediaType, ( SampleDescriptionHandle ) imageDesc,
            inGlobals->fTimeScale, NULL);
    }

    else
        err = memFullErr;
}
return err;
}
```

Set Up and Information Functions

If you do not open a stream handler and set its timescale during initialization, you must implement the `RTPRssmGetTimeScaleFromPacket` function. If you have not set the stream handler's timescale, the base reassembler will call your reassembler's `GetTimeScaleFromPacket` function with every incoming packet until a stream handler is open and its time scale is set.

If you cannot determine the timescale based on the contents of the packet, return `qtsUnknownValueErr`, or a zero timescale.

```
EXTERN_API( ComponentResult )
RTPRssmGetTimeScaleFromPacket(
    RTPReassembler rtpr,
    QTSSStreamBuffer *inStreamBuffer,
    TimeScale * outTimeScale);
```

The `RTPRssmGetInfo`, `RTPRssmSetInfo`, and `RTPRssmHasCharacteristic` functions can be called at any time, even prior to initialization. `RTPRssmHasCharacteristic` is called to determine what features your reassembler supports. `RTPRssmGetInfo` is used to get information from your reassembler. `RTPRssmSetInfo` could be used to send information to your reassembler, but there are currently no selectors that do this.

These functions are commonly used if your reassembler supports passing of non-media data, such as transformation matrices. The functions are typically called after your reassembler reports a change in a non-media parameter by calling `RTPRssmSendStreamHandlerChange`, as described in “Passing Non-Media Data.” The selectors used for passing non-media data are the same for `RTPRssmHasCharacteristic` and `RTPRssmGetInfo`. These selectors begin with `kQTSSource` and are defined in `QTStreaming.h`.

Delegate any selectors you do not support or do not understand to the base reassembler for all of these functions.

```
EXTERN_API( ComponentResult )
RTPRssmSetInfo      (RTPReassembler rtpr,
                    OSType  inSelector,
                    void *  ioParams) ;

EXTERN_API( ComponentResult )
RTPRssmGetInfo      (RTPReassembler rtpr,
                    OSType  inSelector,
                    void *  ioParams) ;

EXTERN_API( ComponentResult )
RTPRssmHasCharacteristic(RTPReassembler rtpr,
                        OSType  inCharacteristic,
                        Boolean * outHasIt) ;
```

Handling Packets Yourself

Your packet reassembler can be called when each packet is received. If you have not yet opened a stream handler and set its timescale, you will receive a `RTPRssmGetTimeScaleFromPacket` call for each received packet.

Once the stream handler’s timescale is set, the default behavior for the base reassembler is to fill out an `RTPRssmPacket` struct for each incoming packet, then to add each packet to the current packet list. The base reassembler starts a new

packet list when a packet's RTP marker bit is set, or the RTP timestamp changes, or if your reassembler has set the `kRTPRssmEveryPacketAChunkFlag` in `RTPRssmSetCapabilities`.

If you want to fill-out or modify the `RTPRssmPacket` struct for each packet yourself, because you use variable payload header lengths, for example, implement the `RTPRssmAdjustPacketParams` function.

```
EXTERN_API( ComponentResult )
RTPRssmAdjustPacketParams(
    RTPReassembler rtpr,
    RTPRssmPacket *inPacket,
    SInt32          inFlags) ;
```

The `inPacket` parameter points to the `RTPRssmPacket` struct for the current packet. This structure has already been filled-in by the base reassembler.

```
struct RTPRssmPacket {
    struct RTPRssmPacket *next;
    struct RTPRssmPacket *prev;
    QTSSstreamBuffer *streamBuffer;
    Boolean paramsFilledIn;
    UInt8  pad[1];
    UInt16 sequenceNum;
    UInt32 transportHeaderLength; /* filled in by base*/
    UInt32 payloadHeaderLength; /* derived adjusts this */
    UInt32 dataLength;
    SHServerEditParameters serverEditParams;
    TimeValue64 timestamp;
    /* lower 32 bits is original rtp timestamp*/
    SInt32 chunkFlags; /* these are or'd together*/
    SInt32 packetFlags;
};
typedef struct RTPRssmPacket RTPRssmPacket;
```

`prev` The previous packet in the list; NULL if this is the first packet
`streamBuffer` The stream buffer containing the packet data

Important: Do not modify the data in the stream buffer. For example, if you have to flip bytes for endian differences between network byte order and the native byte order, copy the resulting data. Do not flip the bytes in place. Other components might have references to the same data.

`paramsFilledIn`

True if the rest of the fields in this structure have been filled-in

`sequenceNum`

The sequence number associated with the packet. Sequence numbers are unsigned 16 bit numbers and wrap around. e.g. 65535 is followed by 0, 1, 2, etc. Sequence numbers start at arbitrary numbers.

`transportHeaderLength`

The length of the rtp header. The payload specific part of the packet begins immediately after the transport header. Important - do not assume that the rtp header length is 12.

`payloadHeaderLength`

The length of the payload header. The payload header (if any) starts immediately after the rtp header. This number gets filled in by the base rsm by using the number passed into `RTPRsmSetPayloadHeaderLength`. It can also be modified by the derived rsm.

`dataLength`

The length of the payload data. This is usually the length of the packet minus transport header length minus payload header length. The base rsm will set the default value to that. The your rsm can modify the number.

`serverEditParams`

`timeStamp`

The 64 bit timestamp associated with the packet, in the stream timebase, not the movie timebase. The timestamp sent in the rtp header is 32 bits and wraps around. The 64 bit timestamp in this structure accounts for the wraparound. The lower 32 bits are exactly the timestamp sent in the rtp header. The upper 32 bits are the number of wraparounds.

`chunkFlags`

Flags that should be set in the chunk that is sent to the stream handler. The base rsm calculates the value that is set in the `SHChunkRecord` structure by ORing all the chunk flags in the

RTPRssmPacket list. Your rssm should fill in these flags as appropriate.

Flags you could set:

kSHChunkFlagSyncSample - set this flag if the chunk is a sync sample

kSHChunkFlagDataLoss - set this flag if the chunk has data loss in it - e.g. there was data loss in the video frame and the rssm did partial recovery

packetFlags The base rssm fills-in this value. Defined flags are:

kRTPRssmPacketHasMarkerBitSet = 0x00000001, The base rssm fills in this value from the rtp header.

kRTPRssmPacketHasServerEditFlag = 0x00010000, The derived rssm should set this flag to 1 if this packet has a server edit. You should also fill in the serverEditParams

Handling Chunks Yourself

The base reassembler automatically builds chunks from packets and sends them to the stream handler you have opened. If any packets are missing from the chunk, the base reassembler discards the entire chunk.

You can take over from the base reassembler at several points in the chunk-building process by implementing the appropriate function. Do this if your reassembler performs loss recovery, or if the base reassembler's default behavior needs to be modified to correctly build chunks for the stream handler.

Your reassembler is responsible for creating and sending the chunk, beginning at whatever point you take over, and continuing until the chunk is sent or discarded.

The process of creating a chunk begins when the base reassembler has a complete packet list. To take over at this point, implement `RTPRssmSendPacketList`. You might want to do this if you were altering the packet list for loss recovery.

If you implement `RTPRssmSendPacketList`, you are responsible for deleting the packet list when you through with it by calling `RTPRssmReleasePacketList`.

Next, the base reassembler calculates the chunk size. It does this by summing the payload size for each packet in the list. The payload size is calculated by

subtracting the RTP header and the payload header from the packet size. To take over at this point, implement `RTPRssmComputeChunkSize`. You might want to do this if you need to add data to the chunk that isn't in the packets, or to use a different method for calculating the payload size.

If you implement `RTPRssmComputeChunkSize`, you will need to allocate a chunk of the appropriate size. To do this, call `RTPRssmGetChunkAndIncrRefCount`. This will allocate the chunk and set the number of references to it to "1". QuickTime will maintain the chunk until the number of references to it is "0". The reference count is automatically decremented when you send the chunk. You can increment the counter by calling `RTPRssmIncrChunkRefCount` if you want QuickTime to preserve the chunk for later use (by substituting for a lost chunk, for example).

Once the chunk size has been calculated and the chunk has been allocated, the base reassembler moves the data from the packets into the chunk. The data moved will be a simple concatenation of the packet payloads. The payload size and offset within each packet are calculated in the same manner as used for calculating chunk size. To take over at this point, implement `RTPRssmCopyDataToChunk`. You would need to do this to modify the bytes at packet boundaries for an H.261 packet reassembler, for example.

Bear in mind that if you implement `RTPRssmSendPacketList`, you are responsible for the steps performed in `RTPRssmComputeChunkSize` and `RTPRssmCopyDataToChunk`. Similarly, if you implement `RTPRssmComputeChunkSize`, you must perform the steps for `RTPRssmCopyDataToChunk`. Only the first of these functions that you implement will be called. You take over the chunk-building process from there.

If you implement any of the three functions just discussed, you are responsible for sending the chunk by calling `RTPRssmSendChunkAndDecrCount`.

Other things your reassembler might do at this point are to

- Tell the stream handler that the sample description has changed (`RTPRssmSetSampleDescription`)
- Tell the stream handler that some non-media data has changed, such as a transformation matrix or sound volume (`RTPRssmSendStreamHandlerChanged`). You may receive a series of `GetInfo` calls to determine what has changed.
- Tell the stream handler not to send the chunk. The reference counter will be decremented; if it becomes zero, the chunk is deallocated (`RTPRssmSendLostChunk`).

- Tell the base reassembler to keep a copy of this chunk for your later use (`RTPRssmIncrChunkRefCount`). Be sure to increment the ref count before you send the chunk. You must eventually call `RTPRssmDecrChunkRefCount` for every call to `RTPRssmIncrChunkRefCount` or you will create a memory leak.

Reset and Clear Cache Functions

Your reassembler can be called at any time with the `RTPRssmReset` function, which you must implement. Reset all your variables, release any chunks being held for you, and prepare for a new run of data. At the end of an `RTPRssmReset` function, your state should be identical to its state after the first `RTPRssmInitialize` call.

You can instruct the base reassembler to release any packets in the packet list it is currently building by calling `RTPRssmClearCachedPackets`. You might do this if you are handling packets yourself and you determine that the list the base reassembler is building should be discarded.

QuickTime Streaming Reference

This section contains the definitions of the media packetizer, packet builder, and packet reassembler functions, as well as a list of common streaming error codes.

Media Packetizer Functions

The following functions can be implemented by media packetizer components. Most of these functions must be implemented in your component, but some are optional, as noted in the discussion section of the optional functions.

RTPMPInitialize

The `RTPMPInitialize` function is called to initialize your media packetizer component. `RTPMPInitialize` will be called after your component is opened, and after the `RTPMPPreflightMedia` call, but before it is sent any data to packetize.

```
ComponentResult RTPMPInitialize (
    RTPMediaPacketizer rtpm,
    SInt32 inFlags);
```

rtpm The component instance of your media packetizer

inFlags A signed 32-bit integer containing the flags for your packetizer at start-up. Defined flags are:
kRTPMPRealtimeModeFlag - your packetizer is being called in a real-time situation (for example, live broadcasting)

DISCUSSION

The calling component must call `RTPMPInitialize` before any `RTPMPSet` calls.

RTPMPPreflightMedia

This function is called to determine whether your packetizer can work with a given media type and sample description. Return `noErr` if you can packetize this type of data. Return `qtsUnsupportedFeatureErr` if you cannot.

```
ComponentResult RTPMPPreflightMedia (
    RTPMediaPacketizer rtpm,
    OSType inMediaType,
    SampleDescriptionHandle inSampleDescription);
```

rtpm The component instance of your media packetizer

inMediaType The media type (such as 'vide')

inSampleDescription The sample description

function result Return `noErr` if you can packetize this type of data. Return `qtsUnsupportedFeatureErr` if you cannot.

DISCUSSION

This function will be called before you are asked to packetize any data. This function must be implemented by your packetizer.

RTPMPIOIdle

This function is called periodically in the application program's event loop to allocate time to your media packetizer. If your packetizer operates synchronously, doing all of its work in the `RTPMPIOSetSampleData` function, the `RTPMPIOIdle` function does nothing. If your packetizer operates asynchronously, it sets `outFlags` to `kRTPMPIOStillProcessingData` in `RTPMPIOSetSampleData` and processes data during `RTPMPIOIdle`. Set `outFlags` to 0 if you have processed all the data passed to you in the last `RTPMPIOSetSampleData`. Set `outFlags` to `kRTPMPIOStillProcessingData` if you need the application to call `RTPMPIOIdle` again.

```
ComponentResult RTPMPIOIdle (
    RTPMediaPacketizer rtpm,
    SInt32 inFlags,
    SInt32 *outFlags);
```

<code>rtpm</code>	The component instance of your media packetizer
<code>inFlags</code>	There are currently no defined flags
<code>outFlags</code>	On return, contains a pointer to a signed 32-bit integer that holds any flags from your packetizer. Defined flags are: <code>kRTPMPIOStillProcessingData</code> Set this flag if there is still data in your queue to be processed. The calling application will call <code>RTPMPIOIdle</code> repeatedly until this flag is not returned. If all data in your queue has been processed, set <code>outFlags</code> to.

DISCUSSION

The packetizer can use this time to process the data in its buffer. Data is placed in the buffer by the `RTPMPIOSetSampleData` function. If the data has not all been processed, the `RTPMPIOSetSampleData` function sets `outFlags` to `kRTPMPIOStillProcessingData`; this causes the calling application to call `RTPMPIOIdle`. The `RTPMPIOIdle` function can also set `outFlags` to `kRTPMPIOStillProcessingData` if it needs more time. The calling application will call `RTPMPIOIdle` repeatedly until you set `outFlags` to 0.

Your packetizer may make calls to the packet builder in response to this call.

RTPMPSetSampleData

This function is called to provide sample data to your media packetizer component. Packetize the data according to your own algorithm and pass it to the selected packet builder. If you can process all the data immediately, set `outFlags` to 0. If you need more data to complete a packet, set `outFlags` to `kRTPMPWantsMoreDataFlag`. If you need more time, set `outFlags` to `kRTPMPStillProcessingData` and continue processing in `RTPMPIdle`.

```
ComponentResult RTPMPSetSampleData (
    RTPMediaPacketizer rtpm,
    const RTPMPSampleDataParams *inSampleData,
    SInt32 *outFlags);
```

<code>rtpm</code>	The component instance of your media packetizer
<code>inSampleData</code>	A pointer to the sample data
<code>outFlags</code>	If you have processed all the data, set <code>outFlags</code> to 0. If you need more data to complete a packet, set <code>outFlags</code> to <code>kRTPMPWantsMoreDataFlag</code> . If you need more time, set <code>outFlags</code> to <code>kRTPMPStillProcessingData</code> and continue processing in <code>RTPMPIdle</code> .

function result `qtsBadDataErr` - the packetizer cannot process the given data

```
/* flags for RTPMPSampleDataParams*/
enum {
    kRTPMPSyncSampleFlag= 0x00000001
};
struct RTPMPSampleDataParams {
    UInt32                version;
    UInt32                timeStamp;
    UInt32                duration;           /* 0 if not
specified*/
    UInt32                playOffset;        /* 0 if not
specified*/
    Fixed                playRate;
    SInt32                flags;
    UInt32                sampleDescSeed;
    Handle                sampleDescription;
    RTPMPSampleRef        sampleRef;
```

```

        UInt32                dataLength;
        const UInt8 *         data;
        RTPMPDataReleaseUPP releaseProc;
        void *                 refCon;
    };

typedef struct RTPMPSampleDataParams RTPMPSampleDataParams;

```

version	Version of the data structure. Currently always 0.
timeStamp	RTP time stamp for the presentation of the sample data. This time stamp has already been adjusted by edits, edit rates, etc.
duration	Duration (in RTP time scale) of the sample.
playOffset	Offset within the media sample itself. This is only used for media formats where a single media sample can span across multiple time units. QuickTime Music is an example of this, where a single sample spans the entire track. For most video and audio formats, this will be 0.
playRate	1.0 (0x00010000) is normal. Higher numbers indicate faster play rates. Note that timeStamp is already adjusted by the rate. This field is generally of interest only to audio packetizers.
flags	kRTPMPSyncSampleFlag is set if the sample is a sync sample (key frame).
sampleDescSeed	If the sample description changes, this number will change.
sampleDescription	The sample description for the given media sample
sampleRef	Private field.
dataLength	Size of media data
data	Pointer to media data
releaseProc	If set, you need to call it when you are finished with the sample data.
refCon	Pass to releaseProc

DISCUSSION

This routine is called to pass media data to a media packetizer. Note that the caller owns the `RTPMPSampleDataParams` struct pointed to by `inSampleData`. Your media packetizer must copy any fields of the struct that it wants to keep. The caller will maintain only the media data pointed to by the `data` field in the struct until you call the release proc.

Your media packetizer must call the release proc when done with the media data.

Calling `RTPMPSetSampleData` adds data cumulatively to any previous calls to `RTPMPSetSampleData`. The data can contain any number of samples (1 or more), or a partial sample.

Set `outFlags` to 0 if your packetizer is able to finish packetizing the sample data, or `kRTPMPWantsMoreDataFlag` if it needs more data to build a packet. Set `outFlags` to `kRTPMPStillProcessingData` if you need more time. This will cause a series of calls to `RTPMPIdle`, which grants time to your packetizer in order to process the data passed in `RTPMPSetSampleData`.

Your packetizer may make calls to the packet builder in response to this call.

RTPMPReset

Stop packetizing your current input, set your packetizer's state to idle, and flush your input buffer. Reset your packetizer's state as if it had just been opened and initialized.

```
ComponentResult RTPMPReset (
    RTPMediaPacketizer rtpm,
    SInt32 inFlags);
```

`rtpm` The component instance of your media packetizer

`inFlags` A signed 32-bit integer containing any flags you being passed to the media packetizer. There are currently no defined flags.

DISCUSSION

Stop the media packetizer and flush its input buffer. This function is normally called to stop transmitting immediately, when skipping forward or backward in the stream, for example, or if the network data connection is interrupted.

RTPMPSetInfo

This function is called to set any one of several parameters for a media packetizer.

```
ComponentResult RTPMPSetInfo (
    RTPMediaPacketizer rtpm,
    OSType inSelector,
    const void *ioParams);
```

rtpm The component instance of the media packetizer

inSelector A selector for the type of information to set. RTPMPSetInfo selectors can be:

```
kQTSSourceTrackIDInfo ('otid') /* UInt32* */
kQTSSourceLayerInfo ('olr') /* UInt16* */
kQTSSourceLanguageInfo ('olng') /* UInt16* */
kQTSSourceTrackFlagsInfo ('otfl') /* SInt32* */
kQTSSourceDimensionsInfo ('odim') /* QTSDimensionParams* */
kQTSSourceVolumesInfo ('ovol') /* QTSVolumesParams* */
kQTSSourceMatrixInfo ('omat') /* MatrixRecord* */
kQTSSourceClipRectInfo ('oclp') /* Rect* */
kQTSSourceGraphicsModeInfo ('ogrm') /*
    QTSGraphicsModeParams* */
kQTSSourceScaleInfo ('oscl') /* Point* */
kQTSSourceBoundingRectInfo ('orct') /* Rect* */
kQTSSourceUserDataInfo ('oudt') /* UserData */
kQTSSourceInputMapInfo ('oimp') /* QTAAtomContainer */
```

Your packetizer can be called with these selectors even if it indicated that it couldn't handle the given characteristic in its 'pcki' resource.

`ioParams` A pointer to a data structure of the appropriate type for the information being passed.

function result Return `qtsBadSelectorErr` if you do not support the selector.

DISCUSSION

This function is used to pass track-level information about the media track to be packetized, such as track ID, layer, and transformation matrix. Return `qtsBadSelectorErr` unless your packetizer is able to transmit this kind of data to your reassembler for use in the client movie.

RTPMPPGetInfo

This function is used to get information of various types from your media packetizer. You will need to return information as a data structure of the appropriate type for the requested information.

```
ComponentResult RTPMPPGetInfo (
    RTPMediaPacketizer rtpm,
    OSType inSelector,
    void *ioParams);
```

`rtpm` The component instance of the media packetizer you want information from.

`inSelector` The selector for the type information requested. Any of the selectors used with `RTPMPPSetInfo` can be used with `RTPMPPGetInfo`. The following additional selectors are defined for `RTPMPPGetInfo` only:

```
/* info selectors - get only */
kRTPMPPayloadTypeInfo ('rtpp'), /* RTPMPPayloadTypeParams* */
kRTPMPRTTimeScaleInfo ('rtpt'), /* TimeScale* */
kRTPMPRequiredSampleDescriptionInfo ('sdsc'),
    /* SampleDescriptionHandle* */
kRTPMPMinPayloadSize ('mins'),
    /* UInt32*, doesn't include rtp header; default is 0 */
kRTPMPMinPacketDuration ('mind'),
    /* UInt32* in milliseconds; default is no min */
```

```

kRTPMPSuggestedRepeatPktCountInfo    ('srpc'), /* UInt32* */
kRTPMPSuggestedRepeatPktSpacingInfo   ('srps'),
    /* UInt32* in milliseconds */
kRTPMPMaxPartialSampleSizeInfo       ('mpss'), /* UInt32* in bytes */
kRTPMPPreferredBufferDelayInfo        ('prbd') /* UInt32* in milliseconds */

```

The `kRTPMPPayloadTypeInfo` selector requires you to fill out an `RTPMPPayloadTypeParams` structure, which describes the payload being used.

```

/* flags for RTPMPPayloadTypeParams */
enum {
    kRTPMPPayloadTypeStaticFlag= 0x00000001,
    kRTPMPPayloadTypeDynamicFlag = 0x00000002
};

struct RTPMPPayloadTypeParams {
    UInt32                flags;
    UInt32                payloadNumber;
    short                 nameLength;          /* in: size of
payloadName buffer (counting null terminator) -- this will be reset to
needed length and paramErr returned if too small */
    char *                payloadName;        /* caller must provide
buffer */
};

typedef struct RTPMPPayloadTypeParams RTPMPPayloadTypeParams;

```

If the `flags` field of the structure is `kRTPMPPayloadTypeStaticFlag`, then the `payloadNumber` field contains the RTP payload number; if the `flags` field of the structure is `kRTPMPPayloadTypeDynamicFlag`, then the `payloadName` field contains the name of the dynamic payload encoding being used. The caller must allocate this `payloadName` field. You need to copy the payload type string to the buffer pointed to by `payloadName` (a null-terminated string). When `RTPMPGetInfo` is called, `RTPMPPayloadTypeParams.nameLength` will be the available size of the input buffer (specified by `payloadName`). If this size is too small for the payload identifier, set `nameLength` to the size of the buffer you need (including the null terminator) and return `paramErr`. This will cause QuickTime to reallocate the buffer and call your packetizer again.

`kRTPMPRTPTimescaleInfo`

return the RTP timescale used by this packetizer if the time scale must be a specific value; otherwise, returns an error for this selector.

`kRTPMPRequiredSampleDescriptionInfo`

return a handle to a sample description specifying that only data with the given sample description is supported; if no such sample description exists, returns an error for this selector.

`kRTPMPMinPayloadSize`

Return the minimum payload size of packets allowed by this packetizer (UInt32).

`kRTPMPMinPacketDuration`

Return the minimum packet duration allowed by this packetizer (UInt32, in milliseconds).

`kRTPMPSuggestedRepeatPktCountInfo`

Return the suggested number of repeat packets to send for this media (UInt32). This will typically be 0 for audio or video, nonzero for text or MIDI.

`kRTPMPSuggestedRepeatedPktSpacingInfo`

Return the suggested temporal distance between repeat packets (UInt32, in milliseconds).

`kRTPMPMaxPartialSampleSizeInfo`

Return the size of the largest partial sample your packetizer can accept (UInt 32 in bytes).

`kRTPMPPreferredBufferDelayInfo`

Return the preferred buffer delay for your packetizer (UInt32, in milliseconds).

`ioParams`

A pointer to a data structure of the appropriate type to hold the information requested.

function result

`qtsBadSelectorErr` -- the selector is not supported, or an appropriate default value should be used.

DISCUSSION

This function can be called at any time. Return `qtsBadSelectorErr` for any selector you do not support. Otherwise, return the requested data in `ioParams`.

RTPMPSetTimeScale

Sets the time scale your media packetizer will use. The time scale is the number of time units that pass in one second (when the media is playing at a rate of 1). The timescale must be set before any calls are made to `RTPMPSetSampleData`.

```
ComponentResult RTPMPSetTimeScale (
    RTPMediaPacketizer rtpm,
    TimeScale inTimeScale);
```

`rtpm` The component instance of your media packetizer

`inTimeScale` The timescale to use

DISCUSSION

This timescale gives meaning to the times used in `RTPMPSetSampleData`.

RTPMPGetTimeScale

Return a pointer to the timescale in use by your media packetizer. The timescale indicates the number of time units that pass in one second (when the media is playing at a rate of 1).

```
ComponentResult RTPMPGetTimeScale (
    RTPMediaPacketizer rtpm,
    TimeScale *outTimeScale);
```

`rtpm` The component instance of your media packetizer

`outTimeScale` Return the timescale in use by your packetizer

DISCUSSION

The timescale is set by in `RTPMPSetTimeScale`.

RTPMPSetTimeBase

This call may be made during set up for a live transmission. It tells your packetizer what time base is in use by the calling application.

```
ComponentResult RTPMPSetTimeBase (
    RTPMediaPacketizer rtpm,
    TimeBase inTimeBase);
```

`rtpm` **Component instance of your packetizer**

`inTimeBase` **The time base in use for this stream**

DISCUSSION

You can query this time base to find out the current time in the stream. Your packetizer should not rely on receiving this call.

RTPMPGetTimeBase

Return the time base passed to you in `RTPMPSetTimeBase`.

```
ComponentResult RTPMPGetTimeBase (
    RTPMediaPacketizer rtpm,
    TimeBase *outTimeBase);
```

`rtpm` **The component instance of your media packetizer**

`outTimeBase` **Return the time base passed to you in `RTPMPSetTimeBase`.**

RTPMPHasCharacteristic

This call is made to determine whether your media packetizer has a particular characteristic, such as whether it supports a user settings dialog.

```
ComponentResult RTPMPSHasCharacteristic (
    RTPMediaPacketizer rtpm,
    OSType inSelector,
    Boolean *outHasIt);
```

<code>rtpm</code>	The component instance of your media packetizer
<code>inSelector</code>	<p>A selector for the characteristic. Defined selectors are:</p> <p><code>kRTPMPNoSampleDataRequiredCharacteristic</code> - if set, the media packetizer does not require the actual sample data to perform packetization. The caller can pass in <code>nil</code> data pointers instead of pointers to the actual media data (see <code>RTPMPSetSampleData</code>).</p> <p><code>kRTPMPHasUserSettingsDialogCharacteristic</code> - if set, the media packetizer supports the calls <code>RTPMPDoUserDialog</code>, <code>RTPMPGetSettingsIntoAtomContainerAtAtom</code>, and <code>RTPMPSetSettingsFromAtomContainerAtAtom</code>.</p> <p><code>kRTPMPPrefersReliableTransportCharacteristic</code> - if set, the packetizer would prefer its data to be sent reliably (such as Text or Music tracks, etc.)</p> <p><code>kRTPMPRequiresOutOfBandDimensionsCharacteristic</code> - if set, the original visual dimensions of the media data cannot be determined simply by looking at the media packets, and must be transmitted via some other method.</p>
<code>outHasIt</code>	Return a boolean which is <code>true</code> if your media packetizer has this characteristic, <code>false</code> otherwise.
<i>function result</i>	Return <code>qtsBadSelectorErr</code> if your packetizer does not have the given characteristic.

RTPMPSetPacketBuilder

Selects which packet builder your media packetizer will use. A media packetizer always sends its output to a packet builder. The specified packet builder may assemble actual RTP packets, or it may use information about the packet to build a hint track.


```
ComponentResult RTPMPSetPacketBuilder (  
    RTPMediaPacketizer rtpm,  
    ComponentInstance inPacketBuilder);
```

rtpm The component instance of your media packetizer

inPacketBuilder The component instance of the packet builder component to use

DISCUSSION

A media packetizer always sends its output to a packet builder. The packet builder must be set using this call prior to any calls to `RTPMPSetSampleData`. This function can be used to dynamically change the packet builder your media packetizer uses at any time.

RTPMPGetPacketBuilder

Return the component instance of the packet builder component being used by your media packetizer.

```
ComponentResult RTPMPGetPacketBuilder (  
    RTPMediaPacketizer rtpm,  
    ComponentInstance *outPacketBuilder);
```

rtpm The component instance of your media packetizer

outPacketBuilder Return the component instance of the packet builder component in use by this media packetizer

DISCUSSION

The packet builder in use is set by `RTPMPSetPacketBuilder`.

RTPMPSetMediaType

Sets the type of media that your media packetizer will process (for example, `VideoMediaType` or `SoundMediaType`). The media type will be set prior to any calls to `RTPMPSetSampleData`. It will not change after such calls.

```
ComponentResult RTPMPSetMediaType (
    RTPMediaPacketizer rtpm,
    OSType inMediaType);
```

`rtpm` The component instance of your media packetizer

`inMediaType` The media type, such as `VideoMediaType`.

function result `qtsBadDataErr` -- the given media type is not supported.

DISCUSSION

This function will be called before any calls to `RTPMPSetSampleData` are made, and will not be called after a call to `RTPMPSetSampleData`.

RTPMPGetMediaType

Return the media type passed in by `RTPMPSetMediaType` (such as `VideoMediaType` or `SoundMediaType`).

```
ComponentResult RTPMPGetMediaType (
    RTPMediaPacketizer rtpm,
    OSType *outMediaType);
```

`rtpm` The component instance of your media packetizer

`outMediaType` Return the media type

DISCUSSION

The media type is set by a call to `RTPMPSetMediaType`.

RTPMPSetMaxPacketSize

Sets the maximum packet size, in bytes, for packets created by your media packetizer.

```
ComponentResult RTPMPSetMaxPacketSize (
    RTPMediaPacketizer rtpm,
    UInt32 inMaxPacketSize);
```

rtpm The component instance of your media packetizer

inMaxPacketSize An unsigned 32-bit integer specifying the maximum size, in bytes, of packets your packetizer may create.

function result qtsBadDataErr -- **inMaxPacketSize** is smaller than the the minimum payload size for this media packetizer (the value you return in the RTPMPGetInfo call with the kRTPMPMinPayloadSize selector).

DISCUSSION

The media packetizer must not create packets larger than this value. The limit applies only to the payload data. The maximum packet size will not change during a presentation. Streaming will be most efficient if this value is set to the largest packet size that can traverse the network without being split.

RTPMPSetMaxPacketSize will not be called after calling RTPMPSetSampleData. If RTPMPSetMaxPacketSize is not called, use a default value.

RTPMPGetMaxPacketSize

Return the maximum packet size, in bytes, that your packetizer is set to create.

```
ComponentResult RTPMPGetMaxPacketSize (
    RTPMediaPacketizer rtpm,
    UInt32 *outMaxPacketSize);
```

rtpm The component instance of your media packetizer

`outMaxPacketSize`

Return a pointer to a 32-bit integer containing the maximum packet size, in bytes, that the packetizer is set to create.

DISCUSSION

The maximum packet size is set by a call to `RTPMPSetMaxPacketSize`. If no value has been set, return the default value that your packetizer will use.

RTPMPSetMaxPacketDuration

Sets the maximum packet duration, in milliseconds, that the media packetizer is to use.

```
ComponentResult RTPMPSetMaxPacketDuration (
    RTPMediaPacketizer rtpm,
    UInt32 inMaxPacketDuration);
```

`rtpm` The component instance of your media packetizer

`inMaxPacketDuration` An unsigned 32-bit integer containing the maximum packet duration in milliseconds.

function result `qtsBadDataErr` -- The specified value was smaller than the minimum packet duration this media packetizer can support (the value returned from the `RTPMPGetInfo` call with the `kRTPMPMinPacketDuration` selector.)

DISCUSSION

The maximum packet duration will not be changed during a presentation.

`RTPMPSetMaxPacketDuration` will not be called after any calls to `RTPMPSetSampleData`. If `RTPMPSetMaxPacketDuration` is not called, use a default value.

RTPMPGetMaxPacketDuration

Return the maximum packet duration, in milliseconds, currently set for this packetizer.

```
ComponentResult RTPMPGetMaxPacketDuration (
    RTPMediaPacketizer rtpm,
    UInt32 *outMaxPacketDuration);
```

`rtpm` The component instance of your media packetizer

`outMaxPacketDuration` Return a pointer to a 32-bit integer containing the maximum packet duration, in milliseconds, that the packetizer is set to use.

DISCUSSION

The maximum packet duration is set by a call to `RTPMPSetMaxPacketDuration`. If a duration has not been set, return the default value that your packetizer will use.

RTPMPDoUserDialog

Obtain media-specific settings from the user through a dialog box.

```
ComponentResult RTPMPDoUserDialog (
    RTPMediaPacketizer rtpm,
    ModalFilterUPP inFilterUPP,
    Boolean *canceled);
```

`rtpm` The component instance of your media packetizer

`inFilterUPP` A modal dialog filter which may be used in a call to `ModalDialog()`

`canceled` Return a boolean which is `true` if the user pressed the cancel button in the dialog box. If this parameter is returned `true`, the settings prior to calling this function should be retained.

DISCUSSION

This is an optional function.

If a media packetizer supports a user dialog (see `RTPMPHasCharacteristic`), it can put up a dialog allowing the user to enter media-specific settings. The settings can be requested by the calling application by calling

`RTPMPGetSettingsIntoAtomContainerAtAtom`, and can be restored or set directly from an application by calling `RTPMPSetSettingsFromAtomContainerAtAtom`.

This function may be called at any time.

RTPMPSetSettingsFromAtomContainerAtAtom

Sets the media-specific settings of a media packetizer. The data is passed in an atom inside an `AtomContainer`.

```
ComponentResult RTPMPSetSettingsFromAtomContainerAtAtom (
    RTPMediaPacketizer rtpm,
    QTAtomContainer inContainer,
    QTAtom inParentAtom);
```

`rtpm` The component instance of your media packetizer

`inContainer` The `AtomContainer` that holds the settings atom

`inParentAtom` The atom that holds the settings

DISCUSSION

This function should only be called if your media packetizer supports packetizer-specific settings. To determine if a media packetizer supports this function, the application will call `RTPMPHasCharacteristic`.

You can be asked to return the the packetizer-specific settings through the call `RTPMPGetSettingsIntoAtomContainerAtAtom`. You can be asked to bring up a dialog which allows the user to change the settings through the `RTPMPDoUserDialog` call.

RTPMPGetSettingsIntoAtomContainerAtAtom

Return the media-specific setting of a media packetizer. The caller must allocate a QT AtomContainer to hold the returned settings.

```
ComponentResult RTPMPGetSettingsIntoAtomContainerAtAtom (
    RTPMediaPacketizer rtpm,
    QTAtomContainer inOutContainer,
    QTAtom inParentAtom);
```

`rtpm` The component instance of your media packetizer

`inOutContainer` The AtomContainer that holds the settings atom.

`inParentAtom` The atom that will hold the settings.

DISCUSSION

This function should only be called if the media packetizer supports packetizer-specific settings. To determine if your media packetizer supports this function, the application will call `RTPMPHasCharacteristic`.

The packetizer-specific settings can be set by through the `RTPMPSetSettingsFromAtomContainerAtAtom` call. You may be asked to bring up a dialog which allows the user to change the settings through the `RTPMPDoUserDialog` call.

RTPMPGetSettingsAsText

Return the media-specific settings of a media packetizer as text in a format presentable to the user.

```
ComponentResult RTPMPGetSettingsAsText (
    RTPMediaPacketizer rtpm,
    Handle *text);
```

`rtpm` The component instance of your media packetizer

`text` Return a handle to a copy of your user settings in text format. The text is formatted as simple array of characters. There is no size byte or null termination. Allocate the handle to fit the text precisely.

DISCUSSION

This function should only be called if the media packetizer supports packetizer-specific settings. To determine if your media packetizer supports this function, the application will call `RTPMPHasCharacteristic`.

The packetizer-specific settings can be set by the `RTPMPSetSettingsFromAtomContainerAtAtom` call. You may be asked to bring up a dialog which allows the user to change the settings through the `RTPMPDoUserDialog` call.

The `RTPMPGetSettingsAsText` function expects you to return your user settings as text.

Packet Builder Functions

The following functions may be called by media packetizers during execution of an `RTPMPSetSampleData` or `RTPMPIdle` routine. Use them to tell a packet builder component to begin a packet group, to begin a packet, to insert data into a packet, to end a packet, and to end a packet group. These functions are only called by media packetizer components.

RTPPBBeginPacketGroup

Tell a packet builder to create a new packet group. The function returns a data reference which will be used when creating new packets or inserting data into packets.

```
ComponentResult RTPPBBeginPacketGroup (
    RTPPacketBuilder rtpb,
    SInt32 inFlags,
    UInt32 inTimeStamp,
    RTPPacketGroupRef *outPacketGroup);
```


<code>rtpb</code>	The component instance of the packet builder component
<code>inFlags</code>	A signed 32-bit integer containing any flags you are passing. There are currently no defined flags.
<code>inTimeStamp</code>	A unsigned 32-bit integer containing the time stamp for this packet group
<code>outPacketGroup</code>	On return, contains a pointer to a reference to the packet group. Use this data reference when creating a new packet or inserting data into a packet that belongs to this group

DISCUSSION

A media packetizer creates a packet group using this function. The data reference returned by this function is then used to create a series of packets that belong to this group. The data reference is also required when inserting data into packets. When the packet group is complete, call `RTPPEndPacketGroup`.

RTPPBeginPacket

Tell a packet builder to create a new packet . The function returns a data reference which will be used when creating inserting data into the packet.

```
ComponentResult RTPPBeginPacket (
    RTPPacketBuilder rtpb,
    SInt32 inFlags,
    RTPPacketGroupRef inPacketGroup,
    UInt32 inPacketMediaDataLength,
    RTPPacketRef *outPacket);
```

<code>rtpb</code>	The component instance of the packet builder component
<code>inFlags</code>	A signed 32-bit integer containing any flags you are passing. There are currently no defined flags.
<code>inPacketGroup</code>	The packet group containing the new packet. This is normally a reference returned by <code>RTPPBeginPacketGroup</code> .

`inPacketMediaDataLength`

An unsigned 32-bit integer specifying the maximum length of data that will be inserted into this packet. This includes the data for all subsequent `RTPPBAddPacketLiteralData`, `RTPPBAddPacketSampleData`, and `RTPPBAddPacketRepeatedData` calls until the packet is closed. `inPacketMediaDataLength` may be larger, but must not be smaller, than the amount of data inserted in the packet.

`outPacket`

On return, contains a pointer to the packet. Use this reference to insert data into the packet.

DISCUSSION

The media packetizer uses this function to create each new packet, prior to inserting any literal, repeated, and/or sample data. A call to `RTPPBBeginPacketGroup` must be made prior to creating the first packet in a group. Data is inserted into the packet using `RTPPBAddPacketLiteralData`, `RTPPBAddPacketRepeatedData`, and/or `RTPPBAddPacketSampleData`. When the packet is complete, call `RTPPEndPacket`.

RTPPBAddPacketLiteralData

Pass literal data directly to a packet builder component. For example, use this function to insert static header information into a packet prior to inserting media sample data. This function will return a data reference you can use to insert the same static information into later packets.

```
ComponentResult RTPPBAddPacketLiteralData (
    RTPPacketBuilder rtpb,
    SInt32 inFlags,
    RTPPacketGroupRef inPacketGroup,
    RTPPacketRef inPacket,
    UInt8 *inData,
    UInt32 inDataLength,
    RTPPacketRepeatedDataRef *outDataRef);
```

`rtpb`

The component instance of the packet builder component

<code>inFlags</code>	A signed 32-bit integer containing any flags you are passing. There are currently no defined flags.
<code>inPacketGroup</code>	The packet group containing the packet into which the data will be placed. This is normally a reference returned by <code>RTPPBBeginPacketGroup</code> .
<code>inPacket</code>	The RTP packet into which the data will be placed. This is normally a reference returned by <code>RTPPBBeginPacket</code> .
<code>inData</code>	A pointer to the data you are passing
<code>inDataLength</code>	An unsigned 32-bit integer containing the length, in bytes, of the data you are passing
<code>outDataRef</code>	On return, contains a pointer to a data reference. Use this reference if you wish to later tell the packet builder to use this same data again, without having to literally pass the data again. Pass in <code>nil</code> if you do not need the packet builder to repeat the data. If you do not pass in <code>nil</code> , you must dispose of the data explicitly by calling <code>RTPPBReleaseRepeatedData</code> .

DISCUSSION

This function will return a reference which can be used to specify the same data repeatedly without having to pass in the data again. This is done by calling `RTPPBAddPacketRepeatedData` with the reference which was returned by `RTPPBAddPacketLiteralData`. When a reference is no longer needed, it should be disposed of by using the call `RTPPBReleaseRepeatedData`. To specify media data to be placed in a packet, a media packetizer should call `RTPPBAddPacketSampleData`.

RTPPBAddPacketSampleData

A media packetizer uses this function to command a packet builder component to insert media sample data into a packet. This typically follows static header information inserted by calling either `RTPPBAddPacketLiteralData` or

`RTPPBAddPacketRepeatedData`. The media packetizer specifies the offset into the media and the length of the sample to insert.

```
ComponentResult RTPPBAddPacketSampleData (
    RTPPacketBuilder rtpb,
    SInt32 inFlags,
    RTPPacketGroupRef inPacketGroup,
    RTPPacketRef inPacket,
    RTPMPSampleDataParams *inSampleDataParams,
    UInt32 inSampleOffset,
    UInt32 inSampleDataLength,
    RTPPacketRepeatedDataRef *outDataRef);
```

<code>rtpb</code>	The component instance of the packet builder component
<code>inFlags</code>	A signed 32-bit integer containing any flags you are passing. There are currently no defined flags.
<code>inPacketGroup</code>	The packet group containing the packet into which the data will be placed. This is normally a reference returned by <code>RTPPBBeginPacketGroup</code> .
<code>inPacket</code>	The RTP packet into which the data will be placed. This is normally a reference returned by <code>RTPPBBeginPacket</code> .
<code>inSampleDataParams</code>	A pointer to a <code>SampleDataParams</code> structure for the sample data you are inserting.
<code>inSampleOffset</code>	A 32-bit unsigned integer containing the offset into the sample media, in bytes
<code>inSampleDataLength</code>	A 32-bit unsigned integer specifying the number of bytes of media sample data to insert into the packet
<code>outDataRef</code>	On return, contains a pointer to a data reference. Use this reference if you wish to later tell the packet builder to use this same sample data again, without having to literally pass the data again. Pass in <code>nil</code> if you do not need the packet builder to repeat the data. If you do not pass in <code>nil</code> , you must dispose of the data explicitly by calling <code>RTPPBReleaseRepeatedData</code> .

DISCUSSION

This function will return a reference which can be used to specify the same data repeatedly without having to pass in the data again. This is done by calling `RTPPBAddPacketRepeatedData` with the reference which was returned by `RTPPBAddPacketLiteralData`. When a reference is no longer needed, it should be disposed of by using the call `RTPPBReleaseRepeatedData`. To specify media data to be placed in a packet, a media packetizer should call `RTPPBAddPacketSampleData`.

RTPPBAddPacketRepeatedData

Tell a packet builder component to insert previously-specified data into a packet. This is typically done to repeat static header information into a series of packets, or to insert previously-sent sample data into a redundant packet. The data is first specified by a call to `RTPPBAddPacketLiteralData` or `RTPPBAddPacketSampleData`, which inserts the data the first time and returns a data reference. Use the data reference with `RTPPBAddPacketRepeatedData` to send the data again.

```
ComponentResult RTPPBAddPacketRepeatedData (
    RTPPacketBuilder rtpb,
    SInt32 inFlags,
    RTPPacketGroupRef inPacketGroup,
    RTPPacketRef inPacket,
    RTPPacketRepeatedDataRef inDataRef);
```

`rtpb` The component instance of the packet builder component

`inFlags` A signed 32-bit integer containing any flags you are passing. There are currently no defined flags.

`inPacketGroup` The packet group containing the packet into which the data will be placed. This is normally a reference returned by `RTPPBBeginPacketGroup`.

`inPacket` The RTP packet into which the data will be placed. This is normally a reference returned by `RTPPBBeginPacket`.

`inDataRef` **A reference to the data to repeat (this is normally a data reference returned from `RTPPBAddPacketLiteralData` or `RTPPBAddPacketSampleData`).**

DISCUSSION

Use this function to cause a packet builder component to repeatedly insert the same data into packets without having to pass the data each time. When you are done sending the repeated data, release the data structure by calling `RTPPBReleaseRepeatedData`.

RTPPBReleaseRepeatedData

Allow a packet builder to deallocate data that will no longer be used.

```
ComponentResult RTPPBReleaseRepeatedData (
    RTPPacketBuilder rtpb,
    RTPPacketRepeatedDataRef inDataRef);
```

`rtpb` **The component instance of the packet builder component**

`inDataRef` **The data reference to the repeated data. This is normally a data reference returned by `RTPPBAddPacketLiteralData` or `RTPPBAddPacketSampleData`.**

DISCUSSION

When a media packetizer passes data to a packet builder using `RTPPBAddPacketLiteralData` or `RTPPBAddPacketSampleData`, the packet builder can return a data reference. The media packetizer can use this data reference to insert the same data into other packets. The media packetizer must release this data when it is no longer needed.

You must release the data if you have allowed `RTPPBAddPacketLiteralData` or `RTPPBAddPacketSampleData` to return a data reference, even if you have not called `RTPPBAddPacketRepeatedData`. You must either pass in a `nil` to the data reference when adding literal or sample data, or you must release the data by calling this function.

RTPPEndPacket

A media packetizer calls this function to tell a packet builder that a packet is complete.

```
ComponentResult RTPPEndPacket (
    RTPPacketBuilder rtpb,
    SInt32 inFlags,
    RTPPacketGroupRef inPacketGroup,
    RTPPacketRef inPacket,
    UInt32 inTimeOffset,
    UInt32 inDuration);
```

rtpb The component instance of the packet builder component

inFlags A signed 32-bit integer containing any flags you are passing. There are currently no defined flags.

inPacketGroup The packet group containing the new packet. This is normally a reference returned by `RTPPBeginPacketGroup`.

inTimeOffset The time offset at which the media sample data contained in this packet begins, in milliseconds. This offset is added to the RTP transmission time in determining when to send the packet. Unsigned 32-bit integer.

inDuration The duration of this packet, specified in milliseconds. Unsigned 32-bit integer.

DISCUSSION

Call this function once when each packet is complete.

RTPPBEndPacketGroup

Tell a packet builder component that a packet group is complete.

```
ComponentResult RTPPBEndPacketGroup (
    RTPPacketBuilder rtpb,
    SInt32 inFlags,
    RTPPacketGroupRef inPacketGroup);
```

`rtpb` The component instance of the packet builder component

`inFlags` A signed 32-bit integer containing any flags you are passing. There are currently no defined flags.

`inPacketGroup` A data reference to the packet group being ended. This is normally a data reference returned by `RTPPBBeginPacketGroup`.

DISCUSSION

This function should be called when all the packets in a group are complete and the media packetizer is either ready to create a new packet group or to terminate the stream.

Packet Reassembler Functions

The following functions can be implemented by packet reassembler components. As noted, some of these functions must be implemented in your component, some can be delegated to the base component, and some are base component utility functions that your component can call.

RTPRssmInitialize

The base reassembler will call this function when it is ready to have your packet reassembler begin handling media packets (this function is not called when the base reassembler opens your component to check its version).


```
ComponentResult RTPRssmInitialize (  
    RTPReassembler rtpr,  
    RTPRssmInitParams *inInitParams);
```

rtpr The component instance of your packet reassembler

inInitParams A pointer to an `RTPRssmInitParams` struct. Use the information contained in this struct to initialize your component.

RTPRssmReset

This function is called to reset all packet reassembler and base reassembler variables for a new run of data.

```
ComponentResult RTPRssmReset (  
    RTPReassembler rtpr,  
    SInt32 inFlags);
```

rtpr The component instance of your reassembler

inFlags A signed 32-bit integer containing any flags being passed. No flags are currently defined.

DISCUSSION

Be sure to release any packet lists you may have created and decrement to zero the reference counters on any chunks you are maintaining.

RTPRssmComputeChunkSize

The base reassembler will call this function to give your packet reassembler the opportunity to compute the size of a chunk, based on the packet list for the chunk, using your own algorithm. This function is called once for each packet list. Implement this function only if you need to override the base reassembler's default computation.

```
ComponentResult RTPRssmComputeChunkSize (
    RTPReassembler rtpr,
    RTPRssmPacket *inPacketListHead,
    SInt32 inFlags,
    UInt32 *outChunkDataSize);
```

rtpr	The component instance of your packet reassembler
inPacketListHead	A pointer to the list of packets that make up this chunk
inFlags	A signed 32-bit integer containing any flags being passed to your packet reassembler.
outChunkDataSize	You should return a pointer to an unsigned 32-bit variable containing the calculated size for this chunk.

DISCUSSION

If you do not implement this call, the base reassembler will compute the chunk size by summing the `dataLength` for all packets in the list.

RTPRssmAdjustPacketParams

This function is called by the base reassembler when it is processing a packet. This allows your packet reassembler to adjust the packet parameters before the packet is processed.

```
ComponentResult RTPRssmAdjustPacketParams (
    RTPReassembler rtpr,
    RTPRssmPacket *inPacket,
    SInt32 inFlags);
```

rtpr	The component instance of your packet reassembler
inPacket	A pointer to the <code>RTPRssmPacket</code> struct whose parameters can be adjusted
inFlags	A signed 32-bit integer containing any flags being passed to your packet reassembler.

DISCUSSION

If your packet reassembler does not implement this function, or takes no action, the default for these parameters will be:

payloadHeaderLength = fixed header length that is set (default is 0)
 dataLength = packetData - transportHeaderLength - payloadHeaderLength
 no serverEditParams
 chunkFlags = 0

RTPRssmCopyDataToChunk

The base reassembler will call this function to allow your packet reassembler the opportunity to write the chunk data, based on the list of packets for the chunk, using your own algorithm. For example, an H.261 packet reassembler must adjust the byte at packet boundaries. Implement this function only if you need to override the base reassembler's default behavior.

```
ComponentResult RTPRssmCopyDataToChunk (
    RTPReassembler rtpr,
    RTPRssmPacket *inPacketListHead,
    UInt32 inMaxChunkDataSize,
    SHChunkRecord *inChunk,
    SInt32 inFlags);
```

rtpr	The component instance of your packet reassembler
inPacketListHead	A pointer to the RTPRssmPacket struct for the first packet in the list
inMaxChunkDataSize	An unsigned 32-bit integer containing the maximum allowable chunk size
inChunk	A pointer to the chunk record. Write the chunk data to this record.
inFlags	A 32-bit signed integer containing any flags being passed to your media packetizer.

DISCUSSION

If you do not implement this function, the base reassembler will write the chunk data by taking `dataLength` bytes from each packet in the list, starting at an offset of (`packetData + transportHeaderLength + payloadHeaderLength`).

RTPRssmSendPacketList

The base reassembler will call this function when it is ready to create a chunk and send it to the stream handler, based on a list of packets. Implement this call if your packet reassembler needs to modify the packet list, or if it overrides the default handling of packet loss.

```
ComponentResult RTPRssmSendPacketList (
    RTPReassembler rtpr,
    RTPRssmPacket *inPacketListHead,
    const TimeValue64 *inLastChunkPresentationTime,
    SInt32 inFlags);
```

`rtpr` The component instance of your packet reassembler

`inPacketListHead` A pointer to the `RTPRssmPacket` struct for the first packet in the list

`inLastChunkPresentationTime` A pointer to a time value which specifies when to present this chunk, in units of the stream's time scale

`inFlags` A signed 32-bit integer containing any flags being passed. The only flag currently defined is:
`kRTPRssmLostSomePackets = 0x00000001`

DISCUSSION

If you do not implement this call, the base reassembler will adjust the packet parameters on all packets in the list, compute the chunk size, and send the chunk. If packet loss has occurred, all the packets will be discarded and the stream handler will be informed that the chunk has been lost. If you implement this function `inFlags` will be set to indicate packet loss if you have instructed to base reassembler to do so in `RTPRssmSetCapabilities`.

RTPRssmGetTimeScaleFromPacket

If your packet reassembler has not specified a timescale as part of `RTPRssmNewStreamHandler`, or by calling `RTPRssmSetTimeScale`, the base reassembler will call this function when it receives packets. This allows your packet reassembler to extract the timescale from a received packet and return it to the base reassembler. The base reassembler will discard received packets until it has been given a valid timescale.

```
ComponentResult RTPRssmGetTimeScaleFromPacket (
    RTPReassembler rtpr,
    QTSSStreamBuffer *inStreamBuffer,
    TimeScale *outTimeScale);
```

<code>rtpr</code>	The component instance of your packet reassembler
<code>inStreamBuffer</code>	A pointer to a received packet from which you may be able to extract a timescale
<code>outTimeScale</code>	Return a pointer to a valid timescale or return an error. If you return a timescale, the packet will be processed normally. Set to zero if you cannot determine the timescale from this packet.

DISCUSSION

Your packet reassembler must set a time scale for the stream handler before the base reassembler can process any incoming packets. If your packet reassembler doesn't know the timescale of its media in advance, because the timescale is contained in the packets for example, the base reassembler will prompt you for a timescale whenever it receives a packet. If your packet reassembler always uses the same timescale, it should set the timescale when it opens a stream handler, and it does not need to implement this function.

RTPRssmSetInfo

This function is called to set various parameters of your packet reassembler. It is also called to set parameters of the base reassembler.

```
ComponentResult RTPRssmSetInfo (
    RTPReassembler rtpr,
    OSType inSelector,
    void *ioParams);
```

rtpr	The component instance of your packet reassembler
inSelector	A selector for the information being set. Ignore any selectors you do not understand. There are currently no selectors defined.
ioParams	A pointer to a data structure containing the information that should be set.

DISCUSSION

Delegate this function to the base reassembler for any selectors you do not understand. If the base reassembler doesn't understand them either, it will return an error to the caller.

RTPRssmGetInfo

The QTS Toolbox calls this function to obtain information about your packet reassembler.

```
ComponentResult RTPRssmGetInfo (
    RTPReassembler rtpr,
    OSType inSelector,
    void *ioParams);
```

rtpr	The component instance of your packet reassembler
inSelector	A selector for the information desired. There are currently no selectors defined.
ioParams	A pointer to a data structure appropriate for the type of data requested. If your component understands the selector, write the requested information into the data structure this parameter points to.

DISCUSSION

Implement this function only for the selectors you understand. Delegate this function to the base reassembler for any other selectors. The base reassembler will correctly return an error if it doesn't understand the selector either.

RTPRssmSetCapabilities

Your reassembler calls this function, typically as part of `RTPRssmInitialize`, to set flags that control the base reassembler.

```
ComponentResult RTPRssmSetCapabilities (
    RTPReassembler rtpr,
    SInt32 inFlags,
    SInt32 inFlagsMask);
```

`rtpr` The component instance of the base reassembler

`inFlags` A signed 32 bit integer containing the logical OR of all the flags you are setting. Defined flags are:

`kRTPRssmEveryPacketAChunkFlag` — Tells the base reassembler to treat every packet as a chunk, as would be the case for μ -law audio, for example.

`kRTPRssmQueueAndUseMarkerBitFlag` — Tells the base reassembler to queue incoming packets and assemble a chunk when a packet has the marker bit set or the RTP time stamp changes.

`kRTPRssmTrackLostPacketsFlag` — Tells the base reassembler to check the RTP sequence numbers and set the `kRTPRssmLostSomePackets` flag if any packets are missing from the packet list when it calls your reassembler's `SendPacketList` function.

`kRTPRssmNoReorderingRequiredFlag` — Tells the base reassembler that packets do not need to come in sequence-number order, as would be the case for μ -law audio, for example.

`inFlagsMask` Use this field to preserve the state of any flags you do not wish to alter. If a flag is set in this field, and is not set in the `inFlags` field, it will not be changed from its current setting.

DISCUSSION

Your packet reassembler can call this function at any time.

RTPRssmGetCapabilities

Your packet reassembler can call this function to obtain the current flag settings for the base reassembler.

```
ComponentResult RTPRssmGetCapabilities (
    RTPReassembler rtpr,
    SInt32 *outFlags);
```

`rtpr` The component instance of the base reassembler

`outFlags` On return, contains a pointer to the current flags (32-bit signed integer). The flags are described in `RTPRssmSetCapabilities`.

DISCUSSION

Your packet reassembler can call this function at any time.

RTPRssmSetPayloadHeaderLength

Your packet reassembler calls this function to set a fixed payload header length. If this is not set, the default value is zero.

```
ComponentResult RTPRssmSetPayloadHeaderLength (
    RTPReassembler rtpr,
    UInt32 inPayloadHeaderLength);
```

`rtpr` The component instance of the base reassembler

`inPayloadHeaderLength`

An unsigned 32-bit integer specifying the payload header length, in bytes

DISCUSSION

Call this function during initialization if your packet reassembler works with RTP payloads that contain payload headers of a fixed length.

RTPRssmGetPayloadHeaderLength

Your packet reassembler can call this function to obtain the current value of the fixed payload header length from the base reassembler.

```
ComponentResult RTPRssmGetPayloadHeaderLength (  
    RTPReassembler rtpr,  
    UInt32 *outPayloadHeaderLength);
```

`rtpr` The component instance of the base reassembler component

`outPayloadHeaderLength`

On return, contains a pointer to an unsigned 32-bit integer containing the length of the payload header in bytes.

DISCUSSION

Your packet reassembler can call this function at any time.

RTPRssmSetTimeScale

Your packet reassembler uses this function to set the timescale for the stream handler that will render your output. The timescale is the number of time units that pass in one second for the media whose sample data is carried in this stream. The stream handler's timescale must be set before it can deliver any data to the user.

```
ComponentResult RTPRssmSetTimeScale (  
    RTPReassembler rtpr,  
    TimeScale inSHTimeScale);
```

rtpr **The component instance of the base reassembler**

inSHTimeScale **The time scale for the stream handler to use**

DISCUSSION

This function is normally called by your packet reassembler when the timescale to use is not initially known. You do not need to call this function if you specified a timescale when the stream handler was opened.

RTPRssmGetTimeScale

Your packet reassembler can call this function to obtain the current timescale from the base reassembler.

```
ComponentResult RTPRssmGetTimeScale (  
    RTPReassembler rtpr,  
    TimeScale *outSHTimeScale);
```

rtpr **The component instance of the base reassembler**

outSHTimeScale **On return, contains a pointer to the timescale in use by the stream handler that is processing your output**

RTPRssmNewStreamHandler

This function causes the base reassembler to open a new stream handler. Any currently-opened stream handler will be closed.

```
ComponentResult RTPRssmNewStreamHandler (  
    RTPReassembler rtpr,  
    OSType inSHType,  
    SampleDescriptionHandle inSampleDescription,  
    TimeScale inSHTimeScale,  
    ComponentInstance *outHandler);
```

rtpr The component instance of the base reassembler

inSHType The stream handler type. Use the same constant you would use to specify a track of this type.

inSampleDescription
A handle to a sample description appropriate for this media type. Pass in NULL if you don't know the media type yet. The sample description is passed by reference; the caller is responsible for maintaining it.

inSHTimeScale
The time scale for the stream handler to use. Pass in 0 if the time scale is not yet known.

outHandler On return, contains a pointer to the component instance of the stream handler that has been opened.

DISCUSSION

Your packet reassembler should call this function when it is initialized.

You must pass in a valid sample description and time scale before the stream handler can process packets. If you do not pass them as part of this function, do so using `RTPRssmSetTimeScale` and `RTPRssmSetSampleDescription`.

RTPRssmSendStreamHandlerChanged

Your packet reassembler should call this function when something has changed in the stream and you want the notification propagated (for example, you have changed the dimensions of the video).

```
ComponentResult RTPRssmSendStreamHandlerChanged (RTPReassembler rtpr);
```

`rtp` The component instance of the base reassembler

DISCUSSION

You will typically receive a series of `RTPRssmGetInfo` calls to determine what has changed.

RTPRssmSetSampleDescription

Your packet reassembler uses this function to change the sample description being used by the stream handler. All subsequent samples will be marked with this new sample description.

```
ComponentResult RTPRssmSetSampleDescription (  
    RTPReassembler rtp,  
    SampleDescriptionHandle inSampleDescription);
```

`rtp` The component instance of the base reassembler

`inSampleDescription`
 The handle of a sample description to use. You are responsible for keeping the handle and the data structure valid during subsequent operations.

DISCUSSION

The sample description is not passed on a per-packet basis, but a per-sample basis, so the sample description should not be changed until a complete sample (sometimes called a "frame" or "chunk") has been reassembled.

RTPRssmGetChunkAndIncrRefCount

Your packet reassembler uses this function to cause the base reassembler to create a chunk for you manually.

```
ComponentResult RTPRssmGetChunkAndIncrRefCount (
    RTPReassembler rtpr,
    UInt32 inChunkDataSize,
    const TimeValue64 *inChunkPresentationTime,
    SHChunkRecord **outChunk);
```

rtpr	The component instance of the base reassembler component
inChunkDataSize	An unsigned 32-bit integer containing the size of the chunk's data portion, in bytes
inChunkPresentationTime	A pointer to a 64 -bit time value specifying the time at which this chunk should be presented, in units of the stream's timescale
outChunk	On return, contains a pointer to the newly-created chunk record

DISCUSSION

This function is useful if you are overriding the `RTPRssmSendPacketList` behavior and constructing the chunk yourself. You must explicitly dispose of the chunk when you are done with it by calling either `RTPRssmDecrChunkRefCount` or `RTPRssmSendChunkAndDecrRefCount`.

RTPRssmSendChunkAndDecrRefCount

Your packet reassembler can call this function when it has finished constructing a chunk and wants the base reassembler to send it to the stream handler.

```
ComponentResult RTPRssmSendChunkAndDecrRefCount (
    RTPReassembler rtpr,
    SHChunkRecord *inChunk,
    const SHServerEditParameters *inServerEdit);
```

rtpr	The component instance of the base reassembler
inChunk	A pointer to a stream handler chunk record

`inServerEdit` A pointer to the server edit parameters. Pass in `NULL` if there is no server edit.

DISCUSSION

Use this function to manually send a chunk if you have overridden the default behavior of `RTPRssmSendPacketList`. This function will decrement the refcount of the chunk. If the refcount is then zero, the chunk is deallocated.

RTPRssmSendLostChunk

Your packet reassembler can use this function to cause the base reassembler to send loss notification to the stream handler.

```
ComponentResult RTPRssmSendLostChunk (
    RTPReassembler rtpr,
    const TimeValue64 *inChunkPresentationTime);
```

`rtpr` The component instance of the base reassembler

`inChunkPresentationTime` A pointer to a 64-bit time value indicating when the chunk would have been presented, in units of the stream's time scale

DISCUSSION

Loss notification is normally performed automatically by the base reassembler. Use this function if you are handling loss and sending chunks manually. Call this function if the chunk you have built will not be sent.

RTPRssmClearCachedPackets

Your packet reassembler uses this function to cause the base reassembler to flush all packets currently queued in its lists.

```
ComponentResult RTPRssmFlushPackets (RTPReassembler rtpr);
```

`rtpr` The component instance of the base reassembler

DISCUSSION

This function is only useful when the base reassembler is operating with the `kRTPRssmQueueAndUseMarkerBit` flag set.

RTPRssmReleasePacketList

Use this function to release the memory associated with a packet list that your packet reassembler created itself, or a list your reassembler took ownership of as a result of implementing `RTPRssmSendPacketList`.

```
ComponentResult RTPRssmReleasePacketList (  
    RTPReassembler rtpr,  
    RTPRssmPacket *inPacketListHead);
```

`rtpr` The component instance of the base reassembler

`inPacketListHead`

A pointer to the `RTPRssmPacket` struct of the first packet in packet list to dispose of

DISCUSSION

This is a housekeeping function that you do not need to perform for packet lists created and handled by the base reassembler, only for packet lists that you create or take ownership of yourself.

RTPRssmIncrChunkRefCount

Use this function to tell the base reassembler to keep a copy of a chunk, to assist in loss recovery, for example.

```
ComponentResult RTPRssmIncrChunkRefCount (
    RTPReassembler rtpr,
    SHChunkRecord *inChunk);
```

`rtpr` **The component instance of the base reassembler**
`inChunk` **A pointer to the chunk record you want to preserve**

DISCUSSION

You must call `RTPRssmDecrChunkRefCount` to release the chunk when you no longer need it.

RTPRssmDecrChunkRefCount

Tell the base reassembler to decrement the reference counter of a chunk that it has created or preserved for you. If the reference count becomes zero, the chunk is deallocated.

```
ComponentResult RTPRssmDecrChunkRefCount (
    RTPReassembler rtpr,
    SHChunkRecord *inChunk);
```

`rtpr` **The component instance of the base reassembler component**
`inChunk` **A pointer to the chunk record to dispose**

DISCUSSION

If you have overridden the `RTPRssmSendPacketList` behavior, and are instructing the base reassembler to construct chunks manually, your packet assembler must explicitly dispose of the chunks by calling either `RTPRssmDecrChunkRefCount` or `RTPRssmSendChunkAndDecrRefCount`. This function is also used to release a chunk you have preserved using the `RTPRssmIncrChunkRefCount` function.

Common Streaming Error Codes

This is a list of common streaming error codes. Applications that play streaming movies should deal gracefully with these errors.

Streaming Errors

- -5400 - qtsBadSelectorErr
- -5401 - qtsBadStateErr
- -5402 - qtsBadDataErr
- -5403 - qtsUnsupportedDataTypeErr
- -5404 - qtsUnsupportedRateErr
- -5405 - qtsUnsupportedFeatureErr
- -5406 - qtsTooMuchDataErr

Network Errors

- -5420 - connection failed (couldn't connect to the server)

Open Transport (Mac only: -3150 to -3180, -3200 to -3285)

look for the full range of OT error codes in `OpenTransport.h`

- -3150 - kOTBadAddressErr - a bad address was specified
- -3170 - kOTBadNameErr; couldn't do name lookup

Windows only

10061 - couldn't connect to server

Mac Toolbox Errors

Streaming can return all QuickTime and Mac Toolbox errors. Common error codes returned include:

QuickTime Errors

- -223 - siInvalidCompression; codec not installed

Windows Specific Errors

- -2092 - componentDllEntryNotFoundErr; Windows specific error when component is loading

RTSP Errors

Most of these error codes are the same as for HTTP. These errors are usually generated from the server.

- 3xx - redirect
- 400 - bad request
- 401 - unauthorized
- 402 - payment required
- 403 - forbidden
- 404 - not found (specified movie not found on server side)
- 405 - method not allowed
- 406 - not acceptable
- 407 - proxy authentication required
- 408 - request time-out
- 409 - conflict
- 410 - gone
- 411 - length required
- 412 - precondition failed
- 413 - request entity too large
- 414 - request URI too large
- 415 - unsupported media type
- 451 - parameter not understood
- 452 - conference not found
- 453 - not enough bandwidth
- 454 - session not founds
- 455 - method not valid in this state
- 456 - header field not valid for resource
- 457 - invalid range
- 458 - parameter is read only
- 459 - aggregate operation not allowed
- 460 - only aggregate operation allowed

- 461 - unsupported transport
- 462 - destination unreachable
- 500 - internal server error
- 501 - not implemented
- 502 - bad gateway
- 503 - service unavailable
- 504 - gateway timeout
- 505 - version not supported
- 551 - option not supported

Hint Track Format

You prepare a QuickTime movie for RTP streaming by adding hint tracks. Hint tracks allow an RTP server to stream QuickTime movies without requiring the RTP server to understand QuickTime media types, codecs, or packing. The RTP server needs to understand the QuickTime file format sufficiently to find tracks and media samples in a QuickTime movie, however.

Each track in a QuickTime movie is sent as a separate RTP stream, and the recipe for packetizing each stream is contained in a corresponding hint track. Each sample in a hint track tells the RTP server how to packetize a specific amount of media data. The hint track sample contains any data needed to build a packet header of the correct type, and also contains a pointer to the block of media data that belongs in the packet.

There is at least one hint track for each media track to be streamed. It is possible to create multiple hint tracks for a given track's media, optimized for streaming the same media over networks with different packet sizes, for example. The hinter included in this release of QuickTime creates one hint track for each track to be streamed.

Hint tracks are structured in the same way as other QuickTime movie tracks. Standard QuickTime data structures, such as track references, are used to point to data. In order to serve a QuickTime movie, your RTP server must locate the hint tracks contained in the movie and parse them to find the packet data they point to. You should review the *QuickTime File Format* documentation before

you read the remainder of this hint track information. You will also find it helpful to refer to the *QuickTime File Format* periodically in the course of reading this document.

Locating Hint Tracks in a QuickTime Movie

In order to find the hint tracks in a QuickTime movie, you need to “walk” the atom structure of the movie. QuickTime provides an API to simplify this process, but it is relatively straightforward to do on your own if QuickTime is not available for your server’s OS.

Note that all data in a movie atom structure is in big-endian format.

A QuickTime movie is made of data structures called **atoms**. Each atom has a header which includes its size and type. The atom’s type tells you what kind of data it contains. The type field is a raw four-character string with no count byte or delimiters. The size field is a 32-bit integer. A given atom is normally identified by its byte offset within an atom structure. Adding the size of an atom to its offset will take you to the offset of the next atom in the structure.

Some atoms contain other atoms, called **child atoms**. The size field of an atom includes any child atoms, so you can skip an atom and all of its children in one step. If an atom contains child atoms, you can walk its internal structure, adding the size of each child atom to its offset to find the next child atom.

A QuickTime movie is an atom of type ‘moov’. It contains a child atom of type ‘mvhd’ (the movie header), a series of ‘trak’ atoms (the media tracks and hint tracks), and a movie user data atom (‘udta’). A diagram of a typical movie atom, with two media tracks and two hint tracks, is shown below. Most of these atoms contain child atoms. The diagram is “collapsed” to show only the highest-level atoms.

```
'moov' - Movie
    'mvhd' - Movie Header
    'trak' - Track
    'trak' - Track
    'trak' - Track
    'trak' - Track
    'udta' - User Data
```

Hint tracks are atoms of type 'trak'. A hint track atom contains a track header atom ('tkhd'), an edits atom ('edts'), a track reference atom pointing to the track being hinted ('tref'), a media atom ('mdia'), and usually a track user data atom ('udta'). A diagram of a typical hint track atom is shown below.

```
'trak' - Track
    'tkhd' - Track Header
    'edts' - Edits
    'tref'
    'mdia' - Media
    'udta' - User Data
```

The media atom ('mdia') in a hint track contains a media handler child atom ('hdlr'). Hint tracks can be identified by their media handler, which is of the generic component type 'mhlr' (bytes 12-15), component subtype 'hint' (bytes 16-19). A diagram of an 'hdlr' atom for a hint track is shown below.

```
'trak' - Track
    'tkhd' - Track Header
    'edts' - Edits
    'tref'
    'mdia' - Media
        'mdhd' - Media Handler Header
        'hdlr' - Handler Description
            flags      $00000000  type
            componentFlags $00000001 subType
            componentFlagsMask $000101AE name
```

mhlr	manufacturer	appl
hint	hint media handler	

To locate all the hint tracks in a movie, walk the movie to find atoms of type 'trak'. Walk the 'trak' atoms to find their 'mdia' atoms, and walk the 'mdia' atoms to find the 'hdlr' atom. Check the contents of the 'hdlr' atom for a component type of 'mhlr' and a subtype of 'hint'. Refer to the *QuickTime File Format* documentation for details on the structure of these atom types.

Finding the Hinted Track from a Hint Track

Each hint track contains a track reference atom (`'tref'`) that contains a child atom of type `'hint'` which holds the track ID of the track being hinted. A diagram of a hint track reference atom is shown below.

Atom/Field	Bytes	Description
Track Reference Atom		Contains a 'hint' atom
Size	4	Size of track reference atom (including 'hint' atom). 32-bit integer.
Type	4	'tref'
..Track Reference Atom		Contains the track ID of the track being hinted
..Size	4	Size of this child atom. 32-bit integer.
..Type	4	'hint'
..Track ID	4	Track ID of the track being hinted. 32-bit integer.

To find a hinted track from its hint track, walk the hint track atom until you find an atom of type `'tref'`. Walk the `'tref'` atom until you find a child atom of type `'hint'`. This will normally be the first child atom. The child atom's type (`'hint'`) will be followed immediately by the track ID of the track being hinted. The track ID is a 32-bit integer.

Track references can be one-to-many, so there could theoretically be a list of track IDs, which you would detect by examining the `'hint'` atom's size. The hinter provided by Apple currently creates a track reference with a single track ID.

QuickTime functions that work with track references are designed to deal with lists of track IDs, so they expect an index parameter to determine which item on the list you are interested in. Pass in 0 as the index parameter to get the first item if you use a function such as `GetTrackReference` to obtain the media track ID.

Each track contains a track header atom (`'tkhd'`) that contains the track's ID. To find the track whose ID is referenced by a hint track, walk the track atoms of the movie until you find the one whose track header atom contains the referenced track ID.

Currently, only one hint track is created for each streamed track, but the hint track structure allows multiple hint tracks to reference the same track. Once you

have determined which track belongs to each hint track, you should check for multiple hint tracks pointing to the same track. If there are duplicates, use the hint track best optimized for your network.

Finding a Hint Track from a Hinted Track

Tracks can contain track reference atoms of type 'hinn' for each hint track that references them, but QuickTime does not currently insert such atoms when a track is hinted. The only reliable way to determine hint track relationships is to work in the other direction, from the hint track to the track being hinted. Multiple hint tracks could theoretically reference the same track, so you need to find all the hint tracks and check their track reference atoms to be certain.

Hint Track Structure

Hint tracks are atoms of type 'trak'. A hint track atom contains a track header atom ('tkhd'), an edits atom ('edts'), a track reference atom ('tref'), a media atom ('mdia'), and usually a track user data atom ('udta'). The atoms are normally present in the order described, but this is not a requirement. Most of

these atoms have child atoms. An expanded diagram of a typical hint track atom is shown below

```



```

A detailed description of each atom's contents follows. This document focuses on atom contents that are specific to hint tracks. Refer to the *QuickTime File Format* documentation for generic information about the structure and contents of the atom types found in track atoms.

Note that the actual hint samples are not part of the hint track atom structure. The hint track atom structure contains the information you need to locate and interpret the samples. A description of hint samples follows the description of the hint track's atoms.

Track Header Atom ('tkhd')

The flags field of the track header atom must be 0x000000, indicating the track is inactive, and not part of the movie, preview, or poster. Hint tracks must be marked as inactive for the movie to play locally.

The creation time, modification time, and track ID fields are set as they would be for any track.

The duration field is also the duration of the media track being hinted, expressed in movie time scale.

The layer field is not used.

The alternate group field can be used to distinguish alternate language or alternate data rate tracks in streaming movies. It is not required that all servers support this feature. It is not recommended that applications attempt to make use of this feature at this time. This field is normally set to 1.

The remaining track header fields are not used for hint tracks.

Unused fields should be set to 0 when creating hint tracks and ignored when using them.

Edits Atom ('edts')

The edits atom contains an edits list atom ('elst'). Edit lists can be used for hint tracks as they can for any other track type.

The fields of the edits list atom ('elst') are used as follows:

The flags field is not used, and should be set to 0.

The numEntries field is the number of list entries. Unless the hint track has been edited, it will have either 1 or 2 entries.

Each list entry consists of a duration, a media time, and a rate.

If the track should begin playing after the movie begins, there is an “empty edit” whose duration is the amount of time that passes until the track begins playing, whose media time is -1, and whose media rate is 1. No data should be sent for this track until the number of movie time units specified by the duration field has passed.

Whether or not there is an empty edit, there will typically be one entry whose duration is the media duration in movie timescale units, whose media time is 0, and whose rate is 1.

Track Reference Atom ('tref')

Each hint track refers to a media track. The hint track contains a track reference atom ('tref') containing a child atom of type 'hint' which holds the track ID of the media track. There is a diagram of a hint track reference atom below.

Atom/Field	Bytes	Description
Track Reference Atom		Contains a 'hint' atom
Size	4	Size of track reference atom (including 'hint' atom). 32-bit integer.
Type	4	'tref'
..Track Reference Type Atom		Contains the track ID of the media track being hinted
..Size	4	Size of this child atom. 32-bit integer.
..Type	4	'hint'
..Track ID	4	Track ID of the media track being hinted. 32-bit integer.

The Track ID field of the 'hint' atom contains the track ID of the media track being hinted. The target track ID can be found in the media track's track header atom ('tkhd'). All media sample data should be taken from the specified media track.

There could theoretically be a list of track ID's, for a hint track that hinted multiple media tracks, but the current hinter only references one media track per hint track.

Media Atom

The media atom is a large and complex atom structure that holds the sample data tables for the hint track. The media atom contains three child atoms: the media handler header ('mdhd'), the handler description ('hdlr'), and the media information atom ('minf'). The media information atom contains several child atoms of its own, including the sample tables. A top-level diagram of a typical media atom is shown below.

```

'mdia' - Media
  'mdhd' - Media Handler Header
  'hdlr' - Handler Description
  'minf' - Media Information

```

The next three sections describe these child atoms in detail. A fourth section describes the sample table atom contained in the media information atom.

Media Handler Header Atom ('mdhd')

The media handler header atom holds the media timescale and duration.

The flags field is not used, and should be set to 0.

The creation time and modification time fields are used normally.

The time scale is the number of time units that pass in one second for the hint track. This may not be the same time scale in use by the track being hinted.

The duration is the duration of the media track in media time scale units.

The language and quality fields mirror the settings of the media track being hinted.

Media Handler Description Atom ('hdlr')

The media handler description atom ('hdlr') identifies a track as a hint track. The media handler description atom is described in the *QuickTime File Format* documentation as a "Handler reference atom".

The flags field is not used, and should be set to 0.

The component type field must be set to 'mhlr'.

The component subtype field must be set to 'hint'. This identifies the track as a hint track.

The manufacturer field is set to 'appl' by the QuickTime hinter component.

The component flags and components flags mask can be ignored by RTP servers.

The Media Information Atom ('minf')

The media information atom ('minf') contains four child atoms: the generic media header atom ('gmhd'), the handler description atom ('hdlr'), the data

information atom ('dinf'), and the sample table atom ('stbl'). A diagram of a 'minf' atom is shown below.

```

'minf' - Media Information
    'gmhd' - Generic Media Header
    'hdlr' - Handler Description
    'dinf' - Data Information
    'stbl' - Sample Table
  
```

The remainder of this section describes the contents of the generic media header atom, handler description atom, and data information atom. The sample table atom is described in the next section.

The Generic Media Header Atom ('gmhd')

The generic media header atom ('gmhd') contains a single child atom, the generic media information atom ('gmin'). The information in this atom is not used by RTP servers, and can be ignored.

The Handler Description Atom ('hdlr')

The handler description atom ('hdlr') found in the media information atom ('minf') contains information that is not used for hint tracks. It can be ignored by RTP servers.

IMPORTANT

Do not mistake the 'hdlr' atom in the media information atom ('minf') for the 'hdlr' atom in the media atom ('mdia'). The 'hdlr' atom in the media atom identifies this track as a hint track. The 'hdlr' atom in the media information atom can be ignored.

The Data Information Atom ('dinf')

The data information atom ('dinf') contains a single child atom: the data reference atom ('dref'). The data reference atom contains a table of data references which are formatted like child atoms. These data references tell you where the hint track sample data can be found. Bear in mind that this is a reference to the hint track's data, not the data in the media track being hinted.

Media tracks can have their sample data spread across multiple files, so the data references are in table format, allowing multiple entries. A hint track will have a

single data reference entry. This will normally be a reference of type 'alis', which indicates a file reference. The reference can also be of type 'rsrc', indicating that the data is stored in a resource file or the resource fork of a Macintosh file.

This is followed by a three-byte flag. The flag is normally set to 0x0001. This indicates that the hint track sample data is in the same file as the movie atom you are parsing. Note that this flag is set to 0x0001 even if the data is in a different fork of the same file.

If this flag is **not** 0x0001, it will be followed by a file specification in Macintosh alias format (see *Inside Macintosh: Files*). This specifies the file that contains the hint track sample data.

If the data reference is of type 'rsrc', a resource type (32-bit unsigned integer) and resource ID (16-bit signed integer) are appended. This resource contains the hint track sample data.

The Sample Table Atom ('stbl')

The sample table atom ('stbl') contains the information you need to find a sample number based on a time and to find the sample's location based on the sample number. Samples are organized into **chunks**, containing one or more samples. The sample table atom contains the information you need to find out which chunk holds a given sample, where that chunk begins, and where in that chunk to find your sample.

The sample table atom contains five or six child atoms: the sample descriptions atom ('std'), the time-to-sample atom ('stts'), the sample-to-chunk atom ('stsc'), the sample sizes atom ('stsz'), and the chunk offset table atom ('stco') are mandatory. The synch samples atom ('stss') is optional. A diagram is shown below.

```
'stbl' - Sample Table
    'std' - Sample Descriptions
    'stts' - Time To Sample
    'stss' - Sync Samples
    'stsc' - Sample to Chunk
    'stsz' - Sample Sizes
    'stco' - Chunk Offset Table
```

- The sample descriptions atom describes the data format of the hint samples and contains track-level information, such as the RTP timescale for this track.

- The time-to-sample atom tells you the duration of each sample, which allows you to determine what hint sample corresponds to what time.
- The sample-to-chunk atom tells you which chunks hold which samples.
- The sample sizes atom tells you the size of each sample.
- The chunk offset table atom gives you the byte offset into a file for each chunk.
- The synch samples atom (optional) is a list of samples you can use as random-access points, such as video key frames. If all the samples are synch samples, this atom is not present.

How to Find a Sample Using the Sample Table Atom

Before we dive into the structural details of the sample table child atoms, you might want to know how they're used. You'll need to examine the structure of the child atoms in detail to implement an actual algorithm.

Let's assume you know what the current movie time is and you want to find the appropriate hint sample to packetize some movie data. Here are the steps you take:

Begin by converting the movie time into timescale units for this hint track.

Note: Time scale for the track is in the media handler header ('mdhd') atom.

Look in the time-to-sample atom ('stts') to determine which sample corresponds to the target time. The time-to-sample atom contains a table of samples and durations. You need to do some integer arithmetic to calculate the sample number from the time.

Once you have the sample number, use the sample-to-chunk atom to find out what chunk it's in. The sample-to-chunk atom contains a table listing chunks and samples-per-chunk. You need to do some integer arithmetic to calculate the chunk number from the sample number. You also need to do some modulo arithmetic to calculate which sample within that chunk is the one you want.

Find out where the chunk starts by looking in the chunk offset table atom ('stco'). It's a simple table with the byte offset of each chunk. Note: The byte offset is into the file or resource specified by the data reference atom ('dref') inside the data information atom ('dinf').

Use the sample sizes atom ('stsz') to determine the byte offset within the chunk for your sample. The sample sizes atom contains either a number (the sample size, if all samples are of equal size) or a table with the size of each

sample. You need to sum the size of every sample in your chunk prior to your target sample to obtain the byte offset of your sample within the chunk.

Add the byte offset of your sample to the byte offset of the chunk. That's where your hint track sample begins.

Now let's look at the sample table child atoms individually. Refer to the *QuickTime File Format* documentation for the structural details of each atom type.

Sample Description Atom ('stsd')

The sample description atom ('stsd') contains information about the hint track samples. It specifies the data format (currently only RTP data format is defined) and which data reference to use (if more than one is defined) to locate the hint track sample data. It also contains some general information about this hint track, such as the hint track version number, the maximum packet size allowed by this hint track, and the RTP timescale. It may contain additional information, such as the random offsets to add to the RTP time stamp and sequence number.

The sample description atom can contain a table of sample descriptions to accomodate media that are encoded in multiple formats, but a hint track can be expected to have a single sample description at this time.

The sample description for hint tracks is defined below.

Table 26-1 Hint track sample description

Field	Bytes
Size	4
Data format	4
Reserved	6
Data reference index	2
Hint track version	2
Last compatible hint track version	2
Max packet size	4
Additional data table	variable
Size	A 32-bit integer specifying the size of this sample description in bytes
Data format	A four-character code indicating the data format of the hint track samples. Only 'rtp ' is currently defined. Note that the fourth character in 'rtp ' is an ASCII blank (hex 20). Do not attempt to packetize data whose format you do not recognize.
Reserved	Six bytes that must be set to 0.
Data reference index	This field indirectly specifies where to find the hint track sample data. The data reference is a file or resource specified by the data reference atom ('dref') inside the data information atom ('dinf') of the hint track. The data information atom can contain a table of data references, and the data reference index is a 16-bit integer that tells you which entry in that table should be used. Normally, the hint track will have a single data reference, and this index entry will be set to 0.
Hint track version	A 16-bit unsigned integer indicating the version of the hint track specification. This is currently set to 1.

Last compatible hint track version

A 16-bit unsigned integer indicating the oldest hint track version with which this hint track is backward-compatible. If your application understands the hint track version specified by this field, it can work with this hint track.

Max packet size

A 32-bit integer indicating the packet size limit, in bytes, used when creating this hint track. The largest packet generated by this hint track will be no larger than this limit.

Additional data table

A table of variable length containing additional information. Additional information is formatted as a series of tagged entries.

This field always contains a tagged entry indicating the RTP timescale for RTP data. All other tagged entries are optional.

Three data tags are currently defined for RTP data. One tag is defined for use with any type of data. Additional tags can be created by developers. Tags are identified using four-character codes. Tags using all lowercase letters are reserved by Apple. Ignore any tagged data you do not understand.

Table entries are structured like atoms. The structure of table entries is shown below.

Field	Format	Bytes
Entry length	32-bit integer	4
Data tag	4-char code	4
Data	variable	Entry length - 8

Tagged entries defined for ‘rtp ‘ data format:

Data Tag	Data Format
‘tims’	32-bit integer specifying the RTP timescale. This entry is required for RTP data.
‘tsro’	32-bit integer specifying the offset to add to the stored timestamp when sending RTP packets. If this entry is not present, a random offset should be used, as specified by the IETF. If this entry is 0, use an offset of 0 (no offset).

`'snro'` 32-bit integer specifying the offset to add to the sequence number when sending RTP packets. If this entry is not present, a random offset should be used, as specified by the IETF. If this entry is 0, use an offset of 0 (no offset).

Tagged entries defined for any data format:

Data Tag	Data Format
-----------------	--------------------

<code>'rely'</code>	8-bit flag indicating whether this track should or must be sent over a reliable transport, such as TCP/IP. If this entry is not present, unreliable transport should be used, such as RTP/UDP. The current client software for QuickTime streaming will only receive streaming tracks sent using RTP/UDP.
---------------------	---

Bits 0-5 (hex 0-3F) are reserved, and must be `false`.

Bit 6 `true` (40 hex) indicates reliable transport preferred. Use TCP/IP if available, RTP/UDP otherwise.

Bit 7 `true` (80 hex) indicates reliable transport is required. Use TCP/IP if available. Do not send this track over RTP/UDP.

Time-to-Sample Atom (`'stsd'`)

The time-to-sample atom (`'stsd'`) for hint tracks is used normally, as described in the *QuickTime File Format* documentation (Movie Atoms section, Sample Atoms subsection). It contains a table you can use to determine which hint track sample corresponds to what time in the movie.

Synch Samples Atom (`'stss'`)

The synch samples atom (`'stss'`) for hint tracks is used normally, as described in the *QuickTime File Format* documentation (Movie Atoms section, Sample Atoms subsection). It contains a table listing the samples that can be used for random access into the movie (key frame samples). This atom is optional. If it is not present, all sample frames can be used as key frames for this track.

Sample-to-Chunk Atom (`'stsc'`)

The sample-to-chunk atom (`'stsc'`) for hint tracks is used normally, as described in the *QuickTime File Format* documentation (Movie Atoms section,

Sample Atoms subsection). It contains a table you can use to determine which chunk contains a particular sample.

Sample Sizes Atom ('stsz')

The sample sizes atom ('stsz') for hint tracks is used normally, as described in the *QuickTime File Format* documentation (Movie Atoms section, Sample Atoms subsection). If all the samples are the same size, it holds the size in bytes as a 32-bit integer. Otherwise, it contains a table you can use to determine the size of any sample. You need to use this data to determine the byte offset of a given sample within a chunk.

Chunk Offset Table Atom ('stco')

The chunk offset table atom ('stco') for hint tracks is used normally, as described in the *QuickTime File Format* documentation (Movie Atoms section, Sample Atoms subsection). It contains a table with the byte offset of each chunk. The offset is into a file or resource specified indirectly by the data reference index field of the hint track sample description atom ('stsd'). The actual data reference is in the data reference atom ('dref') inside the hint track's data information atom ('dinf').

The Hint Track User Data Atom ('udta')

A hint track may contain a user data atom ('udta') holding information atoms specific to the hint track. This will normally contain three child atoms:

- A name atom ('name') with the track's name (such as "Hinted sound track")
- A hint track information atom ('hinf') with track statistics (such as packet count, packet size, and average bit rate)
- A hint information atom ('hnti') containing the SDP data for this track

Note that there are user data atoms ('udta') at the movie level as well as the track level. The 'hnti' child atom in the movie-level user data atom contains hint information which applies to all tracks in the movie. The 'hnti' child atom of the track-level user data atom contains hint information specific to this track.

The User Data Name Atom ('name')

The user data name atom ('name') for hint tracks is used normally, as described in the *QuickTime File Format* documentation (Movie Atoms section, User Data

Atoms subsection). It contains a string with the name of the track, such as “Hinted sound track”. This string is displayed to identify the track when the “Get Info” window for a movie is opened using MoviePlayer.

Hint Information Atom (`'hinf'`)

The hint information atom (`'hinf'`) contains a series of child atoms holding statistics about the track, such as its maximum packet size, number of packets, and average data rate. The defined statistics atoms are listed below. Child atoms of unknown type should be ignored.

Table 26-2 Defined child atoms of 'hinf' user data atoms

Atom type	Data Size	Description
'trpy'	64 bits	Total number of bytes that will be sent, including 12-byte RTP headers, but not including any network headers
'nump'	64 bits	Total number of network packets that will be sent (if the application knows there is a 28-byte network header, it can multiply 28 by this number and add it to the 'trpy' value to get the true number of bytes sent)
'tpyl'	64 bits	Total number of bytes that will be sent, not including 12-byte RTP headers
'maxr'	8 bytes Two 32-bit values	Maximum data rate in bits per second. This atom contains two numbers: <i>g</i> , followed by <i>m</i> (both 32-bit values). <i>g</i> is the granularity, in milliseconds. <i>m</i> is the maximum data rate for that granularity. For example, if <i>g</i> is 1 second, then <i>m</i> is the maximum data rate over any 1 second. There may be multiple 'maxr' atoms, with different values for <i>m</i> and <i>g</i> . The maximum data rate calculation does not include any network headers, but does include 12-byte RTP headers. Currently, movies hinted using the QuickTime hinter component include 'maxr' atoms for 1 second (<i>m</i> =1000) and 1 minute (<i>m</i> =60000).
'dmed'	64 bits	Total number of bytes from the media track to be sent
'dimm'	64 bits	Number of bytes of immediate data to be sent
'drep'	64 bits	Number of bytes of repeated data to be sent
'tmin'	32 bits	Smallest relative transmission time, in milliseconds. This is a signed 32-bit integer, typically from -2000 to 0. Packets which should be sent earlier than the expected transmission time—for data smoothing, for example—use a negative relative transmission time.
'tmax'	32 bits	Largest relative transmission time, in milliseconds. This is a signed 32-bit integer, typically 0.
'pmax'	32 bits	Largest packet, in bytes, including 12-byte RTP header
'dmax'	32 bits	Longest packet duration, in milliseconds
'payt'	variable	32-bit payload type number, followed by rtpmap payload string (Pascal string)

Hint Track Information Atom ('hnti')

Hint track information atoms ('hnti') are defined both for movie-level user data atoms and for track-level user data atoms. The 'hnti' atom in the hint track's user data atom contains hint information specific to this track.

The 'hnti' atom contains child atoms that hold the actual hint information. Currently, only SDP child atoms ('sdp') are defined. Servers should ignore child atoms whose type they do not recognize.

An SDP child atom has a 32-bit size field, a four-character type ('sdp'), and a data field. Note that the fourth character of the atom type 'sdp' is an ASCII blank (hex 20).

The data field of an SDP atom contains a string with the track-specific information required to build an SDP file, such as "m=audio 0 RTP/AVP 96". The string itself has no count byte or delimiter. Its length is calculated by subtracting 8 from the atom size. A single string can contain multiple lines of SDP text, separated by a carriage-return and linefeed (0D 0A hex).

The SDP text must be inserted in the proper place in the SDP file in order to build an SDP file for a movie. The SDP text for each track is inserted after any common SDP text, including SDP text taken from the movie-level 'hnti' atoms. Refer to the RFC for Session Description Protocol for precise line-ordering rules.

Hint Track Samples

Hint track samples either contain or point to the data to be sent in each packet. Hint track samples are not part of the hint track atom structure, although they are usually found in the same file. Use the hint track data reference atom ('dref') and sample table atom ('stbl') to find the file specification and byte offset for a particular sample.

The hint track structure is generalized to support hint samples in multiple data formats. The data format of a hint track sample is specified by the data format field of the sample description atom ('stsd'). Currently, only the data format type of 'rtp' is defined. Note that the fourth character in the type field is an ASCII blank (hex 20). Hint track samples are not atoms, and do not contain size or type fields. The type is found in the sample description, and the size is found in the sample table.

This section describes hint track samples in 'rtp' data format.

- Each hint track sample is used to generate one or more RTP packets.

- All packets in a sample have the same RTP timestamp, which is also the hint sample time.
- Each sample contains a packet table with one entry for each packet to be sent.
- Each entry in the packet table contains RTP header and sequence information for a particular packet, as well as a data table.
- The data table has one entry for each of block of contiguous data that belongs in the packet.
- A typical packet is constructed from more than one block of data. A typical data table might have an entry for a small block of data contained in the data table itself, a second entry pointing to data in another hint track sample, and a third entry pointing to data in a media sample.

Bear in mind that media samples may not reside in the same file or resource as hint track samples. You locate media samples in the same way you locate hint track samples, by walking the media track atom structure and using the data reference atom ('dref') and sample table atom ('stbl') found there.

Note also that hint track samples may not have a one-to-one correspondence to packets or media samples. A packet may contain several small media samples, and a large media sample may be broken into several packets.

Hint track sample data is byte-aligned, but not necessarily 32-bit aligned. Hint track sample data is always in big-endian format.

Table 26-3 Hint track sample format

Field	Bytes	Description
Entry count	2	16-bit integer specifying the number of entries in the packet table. Each entry specifies an RTP packet. A sample with an entry count of 0 must be skipped.
Reserved	2	Set to zero if creating. Ignore if using.
Packet table	variable	Each entry in this table specifies an RTP packet. Packet table entries contain data tables. See the packet table format below for more information.
Additional data	variable	If data table entries point to data stored in hint samples, that data is stored here.

Packet Table Format

Each entry in the packet table specifies one RTP packet. All packets specified in the table have the same RTP timestamp, which is the hint sample time. Multiple entries indicate that a media sample has been split into multiple packets.

Each packet table entry includes a data table. Each entry in the data table indicates a block of data to include in the packet. A packet table entry can include multiple data table entries.

A packet table entry can include an Extra Information TLV table that can be used to extend the RTP hint sample format on a per-packet basis.

The size of a packet table entry is 12 bytes, plus 16 bytes for each data table entry, plus the size of the Extra Information TLV table, if present. The format of packet table entries is shown below.

Table 26-4 Packet table entry format

Field	Bytes	Description
Relative transmission time	4	32-bit signed integer containing a transmission time offset in hint track timescale units. Add this number to the RTP transmission time, which is also the hint sample time. Negative numbers are used to send data early for data smoothing. Positive numbers are used for repeating data.
RTP header info	2	A 16-bit field specifying various RTP header values. Bits are defined as follows: 0-1 — reserved 2 — corresponds to the 'P' packing bit in the RTP header 3 — corresponds to the 'X' extension bit in the RTP header 4-7 — reserved 8 — corresponds to the 'M' marker bit in the RTP header 9-15 — The seven-bit payload type corresponding to the 'PT' payload type field in the RTP header. Note: Bits 2 and 3 should probably not be set.
RTP sequence number	2	16-bit integer specifying the RTP sequence number for this packet. The server adds a random offset to this number. Packets with different transmission times and the same sequence number can be used to periodically re-transmit data.
Flags	2	16-bit field specifying various attributes of this packet. Bits are defined as follows: 0-12 — reserved 13 — X, if set, this table entry contains Extra Information TLVs 14 — B, if set, this is a B-frame packet. Skip if necessary. 15 — R, if set, this is a repeated packet. Skip if necessary. If your server is running out of real time, it can skip B-frame packets and/or repeated packets until it catches up.
Entry count	2	A 16-bit integer specifying the number of entries in the data table
Extra Information TLVs	variable	If the X bit in the flags field is set, an Extra Information TLV table goes here. The Extra Information TLV table is defined below.
Data table	variable	Each entry in the data table includes or points to a block of data that your server must include in the packet. The data table format is defined below.

Extra Information TLV Table Format

The Extra Information TLV table is present in a packet table entry if, and only if,

the X bit is set in the flags field of that table entry. The Extra Information TLVs provide a way to extend the RTP hint sample format on a per-packet basis.

The Extra Information TLV table has a size field indicating the whole size of the table. Each entry in the table is a TLV, formatted like an atom, with a 4-character type field, a 64-bit size field, and a data field. The data field in a TLV is padded to a four-byte boundary. The size field indicates the whole size of the TLV, including its data, size, and type fields, and any padding.

Walk the TLV table as you would an atom structure, ignoring any TLVs whose type you do not recognize.

Table 26-5 Extra Information TLV table structure

Field	Bytes	Description
Table size	4	32-bit integer indicating the whole size of the table, including this field
TLV 1	variable	Each TLV has a 32-bit size field, a 4-character type field, and a variable-length data field. The data field is padded to a 4-byte boundary.
TLV 2	variable	
...	variable	
TLV n	variable	

The only TLV currently defined is the RTP timestamp offset. This can be used for data types whose RTP timestamp does not monotonically increase. The definition of the RTP timestamp offset TLV follows.

Size	A 64-bit integer containing the size of the TLV in bytes. Size for this TLV is always 12.
Type	'rtpo'
Data	A signed 32-bit integer which should be added to the RTP timestamp for this packet.

Data Tables

Data tables are located inside packet table entries. A data table has one entry for each contiguous block of data to be inserted into the packet. Your server should

concatenate the data from each block. Once all the blocks are inserted, the packet is complete.

Each data table entry is 16 bytes long. The first byte indicates the data source. There are currently four defined sources:

- No Data— No data should be inserted
- Immediate Data — The data follows as part of this data table entry
- Sample Data — The data resides in a track's sample
- Sample Description — The data resides in a track's sample description

The 15 bytes following the source-indicator byte contain either immediate data or the information needed to locate the data. The format of the information depends on the data source. All entries are padded to exactly 16 bytes.

The generic structure of a data table entry is shown below. The information format for each defined data source follows.

Table 26-6 Data table entry format

Field	Bytes	Description
Source	1	An 8-bit integer indicating the source of the data block. The following sources are defined: 0 — No data 1 — Immediate data 2 — Sample data 3 — Sample description data
Information	15	Either the immediate data or the necessary information to find the data, depending on the data source

No Data

If the source field is 0, there is no data. Ignore the following 15 bytes. Insert no data into the packet for this entry.

Field	Bytes	Description
Source	1	Always 0
Unused	15	Padding

Immediate Data

If the source field is 1, the data is in the information field of this entry. There is a 1-byte data length field, followed by up to 14 bytes of data.

Field	Bytes	Description
Source	1	Always 1
Length	1	8-bit value indicating the number of data bytes that follow
Data	14	The number of data bytes indicated by the length field, plus any padding to equal 14 bytes

Sample Data

If the source field is 2, the data is in a track's data sample. The track is either the media track being hinted or the hint track itself.

Field	Bytes	Description
Source	1	Always 2
Track ref index	1	8-bit integer indicating the track reference by index
Length	2	16-bit integer specifying the number of bytes to copy
Sample number	4	32-bit integer specifying the sample number to copy the data from
Offset	4	32-bit integer specifying the byte offset into the sample at which to begin copying data
Bytes per compression block	2	16-bit unsigned integer specifying the number of bytes in a compression block, after compression. A value of 0 is equivalent to a value of 1.
Samples per compression block	2	16-bit unsigned integer specifying the number of uncompressed samples in a compression block. A value of 0 is equivalent to a value of 1.

Source An 8-bit integer, value 2, indicating the data source is sample data

Track ref index
An signed 8-bit integer used to determine which track's sample data should be used.

A value of 0 indicates that the hint track refers to one media

track. The reference is stored in the hint track's 'tref' atom. Use that media track's sample data.

A value of 1-127 indicates that the hint track refers to multiple media tracks. The references are stored in a table in the hint track's 'tref' atom. The value of the track ref index field indicates which entry holds the track ID of the media track whose samples should be used.

A value of -1 (FF hex) indicates that the data resides in a sample in the hint track itself.

Length A 16-bit integer specifying the number of bytes to insert into the packet

Sample number

A 32-bit integer specifying the sample number. Use the sample table atom ('stbl') in the indicated track to find the file specification and byte offset for this sample.

Offset A 32-bit integer specifying where in the sample to begin copying the data from

Bytes per compression block

The number of bytes in a compression block, after compression. If this entry is 0 or 1, compression blocks are not used. See note below.

Samples per compression block

The number of uncompressed samples contained in a compression block. If this entry is 0 or 1, compression blocks are not used. See note below.

Note

Normally, the byte offset for a sample is calculated by determining the sample number within a chunk, then adding the size of the preceding samples in that chunk to the chunk offset. If multiple samples are stored in compression blocks, as is the case with compressed audio, the sample size cannot be used to determine the offset within the chunk. Once you have determined the sample offset (the zero-based sample number within a chunk), the byte offset is determined using this formula:

SO = sample offset (0 if first sample, 1 if second sample, and so on)
 CO = chunk offset
 BCB = bytes per compression block
 SPC = samples per compression block
 byte offset = $(SO * BCB / SPC) + CO$

As an example, GSM audio typically compresses 160 samples into a 33-byte block, so a sample from a GSM audio track would have a BCB of 33 and an SPC of 160. The 161st sample in a chunk would have a sample offset of 160, so the byte offset of the sample would be $(160 * 33 / 160)$ plus the chunk offset.

Sample Description Data

If the source field is 3, the data is in a sample description atom ('stsd') either in the media track being hinted or in the hint track itself.

Field	Bytes	Description
Source	1	Always 3
Track ref index	1	8-bit integer indicating the track reference by index
Length	2	16-bit integer specifying the number of bytes to copy
Sample description index	4	32-bit integer specifying which entry in the sample description table to copy the data from
Offset	4	32-bit integer specifying the byte offset into the sample at which to begin copying data
Reserved	4	4 bytes that must be set to zero.

Source An 8-bit integer, value 3, indicating the data source is a sample description.

Track ref index

An signed 8-bit integer used to determine which track's sample description should be used.

A value of 0 indicates that the hint track refers to one media track. The reference is stored in the hint track's 'tref' atom. Use that media track's sample description.

A value of 1-127 indicates that the hint track refers to multiple media tracks. The references are stored in a table in the hint track's 'tref' atom. The value of the track ref index field

	indicates which entry in the 'tref' atom holds the track ID of the media track whose sample description should be used.
	A value of -1 (FF hex) indicates that the data resides in the sample description of the hint track itself.
Length	A 16-bit integer specifying the number of bytes to insert into the packet
Sample description index	A 32-bit integer specifying the sample description by index. Sample descriptions are table entries in a sample description child atom ('tsd') of a track's sample table atom ('stbl'). The sample description index corresponds to the table entry for a sample description.
Offset	A 32-bit integer specifying the byte in the sample description from which to begin copying data.

Hint Track Information Atom ('hnti') in Movie User Data Atom

Hint track information atoms ('hnti') are defined both for movie-level user data atoms and for track-level user data atoms. The 'hnti' atom in the movie's user data atom contains hint information for the movie as a whole. This atom is not part of the hint track atom structure. It is contained in the user data atom ('udta') belonging to the movie atom that contains the hint track.

The 'hnti' atom contains child atoms that hold the actual hint information. Currently, only child atoms of type 'rtsp', subtype 'sdp' are defined for 'hnti' atoms at the movie level. Servers should ignore child atoms whose type they do not recognize.

An RTP/SDP child atom has a 32-bit size field, a four-character type ('rtsp'), a four-character subtype ('sdp'), and a data field. Note that the fourth character of the type 'rtsp' and the of the subtype 'sdp' are both ASCII blanks (hex 20). Note also that this is different from the child atom of type 'sdp' found in the track-level user data atom.

The data field of an RTP/SDP atom contains a string with text used to build an SDP file, such as "a=x-qt-text-inf:Killer Streaming Movie". The string itself has no count byte or delimiter. It's length is calculated by subtracting 8 from the atom size. A single string can contain multiple lines of SDP text, separated by a carriage-return and linefeed (0D 0A hex).

The SDP text must be inserted in the proper place in the SDP file in order to build an SDP file for this movie. SDP text taken from the movie-level ‘`hnti`’ atoms is inserted prior to SDP text taken from track-level ‘`hnti`’ atoms. Refer to the RFC for Session Description Protocol for precise line-ordering rules.