QuickTime Streaming Server Modules Programming Guide

(Legacy)

 Developer

Contents

Contents

Contents

Contents

Figures, Tables, and Listings

8

About This Manual

Framework	None.
Declared in	QTSS.h

> **Important** Apple recommends that developers explore QTKit and Core Video for new development in this technology area. See *QTKit Framework Reference* and *Core Video Programming Guide* for more information.

This manual describes version 5.0 of the programming interface for creating QuickTime Streaming Server (QTSS) modules for the open source Darwin Streaming Server. The QTSS programming interface provides an easy way for developers to add new functionality to the Streaming Server. This version of the programming interface is compatible with QuickTime Streaming Server version 5.5.

This chapter describes the callback rotuines and data types that modules use to call the QuickTime Streaming Server.

What's New

Version 5.0 of the QTSS programming interface provides the following new features:

- These new internal server preferences have been added: `disable_thinning`, `player_requires_rtp_header_info`, and `player_requires_bandwidth_adjustment`.

- These new preferences have been added to the `QTSSFileModule` module for compatibility with 3rd Generation Partnership Project (3GPP) players: `compatibility_adjust_sdp_media_bandwidth_percent` and `enable_player_compatibility`.

- RTP play information is now enabled by default in the `QTSSReflectorModule` module. Use the new preference, `disable_rtp_play_info` to disable RTP Play information. Another new `QTSSReflectorModule` preference is `reflector_rtp_info_offset_msec`. For compatibility with 3GPP players, these preferences have also been added to the `QTSSReflectorModule` module: `enable_play_response_range_header`, `enable_player_compatibility`, and `force_rtp_info_sequence_and_time`.

- This new preference has been added to the `QTSSFlowControlModule` module: `flow_control_udp_thinning_module_enabled`.

The following changes have been made to existing preferences:

- Default size of the `QTSSFileModule` preference `shared_buffer_unit_k_size` has been increased from 32 to 64.

- Default size of the `QTSSFileModule` preference `private_buffer_unit_k_size` has been increased from 32 to 64.

The `enable_rtp_play_info` preference has been removed from the `QTSSReflectorModule` module.

The `–D` command line option has been added to the StreamingServer. When specified, the `–D` option outputs performance status information.

The file `streamingloadtool.conf`, installed in `/Library/QuickTimeStreaming/Config`, has new file tags:

- `player` *text*, where *text* is the name of the RTSP player. The information is sent to the server as the UserAgent header.

- `sendoptions` *setting*, where *setting* is `yes` or `no`. If `yes`, a Send Options request is made before the `DESCRIBE` statement.

- `requestrandomdata` *setting*, where *setting* is `yes` or `no`. Set *setting* to `yes` to ask for random data from the server.

- `randomdatasize` *setting*, where *setting* is a number from 0 to 262144 that specifies the number of random bytes the server should send.

Conventions Used in This Manual

The `Letter Gothic` font is used to indicate text that you type or see displayed. This manual includes special text elements to highlight important or supplemental information:

> **Note:** Text set off in this manner presents sidelights or interesting points of information.

> **Important:**   Text set off in this manner—with the word Important—presents important information or instructions.

> ⚠ **Warning:** Text set off in this manner—with the word Warning—indicates potentially serious problems.

| 2009-06-01 | © 2002, 2009 Apple Inc. All Rights Reserved.

10

For More Information

Go to http://www.opensource.apple.com to register as a member of the Apple open source community. Then download the source code for the Darwin Streaming Server at http://www.publicsource.apple.com/projects/streaming. The source code's Documentation directory contains valuable information:

- `AboutTheSource.html`

- `DevNotes.html`

- `SourceCodeFAQ.html`

The following RFCs provide additional information of interest to developers of QuickTime Streaming Server modules and are available at many locations on the Internet:

- RFC 2326, Real Time Streaming Protocol (RTSP)

- RFC 1889, RTP: A Transport Protocol for Real-Time Applications

- RFC 2327, SDP: Session Description Protocol

- RFC 2616, HTTP 1.1

For an overview of the Darwin Streaming Server and links to the latest QuickTime information, go to http://developer.apple.com/darwin/projects/streaming.

Go to http://developer.apple.com/documentation/quicktime for QuickTime developer documentation.

Communicate with other Darwin Streaming Server developers by joining the discussion list at http://lists.apple.com/mailman/listinfo/streaming-server-dev.

See what with other Darwin Streaming Server developers are doing by joining the discussion list at http://lists.apple.com/mailman/listinfo/publicsource-modifications.

Concepts

This manual describes version 4.0 of the programming interface for creating QuickTime Streaming Server (QTSS) modules. This version of the programming interface is compatible with QuickTime Streaming Server version 5.

QTSS is an open-source, standards-based streaming server that runs on Windows NT and Windows 2000 and several UNIX implementations, including Mac OS X, Linux, FreeBSD, and the Solaris operating system. To use the programming interface for the QuickTime Streaming Server, you should be familiar with the following Internet Engineering Task Force (IETF) protocols that the server implements:

- Real Time Streaming Protocol (RTSP)
- Real Time Transport Protocol (RTP)
- Real Time Transport Control Protocol (RTCP)
- Session Description Protocol (SDP)

This manual describes how to use the QTSS programming interface to develop QTSS modules for the QuickTime Streaming Server. Using the programming interface described in this manual allows your application to take advantage of the server's scalability and protocol implementation in a way that will be compatible with future versions of the QuickTime Streaming Server. Most of the core features of the QuickTime Streaming Server are implemented as modules, so support for modules has been designed into the core of the server.

You can use the programming interface to develop QTSS modules that supplement the features of the QuickTime Streaming server. For example, you could write a module that

- acts as an RTSP proxy, which would be useful for streaming clients located behind a firewall
- supports virtual hosting, allowing a single server to serve multiple domains from multiple document roots.
- logs statistical information for particular RTSP and client sessions
- supports additional ways of storing content, such as storing movies in databases
- configures users' QuickTime Streaming Server preferences
- monitors and reports statistical information in real time
- tracks pay-per-view accounting information

Server Architecture

The Streaming Server consists of one parent process that forks a child process, which is the core server. The parent process waits for the child process to exit. If the child process exits with an error, the parent process forks a new child process.

The core server acts as an interface between network clients, which use RTP and RTSP to send requests and receive responses, and server modules, which process requests and send packets to the client. The core server does its work by creating four types of threads:

- the server's own Main thread. The Main thread checks to see if the server needs to shut down, log status information, or print statistics.

- the Idle Task thread. The Idle Task thread manages a queue of tasks that occur periodically. There are two types of task queues: timeout tasks and socket tasks.

- the Event thread. The Event thread listens for socket events such as a received RTSP request or RTP packet and forwards them to a Task thread.

- one or more Task threads. Tasks threads receive RTSP and RTP requests from the Event thread. Tasks threads foward requests to the appropriate server module for processing and send packets to the client. By default, the core server creates one Task thread per processor.

Figure 1-1 summarizes the relationship between clients, the core server's threads, and server modules.

Figure 1-1   Server architecture
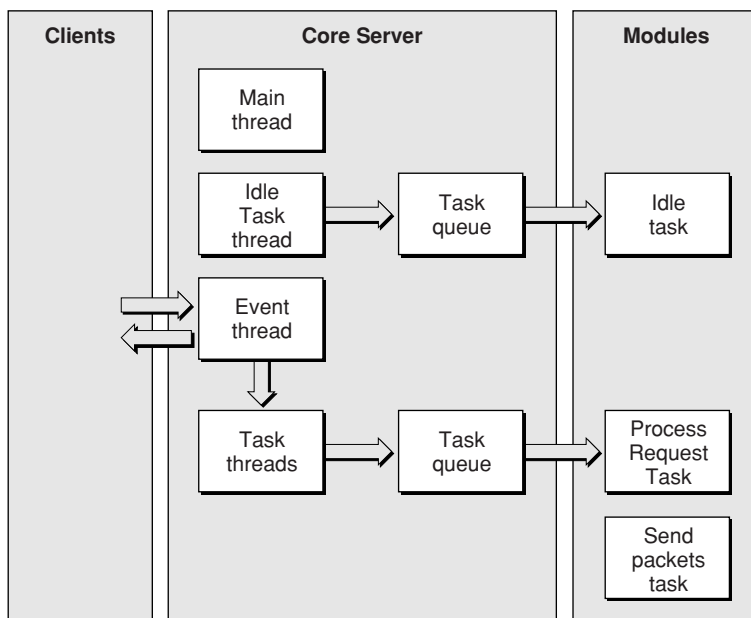
Because the server is largely asynchronous, there needs to be a communication mechanism for events. For instance, when a socket used for an RTSP connection gets data, something has to be notified so that data can be processed. The Task object is a generalized mechanism for performing this communication.

Each Task object has two major methods: Signal and Run. Signal is called by the server to send an event to a Task object. Run is called to give time to the Task for processing the event.

The goal of each Task object is to implement server functionality using small non-blocking time slices. Run is a pure virtual function that is called when a Task object has events to process. Inside the Run function, the Task object can call GetEvents to receive and automatically dequeue all its current and previously signaled events. The Run function is never re-entered: if a Task object calls GetEvents in its Run function, and is then signaled before the Run function completes, the Run function will be called again for the new event only after exiting the function. In fact, the Task's Run function will be called repeatedly until the all the Task object's events have been cleared with GetEvents.

This core concept of event-triggered tasks is integrated into almost every Streaming Server subsystem. For example, a Task object can be associated with a Socket object. If the Socket gets an event (through a select() notification or through the Mac OS X Event Queue, the corresponding Task object will be signaled. In this case, the body of the Run function will contain the code for processing whatever event was received on that Socket.

Task objects make it possible for the Streaming Server use a singlethread to run all connections, which is the Streaming Server's default configuration on a single processor system.

Modules

The Streaming Server uses modules to respond to requests and complete tasks. There are three types of modules:

Content-Managing Modules

The content-managing modules manage RTSP requests and responses related to media sources, such as a file or a broadcast. Each module is responsible for interpreting the client's request, reading and parsing their supported files or network source, and responding with RTSP and RTP. In some cases, such as the mp3 streaming module, the module uses HTTP.

The content-managing modules are QTSSFileModule, QTSSReflectorModule, QTSSRelayModule, and QTSSMP3StreamingModule.

| 2009-06-01 | © 2002, 2009 Apple Inc. All Rights Reserved.

14

Server-Support Modules

The server-support modules perform server data gathering and logging functions. The server-support modules are QTSSErrorLogModule, QTSSAccessLogModule, QTSSWebStatsModule, QTSSWebDebugModule, QTSSAdminModule, and QTSSPOSIXFileSystemModule.

Access Control Modules

The access control modules provide authentication and authorization functions as well as URL path manipulation.

The access control modules are QTSSAccessModule, QTSSHomeDirectoryModule, QTSSHttpFileModule, and QTSSSpamDefenseModule.

Protocols

The Streaming Server supports the following protocols:

- RTSP over TCP. The Real Time Streaming Protocol (RTSP) is a client-server multimedia presentation control protocol designed to provide efficient delivery of streamed multimedia over IP networks. RTSP provides a basis for negotiating unicast and multicast transport protocols, such as RTP, and negotiates codecs in a file format independent way. It works well for large audiences as well as single-viewer media-on-demand. RFC 2326 defines the IETF standard for RTSP.

- RTP over UDP. The Realtime Transport Protocol (RTP) is a packet format for multimedia data streams. RTP is used by many standard protocols, such as RTSP for streaming applications and SDP for multicast applications. It provides the data delivery format for RTSP and SDP. RFC 1889 defines the IETF proposed standard for RTP.

- RTP over Apple's Reliable UDP. If an RTP client requests it, the server sends RTP packets using Reliable UDP. Reliable UDP is a set of quality of service enhancements, such as congestion control tuning improvements, retransmit, and thinning server algorithms, that improve the ability to present a good quality RTP stream to RTP clients even in the presence of packet loss and network congestion. For more information, see "Reliable UDP" (page 123).

- RTSP/RTP in HTTP (tunneled). Firewalls often prevent users on private IP networks from receiving QuickTime presentations. On private networks, an HTTP proxy server is often configured to provide users with indirect access to the Internet. To reach such clients, QuickTime 4.1 supports the placement of RTSP and RTP data in HTTP requests and replies, allowing viewers behind firewalls to access QuickTime presentations through HTTP proxy servers. For more information, see "Tunneling RTSP and RTP Over HTTP" (page 125).

- RTP over RTSP (RTP over TCP). Certain firewall designs and other circumstances may require a server to use alternative means to send data to clients. RFC 2326 allows RTSP packets destined for the same control end point to be packed into a single lower-layer protocol data unint (PDU), encapsulated into a TCP stream, or interleaved with RTP and RTCP packets. Interleaving complicates client and server operation and imposes additional overhead and should only be used if RTSP is carried over TCP. RTP packets are encapsulated by

an ASCII dollar sign ($), followed by a one-byte channel identifier (defined in the transport header using the interleaved parameter), followed by the length of the encapsulated binary data as a binary, two-byte integer in network byte order. The stream data follows immediately, without a CRLF, but including the upper-layer protocol headers. Each $ block contains exactly one RTP packet. When the transport is RTP, RTCP messages are also interleaved by the server over the TCP connection. By default, RTCP packets are sent on the first available channel higher than the RTP channel. The client may request RTCP packets on another channel explicitly. This is done by specifying two channels in the interleaved parameter of the transport header. RTCP is used for synchronization when two or more streams are interleaved. Also, this provides a convenient way to tunnel RTP/RTCP packets through the TCP control connection when required by the network configuration and transfer them onto UDP when possible.

In addition, the following modules implement HTTP:

- QTSSAdminModule

- QTSSMP3StreamingModule

- QTSSWebStatsModule

- QTSSHTTPStreamingModule

- QTSSRefMovieModule

- QTSSWebStats

- QTSSWebDebugModule

Data

When a module needs access to a request's RTSP header, it gains access to the request through a request object defined by the QTSS.h API header file. For example, the RTSPRequestInterface class implements the API dictionary elements accessible by the API. Objects whose name ends with "Interface", such as RTSPRequestInterface, RTSPSessionInterface, and QTSSServerInterface, implement the module's API.

The following interface classes are significant:

- QTSServerInterface — This is the internal data storage object tagged as the QTSS_ServerObject in the API. Each of the QTSS_ServerAttributes in the API is declared and implemented in this base class.

- RTSPSessionInterace — This is the internal data storage object tagged as the qtssRTSPSessionObjectType in the API. Each of the QTSS_RTSPSessionAttributes in the API is declared and implemented in this base class.

- RTPSessionInterface — This is the internal data storage object tagged as the QTSS_ClientSessionObject in the API. Each of the QTSS_ClientSessionAttributes in the API is declared and implemented in this base class.

- RTSPRequestInterface — This is the internal data storage object tagged as the QTSS_RTSPRequestObject in the API. Each of the QTSS_RTSPRequestAttributes in the API is declared and implemented in this base class.

Classes

Figure 1-2 shows how the objects in the server reference each other.

Figure 1-2 Server object data model
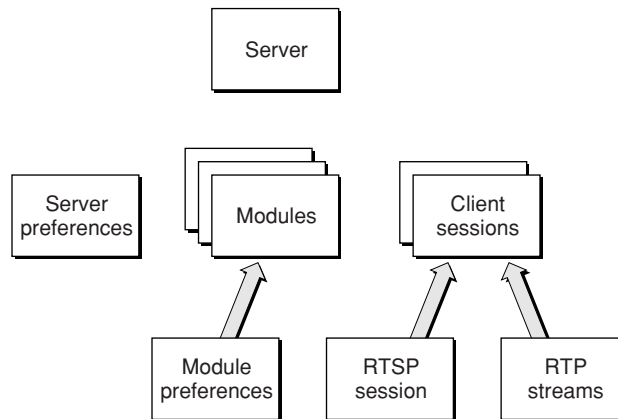
The server object has a a dictionary of preferences. The server owns a list of modules each with a dictionary for their preferences. The server owns a list of RTP client sessions, each of which can have an RTSP session and one or more RTP media streams. It is possible to use the API to walk all of the server's the live sessions and streams.

- QTServer is the core server object, some of which is accessible through the API and the QTSSServerInterface base class.

- Dictionary is a data storage base class that implements key and value access to object data. This base class is inherited by all server objects defined by the API.

- Module is a class for managing modules. Each module instance is responsible for loading, initializing, and executing a static or dynamic API module.

- RTSP and RTP sessions. Reads and writes are managed by the sessions through a stream object. The RTSP session calls each of the modules in their registered RTSP role fromthe session's RTSPSession::Run method. The API module roles that are called are `QTSS_RTSPFilter_Role`, `QTSS_RTSPRoute_Role`, `QTSS_RTSPAuthenticate_Role`, `QTSS_RTSPAuthorize_Role`, `QTSS_RTSPPreProcessor_Role`, `QTSS_RTSPRequest_Role`, `QTSS_RTSPPostProcessor_Role`, and `QTSS_RTSPSessionClosingRole`. The RTSP session also calls modules in their `QTSS_RTSPIncomingData_Role`. The RTP session handles

the following role calls as well as data reads and writes: `QTSS_RTPSendPackets_Role`, `QTSS_RTCPProcess_Role`, and `QTSS_ClientSessionClosing_Role`. For more information about roles, see "Module Roles" (page 30).

Applications and Tools

The Streaming Server comes with the following applications and tools:

- PlayListBroadcaster
- MP3Broadcaster
- StreamingProxy
- QTFileTools (POSIX and Mac OS X only; not maintained)
- WebAdmin
- `qtpasswd`

PlayListBroadcaster

PlaylistBroadcaster broadcasts QuickTime, MPEG4, and 3GPP streaming files to a streaming server, such as QuickTime Streaming Server, which then reflects the media to clients. This lets you create a virtual radio station or TV broadcast that appears to users as a live broadcast of the media.

MP3Broadcaster

The MP3Broadcaster application broadcasts an MP3 file as if it were a live broadcast.

StreamingProxy

POSIX and Mac OS X only.

QTFileTools

QTFileTools are movie-inspection utilities that use the QTFile library. The utillities are:

- QTBroadcaster. This utility requires a target IP address, a source movie having one or more hint track IDs, and an initial port. Every packet referenced by the hint track(s) is broadcast to the specified IP address.

- QTFileInfo. This utility requires a source movie. It displays the movie's name, creation date, and modification date. If the track is a hint track, the utility also displays the total RTP bytes and packets, the average bit rate and packet size, and the total header percentage of the stream.

- QTFileTest. This utility requires a source movie. It parses the Movie Header Atom and displays a trace of the output.

- QTRTPGent. This utility requires a source movie having a hint track ID. It displays the number of packets in each hint track sample and writes the RTP packets to a file named `track.cache`.

- QTRTPFileTest. This utility requires a source movie having a hint track ID. It displays the RTP header (TransmitTime, Cookie, SeqNum, and TimeStamp) for each packet.

- QTSampleLister. This utility requires a source movie and a track ID. It displays the track media sample number, media time, data offset, and sample size for each sample in the track.

- QTSDPGen. This utility requires a list of one or more source movies. It displays the SDP information for all of the hinted tracks in each movie. Use the `−f` option to save the SDP information to the file *moviename* `.sdp` in the same directory as the source movie.

- QTTrackInfo. This utility requires a source movie, a sample table atom type (`stco`, `stsc`, `stsz`, or `stts`) and a track ID. It displays the information in the sample table atom of the specified track.

The following example displays the chunk offset sample table in track 3:

```
./QTTrackInfo −T stco /movies/mystery/.mov 3
```

WebAdmin

WebAdmin is a Perl-based web server. Connect a browser to it, and you can administer the server.

qtpasswd

The `qtpasswd` application generates password files for access control.

Source Organization

The Streaming Server source code is written entirely in C++ and pervasively uses object-oriented concepts such as inheritance and polymorphism. Almost exclusively, there is one C++ class per .h / .cpp file pair, and those file names match the class name.The Streaming Server source is organized as follows:

- "Server.tproj" (page 20)
- "CommonUtilitiesLib" (page 20)
- "QTFileLib" (page 21)
- "APICommonCode" (page 21)
- "APIModules" (page 21)
- "RTSPClientLib" (page 21)
- "RTCPUtilitiesLib" (page 21)
- "APIStubLib" (page 21)

- "HTTPUtilitiesLib" (page 21)

Server.tproj

This directory contains the core server code, which can be divided into three subsystems:

- Server core. Classes in this subsystem are prefixed by QTSS. QTSServer handles startup and shutdown. QTSServerInterface stores server globals and compiles server statistics. QTSSPrefs is a data store for server preferences. QTSSModule, QTSSModuleInterface, and QTSSCallbacks are classes whose sole purpose is to support the QTSS module API.

- RTSP subsystem. These classes handle the parsing and processing of RTSP requests, and implement the RTSP part of the QTSS module API. Several of the classes correspond directly to elements of the QTSS API (for instance, RTSPRequestInterface is a QTSS_RTSPRequestObject). There is one RTSP session object per RTSP TCP connection. The RTSPSession object is a Task object that processes RTSP related events.

- RTP subsystem. These classes handle the sending of media data. The RTPSession object contains the data associated with each RTSP session ID. Each RTPSession is a Task object that can be scheduled to send RTP packets. The RTPStream object represents a single RTP stream. Any number of RTPStream objects can be associated with a single RTPSession. These two objects implement the RTP specific parts of the QTSS module API.

CommonUtilitiesLib

This directory contains a toolkit of thread management, data structure, networking, and text parsing utilities. Darwin Streaming Server and associated tools use these classes to reduce repeated code by abstracting similar or identical tasks, to simplify higher level code through encapsulation, and to separate out platform-specific code. Here is a short description of the classes in the CommonUtilitiesLib directory:

- OS Classes. These classes provide platform-specific code abstractions for timing, condition variables, mutexes, and threads. The classes are OS, OSCond, OSMutex, OSThread, and OSFileSource. The data structures are OSQueue, OSHashTable, OSHeap, and OSRef.

- Sockets. These classes provide platform-specific code abstractions for TCP and UDP networking. Socket classes are generally asynchronous (or non-blocking), and can send events to Task objects. The classes are EventContext, Socket, UDPSocket, UDPDemuxer, UDPSocketPool, TCPSocket, and TCPListenerSocket.

- Parsing Utilities. These classes parse and format text. The classes are StringParser, StringFormatter, StrPtrLen, and StringTranslator.

- Tasks: These classes implement the server's asynchronous event mechanism.

QTFileLib

A major feature of the Streaming Server is its ability to serve hinted QuickTime movie files over RTSP and RTP. This directory contains source code for the QTFile library, which contains all of the code for parsing hinted QuickTime files. The server's RTPFileModule calls the QTFile library to retrieve packets and meta-data from hinted QuickTime files. The QTFile library parses the following movie file types: .mov, .mp4 (a modification of .mov), and .3gpp (a modification of .mov).

APICommonCode

This directory contains source code for API-related classes, such as moduletils, or common module functions, such as log file management.

APIModules

This directory contains a directory for each Streaming Server module.

RTSPClientLib

This directory contains source code that implements the server's RTSP client, which can be used to connect to the server using any of the supported protocols.

RTCPUtilitiesLib

This directory contains source code for parsing RTCP requests.

APIStubLib

This directory contains API definition and support files.

HTTPUtilitiesLib

This directory contains source code for parsing HTTP requests.

Server Preference Naming

The file `QTSS,h` defines server preferences. Each server preference defined in `QTSS.h` has a name, such as `qtssPrefsEnableMonitorStatsFile`, a numeric ID, such as 57, and a string constant, such as `enable_monitor_stats_file`.

To get the current setting of server preferences, the server reads the file `StreamingServer.xml` when it starts up or when signaled to reread that file. In the `StreamingServer.xml` file, string constant names are used to refer to preferences.

| 2009-06-01 | © 2002, 2009 Apple Inc. All Rights Reserved.

21

The modules that come with the server use the built-in preference file support provided by the API to generate preferences and a unique ID if the preference is not already defined in the module's preference object. The check and creation of preferences is usually done in `QTSS_Initialize_Role` but the code to generate the preference and an ID from the string name for the preference is also run in `QTSS_RereadPrefs_Role`. The modules that come with the server use the `QTSSModuleUtils` object to encapsulate API calls such as `QTSS_AddInstanceAttribute` and `QTSS_GetAttrInfoByName`.

Module developers who want to use theserver's built-in preference storage support should use the utility method `QTSSModuleUtils::GetAttribute` or examine and call the QTSS API callbacks used by `QTSSModuleUtils::GetAttribute`. The implementation and header file for `QTSSModuleUtils` can be found in the `APICommonCode` directory.

Requirements for Modules

Every QTSS module must implement two routines:

- a main routine, which the server calls when it starts up to initialize the QTSS stub library with your module
- a dispatch routine, which the server uses when it calls the module for a specific purpose

Main Routine

Every QTSS modules must provide a main routine. The server calls the main routine as the server starts up and uses it to initialize the QTSS stub library so the server can invoke your module later.

For modules that are compiled into the server, the address of the module's main routine must be passed to the server's module initialization routine, as described in the section "Compiling a QTSS Module into the Server".

The body of the main routine must be written like this:

```
QTSS_Error MyModule_Main(void* inPrivateArgs)
{
    return _stublibrary_main(inPrivateArgs, MyModuleDispatch);
}
```

where MyModuleDispatch is the name of the module's dispatch routine, which is described in the following section, "Dispatch Routine" (page 23).

> **Important:** For code fragment modules, the main routine must be named `MyModule_Main` where `MyModule` is the name of the file that contains the module.

Dispatch Routine

Every QTSS module must provide a dispatch routine. The server calls the dispatch routine when it invokes a module for a specific task, passing to the dispatch routine the name of the task and a task-specific parameter block. (The programming interface uses the term "role" to describe specific tasks. For information about roles, see "Module Roles" (page 30).)

The dispatch routine must have the following prototype:

```
void MyModuleDispatch(QTSS_Role inRole, QTSS_RoleParamPtr inParams);
```

where MyModuleDispatch is the name specified as the name of the dispatch routine by the module's main routine, `inRole` is the name of the role for which the module is being called, and `inParams` is a structure containing values of interest to the module.

Overview of QuickTime Streaming Server Operations

The QuickTime Streaming Server works with modules to process requests from clients by invoking modules in a particular role. Each role is designed to perform a particular task. This section describes how the server works with roles when it starts up and shuts down and how the server works with roles when it processes client requests.

| 2009-06-01 | © 2002, 2009 Apple Inc. All Rights Reserved.

23

Server Startup and Shutdown

Figure 2-1 shows how the server works with the Register, Initialize, and Shutdown roles when the server starts up and shuts down.

Figure 1-3   QuickTime Streaming Server startup and shutdown
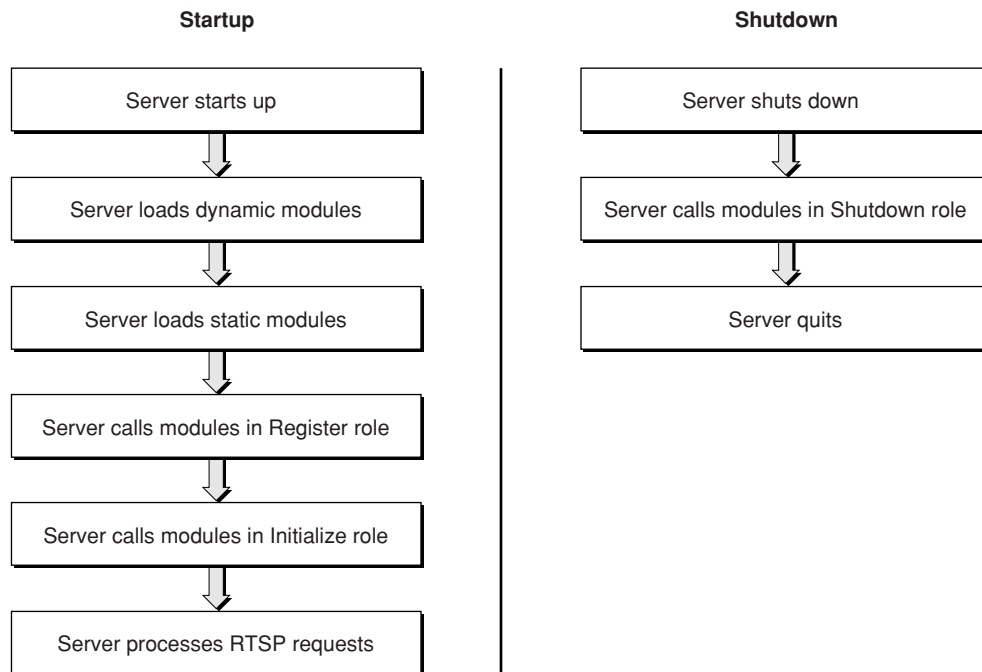
When the server starts up, it first loads modules that are not compiled into the server (dynamic modules) and then loads modules that are compiled into the server (static modules). If you are writing a module that replaces existing server functionality, compile it as a dynamic module so that it is loaded first.

Then the server invokes each QTSS module in the Register role, which is a role that every module must support. In the Register role, the module calls `QTSS_AddRole` to specify the other roles that the module supports.

Next, the server invokes the Initialize role for each module that has registered for that role. The Initialize role performs any initialization tasks that the module requires, such as allocating memory and initializing global data structures.

At shutdown, the server invokes the Shutdown role for each module that has registered for that role. When handling the Shutdown role, the module should perform cleanup tasks and free global data structures.

24

RTSP Request Processing

After the server calls each module that has registered for the Initialize role, the server is ready to receive requests from the client. These requests are known as RTSP requests. A sample RTSP request is shown in Figure 1-4.

Figure 1-4   Sample RTSP request

```
DESCRIBE rtsp://streaming.site.com/foo.mov RTSP/1.0
CSeq: 1
Accept: application/sdp
User-agent: QTS/1.0
```

When the server receives an RTSP request, it creates an RTSP request object, which is a collection of attributes that describe the request. At this point, the `qtssRTSPReqFullRequest` attribute is the only attribute that has a value and that value consists of the complete contents of the RTSP request.

Next, the server calls modules in specific roles according to a predetermined sequence. That sequence is shown in Figure 1-5 (page 26).

Note: The order in which the server calls any particular module for any particular role is undetermined.

Figure 1-5 Summary of RTSP request processing
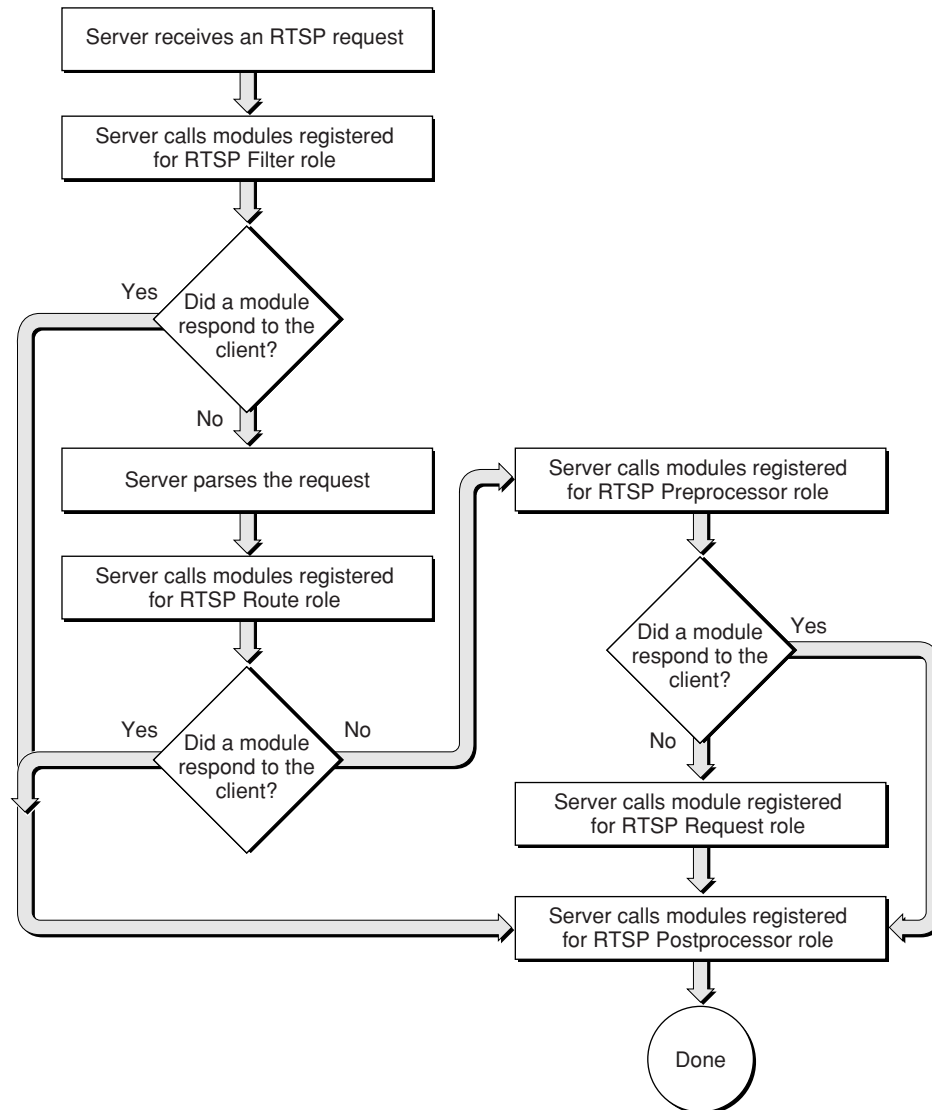
When processing an RTSP request, the first role that the server calls is the RTSP Filter role. The server calls each module that has registered for the RTSP Filter role and passes to it the RTSP request object. Each module's RTSP Filter role has the option of changing the value of the `qtssRTSPReqFullRequest` attribute. For example, an RTSP Filter role might change `/foo/foo.mov` to `/bar/bar.mov`, thereby changing the folder that will be used to satisfy this request.

> **Important:** Any module handling the RTSP Filter role that responds to the client causes the server to skip other modules that have registered for the RTSP Filter role, skip modules that have registered for other RTSP roles, and immediately call the RTSP Postprocessor role of the responding module. A response to a client is defined as any data the module may send to the client.

When all RTSP Filter roles have been invoked, the server parses the request. Parsing the request consists of filling in the remaining attributes of the RTSP object and creating two sessions:

- an RTSP session, which is associated with this particular request and closes when the client closes its RTSP connection to the server

- a client session, which is associated with the client connection that originated the request and remains in place until the client's streaming presentation is complete

After parsing the request, the server calls the RTSP Route role for each module that has registered in that role and passes the RTSP object. Each RTSP Route role has the option of using the values of certain attributes to determine whether to change the value of the `qtssRTSPReqRootDir` attribute, thereby changing the folder that is used to process this request. For example, if the language type is French, the module could change the `qtssRTSPReqRootDir` attribute to a folder that contains the French version of the requested file.

> **Important:** Any module handling the RTSP Route role that responds to the client causes the server to skip other modules that have registered for the RTSP Route role, skip modules that have registered for other RTSP roles, and immediately calls the RTSP Postprocessor role of the responding module.

After all RTSP Route roles have been called, the server calls the RTSP Preprocessor role for each module that has registered for that role. The RTSP Preprocessor role typically uses the `qtssRTSPReqAbsoluteURL` attribute to determine whether the request matches the type of request that the module handles.

If the request matches, the RTSP Preprocessor role responds to the request by calling `QTSS_Write` or `QTSS_WriteV` to send data to the client. To send a standard response, the module can call `QTSS_SendStandardRTSPResponse`, or `QTSS_AppendRTSPHeader` and `QTSS_SendRTSPHeaders`.

> **Important:** Any module handling the RTSP Preprocessor role that responds to the client causes the server to skip other modules that have registered for the RTSP Preprocessor role, skip modules that have registered for other RTSP roles, and immediately calls the RTSP Postprocessor role of the responding module.

If no RTSP Preprocessor role responds to the RTSP request, the server invokes the RTSP Request role of the module that successfully registered for this role. (The first module that registers for the RTSP Request role is the only module that can register for the RTSP Request role.) The RTSP Request role is responsible for responding to all RTSP Requests that are not handled by modules registered for the RTSP Preprocessor role.

After the RTSP Request role processes the request, the server calls modules that have registered for the RTSP Postprocessor role. The RTSP Postprocessor role typically performs accounting tasks, such as logging statistical information.

A module handling the RTSP Preprocessor or RTSP Request role may generate the media data for a particular client session. To generate media data, the module calls `QTSS_Play`, which causes that module to be invoked in the RTP Send Packets role, as shown in .

Figure 1-6   Summary of the RTSP Preprocessor and RTSP Request roles
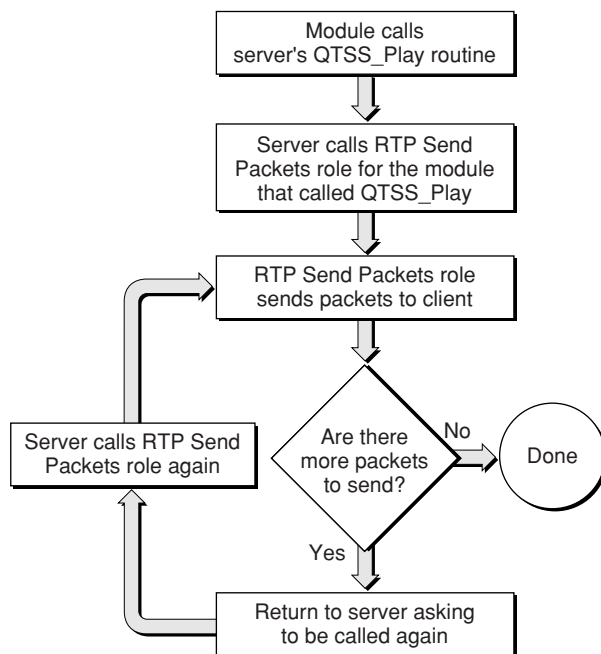
The RTP Send Packets role calls `QTSS_Write` or `QTSS_WriteV` to send data to the client over the RTP session. When the RTP Send Packets role has sent some packets, it returns to the server and specifies the time that is to elapse before the server calls the module's RTP Send Packets role again. This cycle repeats until all of the packets for the media have been sent or until the client requests that the client session be paused or torn down.

Runtime Environment for QTSS Modules

QTSS modules can spawn threads, use mutexes, and are completely free to use any operating system tools.

The QuickTime Streaming Server is fully multi-threaded, so QTSS modules must be prepared to be preempted. Global data structures and critical sections in code should be protected with mutexes. Unless otherwise noted, assume that preemption can occur at any time.

The server usually runs all activity from very few threads or possibly a single thread, which requires the server to use asynchronous I/O whenever possible. (The actual behavior depends on the platform and how the administrator configures the server.)

QTSS modules should adhere to the following rules:

- Perform tasks and return control to the server as quickly as possible. Returning quickly allows the server to load balance among a large number of clients.

- Be prepared for `QTSS_WouldBlock` errors when performing stream I/O. The `QTSS_Write`, `QTSS_WriteV`, and `QTSS_Read` callback routines return `QTSS_WouldBlock` if the requested I/O would block. For more information about streams, see "QTSS Streams" (page 92).

- Avoid using synchronous I/O wherever possible. An I/O operation that blocks may affect streaming quality for other clients.

Server Time

The QuickTime Streaming Server handles real-time delivery of media, so many elements of QTSS module programming interface are time values.

The server's internal clock counts the number of milliseconds that have elapsed since midnight, January 1st, 1970. The data type `QTSS_TimeVal` is used to store the value of the server's internal clock. To make it easy to work with time values, every attribute, parameter, and callback routine that deals with time specifies the time units explicitly. For example, the `qtssRTPStrBufferDelayInSecs` attribute specifies the client's buffer size in seconds. Unless otherwise noted, all time values are reported in milliseconds from the server's internal clock using a `QTSS_TimeVal` data type.

To get the current value of the server's clock, call `QTSS_Milliseconds` or get the value of the `qtssSvrCurrentTimeMilliseconds` attribute of the server object (`QTSS_ServerObject`). To convert a time obtained from the server's clock to the current time, call `QTSS_MilliSecsTo1970Secs`.

Naming Conventions

The QTSS programming interface uses a naming convention for the data types that it defines. The convention is to use the size of the data type in the name. Here are the data types that the QTSS programming interface uses:

- `Bool16` — A 16-bit Boolean value

- `SInt64` — A signed 64-bit integer value

- `SInt32` — A signed 32-bit integer value

- `UInt16` — An unsigned 16-bit integer value

- `UInt32` — An unsigned 32-bit integer value

Parameters for callback functions defined by the QTSS programming interface follow these naming conventions:

- Input parameters begin with `in`.

- Output parameters begin with `out`.

- Parameters that are used for both input and output begin with `io`.

Module Roles

Roles provide modules with a well-defined state for performing certain types of processing. A selector of type `QTSS_Role` defines each role and represents the internal processing state of the server and the number, accessibility, and validity of server data. Depending on the role, the server may pass to the module one or more QTSS objects. In general, the server uses objects to exchange information with modules. For more information about QTSS objects, see "QTSS Objects" (page 44).

Table 1-1 (page 30) lists the roles that this version of the QuickTime Streaming Server supports.

Table 1-1   Module roles

Name	Constant	Task
Register role	`QTSS_Register_Role`	Registers the roles the module supports.
Initialize role	`QTSS_Initialize_Role`	Performs tasks that initialize the module.
Shutdown role	`QTSS_Shutdown_Role`	Performs cleanup tasks.
Reread Preferences role	`QTSS_RereadPrefs_Role`	Rereads the modules's preferences.
Error Log role	`QTSS_ErrorLog_Role`	Logs errors.
RTSP Filter role	`QTSS_RTSPFilter_Role`	Makes changes to the contents of RTSP requests.
RTSP Route role	`QTSS_RTSPRoute_Role`	Routes requests from the client to the appropriate folder.
RTSP Preprocessor role	`QTSS_RTSPPreProcessor_-Role`	Processes requests from the client before the server processes them.

Name	Constant	Task
RTSP Request role	`QTSS_RTSPRequest_Role`	Processes a request from the client if no other role responds to the request.
RTSP Postprocessor role	`QTSS_RTSPPostProcessor_-Role`	Performs tasks, such as logging statistical information, after a request has been responded to.
RTP Send Packets role	`QTSS_RTPSendPackets_-Role`	Sends packets.
Client Session Closing role	`QTSS_-ClientSessionClosing_-Role`	Performs tasks when a client session closes.
RTCP Process role	`QTSS_RTCPProcess_Role`	Processes RTCP receiver reports.
Open File Preprocess role	`QTSS_-OpenFilePreProcess_Role`	Processes requests to open files.
Open File role	`QTSS_OpenFile_Role`	Processes requests to open files that are not handled by the Open File Preprocess role.
Advise File role	`QTSS_AdviseFile_Role`	Responds when a module (or the server) calls the `QTSS_Advise` callback for a file object.
Read File role	`QTSS_ReadFile_Role`	Reads a file.
Request Event File role	`QTSS_RequestEventFile_-Role`	Handles requests for notification of when a file becomes available for reading.
Close File role	`QTSS_CloseFile_Role`	Closes a file that was previously opened.

With the exception of the Register, Shutdown, and Reread Preferences roles, when the server invokes a module for a role, the server passes to the module a structure specific to that particular role. The structure contains information that the modules uses in the execution of that role or provides a way for the module to return information to the server.

The RTSP roles have the option of responding to the client. A response is defined as any data that a module sends to a client. Modules can send data to the client in a variety of ways. They can, for example, call `QTSS_Write` or `QTSS_WriteV`.

| 2009-06-01 | © 2002, 2009 Apple Inc. All Rights Reserved.

31

> **Note:** The order in which modules are called for any particular role is undetermined.

Register Role

Modules use the Register role to call `QTSS_AddRole` to tell the server the roles they support.

Modules also use the Register role to call `QTSS_AddService` to register services and to call `QTSS_AddStaticAttribute` to add static attributes to QTSS object types. (QTSS objects are collections of attributes, each having a value.)

The server calls a module's Register role once at startup. The Register role is always the first role that the server calls.

A module that returns any value other than `QTSS_NoErr` from its Register role is not loaded into the server.

Initialize Role

The server calls the Initialize role of those modules that have registered for this role after it calls the Register role for all modules. Modules use the Initialize role to initialize global and private data structures.

The server passes to each module's Initialize role objects that can be used to obtain the server's global attributes, preferences, and text error messages. The server also passes the error log stream reference, which can be used to write to the error log. All of these objects are globals, so they are valid for the duration of this run of the server and may be accessed at any time.

When called in the Initialize role, the module receives a `QTSS_Initialize_Params` structure which is defined as follows:

```
typedef struct
{
    QTSS_ServerObject    inServer;
    QTSS_PrefsObject     inPrefs;
    QTSS_TextMessagesObjectinMessages;
    QTSS_ErrorLogStream inErrorLogStream;
    QTSS_ModuleObject    inModule;
} QTSS_Initialize_Params;
```

`inServer`

> A `QTSS_ServerObject` object containing the server's global attributes and an attribute that contains information about all of the modules in the running server. For a description of each attribute, see the section "qtssServerObjectType" (page 84).

`inPrefs`

> A `QTSS_PrefsObject` object containing the server's preferences. For a description of each attribute, see the section "qtssPrefsObjectType" (page 61).

`inMessages`

> A `QTSS_TextMessagesObject` object that a module can use for providing localized text strings. See the section "qtssTextMessageObjectType" (page 88).

`inErrorLogStream`

> A `QTSS_ErrorLogStream` stream reference that a module can use to write to the server's error log. Writing to this stream causes the module to be invoked in its Error Log role.

`inModule`

> A `QTSS_ModuleObject` object that a module can use to store information about itself, including its name, version number, and a description of what the module does. See the section "qttsModuleObjectType" (page 50).

A module that wants to be called in the Initialize role must in its Register role call `QTSS_AddRole` and specify `QTSS_Initialize_Role` as the role.

A module that returns any value other than `QTSS_NoErr` from its Initialize role is not loaded into the server.

Shutdown Role

The server calls the Shutdown role of those modules that have registered for this role when the server is getting ready to shut down.

The server calls a module's Shutdown role without passing any parameters.

The module uses its Shutdown role to delete all data structures it has created and to perform any other cleanup task

A module that wants to be called in the Shutdown role must in its Register role call `QTSS_AddRole` and specify `QTSS_Shutdown_Role` as the role.

Modules should always return `QTSS_NoErr` when they finish handling this role.

The server guarantees that the Shutdown role is the last time that the module is called before the server shuts down.

Reread Preferences Role

The server calls the Reread Preferences role of those modules that have registered for this role and rereads its own preferences when the server receives a `SIGHUP` signal or when a module calls the Reread Preferences service described in the section "QTSS Services" (page 94).

When called in this role, the module should reread its preferences, which may be stored in a file or in a QTSS object.

A module that wants to be called in the Reread Preferences role must in its Register role call `QTSS_AddRole` and specify `QTSS_RereadPrefs_Role` as the role.

Modules should always return `QTSS_NoErr` when they finish handling this role.

Error Log Role

The server calls the Error Log role of those modules that have registered for this role when an error occurs. The module should process the error message by, for example, writing the message to a log file.

When called in the Error Log role, the module receives a `QTSS_ErrorLog_Params` structure, which is defined as follows:

```
typedef struct
{
    QTSS_ErrorVerbosity inVerbosity;
    char * inBuffer;
} QTSS_ErrorLog_Params;
```

`inVerbosity`
>   Specifies the verbosity level of this error message. Modules should use the `inFlags` parameter of `QTSS_Write` to specify the verbosity level. The following constants are defined:
>
>   `qtssFatalVerbosity = 0, qtssWarningVerbosity = 1, qtssMessageVerbosity = 2,`
>   `qtssAssertVerbosity = 3, qtssDebugVerbosity = 4,`

`inBuffer`
>   Points to a null-terminated string containing the error message.

Writing an error message at the level `qtssFatalVerbosity` causes the server to shut down immediately.

Writing to the error log cannot result in an `QTSS_WouldBlock` error.

A module that wants to be called in the Error Log role must in its Register role call `QTSS_AddRole` and specify `QTSS_ErrorLog_Role` as the role.

Modules should always return `QTSS_NoErr` when they finish handling this role.

RTSP Roles

When the server receives an RTSP request, it goes through a series of steps to process the request and ensure that a response is sent to the client. The steps consist of calling certain roles in a predetermined order. This section describes each role in detail. For an overview of roles and the sequence in which they are called, see the section "Overview of QuickTime Streaming Server Operations" (page 23).

> **Note:** All RTSP roles have the option of responding directly to the client. When any RTSP role responds to a client, the server immediately skips the RTSP roles that it would normally call and calls the RTSP Postprocessor role of the module that responded to the RTSP request.

RTSP Filter Role

The server calls the RTSP Filter role of those modules that have registered for the RTSP Filter role immediately upon receipt of an RTSP request. Processing the Filter role gives the module an opportunity to respond to the request or to change the RTSP request.

When called in the RTSP Filter role, the module receives a `QTSS_StandardRTSP_Params` structure, which is defined as follows:

```
typedef struct
{
    QTSS_RTSPSessionObject      inRTSPSession;
    QTSS_RTSPRequestObject      inRTSPRequest;
    char**                      outNewRequest;
} QTSS_StandardRTSP_Params;
```

`inRTSPSession`
    The `QTSS_RTSPSessionObject` object for this RTSP session. See the section "qtssRTSPSessionObjectType" (page 83) for information about RTSP session object attributes.

`inRTSPRequest`
    The `QTSS_RTSPRequestObject` object for this RTSP request. When called in the RTSP Filter role, only the `qtssRTSPReqFullRequest` attribute has a value. See the section "qtssRTSPRequestObjectType" (page 80) for information about RTSP request object attributes.

outNewRequest

> A pointer to a location in memory.

The module calls `QTSS_GetValuePtr` to get from the `qtssRTSPReqFullRequest` attribute the complete RTSP request that caused the server to call this role. The `qtssRTSPReqFullRequest` attribute is a read-only attribute. To change the RTSP request, the module should call `QTSS_New` to allocate a buffer, write the modified request into that buffer, and return a pointer to that buffer in the `outNewRequest` field of the `QTSS_StandardRTSP_Params` structure.

While a module is handling the RTSP Filter role, the server guarantees that the module will not be called for any other role referencing the RTSP session represented by `inRTSPSession`.

If module handling the RTSP Filter role responds directly to the client, the server next calls the responding module in the RTSP Postprocessor role. For information about that role, see the section "RTSP Postprocessor Role" (page 40).

A module that wants to be called in the RTSP Filter role must in its Register role call `QTSS_AddRole` and specify `QTSS_RTSPFilter_Role` as the role.

Modules should always return `QTSS_NoErr` when they finish handling this role.

RTSP Route Role

The server calls the RTSP Route role after the server has called all modules that have registered for the RTSP Filter role. It is the responsibility of a module handling this role to set the appropriate root directory for each RTSP request by changing the `qtssRTSPReqRootDir` attribute for the request.

When called, an RTSP Route role receives a `QTSS_StandardRTSP_Params` structure, which is defined as follows:

```
typedef struct
{
    QTSS_RTSPSessionObject      inRTSPSession;
    QTSS_RTSPRequestObject      inRTSPRequest;
    QTSS_RTSPHeaderObject       inRTSPHeaders;
    QTSS_ClientSessionObject    inClientSession;
} QTSS_StandardRTSP_Params;
```

`inRTSPSession`

> The `QTSS_RTSPSessionObject` object for this RTSP session. See the section "qtssRTSPSessionObjectType" (page 83) for information about RTSP session object attributes.

`inRTSPRequest`

> The `QTSS_RTSPRequestObject` object for this RTSP request. In the Route role and all subsequent RTSP roles, all of the attributes are filled in. See the section "qtssRTSPRequestObjectType" (page 80) for information about RTSP request object attributes.

`inRTSPHeaders`

> The `QTSS_RTSPHeaderObject` object for the RTSP headers. See the section "qtssRTSPHeaderObjectType" (page 79) for information about RTSP header object attributes.

`inClientSession`

> The `QTSS_ClientSessionObject` object for the client session. See the section "qtssClientSessionObjectType" (page 45) for information about client session object attributes.

Before calling modules in the RTSP Route role, the server parses the request. Parsing the request consists of filling in all of the attributes of the `QTSS_RTSPSessionObject` and `QTSS_RTSPRequestObject` members of the `QTSS_StandardRTSP_Params` structure.

A module processing the RTSP Route role has the option of changing the `qtssRTSPReqRootDir` attribute of the `QTSS_RTSPRequestObject` member of the `QTSS_StandardRTSP_Params` structure. Changing the `qtssRTSPReqRootDir` attribute changes the root folder for this RTSP request.

While a module is handling the RTSP Route role, the server guarantees that the module will not be called for any other role referencing the RTSP session represented by inRTSPSession.

If a module that is processing the RTSP Route role responds directly to the client, the server immediately skips the processing of any other roles and calls the responding module's RTSP Postprocessor role. For information about that role, see the section "RTSP Postprocessor Role" (page 40).

A module that wants to be called in the RTSP Route role must in its Register role call `QTSS_AddRole` and specify `QTSS_RTSPRoute_Role` as the role.

Modules should always return `QTSS_NoErr` when they finish handling this role.

RTSP Preprocessor Role

The server calls the RTSP Preprocessor role after the server has called all modules that have registered for the RTSP Route role. If the module handles the type of RTSP request for which the module is called, it is the responsibility of a module handling this role to send a proper RTSP response to the client.

When called, an RTSP Preprocessor role receives a `QTSS_StandardRTSP_Params` structure, which is defined as follows:

```
typedef struct
{
    QTSS_RTSPSessionObject   inRTSPSession;

    QTSS_RTSPRequestObject   inRTSPRequest;

    QTSS_RTSPHeaderObject    inRTSPHeaders;

    QTSS_ClientSessionObjectinClientSession;
} QTSS_StandardRTSP_Params;
```

inRTSPSession

> The `QTSS_RTSPSessionObject` object for this RTSP session. See the section
> "qtssRTSPSessionObjectType" (page 83) for information about RTSP session object attributes.

inRTSPRequest

> The `QTSS_RTSPRequestObject` object for this RTSP request with a value for each attribute. See the
> section "qtssRTSPRequestObjectType" (page 80) for information about RTSP request object attributes.

inRTSPHeaders

> The `QTSS_RTSPHeaderObject` object for the RTSP headers. See the section
> "qtssRTSPHeaderObjectType" (page 79) for information about RTSP header object attributes.

inClientSession

> The `QTSS_ClientSessionObject` object for the client session. See the section
> "qtssClientSessionObjectType" (page 45) for information about client session object attributes.

The RTSP Preprocessor role typically uses the `qtssRTSPReqFilePath` attribute of the `inRTSPRequest`
member of the `QTSS_StandardRTSP_Params` structure to determine whether the request matches the type
of request that the module handles. For example, a module may only handle URLs that end in `.mov` or `.sdp`.

If the request matches, the module handling the RTSP Preprocessor role responds to the request by calling
`QTSS_SendStandardRTSPResponse`, `QTSS_Write`, or `QTSS_WriteV`, or by calling `QTSS_AppendRTSPHeader`,
and `QTSS_SendRTSPHeaders`. If this module is also responsible for generating RTP packets for this client
session, it should call `QTSS_AddRTPStream` (page 171) to add streams to the client session, and `QTSS_Play`,
which causes the server to invoke the RTP Send Packets role of the module whose RTSP Preprocessor role calls
`QTSS_Play`.

While a module is handling the RTSP Preprocessor role, the server guarantees that the module will not be
called for any other role referencing the RTSP session specified by `inRTSPSession` or the client session
specified by `inClientSession`.

A module that wants to be called in the RTSP Preprocessor role must in its Register role call `QTSS_AddRole`
and specify `QTSS_RTSPPreProcessor_Role` as the role.

Modules should always return `QTSS_NoErr` when they finish handling this role.

RTSP Request Role

The server calls the RTSP Request role if no RTSP Preprocessor role responds to an RTSP request. Only one module is called in the RTSP Request role, and that is the first module to register for the RTSP Request role when the server starts up.

When called, the RTSP Request role receives a `QTSS_StandardRTSP_Params` structure, which is defined as follows:

```
typedef struct
{
    QTSS_RTSPSessionObject      inRTSPSession;

    QTSS_RTSPRequestObject      inRTSPRequest;

    QTSS_RTSPHeaderObject       inRTSPHeaders;

    QTSS_ClientSessionObject    inClientSession;
} QTSS_StandardRTSP_Params;
```

`inRTSPSession`
> The `QTSS_RTSPSessionObject` object for this RTSP session. See the section "qtssRTSPSessionObjectType" (page 83) for information about RTSP session object attributes.

`inRTSPRequest`
> The `QTSS_RTSPRequestObject` object for this RTSP request with a value for each attribute. See the section"qtssRTSPRequestObjectType" (page 80) for information about RTSP request object attributes.

`inRTSPHeaders`
> The `QTSS_RTSPHeaderObject` object for the RTSP headers. See the section "qtssRTSPHeaderObjectType" (page 79) for information about RTSP header object attributes.

`inClientSession`
> The `QTSS_ClientSessionObject` object for the client session. See the section "qtssClientSessionObjectType" (page 45) for information about client session object attributes.

Like a module processing the RTSP Preprocessor role, a module that processes the RTSP Request Role should use an attribute, such as the `qtssRTSPReqFilePath` attribute of the `inRTSPRequest` member of the `QTSS_StandardRTSP_Params` structure, to determine whether the request matches the type of request that the module can handle.

A module handling the RTSP Request role should respond to the request by

- Sending an RTSP response to the client by calling `QTSS_AppendRTSPHeader` and `QTSS_SendRTSPHeaders`, by calling `QTSS_SendStandardRTSPResponse`, or by calling `QTSS_Write` or `QTSS_WriteV`.

- Preparing the `QTSS_ClientSessionObject` for streaming by using the RTP callbacks, such as `QTSS_AddRTPStream` and `QTSS_Play`. If `QTSS_Play` is called, the server will invoke the calling module in the RTP Send Packets role, at which time the module will be expected to generate RTP packets to send to the client.

A module that wants to be called in the RTSP Request role must in its Register role call `QTSS_AddRole` and specify `QTSS_RTSPRequest_Role` as the role. The first module that successfully calls `QTSS_AddRole` and specifies `QTSS_RTSPRequest_Role` as the role is the only module that is called in the RTSP Request role.

Modules should always return `QTSS_NoErr` when they finish handling this role.

RTSP Postprocessor Role

The server calls a module's RTSP Postprocessor role whenever the module responds to an RTSP request if that module has registered for this role.

Modules can use the RTSP Postprocessor role to log statistical information.

When called, the RTSP Postprocessor role receives a `QTSS_StandardRTSP_Params` structure, which is defined as follows:

```
typedef struct
{
    QTSS_RTSPSessionObject      inRTSPSession;
    QTSS_RTSPRequestObject      inRTSPRequest;
    QTSS_RTSPHeaderObject       inRTSPHeaders;
    QTSS_ClientSessionObject    inClientSession;
} QTSS_StandardRTSP_Params;
```

`inRTSPSession`
> The `QTSS_RTSPSessionObject` object for this RTSP session. See the section "qtssRTSPSessionObjectType" (page 83) for information about RTSP session object attributes.

`inRTSPRequest`
> The `QTSS_RTSPRequestObject` object for this RTSP request with a value for each attribute. See the section "qtssRTSPRequestObjectType" (page 80) for information about RTSP request object attributes.

`inRTSPHeaders`

> The `QTSS_RTSPHeaderObject` object for the RTSP headers. See the section "qtssRTSPHeaderObjectType" (page 79) for information about RTSP header object attributes.

`inClientSession`

> The `QTSS_ClientSessionObject` object for the client session. See the section "qtssClientSessionObjectType" (page 45) for information about client session object attributes.

While a module is handling the RTSP Postprocessor role, the server guarantees that the module will not be called for any role referencing the RTSP session specified by inRTSPSession or the client session specified by inClientSession.

A module that wants to be called in the RTSP Postprocessor role must in its Register role call `QTSS_AddRole` and specify `QTSS_RTSPPostProcessor_Role` as the role.

Modules should always return `QTSS_NoErr` when they finish handling this role.

RTP Roles

This section describes RTP roles, which are used to send data to clients and to handle the closing of client sessions.

RTP Send Packets Role

The server calls a module's RTP Send Packets role when the module calls `QTSS_Play`. It is the responsibility of the RTP Send Packets role to send media data to the client and tell the server when the module's RTP Send Packets role should be called again.

When called, the RTP Send Packets role receives a `QTSS_RTPSendPackets_Params` structure, which is defined as follows:

```
typedef struct
{
    QTSS_ClientSessionObject    inClientSession;
    SInt64                      inCurrentTime;
    QTSS_TimeVal                outNextPacketTime;
} QTSS_RTPSendPackets_Params;
```

`inClientSession`

> The `QTSS_ClientSessionObject` object for the client session. See the section "qtssClientSessionObjectType" (page 45) for information about client session object attributes.

`inCurrentTime`

> The current time in server time units.

`outNextPacketTime`

> A time offset in milliseconds. Before returning from this role, a module should set `outNextPacketTime` to the amount of time that the server should allow to elapse before calling the RTP Send Packets role again for this session.

The RTP Send Packets role is invoked whenever a module calls `QTSS_Play` for that client session. The module calls `QTSS_Write` or `QTSS_WriteV` to send data to the client.

While a module is handling the RTP Send Packets role, the server guarantees that the module will not be called for any role referencing the client session specified by `inClientSession`.

A module that wants to be called in the RTP Send Packets role must in its Register role call `QTSS_AddRole` and specify `QTSS_RTPSendPackets_Role` as the role.

Modules should always return `QTSS_NoErr` when they finish handling this role.

Client Session Closing Role

The server calls a module's Client Session Closing role to allow the module to process the closing of client sessions.

When called, the Client Session Closing role receives a `QTSS_ClientSessionClosing_Params` structure, which is defined as follows:

```
typedef struct
{
    QTSS_ClientClosing          inReason;
    QTSS_ClientSessionObject    inClientSession;
} QTSS_ClientSessionClosing_Params;
```

`inReason`

> The reason why the session is closing. The session may be closing because the client sent an RTSP teardown (`qtssCliSesClosClientTeardown`), because this session has timed out (`qtssCliSesClosTimeout`), or because the client disconnected without issuing a teardown (`qtssCliSesClosClientDisconnect`).

`inClientSession`

> The `QTSS_ClientSessionObject` object for the client session that is closing.

The Client Session Closing role is called whenever the client session specified by `inClientSession` is about to be torn down.

While a module is handling the Client Session Closing role, the server guarantees that the module will not be called for any role referencing the client session specified by `inClientSession`.

A module that wants to be called in the Client Session Closing role must in its Register role call `QTSS_AddRole` and specify `QTSS_ClientSessionClosing_Role` as the role.

Modules should always return `QTSS_NoErr` when they finish handling this role.

RTCP Process Role

The server calls a module's RTCP Process role whenever it receives an RTCP receiver report from a client.

RTCP receiver reports contain feedback from the client on the quality of the stream. The feedback includes the percentage of lost packets, the number of times the audio has run dry, and frames per second. Many attributes in the `QTSS_RTPStreamObject` correlate directly to fields in the receiver report.

When called, the RTP Process role receives a `QTSS_RTCPProcess_Params` structure, which is defined as follows:

```
typedef struct
{
    QTSS_RTPStreamObject        inRTPStream;
    QTSS_ClientSessionObject    inClientSession;
    void*                       inRTCPPacketData;
    UInt32                      inRTCPPacketDataLen;
} QTSS_RTCPProcess_Params;
```

`inRTPStream`

    The `QTSS_RTPStreamObject` object for the RTP stream that this RTCP packet belongs to. See the section "qtssRTPStreamObjectType" (page 76) for information about RTP stream object attributes.

`inClientSession`

    The `QTSS_ClientSessionObject` object for the client session. See the section "qtssClientSessionObjectType" (page 45) for information about client session object attributes.

`inRTCPPacketData`

    A pointer to a buffer containing the packets that are to be processed.

`inRTCPPacketDataLen`

> The length of valid data in the buffer pointed to by `inRTCPPacketData`.

A module handling the RTCP Process role typically monitors the status of the connection. It might, for example, track the percentage of packets lost for each connected client and update its counters.

While a module is handling the RTCP Process role, the server guarantees that the module will not be called for any role referencing the RTP stream specified by `inRTPStream`.

A module that wants to be called in the RTCP Process role must in its Register role call `QTSS_AddRole` and specify `QTSS_RTCPProcess_Role` as the role.

Modules should always return `QTSS_NoErr` when they finish handling this role.

QTSS Objects

QTSS objects provide a way for modules and the server to exchange data with each other. QTSS objects consist of attributes that are used to store data. Every attribute has a name, an attribute ID, a data type, and permissions for reading and writing the attribute's value. Built-in attributes are attributes that the server always defines for an object type. For example, the `QTSS_RTSPRequestObject` object has a built-in URL attribute that other modules can read to obtain the URL associated with a particular RTSP request.

This section describes the attributes for each object type. The object types are

- `qtssAttrInfoObjectType` (page 45)

- `qtssClientSessionObjectType` (page 45)

- `qtssConnectedUserObjectType` (page 48)

- `qtssDynamicObjectType` (page 50)

- `qtssFileObjectType` (page 50)

- `qttsModuleObjectType` (page 50)

- `qtssPrefsObjectType` (page 61)

- `qtssRTPStreamObjectType` (page 76)

- `qtssRTSPHeaderObjectType` (page 79)

- `qtssRTSPRequestObjectType` (page 80)

- `qtssRTSPSessionObjectType` (page 83)

- `qtssServerObjectType` (page 84)

| 2009-06-01 | © 2002, 2009 Apple Inc. All Rights Reserved.

44

- `qtssTextMessageObjectType` (page 88)

qtssAttrInfoObjectType

An object of type `qtssAttrInfoObjectType` consists of attributes whose values describe an attribute: the attribute's name, attribute ID, data type, and permissions for reading and writing the attribute's value. An attribute information object (`QTSS_AttrInfoObject`) is an instance of this object type. There is one `QTSS_AttrInfoObject` for every attribute.

Table 1-2 (page 45) lists the attributes for objects of type `qtssAttrInfoObjectType`.

Note: All of these attributes are preemptive safe, so they can be read by calling `QTSS_GetValue`, `QTSS_GetValueAsString,` or `QTSS_GetValuePtr`.

Table 1-2   Attributes of objects of type `qtssAttrInfoObjectType`

Attribute Name and Description	Access	Data Type
`qtssAttrID` The attribute's identifier.	Readable, preemptive safe	QTSS_AttributeID
`qtssAttrDataType` The attribute's data type.	Readable, preemptive safe	QTSS_AttrDataType
`qtssAttrName` The attribute's name.	Readable, preemptive safe	char
`qtssAttrPermissions` Permissions for reading and writing the attribute's value, and whether getting the attribute's value is preemptive safe.	Readable, preemptive safe	QTSS_AttrPermission

qtssClientSessionObjectType

An object of type `qtssClientSessionObjectType` consists of attributes that describe a client session, where a client session is defined as a single client streaming presentation. A client session object (`QTSS_ClientSessionObject`) is an instance of this object type. The attributes of a client session object are valid for all roles that receive a value of type `QTSS_ClientSessionObject` in the structure the server passes to them.

Table 1-3 (page 46) lists the attributes for objects of type `qtssClientSessionObjectType`.

Note: All of these attributes are preemptive safe, so they can be read by calling QTSS_GetValue, QTSS_GetValueAsString, or QTSS_GetValuePtr.

Table 1-3   Attributes of objects of type `qtssClientSessionObjectType`

Attribute Name and Description	Access	Data Type
`qtssCliSesAdjustedPlayTimeInMsec` The time in milliseconds at which the most recent play was issued, adjusted forward to delay sending packets until the play response is issued.	Readable, preemptive safe	QTSS_TimeVal
`qtssCliSesCounterID` A counter-based unique ID for the session.	Readable, preemptive safe	UInt32
`qtssCliSesCreateTimeInMsec` The time in milliseconds that the session was created.	Readable, preemptive safe	QTSS_TimeVal
`qtssCliSesCurrentBitRate` The movie bit rate.	Readable, preemptive safe	UInt32
`qtssCliSesFirstPlayTimeInMsec` The time in milliseconds at which `QTSS_Play` was first called.	Readable, preemptive safe	QTSS_TimeVal
`qtssCliSesFullURL` The full presentation URL for this session. Same as the `qtssCliSesPresentationURL` attribute but includes the `rtsp://`*domain_name* prefix.	Readable, preemptive safe	char
`qtssCliSesHostName` The host name for this session. Also the *domain_name* `portion of the qtssCliSesFullURL attribute`.	Readable, preemptive safe	char
`qtssCliSesMovieAverageBitRate` The average bits per second based on total RTP bits/movie duration. The value is zero unless set by a module.	Readable, writable, preemptive safe	UInt32

Attribute Name and Description	Access	Data Type
qtssCliSesMovieDurationInSecs Duration of the movie for this session in seconds. The value is zero unless set by a module.	Readable, writable, preemptive safe	Float64
qtssCliSesMovieSizeInBytes Movie size in bytes. The value is zero unless set by a module.	Readable, writable, preemptive safe	UInt64
qtssCliSesPacketLossPercent Percentage of packets lost; for example, .5 = 50%	Readable, preemptive safe	Float32
qtssCliSesPlayTimeInMsec The time in milliseconds at which QTSS_Play was most recently called.	Readable, preemptive safe	QTSS_TimeVal
qtssCliSesPresentationURL The presentation URL for this session. This URL is the "base" URL for the session. RTSP requests to the presentation URL are assumed to affect all streams of the session.	Readable, preemptive safe	char
qtssCliSesReqQueryString The query string from the request that created this client session.	Readable, preemptive safe	char
qtssCliSesRTPBytesSent The number of RTP bytes sent for this session.	Readable, preemptive safe	SInt32
qtssCliSesRTPPacketsSent The number of RTP packets sent for this session.	Readable, preemptive safe	SInt32
qtssCliSesState The state of this session. Possible values are qtssPausedState and qtssPlayingState.	Readable, preemptive safe	QTSS_RTPSessionState
qtssCliSesStreamObjects Iterated attribute containing all RTP stream references (QTSS_RTPStreamObject) belonging to this session.	Readable, preemptive safe	QTSS_RTPStreamObject
qtssCliSesTimeConnectedinMsec Time in milliseconds that the client session has been connected.	Readable, preemptive safe	SInt64

| 2009-06-01 | © 2002, 2009 Apple Inc. All Rights Reserved.

47

Attribute Name and Description	Access	Data Type
`qtssCliRTSPReqRealStatusCode` The status from the most recent request. (Same as the `qtssRTSPReqRealStatusCode` session.)	Readable, preemptive safe	UInt32
`qtssCliRTSPReqRespMsg` The error message sent to the client for the most recent request if the response was an error.	Readable, preemptive safe	char
`qtssCliRTSPSessLocalAddrStr` The local IP address for this RTSP connection in dotted decimal format.	Readable, preemptive safe	char
`qtssCliRTSPSessLocalDNS` The DNS name of the local IP address for this RTSP connection.	Readable, preemptive safe	char
`qtssCliRTSPSessRemoteAddrStr` The IP address of the client in dotted decimal format.	Readable, preemptive safe	char
`qtssCliRTSPSesURLRealm` The realm from the most recent request.	Readable, preemptive safe	char
`qtssCliRTSPSesUserName` The name of the user from the most recent request.	Readable, preemptive safe	char
`qtssCliTeardownReason` The teardown reason. If not requested by the client, the reason for the disconnection must be set by the module that calls `QTSS_Teardown`.	Readable, writable, preemptive safe	QTSS_CliSesTeardownReason

qtssConnectedUserObjectType

An object of type `qtssConnectedUserObjectType` consists of attributes associated with a connected user, irrespective of the transport. Users connecting to a QuickTime movie are already represented by objects of type `qtssClientSessionObjectType`, so this object is used for other connected users, such as those requesting MP3 streams.

A connected user object (`QTSS_ConnectedUserObject`) is an instance of this object type. A `QTSS_ConnectedUserObject` can be created in any module. It can be added to the `qtssSvrConnectedUsers` attribute of the `QTSS_ServerObject` (described in the section "qtssServerObjectType" (page 84)).

Table 2–4 (page 49) lists the attributes for objects of type `qtssConnectedUserObjectType.`

Note: All of these attributes are preemptive safe, so they can be read by calling QTSS_GetValue, QTSS_GetValueAsString, QTSS_GetValuePtr.

Table 1-4   Attributes of objects of type `qtssConnectedUserObjectType`

Attribute Name and Description	Access	Data Types
`qtssConnectionBytesSent` Number of RTP bytes sent so far for this session.	Readable, preemptive safe	UInt32
`qtssConnectionCreateATimeInMsec` The time in milliseconds at which the session was created.	Readable, preemptive safe	QTSS_TimeVal
`qtssConnectionCurrentBitRate` Combined current bit rate in bits per second of all of the streams for this session. This is not an average.	Readable, preemptive safe	UInt32
`qtssConnectionHostName` The host name of the connected client.	Readable, preemptive safe	char
`qtssConnectionMountPoint` Presentation URL for this session. This URL is the "base" URL for the session. RTSP requests to this URL are assumed to affect all of the session's streams.	Readable, preemptive safe	char
`qtssConnectionPacketLossPercent` Combined current percent loss as a fraction; for example, .5 = 50%. This is not an average.	Readable, preemptive safe	Float32
`qtssConnectionTimeConnectedInMsec` Time in milliseconds the session has been connected.	Readable, preemptive safe	QTSS_TimeVal
`qtssConnectionType` The user's connection type, such as "MP3".	Readable, preemptive safe	char
`qtssConnectionSessLocalAddrStr` Local IP address for this connection in dotted-decimal format.	Readable, preemptive safe	char
`qtssConnectionSessRemoteAddrStr` IP address of the client in dotted-decimal format.	Readable, preemptive safe	char

qtssDynamicObjectType

An object of type `qtssDynamicObjectType` can be used to create an object that doesn't have any static attributes.

qtssFileObjectType

An object of type `qtssFileObject` consists of attributes that describe a file that has been opened. A file object (`QTSS_FileObject`) is an instance of this object type. These attributes are valid for all roles that receive a `QTSS_FileObject` in the structure the server passes to them.

Table 1-5 (page 50) lists the attributes for objects of type `qtssFileObjectType`.

> **Note:** All of these attributes are preemptive safe, so they can be read by calling `QTSS_GetValue`, `QTSS_GetValueAsString,` or `QTSS_GetValuePtr`.

Table 1-5   Attributes of objects of type qtss`FileObjectType`

Attribute Name and Description	Access	Data Type
`qtssFlObjStream` The stream reference for this file object.	Readable, preemptive safe	QTSS_StreamRef
`qtssFlOjFileSysModuleName` The name of the file system module that handles this file object	Readable, preemptive safe	char
`qtssFlObjLength` The length of the file in bytes.	Readable, writable, preemptive safe	UInt64
`qtssFlObjPosition` The current position in bytes of the file's file pointer from the beginning of the file (byte zero).	Readable, writable, preemptive safe	UInt64
`qtssFlObjModDate` The date and time of the last time the file was modified.	Readable, writable, preemptive safe	QTSS_TimeVal

qttsModuleObjectType

An object of type `qtssModuleObject` consists of attributes that describe a particular QTSS module, including its name, version number, a description of what the module does, its preferences, and the roles the module is registered for. A module object (`QTSS_ModuleObject`) is an instance of this object type. These attributes are valid for all roles that receive a `QTSS_ModuleObject` in the structure the server passes to them.

For each module the server loads, the server creates a module object and passes it to the module in the module's Initialize role. Modules can get information about other modules the server has loaded by accessing the `qtssSvrModuleObject` attribute of the `QTSS_ServerObject` object.

In addition to the attributes that store the module's name, version number and description, this object type has a module preferences attribute, `qtssModPrefs`. The `qtssModPrefs` attribute itself is an object whose attributes store the module's preferences as instance attributes. All modifications to the `qtssModPrefs` attribute are persistent between invocations of the server because the contents of each module's `qtssModPrefs` attribute are written to the server's configuration file, which is read when the server starts up.

Table 1-6 (page 51) lists the attributes for objects of type `qtssModuleObjectType`.

> **Note:** With the exception of `qtssModDesc` and `qtssModVersion`, these attributes are preemptive safe and can be read by calling `QTSS_GetValue`, `QTSS_GetValueAsString`, or `QTSS_GetValuePtr`.

Table 1-6   Attributes of objects of type `qtssModuleObjectType`

Attribute Name and Description	Access	Data Type
`qtssModAttributes` An object that modules can use to store any local attributes other than preferences.	Readable, writable, preemptive safe	QTSS_Object
`qtssModDesc` A description of what the module does.	Readable, writable not preemptive safe	char
`qtssModName` The module's name.	Readable, preemptive safe	char
`qtssModPrefs` An object whose attributes store the preferences for this module.	Readable, preemptive safe	QTSS_ModulePrefsObject
`qtssModRoles` A list of all the roles for which this module is registered.	Readable, preemptive safe	QTSS_Role
`qtssModVersion` The module's version number in the format 0x*MM*.*m*.*v*.*bbbb*, where *MM* = major version, *m* = minor version, *v* = very minor version, and *b* = build number.	Readable, writable, not preemptive safe	UInt32

qtssModulePrefsObjectType

An object of type `QTSS_ModulePrefsObject` consists of attributes that contain a module's preferences. A module preferences object `QTSS_ModulePrefsObject`) is an instance of this object type.

Each module is reponsible for adding attributes to its module preferences object and setting their values. The values of the preferences in the module preferences object are persistent between invocations of the server because the server writes the module preferences object for each module to a configuration file that the server reads when it is started.

QTSSAccessLogModule Preferences

Table 1-7 (page 52) lists the attributes for preferences of the module `QTSSAccessLogModule`. These preferences are maintained in the `streamingserver.xml` file.

Table 1-7   Attributes for preferences of the module `QTSSAccessLogModule`

Attribute Name and Description	Access	Data Type
`request_logging` By default, the value of this attribute is `true`.	Readable, writable, not preemptive safe	Bool16
`request_logfile_size` By default, the value of this attribute is 10240000.	Readable, writable, not preemptive safe	UInt32
`request_logfile_interval` By default, the value of this attribute is 7.	Readable, writable, not preemptive safe	UInt32
`request_logfile_in_gmt` Set to `true` to use Greenwich Mean Time (GMT) instead of local time in access log file entries. By default, the value of this attribute is `true`.	Readable, writable, not preemptive safe	Bool16
`request_logfile_dir` By default, the value of this attribute is `/Library/QuickTime-Streaming/Logs/`.	Readable, writable, not preemptive safe	char
`request_logfile_name` By default, the value of this attribute is `StreamingServer`.	Readable, writable, not preemptive safe	char

QTSSAccessModule Preferences

Table 1-8 (page 53) lists the attributes for preferences of the module `QTSSAccessModule`. These preferences are maintained in the `streamingserver.xml` file.

Table 1-8   Attributes for preferences of the module `QTSSAccessModule`

Attribute Name and Description	Access	Data Type
`modAccess_groupsfilepath` By default, the value of this attribute is `/Library/QuickTime-Streaming/Config/qtgroups`.	Readable, writable, not preemptive safe	char
`modAccess_qtaccessfilename` By default, the value of this attribute is `qtaccess`.	Readable, writable, not preemptive safe	char
`modAccess_usersfilepath` By default, the value of this attribute is `/Library/QuickTime-Streaming/Config/qtusers`.	Readable, writable, not preemptive safe	char

QTSSAdminModule Preferences

Table 1-9 (page 53) lists the attributes for preferences of the module `QTSSAdminModule`. These preferences are maintained in the `streamingserver.xml` file.

Table 1-9   Attributes for preferences of the module `QTSSAdminModule`

Attribute Name and Description	Access	Data Type
`AdministratorGroup` By default, the value of this attribute is `admin`.	Readable, writable, not preemptive safe	char
`Authenticate` By default, the value of this attribute is `true`.	Readable, writable, not preemptive safe	Bool16
`enable_remote_admin` By default, the value of this attribute is `true`.	Readable, writable, not preemptive safe	Bool16
`IPAccessList` Set to a list of IP addresses to allow remote admin access from the specified IPs only. By default, the value of this attribute is 127.0.0.*.	Readable, writable, not preemptive safe	char
`LocalAccessOnly` Set to `true` to allow local admin server requests only. By default, the value of this attribute is `true`.	Readable, writable, not preemptive safe	Bool16
`RequestTimeIntervalMilli` By default, the value of this attribute is 50.	Readable, writable, not preemptive safe	UInt32

QTSSFileModule Preferences

Table 1-10 (page 54) lists the attributes for preferences of the module `QTSSFileModule`. These preferences are maintained in the `streamingserver.xml` file.

Table 1-10 Attributes for preferences of the module `QTSSFileModule`

Attribute Name and Description	Access	Data Type
`add_seconds_to_client_buffer_delay` Adds the specified number of seconds to the normal buffer delay. By default, the value of this attribute is 0.000000.	Readable, writable, not preemptive safe	Float32
`admin_email` By default, this attribute does not have a value.	Readable, writable, not preemptive safe	char
`compatibility_adjust_sdp_media_- bandwidth_percent` Used to adjust the SDP media bandwidth percentage for compatibility with certain players. By default, the value of this attribute is 100.	Readable, writable, not preemptive safe	UInt32
`enable_movie_file_sdp` Set to `true` to override the movie's built-in SDP information. By default, the value of this attribute is `false`.	Readable, writable, not preemptive safe	Bool16
`enable_player_compatibility` Enables player compatibility with certain players. By default, the value of this attribute is `true`.	Readable, writable, not preemptive safe	Bool16
`enable_private_file_buffers` By default, the value of this attribute is `true`.	Readable, writable, not preemptive safe	Bool16
`enable_shared_file_buffers` By default, the value of this attribute is `true`.	Readable, writable, not preemptive safe	Bool16
`flow_control_probe_interval` By default, the value of this attribute is 10.	Readable, writable, not preemptive safe	UInt32
`max_allowed_speed` By default, the value of this attribute is 4.000000.	Readable, writable, not preemptive safe	Float32
`max_private_buffer_units_per_buffer` By default, the value of this attribute is 8.	Readable, writable, not preemptive safe	UInt32

Attribute Name and Description	Access	Data Type
`max_shared_buffer_units_per_buffer` By default, the value of this attribute is 8.	Readable, writable, not preemptive safe	UInt32
`num_private_buffer_units_per_buffer` By default, the value of this attribute is 1.	Readable, writable, not preemptive safe	UInt32
`num_shared_buffer_increase_per_session` By default, the value of this attribute is 2.	Readable, writable, not preemptive safe	UInt32
`num_shared_buffer_units_per_buffer` By default, the value of this attribute is 0.	Readable, writable, not preemptive safe	UInt32
`private_buffer_unit_k_size` Size of private file I/O buffers. By default, the value of this attribute is 64.	Readable, writable, not preemptive safe	UInt32
`record_movie_file_sdp` Set to `true` to cause SDP information to be provided when the movie is played. By default, the value of this attribute is `false`.	Readable, writable, not preemptive safe	Bool16
`sdp_url` By default, this attribute does not have a value.	Readable, writable, not preemptive safe	char
`shared_buffer_unit_k_size` Size of shared file I/O buffers. By default, the value of this attribute is 64.	Readable, writable, not preemptive safe	UInt32

QTSSFlowControlModule Preferences

Table 1-11 (page 55) lists the attributes for preferences of the module `QTSSFlowControlModule`. These preferences are maintained in the `streamingserver.xml` file.

Table 1-11   Attributes for preferences of the module `QTSSFlowControlModule`

Attribute Name and Description	Access	Data Type
`flow_control_udp_thinning_module_-enabled` By default, the value of this attribute is `true`.	Readable, writable, not preemptive safe	Bool16
`loss_thick_tolerance` By default, the value of this attribute is 5.	Readable, writable, not preemptive safe	UInt32
`loss_thin_tolerance` By default, the value of this attribute is 30.	Readable, writable, not preemptive safe	UInt32

Attribute Name and Description	Access	Data Type
`num_losses_to_thick` By default, the value of this attribute is 6.	Readable, writable, not preemptive safe	UInt32
`num_losses_to_thin` By default, the value of this attribute is 3.	Readable, writable, not preemptive safe	UInt32
`num_worses_to_thin` By default, the value of this attribute is 2.	Readable, writable, not preemptive safe	UInt32

QTSSHomeDirectoryModule Preferences

Table 1-12 (page 56) lists the attributes for preferences of the module `QTSSHomeDirectoryModule`. These preferences are maintained in the `streamingserver.xml` file.

Table 1-12   Attributes for preferences of the module `QTSSHomeDirectoryModule`

Attribute Name and Description	Access	Data Type
`enabled` Enable or disable this module. By default, the value of this attribute is `false`.	Readable, writable, not preemptive safe	Bool16
`movies_directory` By default, this attribute does not have a value.	Readable, writable, not preemptive safe	Bool16
`max_num_cons_per_home_directory` Denies additional client connections greater than the value of this attribute. By default, the value of this attribute is 0.	Readable, writable, not preemptive safe	UInt32
`max_bandwidth_kbps_per_home_directory` Denies additional client connections when the value of this attribute is exceeded. By default, the value of this attribute is 0.	Readable, writable, not preemptive safe	UInt32

QTSSMP3StreamingModule Preferences

Table 1-13 (page 57) lists the attributes for preferences of the module `QTSSMp3StreamingModule`. These preferences are maintained in the `streamingserver.xml` file.

Table 1-13   Attributes for preferences of the module `QTSSMP3StreamingModule`

Attribute Name and Description	Access	Data Type
`mp3_request_logfile_name` By default, the value of this attribute is `mp3_access`.	Readable, writable, not preemptive safe	char
`mp3_request_logfile_dir` By default, the value of this attribute is `/Library/QuickTime-Streaming/Logs`.	Readable, writable, not preemptive safe	char
`mp3_streaming_enabled` By default, the value of this attribute is `true`.	Readable, writable, not preemptive safe	Bool16
`mp3_broadcast_password` By default, the value of this attribute is `true`.	Readable, writable, not preemptive safe	Bool16
`mp3_broadcast_password` By default, this attribute has no value.	Readable, writable, not preemptive safe	char
`mp3_broadcast_buffer_size` By default, the value of this attribute is 8192 bytes.	Readable, writable, not preemptive safe	SInt32
`mp3_max_flow_control_time` Length of the server-side MP3 buffer in milliseconds. By default, the value of this attribute is 10.	Readable, writable, not preemptive safe	UInt32
`mp3_request_logging` By default, the value of this attribute is `true`.	Readable, writable, not preemptive safe	Bool16
`mp3_request_logfile_size` By default, the value of this attribute is 10240000.	Readable, writable, not preemptive safe	UInt32
`mp3_request_logfile_interval` By default, the value of this attribute is 7.	Readable, writable, not preemptive safe	UInt32
`mp3_request_logtime_in_gmt` By default, the value of this attribute is `true`.	Readable, writable, not preemptive safe	Bool16

QTSSReflectorModule Preferences

Table 1-14 (page 58) lists the attributes for preferences of the module `QTSSReflectorModule`. These preferences are maintained in the `streamingserver.xml` file.

Table 1-14 Attributes for preferences of the module `QTSSReflectorModule`

Attribute Name and Description	Access	Data Type
`allow_broadcasts` By default, the value of this attribute is `true`.	Readable, writable, not preemptive safe	Bool16
`allow_duplicate_broadcasts` Set to `true` to allow the acceptance of setups on an existing broadcast stream. By default, the value of this attribute is `false`.	Readable, writable, not preemptive safe	Bool16
`allow_non_sdp_urls` By default, the value of this attribute is `true`.	Readable, writable, not preemptive safe	Bool16
`allow_announced_kill` By default, the value of this attribute is `true`.	Readable, writable, not preemptive safe	Bool16
`authenticate_local_broadcast` By default, the value of this attribute is `false`.	Readable, writable, not preemptive safe	Bool16
`broadcast_dir_list` By default, this attribute has no value.	Readable, writable, not preemptive safe	char
`compatibility_adjust_sdp_media_-bandwidth_percent` This attribute is provided for compatibility with 3GPP players. By default, the value of this attribute is `50`.	Readable, writable, not preemptive safe	UInt32
`disable_rtp_play_info` RTP play information is always enabled. Set this attribute to `true` to disable RTP play information. By default, the value of this attribute is `false`.	Readable, writable, not preemptive safe	Bool16
`disable_overbuffering` By default, the value of this attribute is `false`.	Readable, writable, not preemptive safe	Bool16
`enable_broadcast_announce` Set to `true` to enable broadcaster announce of an SDP file to the server. By default, the value of this attribute is `true`.	Readable, writable, not preemptive safe	Bool16
`enable_broadcast_push` Set to `true` to enable broadcaster RTSP push to the server. By default, the value of this attribute is `true`.	Readable, writable, not preemptive safe	Bool16
`enable_play_response_range_header` This attribute is provided for compatibility with 3GPP players. By default, the value of this attribute is `true`.	Readable, writable, not preemptive safe	Bool16

Attribute Name and Description	Access	Data Type
`enable_player_compatibility` This attribute is provided for compatibility with 3GPP players. By default, the value of this attribute is `true`.	Readable, writable, not preemptive safe	Bool16
`enforce_static_sdp_port_range` By default, the value of this attribute is `false`.	Readable, writable, not preemptive safe	Bool16
`force_rtp_info_sequence_and_time` This attribute is provided for compatibility with 3GPP players. By default, the value of this attribute is `false`.	Readable, writable, not preemptive safe	Bool16
`ip_allow_list` By default, the value of this attribute is 127.0.0.*.	Readable, writable, not preemptive safe	char
`kill_clients_when_broadcast_stops` When set to `true`, clients watching the stream of a broadcaster RTSP session that goes down are also torn down. By default, the value of this attribute is `false`.	Readable, writable, not preemptive safe	Bool16
`max_broadcast_announce_duration_secs` Sets the maximum duration, in seconds, of announced SDPs. By default, the value of this attribute is 0, which allows an infinite duration.	Readable, writable, not preemptive safe	UInt32
`maximum_static_sdp_port` By default, the value of this attribute is 65535.	Readable, writable, not preemptive safe	UInt16
`minimum_statid_sdp_port` By default, the value of this attribute is 2000.	Readable, writable, not preemptive safe	UInt16
`redirect_broadcast_keyword` By default, this attribute has no value.	Readable, writable, not preemptive safe	char
`redirect_broadcasts_dir` By default, this attribute has no value.	Readable, writable, not preemptive safe	char
`reflector_bucket_offset_delay_msc` By default, the value of this attribute is 73.	Readable, writable, not preemptive safe	UInt32
`reflector_buffer_size_sec` By default, the value of this attribute is 10.	Readable, writable, not preemptive safe	UInt32
`reflector_in_packet_receive_time` By default, the value of this attribute is 60.	Readable, writable, not preemptive safe	UInt32

Attribute Name and Description	Access	Data Type
`reflector_rtp_info_offset_msec` An internal value for live player compatibility. By default, the value of this attribute is 500.	Readable, writable, not preemptive safe	UInt32
`reflector_use_in_packet_receive_time` By default, the value of this attribute is `false`.	Readable, writable, not preemptive safe	Bool16
`timeout_broadcaster_session_secs` By default, the value of this attribute is 20.	Readable, writable, not preemptive safe	UInt32
`use_one_SSRC_per_stream` By default, the value of this attribute is `true`.	Readable, writable, not preemptive safe	Bool16

QTSSRefMovieModule Preferences

Table 1-15 (page 60) lists the attributes for preferences of the module `QTSSRefMovieModule`, which allows web developers to put RTSP URLs in web pages. These preferences are maintained in the `streamingserver.xml` file.

Table 1-15   Attributes for preferences of the module `QTSSRefMovieModule`

Attribute Name and Description	Access	Data Type
`refmovie_rtsp_port` The port to use for RTSP request redirection. Technically, this is not a protocol redirect. It is a media or content level redirect. Works the same as if you had a text file on a Web server called `mymovie.mov` that contained the RTSP URL with an rtsptext QuickTime tag. The tag and file name extension would tell the QuickTime client to RTSP stream the file. By default,the value of this attribute is 554.	Readable, writable, not preemptive safe	UInt16
`refmovie_xfer_enabled` For QuickTime clients only, converts, for example,`http: //hostname/mymovie.mov`to `rtsp://hostname: 554/mymovie.mov`. The server creates a text-based ref movie as the HTTP response, which redirects the client to the same movie on the server but as an RTSP request. This conversion is useful for placing streaming movie references on a web server. HTTP requests that do not specify a port go to port 80. However, `http://hostname: 554/mymovie.mov`also works. By default, the value of this attribute is `true`.	Readable, writable, not preemptive safe	Bool16

QTSSRelayModule Preferences

Table 1-16 (page 61) lists the attributes for preferences of the module `QTSSRelayModule`. These preferences are maintained in the `streamingserver.xml` file.

Table 1-16   Attributes for preferences of the module `QTSSRelayModule`

Attribute Name and Description	Access	Data Type
`relay_prefs_file` By default, the value of this attribute is `/Library/QuickTime-Streaming/Config/relayconfig.xml`.	Readable, writable, not preemptive safe	char
`relay_stats_url` By default, this attribute has no value.	Readable, writable, not preemptive safe	char

qtssPrefsObjectType

An object of type `qtssPrefsObjectType` consists of attributes that describe the server's internal preference storage system. A preference object (`QTSS_PrefsObject`) is an instance of this object type. The attribute values for objects of this type are stored in the server's configuration file, `streamingserver.xml`. For each server, there is a single instance of this object type.

In previous versions of the QTSS programming interface, module preferences were stored in this object. Since version 4.0, module preferences have been stored in each module's `QTSS_ModuleObject` object.

Table 1-17 (page 61) lists the attributes for objects of type `qtssPrefsObjectType`.

> **Note:** None of these attributes is preemptive safe, so they can must be read by calling `QTSS_GetValue` or by locking the object, calling `QTSS_GetValuePtr`, and unlocking the object.

Table 1-17   Attributes of objects of type `qtssPrefsObjectType`

Attribute Name and Description	Name in streamingserver.xml	Access	Data Type
`qtssPrefsAckLoggingEnabled` Enables detailed logging of UDP acknowledgement and retransmit packets. By default, the value of this attribute is `false`.	`ack_logging_enabled`	Readable, writable, not preemptive safe	Bool16

| 2009-06-01 | © 2002, 2009 Apple Inc. All Rights Reserved.

61

Attribute Name and Description	Name in streamingserver.xml	Access	Data Type
`qtssPrefsAltTransportIPAddr` If you want an IP address other than the server's IP address appended to the transport header, use this attribute to specify the alternate address. By default, this attribute does not have a value.	`alt_transport_src _ipaddr`	Readable, writable, not preemptive safe	char
`qtssPrefsAlwaysThinDelayInMsec` If a packet is as late in milliseconds as the value of this attribute, the server starts to thin. This attribute is part of the server's thinning algorithm. By default, the value of this attribute is 750.	`always_thin_delay`	Readable, writable, not preemptive safe	SInt32
`qtssPrefsAuthenticationScheme` Set this attribute to the authentication scheme you want the server to use. The currently supported values are `basic`, `digest`, and `none`. By default, the value of this attribute is `digest`.	`authentication _– scheme`	Readable, writable, not preemptive safe	char
`qtssPrefsAutoDeleteSPDFiles` An attribute for a preference that is no longer supported. The attribute remains for API compatibility.	`auto_delete_sdp_ files`	Readable, writable, not preemptive safe	Bool16
`qtssPrefsAutoRestart` If `true`, the server automatically restarts itself if it crashes. By default, the value of this attribute is `true`.	`auto_restart`	Readable, writable, not preemptive safe	Bool16
`qtssPrefsAutoStart` Obsolete and should always be set to `false`.	`auto_start`	Readable, writable, not preemptive safe	Bool16
`qtssPrefsAvgBandwidthUpdate` The interval in seconds between computations of the server's average bandwidth. By default, the value of this 60.	`average_bandwidth _– update`	Readable, writable, not preemptive safe	UInt32

Attribute Name and Description	Name in streamingserver.xml	Access	Data Type
`qtssPrefsBreakOnAssert` If `true`, the server will stop and enter the debugger when an assert condition is hit. By default, the value of this attribute is `false`.	`break_on_assert`	Readable, writable, not preemptive safe	Bool16
`qtssPrefsCloseLogsOnWrite` If set to `true`, the server closes log files after each write. By default, the value of this attribute is `false`.	`force_logs_close_‐ on_write`	Readable, writable, not preemptive safe	Bool16
`qtssPrefsDefault‐ AuthorizationRealm` Specifies the text to display as the login entity "realm" by the client. By default, the value of this attribute is `Streaming Server`. If the value of this attribute is not set, `Streaming Server` is displayed.	`default_‐ authorization_realm`	Readable, writable, not preemptive safe	char
`qtssPrefsDelete‐ SPDFilesInterval` The interval in seconds at which to check SDP files. Changes to this attribute take effect at the end of the current interval. By default, the value of this attribute is 10. The server maintains an internal interval of 1.	`sdp_file_delete _‐ interval_seconds`	Readable, writable, not preemptive safe	Bool16
`qtssPrefsDoReport‐ HTTPConnectionAddress` When behind a round-robin DNS, the client needs to be told the IP address of the machine that is handling its request. This attribute tells the server to report its IP address in the reply to the HTTP GET request when tunneling RTSP through HTTP. By default, the value of this attribute is `false`.	`do_report_http_‐ connection_ip_‐ address`	Readable, writable, not preemptive safe	Bool16

Attribute Name and Description	Name in streamingserver.xml	Access	Data Type
`qtssPrefsDropAll-` `PacketsDelayInMsec` If a packet is as late as the value of this attribute in milliseconds, the server drops it. This attribute is part of the server's thinning algorithm. By default, the value of this attribute is 2500.	`drop_all_packets _-` `delay`	Readable, writable, not preemptive safe	SInt32
`qtssPrefsDropVideo-` `AllPacketsDelayInMsec` If a video packet cannot be sent within the time in milliseconds specified by this attribute, the server drops it. This atttribute is used by the server's thinning algorithm. By default, the value of this attribute is 1750.	`drop_all_video _-` `delay`	Readable, writable, not preemptive safe	SInt32
`qtssPrefsEnable-` `MonitorStatsFile` If set to `true`, the server writes server statistics to the monitor file, which is read by an external monitor application. By default, the value of this attribute is `false`.	`enable_monitor _-` `stats_file`	Readable, writable, not preemptive safe	Bool16
`qtssPrefsEnable-` `PacketHeaderPrintfs` If set to `true`, the server prints the headers of outgoing RTP and RTCP packets on `stdout`. The server must have been started with the `-d` command line option. See the `qtssPrefsPacket-` `HeaderPrintfOptions` attribute for the available print options. By default, the value of this attribute is `false`.	`enable_packet _-` `header_printfs`	Readable, writable, not preemptive safe	Bool16
`qtssPrefsEnable-` `RTSPDebugPrintfs` When set to `true`, the server prints on `stdout` incoming RTSP requests and outgoing RTSP responses. The server must have been started with the `-d` command line option. By default, the value of this attribute is `false`.	`RTSP_debug_printfs`	Readable, writable, not preemptive safe	Bool16

Attribute Name and Description	Name in streamingserver.xml	Access	Data Type
`qtssPrefsEnable–RTSPErrorMessage` If set to `true`, the server appends a content body string error message for reported RTSP errors. By default, the value of this attribute is `false`.	`RTSP_error_message`	Readable, writable, not preemptive safe	Bool16
`qtssPrefsEnableRTSPServerInfo` If set to `true`, the server adds server information to RTSP headers. The informatin includes the server's platform, version number, and build number. By default, the value of this attribute is `true`.	`RTSP_server_info`	Readable, writable, not preemptive safe	Bool16
`qtssPrefsLargeWindowSizeInK` For Reliable UDP, the window size in K bytes used for high bitrate movies. For clients that don't specify a window size, the server may use the value of this attribute. By default, the value of this attribute is 64.	`large_window_size`	Readable, writable, not preemptive safe	UInt32
`qtssPrefsMaxAdvance–SendTimeTimeInSec` The most number of seconds the server sends a packet ahead of time to a client that supports overbuffing. By default, the value of this attribute is 25.	`max_send_ahead_time`	Readable, writable, not preemptive safe	UInt32
`qtssPrefsMaximumBandwidth` The maximum amount of bandwidth the server is allowed to serve in K bits. If the server exceeds this value, it responds to new client requests for additional streams with RTSP error 453, "Not Enough Bandwidth." A value of -1 means the amount of bandwidth the server is allowed to serve is unlimited. By default, the value of this attribute is 102400.	`maximum_bandwidth`	Readable, writable, not preemptive safe	SInt32

Attribute Name and Description	Name in streamingserver.xml	Access	Data Type
`qtssPrefsMaximumConnections` The maximum number of concurrent RTP connections the server allows. A value of -1 means that an unlimited number of connections are allowed. By default, the value of this attribute is 1000.	`maximum_connections`	Readable, writable, not preemptive safe	SInt32
`qtssPrefsMaxRetransDelayInMsec` For Reliable UDP, the maximum interval in milliseconds between when a retransmit is supposed to be sent and when it is actually sent. Lower values result in smoother but slower server performance. By default, the value of this attribute is 500.	`max_retransmit _–delay`	Readable, writable, not preemptive safe	UInt32
`qtssPrefsMax–TCPBufferSizeInBytes` The maximum size in bytes the TCP socket send buffer can be set to. By default, the value of this attribute is 200000.	`max_tcp_buffer_size`	Readable, writable, not preemptive safe	Float32
`qtssPrefsMediumWindowSizeInK` For Reliable UDP, the window size in K bytes used for medium bitrate movies. For clients that don't specify a window size, the server may use the value of this attribute. By default, the value of this attribute is 48.	`medium_window_size`	Readable, writable, not preemptive safe	UInt32
`qtssPrefsMin–TCPBufferSizeInBytes` The minimum size in bytes the TCP socket send buffer can be set to. By default, the value of this attribute is 8192.	`min_tcp_buffer_size`	Readable, writable, not preemptive safe	UInt32

| 2009-06-01 | © 2002, 2009 Apple Inc. All Rights Reserved.

66

Attribute Name and Description	Name in streamingserver.xml	Access	Data Type
`qtssPrefsModuleFolder` The path to the folder containing dynamic loadable server modules. For Mac OS X, this attribute is set to `/Library/QuickTime-Streaming/Modules`. For Darwin platforms, this attribute is set to `/usr/local/sbin/Streaming-Server/Modules`, and for Win32 platforms, this attribute is set to `c:\Program Files\DarwinStreamingServer\QTSSModules`.	`module_folder`	Readable, writable, not preemptive safe	char
`qtssPrefsMovieFolder` The path to the root movie folder. By default, the value of this attribute is `/Library/QuickTime-Streaming/Movies`.	`movie_folder`	Readable, writable, not preemptive safe	char
`qtssPrefsMonitor-StatsFileFileName` Name of the monitor file. By default, the value of this attribute is `server_status`.	`monitor_stats_-file_name`	Readable, writable, not preemptive safe	char
`qtssPrefsMonitor-StatsFileIntervalSec` Interval at which server writes server statistics in the monitor file. By default, the value of this attribute is 10.	`monitor_stats_file_-interval_seconds`	Readable, writable, not preemptive safe	UInt32
`qtssPrefsOverbufferRate` The server uses this attribute to calculate the rate at which to overbuffer. The value of this attribute is multiplied by the data rate. By default, the value of this attribute is `2.0`.	`overbuffer_rate`	Readable, writable, not preemptive safe	Float32

Attribute Name and Description	Name in streamingserver.xml	Access	Data Type
`qtssPrefsPacket–HeaderPrintfOptions` Identifies which packet headers to print when `qtssPrefsEnabled–PacketHeaderPrintfs` is `true`. The options are semicolon (`;`) delimited strings. By default, the value of this attribute is all of the available options, `rtp;rr;sr;app;ack;`, which means that headers of RTP packets (`rtp`), RTCP receiver reports (`rr`), RTCP sender reports (`sr`), RTCP application packets (`app`), and Reliable UDP RTP acknowledgement packets (`ack`) are printed.	`packet_header _–printf_options`	Readable, writable, not preemptive safe	char
`qtssPrefsPIDFile` Specifies the name of the file in which the server's process ID is written. By default, the value of this attribute is `/var/run/QuickTime–StreamingServer.pid`.	`pid_file`	Readable, writable, not preemptive safe	char
`qtssPrefsRealRTSPTimeout` The amount of time in seconds the server actually waits before disconnecting idle RTSP clients. This timer is reset each time the server receives a new RTSP request from the client. A value of zero means that there is no timeout. By default, the value of this attribute is 180.	`real_rtsp_timeout`	Readable, writable, not preemptive safe	UInt32
`qtssPrefsReliableUDP` If set to `true`, the server the uses Reliable UDP transport if requested by the client. By default, the value of this attribute is `true`.	`reliable_udp`	Readable, writable, not preemptive safe	Bool16

Attribute Name and Description	Name in streamingserver.xml	Access	Data Type
`qtssPrefsReliableUDPDirs` This attribute specifies the directories for which Reliable UDP is to be used. The directories are interpreted as relative to the Movies folder (`qtssPrefsMovieFolder`) with a leading slash but no trailing slash. For example, `/reliable_udp_dir`. By default, this attribute does not have a value.	`reliable_udp_dirs`	Readable, writable, not preemptive safe	char
`qtssPrefsReliableUDPPrintfs` When set to `true`, the server prints on `stdout` Reliable UDP statistics when the client disconnects. The server must have been started with the `-d` command line option. The statistics include the URL, maximum congestion window, minimum congestion window, maximim, minimum, and average RTT, number of skipped frames, and the number of late packets dropped. By default, the value of this attribute is `false`.	`reliable_udp_-printfs`	Readable, writable, not preemptive safe	Bool16
`qtssPrefsReliableUDPSlowStart` Set to `true` to enable Reliable UDP slow start. Disabling UDP slow start may lead to an initial burst of packet loss due to mis-estimate of the client's available bandwidth. Enabling UDP slow start may lead to premature reduction of the bit rate (known as "thinning"). By default, the value of this attribute is `true`.	`reliable_udp_slow_start`	Readable, writable, not preemptive safe	Bool16
`qtssPrefsRTCPPoll-IntervalInMsec` A preference that is no longer used. Polling is no longer a feature of RTCP.	`rtcp_poll_interval`	Readable, writable, not preemptive safe	UInt32

Attribute Name and Description	Name in streamingserver.xml	Access	Data Type
`qtssPrefsRTCPSockRcvBufSizeInK` Size of the receive socket buffer for UDP sockets used to receive RTCP packets. The buffer needs to be big enough to absorb bursts of RTCP acknowledgements. By default, the value of this attribute is 768.	`rtcp_rcv_buf_size`	Readable, writable, not preemptive safe	UInt32
`qtssPrefsRTPTimeout` The amount of time in seconds the server will wait before disconnecting idle RTP clients. This timer is reset each time the server receives an RTCP status packet from a client. A value of zero means there is no timeout. By default, the value of this attribute is 120.	`rtp_timeout`	Readable, writable, not preemptive safe	UInt32
`qtssPrefsRTSPIPAddr` Specifies the IP address(es) in dotted-decimal format the server should accept RTSP client connections on. This attribute is useful when the machine has more than one IP address and you want to specify which addresses the server should listen on. A value of 0 means the server should accept connections on all IP addresses that are currently enabled on the system. By default, the value of this attribute is 0.	`bind_ip_addr`	Readable, writable, not preemptive safe	char
`qtssPrefsRTSPPorts` Ports for accepting RTSP client connections. By default, ports 554, 7070, 8000, and 8001 are enabled. Add port 80 to this list if you are streaming across the Internet and want clients behind firewalls to be able to connect to the server.	`rtsp_port`	Readable, writable, not preemptive safe	UInt32
`qtssPrefsRTSPTimeout` Amount of time in seconds the server tells clients it will wait before disconnecting idle RTSP clients. By default, the value of this attribute is 0.	`rtsp_timeout`	Readable, writable, not preemptive safe	UInt32

Attribute Name and Description	Name in streamingserver.xml	Access	Data Type
qtssPrefsRunGroupName Run the server under the specified group name. By default, the value of this attribute is qtss.	run_group_name	Readable, writable, not preemptive safe	char
qtssPrefsRunNumThreads If value of this attribute is non-zero, the server will create the specified number of threads for handling RTSP and RTP streams. Otherwise, the server creates one thread per processor for handling RTSP and RTP streams. By default, the value of this attribute is 0.	run_num_threads	Readable, writable, not preemptive safe	UInt32
qtssPrefsRunUserName Run the server under the specified user name. By default, the value of this attribute is qtss.	run_user_name	Readable, writable, not preemptive safe	char
qtssPrefsSafePlayDuration If the server finds it is serving more than its allowed maximum bandwidth (using the average bandwidth computation), it will attempt to disconnect the most recently connected clients until the average bandwidth drops to acceptable levels. However, it will not disconnect clients if they've been connected for longer than the time in seconds specified by this attribute. If this value is set to zero, the server does not disconnect clients. By default, the value of this attribute is 600.	safe_play_duration	Readable, writable, not preemptive safe	UInt32
qtssPrefsSendInterval The minimum time in milliseconds the server will wait between sending packet data to the client. By default, the value of this attribute is 50.	send_interval	Readable, writable, not preemptive safe	UInt32

Attribute Name and Description	Name in streamingserver.xml	Access	Data Type
`qtssPrefsSmallWindowSizeInK` For Reliable UDP, the window size in K bytes used for low bitrate movies. For clients that don't specify a window size, the server may use the value of this attribute. By default, the value of this attribute is 24.	`small_window_size`	Readable, writable, not preemptive safe	UInt32
`qtssPrefsSrcAddrInTransport` If set to `true`, the server adds its source address to its transport headers. This is necessary on certain networks where the source address is not necessarily known. By default, the value of this attribute is `false.`	`append_source_addr_–in_transport`	Readable, writable, not preemptive safe	Bool16
`qtssPrefsStart–QualityCheckIntervalInMsec` The interval in milliseconds at which server checks thinning and adjusts it if necessary. This attribute is part of the server's thinning algorithm. By default, the value of this attribute is 1000.	`quality_check _–interval`	Readable, writable, not preemptive safe	UInt32
`qtssPrefsStart–ThickingDelayInMsec` If a packet is this late in milliseconds, starting thicking. This attribute is part of the server's thinning algorithm. By default, the value of this attribute is 250.	`start_thicking _–delay`	Readable, writable, not preemptive safe	SInt32
`qtssPrefsStart–ThinningDelayInMsec` If a packet is as late as the value of this attribute, start thinning. By default, the value of this attribute is 0.	`start_thinning _–delay`	Readable, writable, not preemptive safe	SInt32

72

Attribute Name and Description	Name in streamingserver.xml	Access	Data Type
`qtssPrefsTCPSecondsToBuffer` When streaming over TCP, the size of the send buffer is scaled based on the movie's bitrate. Using the bitrate of the movie as a guide, the server will set the TCP send buffer to fit this number of seconds of data. By default, the value of this attribute is .5.	`tcp_seconds_to _buffer`	Readable, writable, not preemptive safe	Float32
`qtssPrefsThickAllThe-WayDelayInMsec` If a packet is this late (negative means it is ahead of time), restore full quality. This attribute is part of the server's thinning algorithm. By default, the value of this attribute is -2000.	`thick_all_the_way _delay`	Readable, writable, not preemptive safe	UInt32
`qtssPrefsThinAllThe-WayDelayInMsec` If a packet is as late in milliseconds as the value of this attribute, the server thins the stream as much as possible. This attribute is part of the server's thinning algorithm. By default, the value of this attribute is 1500.	`thin_all_the_way _delay`	Readable, writable, not preemptive safe	SInt32
`qtssPrefsTotalBytesUpdate` The interval in seconds between updates of the server's total bytes and current bandwidth statistics. By default, the value of this attribute is 1.	`total_bytes_update`	Readable, writable, not preemptive safe	UInt32

Attribute Name and Description	Name in streamingserver.xml	Access	Data Type
`qtssPrefsWindowSize-MaxThreshold` The window size in bytes used to measure reliable UDP bandwidth. If the bit rate is greater than `qtssPrefsWindowSize-MaxThreshold`, the window size is set to `qtssPrefsLargeWindowSizeInK`. If the bit rate is greater than `qtssPrefsWindowSizeThreshold` and and less than or equal to `qtssPrefsWindowSize-MaxThreshold`, the window is set to `qtssPrefsMediumWindowSizeInK`. If the bit rate is less than or equal to `qtssPrefsWindSizeThreshold`, the window size is set to `qtssPrefsSmallWindowSizeInK`. By default, the value of this attribute is 1000.	`window_size_max _- threshold`	Readable, writable, not preemptive safe	UInt32
`qtssPrefsWindowSizeThreshold` For Reliable UDP, if the client doesn't specify its window size, the server uses the value of `qtssPrefsSmall-WindowSizeInK` as the window size if the bitrate is below the value of this attribute measured in K bits/second. By default, the value of this attribute is 200.	`window_size _- threshold`	Readable, writable, not preemptive safe	UInt32
This attribute is used for performance testing. When set to `true`, this attribute forces the server to maintain full bandwidth connections. By default, the value of this attribute is `false`.	`disable_thinning`	Readable, writable, not preemptive safe	Bool16
This attribute is used for compatibility with certain players. It contains a list of players that, for compatibility, require RTP header informatin. By default, the list consists of `Nokia` and `Real`.	`player_requires_- rtp_header_info`	Readable, writable, not preemptive safe	char

Attribute Name and Description	Name in streamingserver.xml	Access	Data Type
This attribute is used for compatibility with certain players.	`player_requires_-bandwidth_adjustment`	Readable, writable, not preemptive safe	char

The built-in error log module that loads before all other modules uses the following seven attributes:

Attribute Name and Description	Name in streamingserver.xml	Access	Data Type
`qtssPrefsErrorLogDir` Sets the path to the directory containing the error log file. By default, the value of this attribute is `/Library/QuickTime-Streaming/Logs`.	`error_logfile_dir`	Readable, writable, not preemptive safe	char
`qtssPrefsErrorLogEnabled` Set to `true` to enable error logging. By default, the value of this attribute is `true`.	`error_logging`	Readable, writable, not preemptive safe	Bool16
`qtssPrefsErrorLogName` Sets the name of the error log file. By default, the value of this attribute is `Error`.	`error_log_name`	Readable, writable, not preemptive safe	char
`qtssPrefsErrorLogVerbosity` Sets the verbosity level of messages the error logger logs. The following values are meaningful: 0 = log fatal errors 1 = log fatal errors and warnings 2 = log fatal errors, warnings, and asserts 3 = log fatal errors, warnings, asserts, and debug messages By default, the value of this attribute is 2.	`error_logfile_-verbosity`	Readable, writable, not preemptive safe	UInt32
`qtssPrefsErrorRollInterval` The interval in days between rolling the error log file. By default, the value of this attribute is 0, which means that the error log file is not rolled.	`error_logfile_-interval`	Readable, writable, not preemptive safe	UInt32
`qtssPrefsMaxErrorLogSize` The maximum size in bytes of the error log. A value of zero means that the server does not impose a limit. By default, the value of this attribute is 256000.	`error_logfile_size`	Readable, writable, not preemptive safe	UInt32

Attribute Name and Description	Name in streamingserver.xml	Access	Data Type
`qtssPrefsScreenLogging` If this attribute is set to `true` , every line in the error log is written to the terminal window. Note that to see the error log, the server must be launched from the command line in foreground mode by using the `-d` flag. By default, the value of this attribute is `true.`	`screen_logging`	Readable, writable, not preemptive safe	Bool16

qtssRTPStreamObjectType

An object of type `qtssRTPStreamObjectType` consists of attributes that describe a particular RTP stream whether it's an audio, video, or text stream. An RTP stream object (`QTSS_RTPStreamObject`) is an instance of this object type and is created by calling `QTSS_AddRTPStream`. An RTP stream object must be associated with a single client session object (`QTSS_ClientSessionObject`). A client session object may be associated with any number of RTP stream objects. These attributes are valid for all roles that receive a `QTSS_RTPStreamObject` in the structure the server passes to them.

Table 1-18 (page 76) lists the attributes for objects of type `qtssRTPStreamObjectType`.

> **Note:** All of these attributes are preemptive safe, so they can be read by calling `QTSS_GetValue`, `QTSS_GetValueAsString,` or `QTSS_GetValuePtr`.

Table 1-18   Attributes of objects of type `qtssRTPStreamObjectType`

Attribute Name and Description	Access	Data Type
`qtssRTPStrBufferDelayInSecs` Size of the client's buffer. The server sets this attribute to three seconds, but the module is responsible for determining the buffer size and setting this attribute accordingly.	Readable, preemptive safe	Float32
`qtssRTPStrFirstSeqNumber` Sequence number of the first packet after the last PLAY request was issued. If known, this attribute must be set by a module before calling `QTSS_Play`. The server uses this attribute to generate a proper RTSP PLAY response.	Readable, writable, preemptive safe	SInt16

Attribute Name and Description	Access	Data Type
`qtssRTPStrFirstTimestamp` RTP timestamp of the first RTP packet generated for this stream after the last PLAY request was issued. If known, this attribute must be set by a module before calling `QTSS_Play`. The server uses this attribute to generate a proper RTSP PLAY response.	Readable, writable, preemptive safe	SInt32
`qtssRTPStrNetworkMode` Network mode for the RTP stream. Possible values are `qtssRTPNetworkModeDefault,qtss-RTPNetworkModeMulticast,and qtssNetworkModeUnicast`.	Readable, preemptive safe	UInt32
`qtssRTPStrPayloadName` Name of the media for this stream. This attribute is empty unless a module explicitly sets it.	Readable, writable, preemptive safe	char
`qtssRTPStrPayloadType` Payload type of the media for this stream. The value of this attribute is `qtssUnknownPayloadType` unless a module sets it to `qtssVideoPayloadType` or `qtssAudioPayloadType`.	Readable, writable, preemptive safe	QTSS_RTPPayloadType
`qtssRTPStrTrackID` Unique ID that identifies each RTP stream.	Readable, writable, preemptive safe	UInt32
`qtssRTPStrTimescale` Timescale for the track. If known, this must be set before calling `QTSS_Play`.	Readable, writable, preemptive safe	SInt32
`qtssRTPStrSSRC` Synchronization source (SSRC) generated by the server. The SSRC is guaranteed to be unique among all streams in the session. The server includes the SSRC in all RTCP Sender Reports that the server generates.	Readable, preemptive safe	UInt32

The values of the following attributes come from the most recent RTCP packet received on a stream. If a field in the most recent RTCP packet is blank, the server sets the value of the corresponding attribute to zero.

Attribute Name and Description	Access	Data Type
`qtssRTPStrAudioDryCount` Number of times the audio has run dry.	Readable, preemptive safe	UInt16

Attribute Name and Description	Access	Data Type
`qtssRTPStrAvgBugDelayInMsec` Average buffer delay in milliseconds.	Readable, preemptive safe	UInt16
`qtssRTPStrAvgLateMilliseconds` Average in milliseconds of packets that the client received late.	Readable, preemptive safe	UInt16
`qtssRTPStrClientBufFill` How full the client buffer is in tenths of a second.	Readable, preemptive safe	UInt16
`qtssRTPStrExpFrameRate` The expected frame rate in frames per second.	Readable, preemptive safe	UInt16
`qtssRTPStrFractionLostPackets` The fraction of packets that have been lost for this stream.	Readable, preemptive safe	UInt32
`qtssRTPStrFrameRate` The current frame rate in frames per second.	Readable, preemptive safe	UInt16
`qtssRTPStrGettingBetter` A non-zero value if the client reports that the stream is getting better.	Readable, preemptive safe	UInt16
`qtssRTPStrGettingWorse` A non-zero value if the client reports that the stream is getting worse.	Readable, preemptive safe	UInt16
`qtssRTPStrIsTCP` If this RTP stream is being sent over TCP, this attribute is `true`. If this RTP stream is being sent over UDP, this attribute is `false`.	Readable, preemptive safe	Bool16
`qtssRTPStrJitter` Cumulative jitter for this stream.	Readable, preemptive safe	UInt32
`qtssRTPStrNumEyes` Number of clients connected to this stream.	Readable, preemptive safe	UInt32
`qtssRTPStrNumEyesActive` Number of clients playing this stream.	Readable, preemptive safe	UInt32
`qtssRTPStrNumEyesPaused` Number of clients connected but currently paused.	Readable, preemptive safe	UInt32

| 2009-06-01 | © 2002, 2009 Apple Inc. All Rights Reserved.

78

Attribute Name and Description	Access	Data Type
`qtssRTPStrPercentPacketsLost` Fixed percentage of lost packets for this stream.	Readable, preemptive safe	UInt16
`qtssRTPStrRecvBitRate` Average bit rate received by the client in bits per second.	Readable, preemptive safe	UInt32
`qtssRTPStrStreamRef` A `QTSS_StreamRef` used for sending RTP or RTCP packets to the client. Use `QTSS_WriteFlags` to specify whether each packet is an RTP or RTCP packet.	Readable, preemptive safe	QTSS_StreamRef
`qtssRTPStrTotalLostPackets` The total number of packets that have been lost for this stream.	Readable, preemptive safe	UInt32
`qtssRTPStrTotPacketsRecv` Total packets received by the client.	Readable, preemptive safe	UInt32
`qtssRTPStrTotPacketsDropped` Total packets dropped by the client.	Readable, preemptive safe	UInt16
`qtssRTPStrTotPacketsLost` Total packets lost.	Readable, preemptive safe	UInt16
`qtssRTPStrTransportType` The transport type.	Readable, preemptive safe	`QTSS_-RTPTransportType`

qtssRTSPHeaderObjectType

An object of type `qtssRTSPHeaderObjectType` consists of attributes containing all of the RTSP request headers associated with an individual RTSP request. An RTSP header object (`QTSS_RTSPHeaderObject`) is an instance of this object type.

The names of the attributes are the names of the RTSP headers associated with that RTSP request. For example, the following RTSP request has a Session header and a User-agent header:

```
DESCRIBE /foo.mov RTSP/1.0

Session: 20fj02ijf

User-agent: QTS/4.0.3
```

In this case, the value of the Session attribute is "20fj02ijf" and the value of the User-agent attribute is "QTS/4.0.3". Modules can get the value of a given header by calling `QTSS_GetValue`, `QTSS_GetValueAsString,` or `QTSS_GetValuePtr`.

qtssRTSPRequestObjectType

An object of type `qtssRTSPRequestObjectType` consists of attributes that describe a particular RTSP request. An RTSP request object (`QTSS_RTSPRequestObject`) is an instance of this object type and exists from the time the server receives a complete RTSP request from a client until the response is sent and the server moves on to the next request. An RTSP request object must be associated with a single RTSP session object (`QTSS_RTSPSessionObject`) for a given request made over a given connection.

With the exception of the RTSP Filter role, the value of each attribute is available in all roles that receive an object of type `QTSS_RTSPRequestObject`. When the RTSP Filter role receives an object of type `QTSS_RTSPRequestObject`, the only attribute that has a value is the `qtssRTSPReqFullRequest` attribute.

Each text name is identical to its enumerated type name.

Table 1-19 (page 80) lists the attributes for objects of type `qtssRTSPRequestObjectType`.

Note: All of these attributes are preemptive safe, so they can be read by calling `QTSS_GetValue`, `QTSS_GetValueAsString,` or `QTSS_GetValuePtr`.

Table 1-19   Attributes of type `qtssRTSPRequestObjectType`

Attribute Name and Description	Access	Data Type
`qtssRTSPReqAbsoluteURL` The full URL starting with "rtsp://".	Readable, preemptive safe	char
`qtssRTSPReqContentLen` Content length of incoming RTSP request body.	Readable, preemptive safe	UInt32
`qtssRTSPReqFileDigit` If the URI ends with one or more digits, this attribute points to those digits.	Readable, preemptive safe	char
`qtssRTSPReqFileName` All characters after the last path separator in the file system path.	Readable, preemptive safe	char
`qtssRTSPReqFilePath` URI for this request, converted to a local file system path.	Readable, preemptive safe	char

Attribute Name and Description	Access	Data Type
`qtssRTSPReqFilePathTrunc` Same as `qtssRTSPReqFilePath`, but without the last element of the path.	Readable, preemptive safe	char
`qtssRTSPReqFullRequest` The complete RTSP request as sent by the client. This attribute is available in every role that receives an object of type `QTSS_RTSPRequestObject`.	Readable, preemptive safe	char
`qtssRTSPReqIfModSinceDate` If the RTSP request contains an If-Modified-Since header, this attribute is the if-modified date converted to a value of type `QTSS_TimeVal`.	Readable, preemptive safe	QTSS_TimeVal
`qtssRTSPReqLateTolerance` Value of the late-tolerance field in the `x-RTP-Options` header, or -1 if not present.	Readable, preemptive safe	Float32
`qtssRTSPReqMethod` The RTSP method as a value of type `QTSS_RTSPMethod`.	Readable, preemptive safe	QTSS_RTSPMethod
`qtssRTSPReqMethodStr` The RTSP method of this request.	Readable, preemptive safe	char
`qtssRTSPReqNetworkMode` Network mode for the request. Possible values are `qtssRTPNetwork-ModeDefault,qtssRTPNetworkMode-Multicast, andqtssRTPNetwork-ModeUnicast`.	Readable, preemptive safe	Bool16
`qtssRTSPReqRealStatusCode` Same as the `qtssRTSPReqStatusCode` attribute but translated from a `QTSS_RTSPStatusCode` to an actual RTSP status code.	Readable, preemptive safe	UInt32
`qtssRTSPReqRespKeepAlive` Set this attribute to `true` if you want the server to keep the connection open after completion of the request. Otherwise, set this attribute to `false` if you want the server to terminate the connection upon completion of the request.	Readable, writable, preemptive safe	Bool16

Attribute Name and Description	Access	Data Type
qtssRTSPReqRespMsg The error message that is sent back to the client if the response was an error. A module sending an RTSP error to the client should set this attribute to be a text message that describes why the error occurred. It is also useful to write this message to a log file. Once the RTSP response has been sent, this attribute contains the response message.	Readable, writable, preemptive safe	char
qtssRTSPReqRootDir The root directory for this request. The default value for this attribute is the server's media folder path. Modules can set this attribute from the RTSP Route role.	Readable, writable, preemptive safe	char
qtssRTSPReqSkipAuthorization Set by a module that wants this request to be allowed by all authorization modules.	Readable, writable, preemptive safe	Bool16
qtssRTSPReqSpeed Value of the speed header.	Readable, preemptive safe	Float32
qtssRTSPReqStartTime The start time specified in the Range header of the PLAY request.	Readable, preemptive safe	Float64
qtssRTSPReqStatusCode The current status code for the request as QTSS_RTSPStatusCode. By default, the value is qtssSuccessOK. If a module sets this attribute and calls QTSS_SendRTSPHeaders, the status code in the header that the server generates contains the value of this attribute.	Readable, writable, preemptive safe	QTSS_RTSPStatusCode
qtssRTSPReqStopTime The stop time specified in the Range header of the PLAY request.	Readable, preemptive safe	Float64
qtssRTSPReqStreamRef A value of type QTSS_StreamRef for sending data to the RTSP client. This stream reference, unlike the one provided as an attribute in the QTSS_-RTSPSessionObject, never returns QTSS_WouldBlock in response to a QTSS_Write or a QTSS_WriteV call.	Readable, preemptive safe	QTSS_StreamRef

Attribute Name and Description	Access	Data Type
`qtssRTSPReqTruncAbsoluteURL` The URL without last element of the path.	Readable, preemptive safe	char
`qtssRTSPReqURI` URI for this request.	Readable, preemptive safe	char
`qtssRTSPReqURLRealm` The authorization entity for the client to display in the following string: "Please enter password for realm at server-name. The default value of this attribute is "Streaming Server."	Readable, writable, preemptive safe	char
`qtssRTSPReqUserName` The decoded user name, if provided by the RTSP request.	Readable, preemptive safe	char

qtssRTSPSessionObjectType

An object of type `qtssRTSPSessionObjectType` consists of attributes associated with an RTSP client-server connection. An RTSP session object (`QTSS_RTSPSessionObject`) is an instance of this object type and exists as long as the RTSP client is connected to the server. These attributes are valid for all roles that receive a `QTSS_RTSPSessionObject` in the structure the server passes to them.

Table 1-20 (page 83) lists the attributes for objects of type `qtssRTSPSessionObjectType`.

Note: All of these attributes are preemptive safe, so they can be read by calling `QTSS_GetValue`, `QTSS_GetValueAsString`, or `QTSS_GetValuePtr`.

Table 1-20 Attributes of objects of type `qtssRTSPSessionObjectType`

Attribute Name and Description	Access	Data Type
`qtssRTSPSesEventCntxt` An event context for the RTCP connection to the client. This attribute should primarily be used to wait for flow-controlled `EV_WR` event when responding to a client.	Readable, preemptive safe	QTSS_EventContextRef
`qtssRTSPSesID` An ID that uniquely identifies each RTSP session since the server started up.	Readable, preemptive safe	UInt32
`qtssRTSPSesLocalAddr` Local IP address for this RTSP session.	Readable, preemptive safe	UInt32

Attribute Name and Description	Access	Data Type
`qtssRTSPSesLocalAddrStr` Local IP address for the RTSP session in dotted-decimal format.	Readable, preemptive safe	char
`qtssRTSPSesLocalDNS` DNS name that corresponds to the local IP address for this RTSP session.	Readable, preemptive safe	char
`qtssRTSPSesLocalPort` Local port for the connection.	Readable, preemptive safe	UInt16
`qtssRTSPSesRemoteAddr` IP address of the client.	Readable, preemptive safe	UInt32
`qtssRTSPSesRemoteAddrStr` IP address of the client in dotted-decimal format.	Readable, preemptive safe	char
`qtssRTSPSesRemotePort` Remote (client) port for the connection.	Readable, preemptive safe	UInt16
`qtssRTSPSesStreamRef` A `QTSS_StreamRef` used for sending data to the RTSP client.	Readable, preemptive safe	QTSS_RTSPSessionStream
`qtssRTSPSesType` The RTSP session type. Possible values are `qtssRTSPSession`, `qtssRTSPHTTPSession` (an HTTP tunneled RTSP session), and `qtssRTSPHTTPInputSession`. Sessions of type `qtssRTSPHTTPInputSession` are usually very short lived.	Readable, preemptive safe	QTSS_RTSPSessionType

qtssServerObjectType

An object of type `qtssServerObjectType` consists of attributes that contain global server information, such as server statistics. A server object (`QTSS_ServerObject` is an instance of this object type.There is a single instance of this object type for each server. These attributes are valid for all roles that receive a `QTSS_ServerObject` in the structure the server passes to them.

Table 1-21 (page 85) lists the attributes for objects of type `qtssServerObjectType`.

> **Note:** Some of these attributes are not preemptive safe, as noted in Table 1-21 (page 85).

Table 1-21   Attributes of objects of type `qtssServerObjectType`

Attribute Name and Description	Access	Data Type
`qtssMP3SvrAvgBandwidth` Average MP3 bandwidth in bits per second that the server is currently sending.	Readable, writable, preemptive safe	UInt32
`qtssMP3SvrCurBandwidth` MP3 bandwidth in bits per second that the server is currently sending.	Readable, writable, preemptive safe	UInt32
`qtssMP3SvrCurConn` Number of currently connected MP3 client sessions.	Readable, writable, preemptive safe	UInt32
`qtssMP3SvrTotalBytes` Total number of MP3 bytes sent since the server started up.	Readable, writable, preemptive safe	UInt32
`qtssMP3TotalConn` Total number of MP3 client sessions since the server started up.	Readable, writable, preemptive safe	UInt32
`qtssRTPSvrAvgBandwidth` Average bandwidth output by the server in bits per second.	Readable, not preemptive safe	UInt32
`qtssRTPSvrCurBandwidth` Current bandwidth being output by the server in bits per second.	Readable, not preemptive safe	UInt32
`qtssRTPSvrCurConn` The number of clients currently connected to the server.	Readable, not preemptive safe	UInt32
`qtssRTPSvrCurPackets` Current packets per second being output by the server.	Readable, not preemptive safe	UInt32
`qtssRTPSvrNumUDPSockets` Number of UDP sockets currently being used by the server.	Readable, not preemptive safe	UInt32
`qtssRTPSvrTotalBytes` Total number of bytes output since the server started up.	Readable, not preemptive safe	UInt64
`qtssRTPSvrTotalConn` Total number of clients that have connected to the server since the server started up.	Readable, not preemptive safe	UInt32

Attribute Name and Description	Access	Data Type
`qtssRTPSvrTotalPackets` Total number of bytes output since the server started up.	Readable, not preemptive safe	UInt64
`qtssRTSPCurrentSessionCount` The number of clients that are currently connected over standard RTSP.	Readable, not preemptive safe	UInt32
`qtssRTSPHTTPCurrentSessionCount` The number of clients that are currently connected over RTSP/HTTP.	Readable, not preemptive safe	UInt32
`qtssServerAPIVersion` The API version supported by this server. The format of this value is 0x$MMMMmmmm$, where M is the major version number and m is the minor version number.	Readable, preemptive safe	UInt32
`qtssSvrDefaultIPAddrStr` The default IP address of the server as a string.	Readable, preemptive safe	char
`qtssSvrClientSessions` An object containing all client sessions stored as indexed `QTSS_-ClientSessionObject` objects.	Read	QTSS_Object
`qtssSvrConnectedUsers` The number of connected clients. The QTSSMP3StreamingModule is the only module that adds `QTSS_-ConnectedUserObject` objects to this attribute, but other modules can add `QTSS_ConnectedUserObject` objects filled in with their own data.	Readable, writable, not preemptive safe	QTSS_ConnectedUserObject
`qtssSvrCPULoadPercent` The percentage of CPU time the server is currently using.	Readable, not preemptive safe	Float32
`qtssSvrCurrentTimeMilliseconds` The server's current time in milliseconds. Getting the value of this attribute is equivalent to calling `QTSS_Milliseconds`.	Readable, not preemptive safe	QTSS_TimeVal
`qtssSvrDefaultDNSName` The "default" DNS name of the server.	Readable, preemptive safe	char

Attribute Name and Description	Access	Data Type
`qtssSvrDefaultIPAddr` The "default" IP address of the server.	Readable, preemptive safe	UInt32
`qtssSvrGMTOffsetInHrs` The time zone in which the server is running (offset from GMT in hours).	Readable, preemptive safe	SInt32
`qtssSvrHandledMethods` The methods that the server supports. Modules should append the methods they support to this attribute in their `QTSS_Initialize_Role`.	Readable, writable, not preemptive safe	QTSS_RTSPMethod
`qtssSvrIsOutOfDescriptors` If the server has run out of file descriptors, this attribute is `true`; otherwise, this attribute is `false`.	Readable, not preemptive safe	Bool16
`qtssSvrMessages` An object containing the server's error messages.	Readable, preemptive safe	QTSS_Object
`qtssSvrModuleObjects` A module object representing each module.	Readable, preemptive safe	QTSS_ModuleObject
`qtssSvrPreferences` An object representing each of the server's preferences.	Readable, preemptive safe	QTSS_PrefsObject
`qtssSvrRTSPPorts` An indexed attribute containing all the ports the server is listening on.	Readable, not preemptive safe	char
`qtssSvrRTSPServerHeader` The header that the server uses when responding to RTSP clients.	Readable, preemptive safe	char
`qtssSvrServerBuildDate` Date that the server was built.	Readable, preemptive safe	char
`qtssSvrServerName` The name of the server.	Readable, preemptive safe	char
`qtssSvrServerVersion` The version of the server.	Readable, preemptive safe	char
`qtssSvrStartupTime` The time at which the server started up.	Readable, preemptive safe	QTSS_TimeVal

Attribute Name and Description	Access	Data Type
`qtssSvrState` The current state of the server. Possible values are `qtssStartingUpState,` `qtssRunningState, qtssRefusing-ConnectionsState,qtssFatalError-State,` and `qtssShuttingDownState, qtssIdleState`. Modules can set the server state. If a module sets the server state, the server responds accordingly. Setting the server state to `qtssRefusing-ConnectionsState` causes the server to refuse new connections.Setting the server state to `qtssFatalErrorState` or to `qtssShuttingDownState` causes the server to quit. The`qtssFatalErrorState`state indicates that a fatal error has occurred but the server is not shutting down yet.	Readable, writable, not preemptive safe	QTSS_ServerState

qtssTextMessageObjectType

An object of type `qtssTextMessageObjectType` consists of attributes whose values are intended for display to the user or that are returned to the client. A text message object (`QTSS_TextMessageObject`) is an instance of this object type. To make localization easier, the attribute values are text strings.

Table 1-22 (page 88) lists the attributes for objects of type `qtssTextMessageObjectType`.

Table 1-22   Attributes of objects of type `qtssTextMessageObjectType`

Attribute Name and Description	Access	Data Type
`qtssListenPortAccessDenied`	Read only, preemptive safe	char
`qtssListenPortError`	Read only, preemptive safe	char
`qtssListenPortInUse`	Read only, preemptive safe	char
`qtssMsgAltDestNotAllowed` Request specifies an alternative destination and the server is not configured to support alternative destinations.	Read only, preemptive safe	char
`qtssMsgBadBase64`	Read only, preemptive safe	char

Attribute Name and Description	Access	Data Type
`qtssMsgBadFormat` The server could not parse the request.	Read only, preemptive safe	char
`qtssMsgBadModule` The server tried to run an invalid module.	Read only, preemptive safe	char
`qtssMsgBadRTSMethod` Request specified an invalid RTS method.	Read only, preemptive safe	char
`qtssMsgCannotCreatePIDFile` The server could not create the process ID file. See the `qtssPrefsPIDFile` attribute of the `qtssPrefsObjectType` described in Table 1-17 (page 61).	Read only, preemptive safe	char
`qtssMsgCannotSetRunUser` The server could not run under the user name specified by the `qtssPrefsRunUser` attribute of the `qtssPrefsObjectType` described in Table 1-17 (page 61).	Read only, preemptive safe	char
`qtssMsgCannotSetRunGroup` The server could not run under the group name specified by the `qtssPrefsRunGroup` attribute of the `qtssPrefsObjectType` described in Table 1-17 (page 61).	Read only, preemptive safe	char
`qtssMsgCantSetupMulticast` Server is not configured for multicast.	Read only, preemptive safe	char
`qtssMsgCantWriteFile`	Read only, preemptive safe	char
`qtssMsgColonAfterHeader` Request's header is not followed by a colon (`:`) character .	Read only, preemptive safe	char
`qtssMsgCouldntListen` This text message is not used.	Read only, preemptive safe	char
`qtssMsgDefaultRTSPAddrUnavail` The IP address specified by the `qtssPrefsRTSPIPAddr` attribute could not be found or failed in some way.	Read only, preemptive safe	char
`qtssMsgFileNameTooLong` Request contains a file name that is too long.	Read only, preemptive safe	char
`qtssMsgInitFailed` The server could not initialize itself.	Read only, preemptive safe	char

Attribute Name and Description	Access	Data Type
`qtssMsgNoClientPortInTransport` Request contains a transport header that does not specify the client's port number.	Read only, preemptive safe	char
`qtssMsgNoEOLAfterHeader` Request's header is not terminated by an end of line character.	Read only, preemptive safe	char
`qtssMsgNoMessage` No message.	Read only, preemptive safe	char
`qtssMsgNoModuleFolder` The server could not find the module folder.	Read only, preemptive safe	char
`qtssMsgNoModuleForRequest` Request specifies a module the server does not have.	Read only, preemptive safe	char
`qtssMsgNoRTSPInURL` Request specifed a URL that does not support RTSP.	Read only, preemptive safe	char
`qtssMsgNoRTSPVersion` Request did not specify an RTSP version.	Read only, preemptive safe	char
`qtssMsgNoPortsSucceeded`	Read only, preemptive safe	char
`qtssMsgNoSesIDOnDescribe` The Describe section of the request's header does not contain a session ID.	Read only, preemptive safe	char
`qtssMsgNoSessionID` Request does not contain a session ID.	Read only, preemptive safe	char
`qtssMsgNotConfiguredForIP` The server is not configured for IP.	Read only, preemptive safe	char
`qtssMsgNoURLInRequest` Request did not contain a URL.	Read only, preemptive safe	char
`qtssMsgOutOfPorts` The server could not accept the request because it is out of ports.	Read only, preemptive safe	char
`qtssMsgRefusingConnections` The server is refusing connections.	Read only, preemptive safe	char
`qtssMsgRegFailed` A module failed to register.	Read only, preemptive safe	char
`qtssMsgRequestTooLong` Request is too long.	Read only, preemptive safe	char

Attribute Name and Description	Access	Data Type
`qtssMsgRTCPPortMustBeOneBigger` Request contains an RTCP port number that is not bigger than the RTP port number by 1.	Read only, preemptive safe	char
`qtssMsgRTPPortMustBeEven` Request contains an RTP port number that is odd instead of even.	Read only, preemptive safe	char
`qtssMsgSockBufSizesTooLarge`	Read only, preemptive safe	char
`qtssMsgTooManyClients` The server has too many connections to accept this connection.	Read only, preemptive safe	char
`qtssMsgTooMuchThroughput` The server is consuming too much bandwidth to accept this request.	Read only, preemptive safe	char
`qtssMsgSomePortsFailed`	Read only, preemptive safe	char
`qtssMsgURLInBadFormat` Request specifed a URL that is properly formatted.	Read only, preemptive safe	char
`qtssMsgURLTooLong` Request contains a URL that is longer than 256 bytes.	Read only, preemptive safe	char
`qtssServerPrefMissing` A required server preference is missing from the server's configuration.	Read only, preemptive safe	char
`qtssServerPrefWrongType` A required server preference is of the wrong type.	Read only, preemptive safe	char

qtssUserProfileObjectType

An object of type `qtssUserProfileObjectType` consists of attributes whose values describe a user's profile.

Table 1-23 (page 91) lists the attributes for objects of type `qtssUserProfileObjectType`.

Table 1-23 Attributes of objects of type `qtssUserProfileObjectType`

Attribute Name and Description	Access	Data Type
`qtssUserPassword` The user's password.	Readable, writable preemptive safe	char

Attribute Name and Description	Access	Data Type
`qtssUserGroups` Groups of which the user is a member. This is a multi-valued attribute. Each group name is a C strings padded with enough nulls to make all of the group names the same length.	Readable, writable preemptive safe	char
`qtssUserName` The user's name.	Readable, preemptive safe	char
`qtssUserRealm` Authentication realm for this user.	Readable, writable preemptive safe	char

QTSS Streams

The QTSS programming interface provides QTSS stream references as a generalized stream abstraction. Streams can be used for reading and writing data to many types of I/O sources, including, but not limited to files, the error log, and sockets and for communicating with the client via RTSP or RTP. In all RTSP roles, for example, modules receive an object of type `QTSS_RTSPRequestObject` that has a `qtssRTSPReqStreamRef` attribute. The value of this attribute is of type `QTSS_StreamRef,` and it can be used for sending RTSP response data to the client.

Unless otherwise noted, all streams are asynchronous. When using the asynchronous QTSS file system callbacks, modules should be prepared to receive the `QTSS_WouldBlock` result code, subject to the restrictions and rules of each stream type described in this section. The `QTSS_WouldBlock` error is returned from a stream callback when completing the requested operation would require the current thread to block. For instance, `QTSS_Write` on a socket will return `QTSS_WouldBlock` if the socket is currently subject to flow control. For information on threading and asynchronous I/O, see the section "Runtime Environment for QTSS Modules" (page 28).

When a module receives the `QTSS_WouldBlock` result code, modules should call the `QTSS_RequestEvent` callback routine to request a notification from the server when the specified stream becomes available for I/O. After calling `QTSS_RequestEvent,` the module should return control immediately to the server. The module will be re-invoked in the same role in the exact same state when the specified stream is available for I/O.

All stream references are of type `QTSS_StreamRef`. The QTSS programming interface uses following stream types:

`QTSS_ErrorLogStream`
> Used for writing binary data to the server's error log. There is a single instance of this stream type, which is passed to each module in the Initialize role. When data is written to this stream, modules that have

registered for the Error Log role are invoked. For information about this role, see the section "Error Log Role" (page 34). All operations on this stream type are synchronous.

QTSS_FileStream

Represents a file and is obtained by making the QTSS_OpenFileStream callback. If the file stream is opened with the qtssFileStreamAsync flag, callers should expect to receive a result code of QTSS_WouldBlock when they call QTSS_Read, QTSS_Write, and QTSS_WriteV.

QTSS_RTSPSessionStream

Used for reading data (QTSS_Read) from an RTSP client and writing data (QTSS_Write or QTSS_WriteV) to an RTSP client. The server may encounter flow control conditions, so modules should be prepared to handle QTSS_WouldBlock result codes when reading from or writing to this stream type. Calling QTSS_Read means that you are reading the request body sent by the client to the server. This stream reference is an attribute of the object QTSS_RTSPSessionObject.

QTSS_RTSPRequestStream

Used for reading data (QTSS_Read) from an RTSP client and writing data (QTSS_Write or QTSS_WriteV) to an RTSP client. This stream is identical to the QTSS_RTSPSessionStream stream except that data written to streams of this type is buffered in memory until a full RTSP response is constructed. Because the data is buffered internally, modules do not receive QTSS_WouldBlock errors when writing to streams of this type. Calling QTSS_Read on this type of stream means that you are reading the request body sent by the client to the server. Modules that call QTSS_Read to read this type of stream should be prepared to handle a result code of QTSS_WouldBlock. This stream reference is an attribute of the object QTSS_RTSPRequestObject.

QTSS_RTPStreamStream

Used for writing data to an RTP client. When writing to a stream of this type, a single write call corresponds to a single, complete RTP packet, including headers. Currently, it is not possible to use the QTSS_RequestEvent callback to receive events for this stream, so if QTSS_Write or QTSS_WriteV returns QTSS_WouldBlock, modules must poll periodically for the blocking condition to be lifted. This stream reference is an attribute of the object QTSS_RTPStreamObject.

QTSS_SocketStream

Represents a socket. This stream type allows modules to use the QTSS stream event mechanism (QTSS_RequestEvent) for raw socket I/O. (In fact, the QTSS_RequestEvent callback is the only stream callback available for this type of stream.) Modules should read sockets asynchronously and should use the operating system's socket function to read from and write to sockets. When those routines reach a blocking condition, the module can call QTSS_RequestEvent to be notified when the blocking condition has cleared.

Table 1-24 (page 94) uses an "X" to summarize the I/O-related callback routines that are appropriate for each type of stream.

Table 1-24   Streams and appropriate callback routines

Stream Type	Read	Seek	Flush	Advise	Write	WriteV	RequestEvent	SignalS
File Stream	X	X		X			X	X
Error Log					X			
Socket Stream							X	
RTSP Session Stream	X		X		X	X	X	
RTSP Request Stream	X		X		X	X	X	
RTP Stream	X		X		X	X		

QTSS Services

QTSS services are services the modules can access. The service may be a built-in service provided by the server or an added service provided by another module. An example of a service would be a logging module that allows other modules to write messages to the error log.

Modules use the callback routines described in the section "Service Callback Routines" (page 168) to register and invoke services. Modules add and find services in a way that is similar to the way in which they add and find attributes of an object.

Every service has a name. To invoke a service, the calling module must know the name of the service and resolve that name into an ID.

Each service has its own specific parameter block format. Modules that export services should carefully document the services they export. Modules that call services should fail gracefully if the service isn't available or returns an error.

A module that implements a service calls `QTSS_AddService` in its Register role to add the service to the server's internal database of services, as shown in the following code:

```
void MyAddService()
{
    QTSS_Error theErr = QTSS_AddService("MyService", &MyServiceFunction);
}
```

The `MyServiceFunction` corresponds to the name of a function that must be implemented in the same module. Here is a stub implementation of the `MyServiceFunction`:

```
QTSS_Error MyServiceFunction(MyServiceArgs* inArgs)
{
    // Each service function must take a single void* argument
    // Implement the service here.
    // Return a QTSS_Error.
}
```

To use a service, a module must get the service's ID by calling `QTSS_IDForService` and providing the name of the service as a parameter. With the service's ID, the module calls `QTSS_DoService` to cause the service to run, as shown in .

Listing 1-1   Starting a service

```
void MyInvokeService()
{
    // Service functions take a single void* parameter that corresponds
    // to a parameter block specific to the service.
    MyServiceParamBlock theParamBlock;

    // Initialize service-specific parameters in the parameter block.
    theParamBlock.myArgument = xxx;
    QTSS_ServiceID theServiceID = qtssIllegalServiceID;
    // Get the service ID by providing the name of the service.
    QTSS_Error theErr = QTSS_IDForService('MyService', &theServiceID);
    if (theErr != QTSS_NoErr)
        return; // The service isn't available.

    // Run the service.
    theErr = QTSS_DoService(theServiceID, &theParamBlock);
}
```

Built-in Services

The QuickTime Streaming Server provides built-in services that modules may invoke using the service routines. In this version of the QTSS programming interface, there is one built-in service:

```
#define QTSS_REREAD_PREFS_SERVICE "RereadPreferences"
```

Invoking the Reread Preferences service causes the server to reread its preferences and invoke each module in the Reread Preferences role, if they have registered for that role.

To invoke a built-in service, retrieve the service ID of the service by calling `QTSS_IDForService`. Then call `QTSS_DoService` to run the service.

Automatic Broadcasting

The Streaming Server can accept RTSP ANNOUNCE requests from QuickTime broadcasters. Support for ANNOUNCE requests and the ability of the server to act as an RTSP client allow the server to initiate new relay sessions. This section describes the two ways in which an automatic broadcast can be initiated, how ANNOUNCE requests work with SDP, and how the `qtaccess` and `qtusers` files control automatic broadcasting.

Automatic Broadcasting Scenarios

QTSS supports two automatic broadcasting scenarios:

- Pull then push. To initiate automatic broadcast, an RTSP client sends standard RTSP requests to request a stream and the server then relays the stream to one or more other streaming servers. This scenario is described in the section "Pull Then Push" (page 97).

- Listen then push. In this scenario, an automatic broadcast is initiated when the streaming server receives an ANNOUNCE request. This scenario is described in the section "Listen Then Push" (page 98).

Pull Then Push

The user can request a stream from a remote source by making standard DESCRIBE/SETUP/PLAY requests and then relay it to one or more destinations. This functionality can be useful when an organization only wants one copy of an outside stream to consume bandwidth on its Internet connection. The relay would sit just inside the corporate network and push the stream to a reflector (possibly itself). `Figure 2–7` (page 97) provides an example of the pull-then-push scenario.

Figure 1-7   Pull-then-push automatic broadcasting
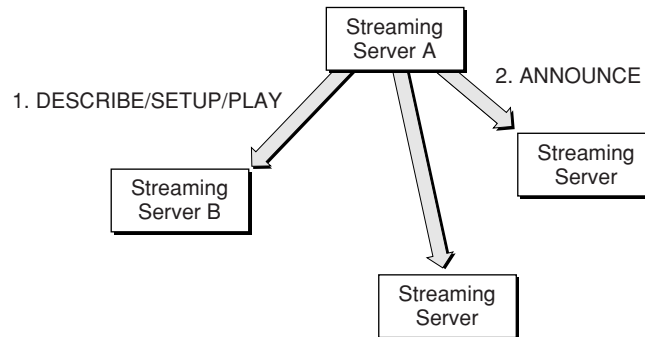
Using `Figure 2–7` (page 97) as a reference, the steps for the pull-then-push scenario are as follows:
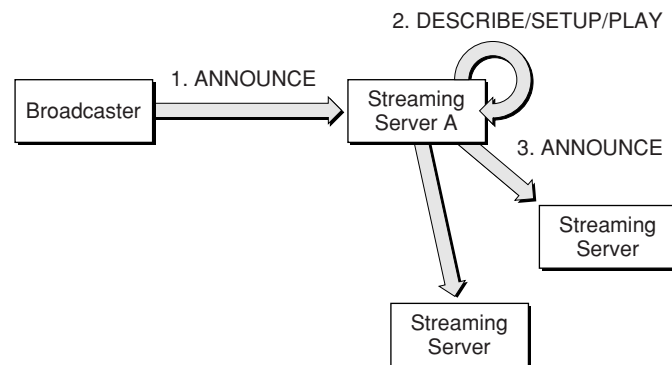
1. Streaming Server A (the relay client) sends standard RTSP client DESCRIBE/SETUP/PLAY requests to a remote server, Streaming Server B.

2. The relay "client" (Streaming Server A) that requested the stream will begin receiving it and then send an ANNOUNCE to all of the destinations listed in the relay configuration for that particular incoming stream.

Listen Then Push

The streaming server can be configured to send incoming streams created by an ANNOUNCE request to one or more destination machines automatically. This can be useful for setting up an automated broadcast network. `Figure 2-8` (page 98) provides an example of the pull-then-push scenario.

Figure 1-8   Listen-then-push automatic broadcasting

Using `Figure 2-8` (page 98) as a reference, the steps for the listen -then-push scenario are as follows:

- A remote machine (a broadcaster or a relay) sends an ANNOUNCE request to Streaming Server A. The streaming server may accept or deny the request. If it accepts the request, the streaming server checks its relay configuration to determined whether the stream should be relayed.

- If the stream should be relayed, the streaming server will send standard RTSP client DESCRIBE/SETUP/PLAY request to itself.

- The relay "client" (Streaming Server A) that requested the stream will begin receiving it and then send an ANNOUCE to all of the destinations listed in its relay configuration for that particular incoming stream.

By default, authentication is required for automatic broadcasts. ANNOUNCE requests from broadcasters are filtered through the authentication mechanism active in the server. To support broadcast authentication, a new WRITE directive has been added to qtaccess file. The new directive allows SDP files to be written to the `movies` folder.

ANNOUNCE Requests and SDP

The ANNOUNCE request contains the Session Description Protocol (SDP) information for the broadcast. The ANNOUNCE request's URI value may contain path delimiters in order to provide name space functionality.

When a broadcast is initiated by an ANNOUNCE request, the SDP information is stored in an in-memory broadcast list. To terminate a broadcast, the broadcaster sends to the server a TEARDOWN request, which causes the server to close the broadcast session and discard the SDP information. Similarly, dropped RTSP connections and broadcasters that do no send RTCP sender reports to the server within a 90-second window cause the server to close the broadcast session and discard the SDP information.

To support multiple SDP references to the same broadcast for announced UDP and TCP broadcasts, the port setting is zero in the ANNOUNCE header. Here is an example:

```
m=audio 0 RTP/AVP
```

The a=x-urlmap tag is required to support sharing streams between broadcasts (where one stream comes from one broadcaster and another stream comes from another broadcaster). The a=x-urlmap tag should appear in the SDP that references the source SDP. Here is an example:

```
a=x-urlmap: someotherbroadcastURL/TrackID=1
```

Access Control of Announced Broadcasts

To control automatic broadcasting, two new user tags have been defined in the qtaccess file. Table 2-25 (page 99) lists the new tags.

Table 1-25   Access control user tags

Tag	Purpose
valid-user	Specifies that the user can have access to the requested movie if the client provides a name and password that match an entry in the qtusers file. The tag is written as `require valid-user`.
any-user	Specifies that any user can have access to the requested movie, with no requirement that the user be defined in the `qtusers` file or that the client provide a name and password that is checked. The tag is written as `require any-user`.

By default, the `qtaccess` file allows read access for all directives in the file. To allow announced broadcasts, the `qtaccess` file must contain a `Limit` directive that allows writing.

The purpose of the `Limit` directive is to restrict the effect of access controls to RTSP readers or writers. The following example limits the `require` access control so that only users defined in the `qtusers` file can RTSP PLAY a broadcast to the server. All other normal client PLAY requests are available to any user:

```
<Limit WRITE>

require valid-user

</Limit>
```

> **Note:** The termination of the Limit directive (`</Limit>`) must be placed on its own line.

The following example allows movie viewing by any user in the `qtusers` file that is in the `movie_watchers` group and the user `john`. Broadcasters must be in the `movie_broadcasters` group to broadcast to this directory or its protected branches.

```
<Limit READ>

require group movie_watchers

require user john

</Limit>

<Limit WRITE>

require group movie_broadcasters

</Limit>
```

> **Note:** Strings in the `qtaccess` file are case-sensitive.

The following example has the same effect as the previous example. It works because the default behavior is to limit access to reading when no limit field is specified.

```
require group movie_watchers

<Limit WRITE>

require group movie_broadcasters

</Limit>


require user john
```

The following example allows movie viewing and broadcasting by any user in the `qtusers` file that is in the `movie_watchers_and_broadcasters` group:

```
<Limit READ WRITE>
```

```
require group movie_watchers_and_broadcasters
</Limit>
```

Broadcaster-to-Server Example

This section shows a typical exchange between a client and a server in order to initiate an announced broadcast. The following example shows a UDP multicast. Announced broadcasts can also set up requests with using unicast RTP/AVP/UDP streams as well as RTP/AVP/TCP interleaved streams. For more information, see RFC 2326.

Client to server:

```
ANNOUNCE rtsp://server.example.com/meeting RTSP/1.0
CSeq: 90
Content-Type: application/sdp
Content-Length: 121
v=0
o=camera1 3080117314 3080118787 IN IP4 195.27.192.36
s=IETF Meeting, Munich – 1
i=The thirty-ninth IETF meeting will be held in Munich, Germany
u=http://www.ietf.org/meetings/Munich.html
e=IETF Channel 1 <ietf39-mbone@uni-koeln.de>
p=IETF Channel 1 +49-172-2312 451
c=IN IP4 224.0.1.11/127
t=3080271600 3080703600
a=tool:sdr v2.4a6
a=type:test
m=audio 0 RTP/AVP 5
a=control:trackID=1
c=IN IP4 224.0.1.11/127
a=ptime:40
m=video 0 RTP/AVP 31
a=control:trackID=2
c=IN IP4 224.0.1.12/127
```

Server to client:

```
RTSP/1.0 200 OK

CSeq: 90
```

Client to server:

```
SETUP rtsp://server.example.com/meeting/trackID=1 RTSP/1.0

CSeq: 91

Transport: RTP/AVP;multicast;destination=224.0.1.11;

client_port=21010-21011;mode=record;ttl=127
```

Server to client:

```
RTSP/1.0 200 OK

CSeq: 91

Session: 50887676

Transport: RTP/AVP;multicast;destination=224.0.1.11;

client_port=21010-21011;serverport=6000-6001;mode=receive;ttl=127
```

Client to server:

```
SETUP rtsp://server.example.com/meeting/trackID=2 RTSP/1.0

CSeq: 92

Session: 50887676

Transport: RTP/AVP;multicast;destination=224.0.1.12;

client_port =61010-61011;mode=record;ttl=127
```

Server to client:

```
RTSP/1.0 200 OK

CSeq: 92

Transport: RTP/AVP;multicast;destination=224.0.1.12;

client_port =61010-61011;serverport=6002-6003;mode=record;ttl=127
```

Client to server:

| 2009-06-01 | © 2002, 2009 Apple Inc. All Rights Reserved.

102

```
    RECORD rtsp://server.example.com/meeting RTSP/1.0

    CSeq: 93

    Session: 50887676
```

Server to client:

```
    RTSP/1.0 200 OK

    CSeq: 93
```

Additional Trace Examples

This section provides three traces. The first trace is from the QuickTime Broadcaster, and it is sending MPEG 4 streams using TCP. The second trace is also from the QuickTime Broadcaster, but it is using UDP. The third trace is from RFC 2326 (RTSP) showing the ANNOUNCE and RECORD RTSP methods using UDP transport.

The broadcaster requests to notice are

- RTSP ANNOUNCE to send the SDP file to the server

- RTSP SETUP to send a Transport header setting `mode=record`; the direction of the stream is implicitly from the perspective of the server

- RTSP RECORD to start the broadcast

The requests mirror the streaming client requests:

- RTSP DESCRIBE to receive the SDP file from the server

- RTSP SETUP to set up each stream

- RTSP PLAY to start the streams

Trace of QuickTime Broadcaster Using TCP

Here is a trace of a QuickTime Broadcaster sending MPEG 4 streams using TCP. A TCP connection uses the same set of RTSP requests with the standard specified transport of RTP/AVP/TCP and the port identifier of `interleaved=` for each stream.

For this example, authentication and authorization has been disabled by a `qtaccess` file to allow any user to annouce a broadcast. The broadcast file is relative to the `movies` directory. If an SDP file already exists for the URL, it is replaced. Clients that are already connected to the URL are not updated with the new SDP as doing so would require a new `DESCRIBE` from the client, and there currently is no way to notify clients of the SDP change.

| 2009-06-01 | © 2002, 2009 Apple Inc. All Rights Reserved.

103

Client to server:

```
ANNOUNCE rtsp://127.0.0.1/mystream.sdp RTSP/1.0\r\n

CSeq: 1\r\n

Content-Type: application/sdp\r\n

User-Agent: QTS (qtver=6.1;cpu=PPC;os=Mac 10.2.3)\r\n

Content-Length: 790\r\n

\r\n

c=IN IP4

127.0.0.1\ra=x-qt-text-nam:test\ra=x-qt-text-cpy:apple\

ra=x-qt-text-aut:john\

ra=x-qt-text-inf:none\

ra=mpeg4-iod:"data:application/

mpeg4-iod;base64,AoF/

AE8BAQEBAQOBEgABQHRkYXRhOmFwcGxpY2F0aW9uL21wZWc0LW9kLWF1L102Jhc2U2NCxBVGdC

R3dVZkF4Y0F5U1FBWlFRTklCRUFGGM0FBQVBvQUFBRERVQVlCQkFFFWkFwOERGUUJsQlFFRTlFC

VUFCOUFBQUQ2QUFBBBQStnQVlCQVQXc9PQQNAQUAAMgAAAAAAAAAAYJAQAAAAAAAAA2EAAkA+

ZGF0YTphcHBCsaWNhdGlvbi9tcGVnNC1iaWZzLWF1L102Jhc2U2NCx3QkFTZ1RBcUJJYSmhCSWhR

UlFVL0FBPT0EEgINAAAUAAAAAAAAAFAwAAQAYJAQAAAAAAAA"\ra=isma-

compliance:1,1.0,1\rm=audio 0 RTP/AVP 96\ra=rtpmap:96

X-QT/8000/1\ra=control:trackid=1\rm=video 0 RTP/AVP 97\ra=rtpmap:97

MP4V-ES\ra=fmtp:97

profile-level-id=1;config=000001B0F3000001B50EE040C0CF0000010000000120008440FA2850
  20F0

A31F\ra=mpeg4-esid:201\ra=cliprect:0,0,240,320\ra=control:trackid=2\r
```

Server to client:

```
RTSP/1.0 200 OK\r\n

Server: QTSS/4.1.3.x (Build/425; Platform/MacOSX; Release/Development;)\r\n

Cseq: 1\r\n

\r\n
```

Client to server:

```
// The broadcaster is trying to determine if RECORD is supported. QTSS 4.0 used
an  Apple

// method of RECEIVE instead of the RECORD.


OPTIONS rtsp://127.0.0.1/mystream.sdp RTSP/1.0\r\n

CSeq: 2\r\n

User-Agent: QTS (qtver=6.1;cpu=PPC;os=Mac 10.2.3)\r\n

\r\n
```

Server to client:

```
RTSP/1.0 200 OK\r\n

Server: QTSS/4.1.3.x (Build/425; Platform/MacOSX; Release/Development;)\r\n

Cseq: 2\r\n

Public: DESCRIBE, SETUP, TEARDOWN, PLAY, PAUSE, ANNOUNCE, SET_PARAMETER, RECORD\r\n

\r\n
```

Client to server:

```
// Here is the first setup with the transport defined from the client to the

// server. The URL is the same as when a client performs a setup requesting

// a stream. QTSS does not allow a SETUP on a stream that is already set up

// and will return an error. This can happen in two ways.

// 1) A broadcast software error that does not change the URL.

// 2) A broadcast dies without performing a teardown. In this case, the

// broadcast session has to timeout and die before another setup can occur.

// The server uses a short timeout of 20 seconds for broadcast sessions. The

// timeout is refreshed by any packet received from the broadcaster.

SETUP rtsp://127.0.0.1/mystream.sdp/trackid=1 RTSP/1.0\r\n

CSeq: 3\r\n

Transport: RTP/AVP/TCP;unicast;mode=record;interleaved=0-1\r\n

User-Agent: QTS (qtver=6.1;cpu=PPC;os=Mac 10.2.3)\r\n

Accept-Language: en-US\r\n

\r\n
```

Server to client:

```
// The server responds with the interleaved values. If the values conflict,

// the client will change them so each stream has a unique set of

// interleaved IDs.


RTSP/1.0 200 OK\r\n

Server: QTSS/4.1.3.x (Build/425; Platform/MacOSX; Release/Development;)\r\n

Cseq: 3\r\n

Cache-Control: no-cache\r\n

Session: 6664885458621367225\r\n

Date: Thu, 13 Feb 2003 21:34:27 GMT\r\n

Expires: Thu, 13 Feb 2003 21:34:27 GMT\r\n

Transport: RTP/AVP/TCP;unicast;mode=record;interleaved=0-1\r\n

\r\n
```

Client to server:

```
SETUP rtsp://127.0.0.1/mystream.sdp/trackid=2 RTSP/1.0\r\n

CSeq: 4\r\n

Transport: RTP/AVP/TCP;unicast;mode=record;interleaved=2-3\r\n

Session: 6664885458621367225\r\n

User-Agent: QTS (qtver=6.1;cpu=PPC;os=Mac 10.2.3)\r\n

Accept-Language: en-US\r\n

\r\n
```

Server to client:

```
RTSP/1.0 200 OK\r\n

Server: QTSS/4.1.3.x (Build/425; Platform/MacOSX; Release/Development;)\r\n

Cseq: 4\r\n

Session: 6664885458621367225\r\n

Cache-Control: no-cache\r\n

Date: Thu, 13 Feb 2003 21:34:27 GMT\r\n

Expires: Thu, 13 Feb 2003 21:34:27 GMT\r\n

Transport: RTP/AVP/TCP;unicast;mode=record;interleaved=2-3\r\n

\r\n
```

Client to server:

```
// This is the equivalent to a client PLAY request. The broadcaster is now
// starting the streams.

RECORD rtsp://127.0.0.1/mystream.sdp RTSP/1.0\r\n
CSeq: 5\r\n
Session: 6664885458621367225\r\n
User-Agent: QTS (qtver=6.1;cpu=PPC;os=Mac 10.2.3)\r\n
\r\n
```

Server to client:

```
// RTCPs will be sent back on the channels to show the number of watching
// clients.
RTSP/1.0 200 OK\r\n
Server: QTSS/4.1.3.x (Build/425; Platform/MacOSX; Release/Development;)\r\n
Cseq: 5\r\n
Session: 6664885458621367225\r\n
RTP-Info: url=trackid=1,url=trackid=2\r\n
\r\n
```

Client to server:

```
PAUSE rtsp://127.0.0.1/mystream.sdp RTSP/1.0\r\n
CSeq: 6\r\n
Session: 6664885458621367225\r\n
User-Agent: QTS (qtver=6.1;cpu=PPC;os=Mac 10.2.3)\r\n
\r\n
```

Server to client:

```
RTSP/1.0 200 OK\r\n
Server: QTSS/4.1.3.x (Build/425; Platform/MacOSX; Release/Development;)\r\n
Cseq: 6\r\n
Session: 6664885458621367225\r\n
```

```
\r\n
```

Client to server:

```
// A TEARDOWN stops the broadcast streams. It does not stop the clients or
// their streams. By default, QTSS allows a restarted or different broadcaster
// to send to the same URL and the clients will receive the new streams. This
// can be both good and bad since the broadcaster can change the stream media
// type on the clients. The streamingserver.xml file provides an attribute
// that allows the server to force clients to disconnect if the broadcaster
// disconnects. The broadcast receiver is recommended to have a way for an
// administrator or the broadcaster to tear down sessions that have failed.
// The server adds a 30 second timeout between SSRC values to prevent someone
// from pirating a stream.  As long as a stream is playing with the initial
// SSRC, another stream arriving on the same ports will not be reflected to
// clients. Attempts to pirate a steam usually occur by accident when users
// manually set their SDP ports.
TEARDOWN rtsp://127.0.0.1/mystream.sdp RTSP/1.0\r\n
CSeq: 7\r\n
Session: 6664885458621367225\r\n
User–Agent: QTS (qtver=6.1;cpu=PPC;os=Mac 10.2.3)\r\n
\r\n
```

Server to client:

```
// The server removes the SDP file from the movies directory on teardown or
// broadcaster timeout.
RTSP/1.0 200 OK\r\n
Server: QTSS/4.1.3.x (Build/425; Platform/MacOSX; Release/Development;)\r\n
Cseq: 7\r\n
Session: 6664885458621367225\r\n
Connection: Close\r\n
\r\n
```

Trace of UDP Broadcast with Negotiated Server Ports

The only significant addition to RFC 2326 is that when receiving a broadcast over UDP, the QuickTime server uses SETUP with `mode=RECORD` to generate and send back to the client a UDP port to use when the SDP contains a port value of 0 for a given stream. Otherwise, the server uses the SDP-defined port to receive the streams. The server's receive port is declared in the SETUP response transport header. The format looks exactly as if a client were performing a SETUP request for a stream from the server and then receiving the port the server is sending from.

Client to server:

```
ANNOUNCE rtsp://127.0.0.1/mystream.sdp RTSP/1.0\r\n

CSeq: 1\r\n

Content-Type: application/sdp\r\n

User-Agent: QTS (qtver=6.1;cpu=PPC;os=Mac 10.2.3)\r\n

Content-Length: 790\r\n

\r\n

c=IN IP4

127.0.0.1\ra=x-qt-text-nam:test\ra=x-qt-text-cpy:apple\

ra=x-qt-text-aut:john\

ra=x-qt-text-inf:none\

ra=mpeg4-iod:"data:application/

mpeg4-iod;base64,AoF/

AE8BAQEBAQOBEgABQHRkYXRhOmFwcGxpY2F0aW9uL21wZWc0LW9kLWF1O2Jhc2U2NCxBVGdC

R3dVZkF4Y0F5U1FBWlFFRTklCRUFGM0FFBQVBvQUFBRERVQVlCQQkFFWkFwOERGGUUJsQlFRTlFC

VUFCOUFBQUQ2QUFBQSStnQVlCCQXc9PQQNAQUAAMgAAAAAAAAAAYJAQAAAAAAAAAAA2EAAkA+

ZGF0YTphcHcHBsaWNhdGlvbi9tcGVnNC1iaWZzLWF1O2Jhc2U2NCx3QkFTZ1RBRBcUJYSmhCSWhR

UlFVL0FFBPT0EEgINAAAAUAAAAAAAAAAAAFAwAAQAYJAQAAAAAAAAA"\ra=isma-

compliance:1,1.0,1\rm=audio 0 RTP/AVP 96\ra=rtpmap:96

X-QT/8000/1\ra=control:trackid=1\rm=video 0 RTP/AVP 97\ra=rtpmap:97

MP4V-ES\ra=fmtp:97

profile-level-id=1;config=000001B0F3000001B50EE040C0CF000001000000120008440FA2850
  20F0

A31F\ra=mpeg4-esid:201\ra=cliprect:0,0,240,320\ra=control:trackid=2\r
```

Server to client:

```
RTSP/1.0 200 OK\r\n
```

| 2009-06-01 | © 2002, 2009 Apple Inc. All Rights Reserved.

109

```
Server: QTSS/4.1.3.x (Build/425; Platform/MacOSX; Release/Development;)\r\n

Cseq: 1\r\n

\r\n
```

Client to server:

```
OPTIONS rtsp://127.0.0.1/mystream.sdp RTSP/1.0\r\n

CSeq: 2\r\n

User-Agent: QTS (qtver=6.1;cpu=PPC;os=Mac 10.2.3)\r\n

\r\n
```

Server to client:

```
RTSP/1.0 200 OK\r\n

Server: QTSS/4.1.3.x (Build/425; Platform/MacOSX; Release/Development;)\r\n

Cseq: 2\r\n

Public: DESCRIBE, SETUP, TEARDOWN, PLAY, PAUSE, ANNOUNCE, SET_PARAMETER, RECORD\r\n

\r\n
```

Client to server:

```
SETUP rtsp://127.0.0.1/mystream.sdp/trackid=1 RTSP/1.0\r\n

CSeq: 3\r\n

Transport: RTP/AVP;unicast;client_port=6974-6975;mode=record\r\n

User-Agent: QTS (qtver=6.1;cpu=PPC;os=Mac 10.2.3)\r\n

Accept-Language: en-US\r\n

\r\n
```

Server to client:

```
RTSP/1.0 200 OK\r\n

Server: QTSS/4.1.3.x (Build/425; Platform/MacOSX; Release/Development;)\r\n

Cseq: 3\r\n

Cache-Control: no-cache\r\n

Session: 1549167172936112945\r\n
```

```
Date: Thu, 13 Feb 2003 21:59:22 GMT\r\n

Expires: Thu, 13 Feb 2003 21:59:22 GMT\r\n

Transport:

RTP/AVP;unicast;client_port=6974-6975;mode=record;source=127.0.0.1;

server_port=6976-6977\r\n

\r\n
```

Client to server:

```
SETUP rtsp://127.0.0.1/mystream.sdp/trackid=2 RTSP/1.0\r\n

CSeq: 4\r\n

Transport: RTP/AVP;unicast;client_port=6972-6973;mode=record\r\n

Session: 1549167172936112945\r\n

User-Agent: QTS (qtver=6.1;cpu=PPC;os=Mac 10.2.3)\r\n

Accept-Language: en-US\r\n

\r\n
```

Server to client:

```
RTSP/1.0 200 OK\r\n

Server: QTSS/4.1.3.x (Build/425; Platform/MacOSX; Release/Development;)\r\n

Cseq: 4\r\n

Session: 1549167172936112945\r\n

Cache-Control: no-cache\r\n

Date: Thu, 13 Feb 2003 21:59:22 GMT\r\n

Expires: Thu, 13 Feb 2003 21:59:22 GMT\r\n

Transport:

RTP/AVP;unicast;client_port=6972-6973;mode=record;source=127.0.0.1;

server_port=6978-6979\r\n

\r\n
```

Client to server:

```
RECORD rtsp://127.0.0.1/mystream.sdp RTSP/1.0\r\n

CSeq: 5\r\n
```

```
Session: 1549167172936112945\r\n

User-Agent: QTS (qtver=6.1;cpu=PPC;os=Mac 10.2.3)\r\n

\r\n
```

Server to client:

```
RTSP/1.0 200 OK\r\n

Server: QTSS/4.1.3.x (Build/425; Platform/MacOSX; Release/Development;)\r\n

Cseq: 5\r\n

Session: 1549167172936112945\r\n

RTP-Info: url=trackid=1,url=trackid=2\r\n

\r\n
```

Trace of ANNOUNCE and RECORD Using UDP Transport

The following trace example of ANNOUNCE and RECORD RTSP methods using UDP transport is from RFC 2326. The conference participant client asks the media server to record the audio and video portions of a meeting. The client uses the ANNOUNCE method to provide meta-information about the recorded session to the server.

Client to server:

```
ANNOUNCE rtsp://server.example.com/meeting RTSP/1.0

CSeq: 90

Content-Type: application/sdp

Content-Length: 121


v=0

o=camera1 3080117314 3080118787 IN IP4 195.27.192.36

s=IETF Meeting, Munich - 1

i=The thirty-ninth IETF meeting will be held in Munich, Germany

u=http://www.ietf.org/meetings/Munich.html

e=IETF Channel 1 <ietf39-mbone@uni-koeln.de>

p=IETF Channel 1 +49-172-2312 451

c=IN IP4 224.0.1.11/127

t=3080271600 3080703600

a=tool:sdr v2.4a6
```

```
a=type:test

m=audio 21010 RTP/AVP 5

c=IN IP4 224.0.1.11/127

a=ptime:40

m=video 61010 RTP/AVP 31

c=IN IP4 224.0.1.12/127
```

Server to client:

```
RTSP/1.0 200 OK

CSeq: 90
```

Client to server:

```
SETUP rtsp://server.example.com/meeting/audiotrack RTSP/1.0

CSeq: 91

Transport: RTP

AVP;multicast;destination=224.0.1.11;port=21010–21011;mode=record;ttl=127
```

Server to client:

```
RTSP/1.0 200 OK

CSeq: 91

Session: 50887676

Transport: RTP

AVP;multicast;destination=224.0.1.11;port=21010–21011;mode=record;ttl=127
```

Client to server:

```
SETUP rtsp://server.example.com/meeting/videotrack RTSP/1.0

CSeq: 92

Session: 50887676

Transport: RTP/

AVP;multicast;destination=224.0.1.12;port=61010–61011;mode=record;ttl=127
```

Server to client:

```
RTSP/1.0 200 OK

CSeq: 92

Transport: RTP

AVP;multicast;destination=224.0.1.12;port=61010–61011;mode=record;ttl=127
```

Client to server:

```
RECORD rtsp://server.example.com/meeting RTSP/1.0

CSeq: 93

Session: 50887676

Range: clock=19961110T1925–19961110T2015
```

Server to client:

```
RTSP/1.0 200 OK

CSeq: 93
```

Stream Caching

This version of QTSS includes RTSP and RTP features that make it as easy for a caching proxy server to capture and manage a pristine copy of a media stream. Some of these features are elements of RTSP that were not supported in previous versions of QTSS, and other features are additions to RTSP and RTP. The features are

- *Speed RTSP header.* This version of QTSS supports the speed header wherever possible. The speed header allows a caching proxy server to request that a stream be delivered faster than real time so that the caching proxy server can move the stream into the cache as quickly as possible. This header is described in the section "Speed RTSP Header" (page 115).

- *x-Transport-Options RTSP header.* This version of QTSS supports the non-standard RTSP header, `x-Transport-Options`. Caching proxy servers can use this header to tell the streaming server how late packets the streaming server can send packets and have them still be useful to the caching proxy server. This header is described in the section "x-Transport-Options Header" (page 116).

- *RTP payload meta-information.* This version of QTSS fully supports RTP payload meta-information (an IETF draft), which includes information such as the packet transmission time, unique packet number, and video frame type. Caching proxy servers can use this information to provide the same quality of service to clients as the originating server. This header is described in the section "RTP Payload Meta-Information" (page 116).

- *x-Packet-Range RTSP header.* This version of QTSS supports the non-standard RTSP header, x-Packet-Range. This header is similar to the Range RTSP header but allows the client to specify a specific range of packets instead of a range of time. A caching proxy server can use the x-Packet-Range header to tell the originating server to selectively retransmit only those packets that the caching proxy server needs in order to fill in holes in its cached copy of the stream. This header is described in the section "x-Packet-Range RTSP Header" (page 123).

The following sections describe each of these features.

Speed RTSP Header

Clients can send to the server the optional Speed RTSP header to request that the server send data to the client at a particular speed. The server must respond by echoing the Speed RTSP header to the client. If the server does not echo the Speed RTSP header, the client must assume that the server cannot accommodate the request at this time. The server may modify the value of the Speed RTSP header argument. If the server modifies the value of the argument, the client must accept the modified value.

The value of the Speed RTSP header argument is expressed as a decimal ratio. The following example asks the server to send data twice as fast as normal:

```
Speed: 2.0
```

> **Note:** An argument of zero is invalid.

If the request also contains a Range argument, the new speed value will take effect at the specified time.

This header is intended for use when preview of the presentation at a higher or lower rate is necessary. Bandwidth for the session may have been negotiated earlier (by means other than RTSP), and therefore re-negotiation may be necessary.

When data is delivered over UDP, it is highly recommended that means such as RTCP be used to track packet loss rates.

x-Transport-Options Header

The optional `x-Transport-Options` RTSP header should be sent from a client (typically a caching proxy server) to the server in an RTSP SETUP request and must echoed by the server. If the server does not echo the `x-Transport-Options` header, the client must assume that the server does not support this header. The server may modify the value of the `x-Transport-Options` header argument. If the server modifies the value of the argument, the client must accept the modified value.

The body of this header contains one or more arguments delimited by the semicolon character. For this version of QTSS, there is only one argument, the late-tolerance argument.

The value of the `late-tolerance` argument is a positive integer that represents the number of seconds late that the server can send a media packet and still have it be useful to the client. The server should use the value of the `late-tolerance` argument as a guide for making a best-effort attempt to deliver all media data so that the delivered data is no older than the late-tolerance value.

Here is an example:

```
x-Transport-Options: late-tolerance=30
```

If this example were for a video stream, the server would send all video frames that are less than 30 seconds old. The server would drop frames that are more than 30 seconds old because they are stale.

Caching proxy servers can use the `x-Transport-Options` header to prevent the media server from dropping frames or lowering the stream bit rate in the event it falls behind in sending media data. If the caching proxy server knows the duration of the media, it can prevent the server from dropping any frames by setting the `late-tolerance` argument to the duration of the media, allowing the cache to receive a complete copy of the media data.

For a live broadcast, a caching proxy server may want to do extra buffering to improve quality for its clients. It could use the `x-Transport-Options` header to advertise the length of its buffer to the server.

RTP Payload Meta-Information

Certain RTP clients, such as caching proxy servers, require per-packet meta information that goes beyond the sequence number and timestamp already provided in the RTP header. For instance, a caching proxy server may want to provide stream thinning to its clients in case those clients are bandwidth constrained. If that stream thinning is based on the type of video frame being sent by the originating server, there is no payload-independent way for the caching proxy server to determine the frame type.

116

The RTP payload meta-information solves this deficiency by including information that RTP clients can use to provide the same quality of service to clients as the originating server. The following section, "RTP Data" (page 117), describes the RTP data that the server delivers in the RTP payload meta-information type.

RTP Data

The server uses the RTP payload meta-information type to provide the following information to the RTP client:

- Transmission time, described in the section "Transmission Time" (page 117)

- Frame type, described in the section "Frame Type" (page 117)

- Packet number, described in the section "Packet Number" (page 118)

- Packet position, described in the section "Packet Position" (page 118)

- Media data, described in the section "Media Data" (page 118)

- Sequence number, described in the section "Sequence Number" (page 118)

Transmission Time

The server sends the transmission time as a single four-octet unsigned integer representing the recommended transmission time of the RTP packet in milliseconds.

The transmission time is always offset from the start of the media presentation. For example, if the SDP response for a URL includes a range of 0-729.45 and the client makes a PLAY request with a range of 100-729.45, the first RTP packet from the server should provide a transmission time value of approximately 100,000. (It may not be exactly 100,000 because the server is free to find a frame nearby the requested time.) If the SDP for a URL does not contain a range, the client can at least use these values as relative offsets.

Frame Type

The server sends the frame type as a single 16-bit unsigned integer value for which several well-known values representing different frame types are defined. The well-known values are as follows:

- 0 represents an unknown frame type

- 1 represents a key frame

- 2 represents a b-frame

- 3 represents a p-frame

> **Note:** The frame type is valid for video RTP streams only.

Packet Number

The server sends the packet number as a single 64-bit unsigned integer value. The value is the packet number offset from the absolute start of the stream. For example, if the SDP response for a URL includes a range of 0-729.45 and the client makes a PLAY request with a range of 0-729.45, the packet number value of the first packet will be 0 and will increment by 1 for each subsequent packet. If there are 1000 packets between in the first 60 seconds of a stream and a client makes a PLAY request of 60-729.45, the packet number of the first packet will be 1001 and will increment by 1 for each subsequent packet.

Packet Position

The server sends the packet position as a single 64-bit unsigned integer value. The value is the byte offset of this packet from the absolute start of the stream. For example, if the SDP response for a URL includes a range of 0-729.45 and the client makes a PLAY request with a range of 100-729.45, the packet position value of the first video RTP packet will be the total number of bytes of the video RTP packets between 0 and 100. Only the RTP packet payload bytes are used to compute each packet position value.

The server cannot provide the packet position for live or dynamic media. In general, if the media SDP has a range attribute, the server can provide the packet position.

Media Data

The server sends media data for the underlying RTP protocol.

Sequence Number

The server sends the RTP sequence number as a two-octet value. The sequence number is useful for mapping RTP meta-information to the underlying payload data that they refer to, if that data is being sent out-of-band.

Standard Format

The RTP payload meta-information returned by the server consists of a series of fields. Each field consists of a header and data. When returned in standard format, the first bit of the header is zero to indicate that the field is in standard format (that is, not compressed).

The first bit is followed by the 15-bit Name subfield. The Name subfield contains two ASCII alphanumeric characters that represent one of the RTP data types listed in the section "RTP Data" (page 117). The first character is seven bits long, so the value of the Name subfield must consist of seven-bit ASCII characters.

Table 2-26 (page 119) lists the Name subfield values for each of the RTP data types.

Table 1-26   Defined Name subfield values

RTP data type	Name subfield value
Transmission time	`tt`
Frame type	`ft`
Packet number	`pn`
Packet position	`pp`
Media	`md`
Sequence number	`sn`

The Name subfield is followed by a two-octet Length subfield that contains the full length of the Data subfield.

`Figure 2–9` (page 119) shows the format of the Name subfield in standard format.

Figure 1-9   Standard RTP payload meta-information format
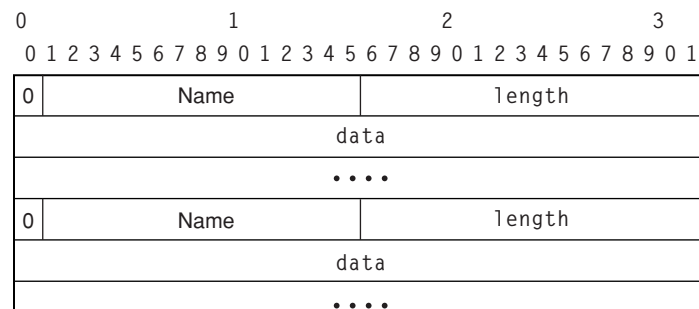
`Figure 2-10` (page 120) shows the format of the RTP data in standard format.

Figure 1-10   RTP data in standard format

	0		1		2		3
	0 1 2 3 4 5 6 7	8 9 0 1 2 3 4 5	6 7 8 9 0 1 2 3	4 5 6 7 8 9 0 1			

Transmit time data
```
0        tt                    4
      (transmit time)
```

Frame type data
```
0        ft                    2
      (frame type)
```

Packet number data
```
0        pn                    8
   (high order packet number)
   (low order packet number)
```

Packet position data
```
0        pp                    8
   (high order packet number)
   (low order packet number)
```

Media data
```
0        md            (field length)
            . . . .
```

Sequence number
```
0        sq                    2
   (sequence number)
```
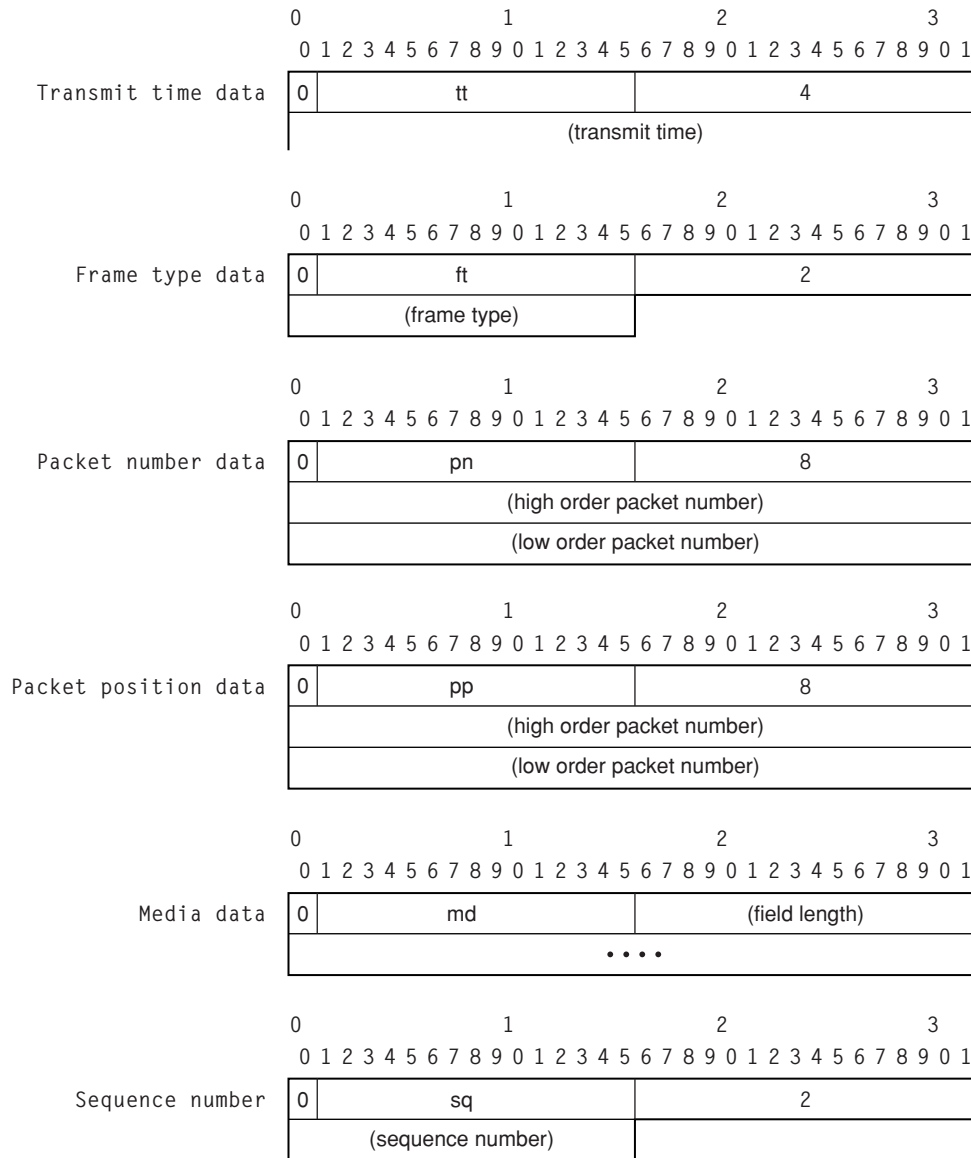
Compressed Format

When the server provides a field of RTP meta-information in compressed format, the field consists of a header and data. The first bit of the header is set to one to indicate that the rest of the header is in compressed format.

The first bit is followed by a seven-bit ID subfield that identifies the type of data in the Data subfield. The meaning of the ID subfield is assigned by the server, as described in the section "`x-RTP-Meta-Info RTSP Header Negotiation`" (page 122).

The ID subfield is followed by the one-octet Length subfield that contains the full length of the Data subfield that follows the Length subfield.

Figure 2–11 (page 121) shows the format of the ID subfield in compressed format.

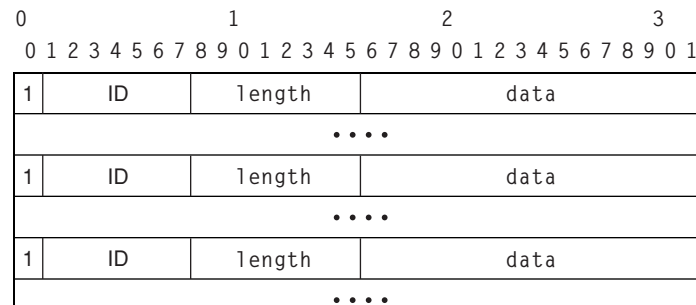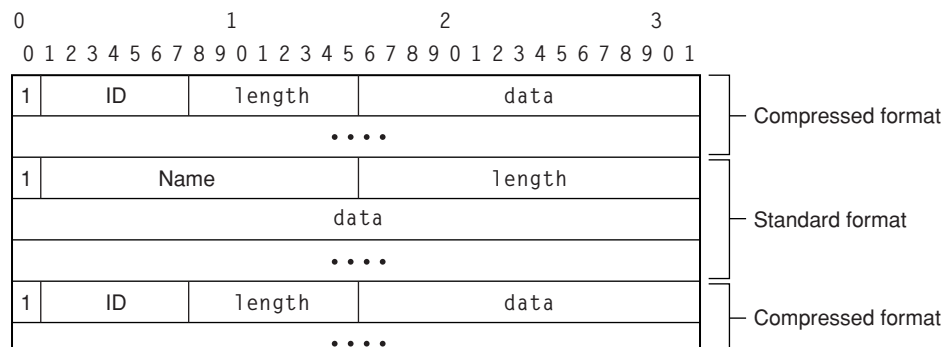**Figure 1-11** Compressed RTP payload meta-information format

Figure 2–12 (page 121) shows an RTP payload meta-information packet when some fields are in compressed format and some fields are in standard format.

Figure 1-12 Mixed RTP payload meta-information format

Negotiation for Use of Compressed Format

Use of the compressed format requires out-of-band negotiation between client and server. During the negotiation process, the server assigns a seven-bit ID for each RTP data type. Instead sending the name of the RTP data type (for example, `ft`) in the RTP payload, only the ID is sent.

Negotiation for using the compressed format can occur in two ways:

- Through the x-RTP-Meta-Info RTSP header, described in the section "x–RTP–Meta–Info RTSP Header Negotiation" (page 122)

- Through the SDP description of the data, described in the section "Describing RTP–Meta–Info Payload in SDP" (page 122)

x-RTP-Meta-Info RTSP Header Negotiation

The client can negotiate compression with the server for any payload by sending an x-RTP-Meta-Info RTSP header to the server in a SETUP request. If the server does not echo the header in its SETUP response, the client must assume that the server does not support this header.

The client's SETUP request specifies the RTP data types the client wants to receive in the specified RTP stream. Here is an example of a client request:

```
x-RTP-Meta-Info: to;bi;bo
```

The server's response lists the names of the RTP data that the server will provide for that RTP stream. If the server supports the compressed format, the response may also contain ID mappings for some or all of the names. The server may return a subset of the names if it doesn't support all of the requested names, or if some requested names don't apply to the RTP stream specified by the SETUP request. Here are two examples of a server response:

```
x-RTP-Meta-Info: to=0;bi;bo=1
x-RTP-Meta-Info: to;bi
```

In the first response, the server indicates that it will provide bi data in standard format. The server will send to data in compressed format and use an ID of 0 to indicate fields that contain to data. The server will send bo data in compressed format and use an ID of 1 to indicate fields that contain bo data. Because IDs are represented by seven bits, an ID must be between 0 and 127.

In the second response, the server indicates that it will provide to and bi data in standard format.

Describing RTP-Meta-Info Payload in SDP

The originator of RTP-Meta-Info payload packets should describe the contents of the payload as part of the SDP description of the media. RTP-Meta-Info descriptions consist of two additional a= headers.

The `a=x-embedded-rtpmap` header tells the client the payload type of the underlying RTP payload.

The `a=x-RTP-Meta-Info` header tells the client the RTP data types the server will provide. Here is an example of an SDP description of the RTP-Meta-Info payload:

```
m=other 5084 RTP/AVP 96
a=rtpmap:96 x-RTP-Meta-Info
a=x-embedded-rtpmap:96 x-QT}
a=x-RTP-Meta-Info: standard;to;bi;bo
```

x-Packet-Range RTSP Header

The x-Packet-Range RTSP header allows the client (typically a caching proxy server) to specify a range of packets that the server should retransmit, thereby allowing the client to fill in holes in its cached copy of the stream. The client should send the x-Packet-Range RTSP header in a PLAY request in place of the Range header. If the server does not support this header, it sends the client a "501 Header Not Implemented" response.

The body of this header contains a start and stop packet number for this PLAY request. The specified packet numbers must be based on the packet number RTP-Meta-Info field. For information on how to request packet numbers as part of the RTP stream, see the RTP-Meta-Info payload format IETF Draft.

The header format consists of two arguments delimited by the semicolon character. The first argument must be the packet number range, with the start and stop packet numbers separated by a hyphen (-). The second argument must be the stream URL to which the specified packets belong.

The following example requests packet numbers 4551 through 4689 for trackID3:

```
x-Packet-Range: pn=4551-4689;url=trackID3
```

The stop packet number must be equal to or greater than the start packet number. Otherwise, the server may return an error or may not send any media data after the PLAY response.

Reliable UDP

Reliable UDP is a set of quality of service enhancements, such as congestion control tuning improvements, retransmit, and thinning server algorithms, that improve the ability to present a good quality RTP stream to RTP clients even in the presence of packet loss and network congestion. Reliable UDP's congestion control mechanisms allow streams to behave in a TCP-friendly fashion without disturbing the real-time nature of the protocol.

To work well with TCP traffic on the Internet, Reliable UDP uses retransmission and congestion control algorithms similar to the algorithms used by TCP. Additionally, these algorithms are time-tested to utilize available bandwidth optimally.

Relibable UDP features include

- Client acknowledgment of packets sent by the server to the client
- Windowing and congestion control so the server does not exceed the currently available bandwidth
- Server retransmission to the client in the event of packet loss
- Faster than real-time streaming known as "overbuffering"

Whether a client uses Reliable UDP is determined by the content of the client's RTSP SETUP request.
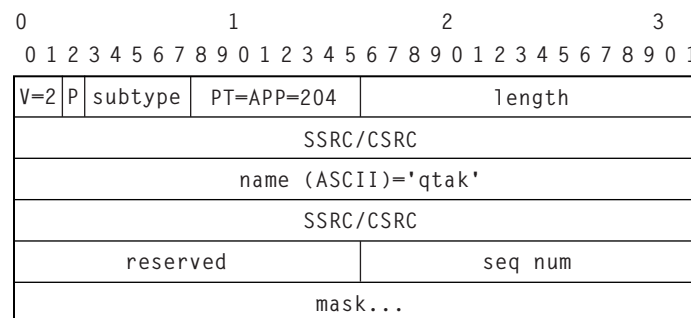
Acknowledgment Packets

When using Reliable UDP, the server expects to receive an acknowledgment for each RTP packet it sends. If the server does not receive an acknowledgment for a packet, it may retransmit the packet. The client does not need to send an acknowledgment packet for each RTP packet it receives. Instead, the client can coalesce acknowledgments for several packets and send them to the server in a single packet.

The Reliable UDP acknowledgment packet format is a type of RTCP APP packet. After the standard RTCP APP packet headers, the payload for an acknowledgment packet consists of an RTP sequence number followed by a variable length bit mask. The sequence number identifies the first RTP packet that the client is acknowledging. Each additional RTP packet being acknowledged is represented by a bit set in the bitmask. The bit mask is an offset from the specified sequence number, where the high order bit of the first byte in the mask is one greater than the sequence number, the second bit is two greater, and so on. Bit masks must be sent in multiples of four octets. Setting a bit to 0 in the mask simply means that the client does not wish to acknowledge this sequence number right now and does not imply a negative acknowledgment.

Figure 2–13 (page 124) shows the format of the Reliable UDP acknowledgment packet.

Figure 1-13 Reliable UDP acknowledgment packet format

```
0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-------+-+---------+-----------------+-------------------------+
|V=2|P| subtype |   PT=APP=204    |          length         |
+-----------------------------------------------------------+
|                       SSRC/CSRC                           |
+-----------------------------------------------------------+
|                   name (ASCII)='qtak'                     |
+-----------------------------------------------------------+
|                       SSRC/CSRC                           |
+---------------------------+-------------------------------+
|         reserved          |           seq num             |
+---------------------------+-------------------------------+
|                        mask...                            |
+-----------------------------------------------------------+
```

RTSP Negotiation

Whether to use Reliable UDP is negotiated out of band in RTSP. If a client wants to use Reliable UDP, it should include an x-Retransmit header in its RTSP SETUP request. The body of the header contains the retransmit protocol name (`our-retransmit`) followed by a list of arguments delimited by the semicolon character.

Currently, one argument can be passed from the client to the server: the window argument. If included, the window argument tells the Reliable UDP server the size of the client's window in KBytes.

Here is an example:

```
x-Retransmit: our-retransmit;window=128
```

The server must echo the header and all parameters. If the x-Retransmit header is not in the SETUP response, the client must assume that Reliable UDP will not be used for this stream. If the server changes the parameter values, the client must use the new values.

Tunneling RTSP and RTP Over HTTP

Using standard RTSP/RTP, a single TCP connection can be used to stream a QuickTime presentation to a user. Such a connection is not sufficient to reach users on private IP networks behind firewalls where HTTP proxy servers provide clients with indirect access to the Internet. To reach such clients, QuickTime 4.1 supports the placement of RTSP and RTP data in HTTP requests and replies. As a result, viewers behind firewalls can access QuickTime presentations through HTTP proxy servers.

The QuickTime HTTP transport is built from two separate HTTP GET and POST method requests initiated by the client. The server then binds the connections to form a virtual full-duplex connection. The protocol that forms this type of connection is must meet the following requirements:

- Work with unmodified RTSP/RTP packets

- Be acceptable to HTTP proxy servers

- Indicate to proxy servers that requests and replies are not to be cached

- Work in an environment where the client originates all requests

- Provide a way to uniquely identify request pairs so that they can be bound together to form a full-duplex connection

- Ensure that related requests connect to the same RTSP server in spite of load-balancing algorithms such as round-robin DNS servers

- Identify any request as one that will eventually tunnel an RTSP conversation and RTP data

The QuickTime HTTP transport exploits the capability of HTTP's GET and POST methods to carry an indefinite amount of data in their reply and message body respectively. In the most simple case, the client makes an HTTP GET request to the server to open the server-to-client connection. Then the client makes a HTTP POST request to the server to open the client-to-server connection. The resulting virtual full-duplex connection (shown in `Figure 2-14` (page 126)) makes it possible to send unmodified RTSP and RTP data over the connection.

Figure 1-14   Required connections for tunneling

HTTP Client Request Requirements
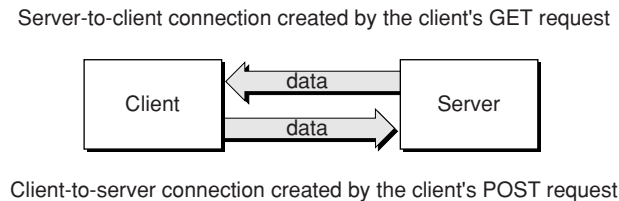
HTTP Client Request Requirements

To work with the QuickTime HTTP transport, client HTTP requests must

- Be made using HTTP version 1.0

- Include in the header an `x-sessioncookie` directive whose value is a globally unique identifier (GUID). The GUID makes it possible for the server to unambiguously bind the two connections by passing it as an opaque token to the C library `strcmp` function

- In POST requests, the `application/x-rtsp-tunneled` MIME type for both the `Content-Type` and `Accept` directives must be specified; this MIME type reflects the data type that is expected and delivered by the client and server

- Direct POST requests to the specified IP address if a server's reply to an initial GET request includes the `x-server-ip-address` directive and an IP address

In addition to these requirements, client HTTP POST request headers may include other directives in order to help HTTP proxy servers handle RTSP streams optimally.

Sample Client GET Request

Here is an example of a client GET request:

```
GET /sw.mov HTTP/1.0
User-Agent: QTS (qtver=4.1;cpu=PPC;os=Mac8.6)
x-sessioncookie: tD9hKgAAfB8ABCftAAAAAw
```

Sample Client POST Request

Here is an example of a client POST request:

```
POST /sw.mov HTTP/1.0

User-Agent: QTS (qtver=4.1;cpu=PPC;os=Mac8.6)

Content-Type: application/x-rtsp-tunnelled

Pragma: no-cache

Cache-Control: no-cache

Content-Length: 32767

Expires: Sun, 9 Jan 1972 00:00:00 GMT
```

Note: The server does not respond to client POST requests. The client will continue to send RTSP data as the message body of this POST request.

The sample client POST request includes three optional header directives that are present to control the behavior of HTTP proxy servers so that they handle RTSP streams optimally:

- The `Pragma: no-cache` directive tells many HTTP 1.0 proxy servers not to cache the transaction.

- The `Cache-Control: no-cache` directive tells many HTTP 1.1 proxy servers not to cache the transaction.

- The `Expires` directive specifies an arbitrary time in the past. This directive is intended to prevent proxy servers from caching the transaction.

HTTP requires that all POST requests have a content-length header. In the sample client POST request, the content length of 32767 is an arbitrary value. In practice, the actual value seems to be ignored by proxy servers, so it is possible to send more than this amount of data in the form of RTSP requests. The QuickTime Server ignores the content-length header.

HTTP Server Reply Requirements

When the server receives an HTTP GET request from a client, it must respond with a reply whose header specifies the `application/x-rtsp-tunneled` MIME type for both the `Content-Type` and `Accept` directives.

> **Note:** The server must reply to all client HTTP GET requests but never replies to client HTTP POST requests.

Server reply headers may optionally include the `Cache-Control: no-store` and `Pragma: no-cache` directives to prevent HTTP proxy servers from caching the transaction. It is recommended that implementations honor these headers if they are present.

Server clusters are often allocated connections by a round-robin DNS or other load-balancing algorithm. To insure that client requests are directed to the same server among potentially several servers in a server farm, the server may optionally include the `x-server-ip-address` directive followed by an IP address in dotted decimal format in the header of its reply to a client's initial GET request. When this directive is present, the client must direct its POST request to the specified IP address regardless of the IP address returned by a DNS lookup.

In the absence of an HTTP error, the server reply header contains "200 OK". An HTTP error in a server reply reflects the inability of the server to form the virtual full-duplex connection; an HTTP error does not imply an RTSP error. When an HTTP error occurs, the server simply closes the connection.

Sample Server Reply to a GET Request

Here is an example of a server reply to a GET request:

```
HTTP/1.0 200 OK
Server: QTSS/2.0 [v101] MacOSX
Connection: close
Date: Thu, 19 Aug 1982 18:30:00 GMT
Cache-Control: no-store
Pragma: no-cache
Content-Type: application/x-rtsp-tunnelled
```

Including the following header directives in a reply is not required but is recommended because the directives they tell proxy servers to behave in a way that allows them to handle RTSP streams optimally:

- The `Date` directive specifies an arbitrary time in the past. This keeps proxy servers from caching the transaction.

- The `Cache-Control: no-cache` directive tells many HTTP 1.1 proxy servers not to cache the transaction.

- The `Pragma: no-cache` directive tells many HTTP 1.0 proxy servers not to cache the transaction.

RTSP Request Encoding

RTSP requests made by the client on the POST connection must be encoded using the base64 method. (See RFC 2045 "Internet Message Bodies", section 6.8, Base64 Content-Transfer-Encoding, and RFC 1421 "Privacy Enhancements for Electronic Mail," section 4.3.2.4, Printable Encoding.) The base64 encoding prevents HTTP proxy server from determining that an embodied RTSP request is a malformed HTTP requests.

Here is a sample RTSP request before it is encoded:

```
DESCRIBE rtsp://tuckru.apple.com/sw.movRTSP/1.0
CSeq: 1
Accept: application/sdp
Bandwidth: 1500000
Accept-Language: en-US
User-Agent: QTS (qtver=4.1;cpu=PPC;os=Mac8.6)
```

Here is the same request after encoding:

```
REVTQ1JJQkUgcnRzcDovL3R1Y2tydS5hcHBsZS5jb20vc3cubW92IFJUU1AvMS4w
DQpDU2VxOiAxDQpBY2N1cHQ6IGFwcGxpY2F0aW9uL3NkcA0KQmFuZHdpZHRo0iAx
NTAwMDAwDQpBY2N1cHQtTGFuZ3VhZ2U6IGVuLVVTDQpVc2VyLUFnZW50OiBRVFMg
KHF0dmVyPTQuMTtjcHU9UFBDO29zPU1hYyA4LjYpDQoNCg==
```

Connection Maintenance

The client may close the POST connection at any time. Doing so frees socket and memory resources at the server that might otherwise be unused for a long time. In QuickTime HTTP streaming, the best time to close the POST connection usually occurs after the PLAY request.

Support For Other HTTP Features

Support for HTTP features that are not documented here is not required in order to implement the tunneling of QuickTime RTSP and RTP over HTTP. The tunnel should mimic a normal TCP connection as closely as possible without adding unnecessary features.

Tasks

This chapter describes common QTSS tasks:

- Building the Streaming Server, described in "Building the Streaming Server" (page 130).

- Compiling and installed a QTSS module, described in "Compiling a QTSS Module into the Server" (page 131).

- Getting and setting attribute values, described in "Working with Attributes" (page 134). This section also tells you how to add your own attributes to an object.

- Using the server's file module to open, read, and close files, described in "Using Files" (page 139). This section also tells you how to implement your own file system module.

- Communicating with the server with the Admin protocol, described in "Using the Admin Protocol" (page 151).

Building the Streaming Server

This section describes the Streaming Server build and install process for Mac OS X, POSIX, and Windows platforms.

Mac OS X

Use the `Buildit` script to build the Streaming Server for Mac OS X. Use the following command line options: `StreamingServer.pbroj -target DSS`. As they are built, the binaries are left in the build directory.

The command `BuildOSXInstallerPkg dss` creates a file named `DarwinStreamingServer.pkg`.

POSIX

Use the `Buildit` script to build the Streaming Server on POSIX platforms. Binaries are left in the source directories. To create the installer, use the `buildtarball` script, which creates an install directory with Install script and tar file.

Windows

Use the `WindowsNTSupport/StreamingServer.dsw` script to build the Streaming Server on Windows platforms. Batch build all. Binaries are left in the Debug and Release directory. The `WindowsNTSupport/makezip.bat` script creates an install directory with an Install.bat file.

Building a QuickTime Streaming Server Module

You can add a QTSS module to the QuickTime Streaming Server by compiling the code directly into the server itself or by building a module as a separate code fragment that is loaded when the server starts up.

Whether compiled into the server or built as a separate module, the code for the module is the same. The only difference is the way in which the code is compiled.

Compiling a QTSS Module into the Server

If you have the source code for the QuickTime Streaming Server, you can compile your module into the server.

> **Note:** The source code for the server is available at http://www.publicsource.apple.com/projects/streaming.

To compile your code into the server, locate the function `QTSServer::LoadCompiledInModules` in `QTSServer.cpp` and add to it the following lines

```
QTSSModule* myModule = new QTSSModule("__XYZ__");
(void)myModule->Initialize(&sCallbacks, &__XYZMAIN__);
(void)AddModule(myModule);
```

where `XYZ` is the name of your module and `XYZMAIN` is your module's main routine.

Some platforms require that each module use unique function names. To prevent name conflicts when you compile a module into the server, make your functions static.

Modules that are compiled into the server are known as static modules.

Building a QTSS Module as a Code Fragment

To have the server load at runtime a QTSS module that is a code fragment, follow these steps:

1. Compile the source for your module as a dynamic shared library for the platform you are targeting. For Mac OS X, the project type must be `loadable bundle`.

2. Link the resulting file against the QTSS API stub library for the platforms you are targeting.

3. Place the resulting file in the `/Library/QuickTimeStreaming/Modules` directory (Mac OS X), `/usr/local/sbin/StreamingServerModules (Darwin platforms)`, and `c:\Program Files\Darwin StreamingServer\QTSSModules`. The server will load your module the next time it restarts.

Some platforms require that each module use unique function names. To prevent name conflicts when the server loads your module, strip the symbols from your module before you have the server load it.

Debugging

Several server preferences in the streamingserver.xml file are available for enabling the generation of debugging information, which is printed on the terminal screen. The following sections provide information on debugging:

RTSP and RTP Debugging

To enable the display of RTSP and RTP informati on the terminal screen, modify the RTSP_debug_printfs preference in the streamingserver.xml file and restart the server:

```
<PREF NAME="RTSP_debug_printfs" TYPE="BOOL16" >true</PREF>
```

To enable the display of packet header information, modify the "enable_packet_header_printfs" preference in the streamingserver.xml file:

```
<PREF NAME="enable_packet_header_printfs" TYPE="BOOL16" >true</PREF>
```

Then specify which packet headers to display by modifying the "packet_header_printf_options" preference. The following example enables the display of all packet headers:

```
<PREF NAME="packet_header_printf_options" >rtp;rr;sr;app;ack;</PREF>
```

In the previous example, `rtp` enables the display of RTP packet headers, `rr` enables the display of RTCP receiver reports, `sr` enables the display of RTCP sender reports, `app` enables the display of RTCP application packets, and `ack` enables the display of Reliable UDP RTP acknowledgement packets.

After enabling RTSP and RTP debugging, restart the Streaming Server in debug mode using this command:

```
QuickTimeStreamingServer —d
```

When you connect a client, debug information is displayed on the terminal screen.

Source File Debugging Support

You can enable debugging in specific source files. For example, in the file CommonUtilitiesLib/Task.h, make the following change:

```
#define TASK_DEBUG 1
```

Rebuild and start the Streaming Server in debug mode:

```
QuickTimeStreamingServer —d
```

Here is some sample output:

```
Task::Signal enque task TaskName=RTSPSession ...

TaskThread::Entry run task TaskName=RTSPSession ...

TaskThread::Entry insert task TaskName=RTSPSession ...

TaskThread::Entry run task TaskName=RTSPSession ...

TaskThread::WaitForTask found timer task TaskName=QTSSAccessLog ...

TaskThread::Entry run task TaskName=QTSSAccessLog ...
```

You can also enable debugging in CommonUtilitiesLib/OSFileSource.cpp:

```
#define FILE_SOURCE_DEBUG 1
```

Here is some sample output:

```
OSFileSource::SetLog=/Library/QuickTimeStreaming/Movies/sample_100kbit.mov

FileMap::AllocateBufferMap shared buffers

OSFileSource::ReadFromCache inPosition =272 ...

OSFileSource::ReadFromCache inPosition =276 ...

OSFileSource::ReadFromCache inPosition =280 ...
```

133

```
...
OSFileSource::ReadFromCache inPosition =80667
```

Working with Attributes

QTSS objects consist of attributes that are used to store data. Every attribute has a name, an attribute ID, a data type, and permissions for reading and writing the attribute's value. There are two attribute types:

- static attributes. Static attributes are present in all instances of an object type. A module can add static attributes to objects from its Register role only. All of the server's built-in attributes are static attributes. For information about adding static attributes to object types, see the section "Adding Attributes" (page 138)

- instance attributes. Instance attributes are added to a specific instance of any object type. A module can use any role to add an instance attribute to an object and can also remove instance attributes that it has added to an object. For information about adding instance attributes to objects, see the section "Adding Attributes" (page 138).

Note: Adding static attributes is more efficient than adding instance attributes, so adding static attributes instead of adding instance attributes is strongly recommended.

Getting Attribute Values

Modules use attributes stored in objects to exchange information with the server, so they frequently get attribute values. Three callback routines get attribute values:

- `QTSS_GetValue`, which copies the attribute value into a buffer provided by the module. This callback can be used to get the value of any attribute, but it is not as efficient as `QTSS_GetValuePtr`.

- `QTSS_GetValueAsString`, which copies the attribute value as a string into a buffer provided by the module. This callback can be used to get the value of any attribute. This is the least efficient way to get the value of an attribute

- `QTSS_GetValuePtr`, which returns a pointer to the server's internal copy of the attribute value. This is the most efficient way to get the value of preemptive safe attributes. It can also be used to get the value of non-preemptive safe attributes, but the object must first be locked and must be unlocked after QTSS_GetValuePtr is called. When getting the value of a single non-preemptive-safe attribute, calling QTSS_GetValue may be more efficient than locking the object, calling QTSS_GetValuePtr and unlocking the object.

The sample code in Listing 2-1 (page 135) calls `QTSS_GetValue` to get the value of the `qtssRTPSvrCurConn` attribute, which is not preemptive safe, from the `QTSS_ServerObject` object.

Listing 2-1   Getting the value of an attribute by calling QTSS_GetValue

```
UInt32 MyGetNumCurrentConnections(QTSS_ServerObject inServerObject)
{
    // qtssRTPSvrCurConn is a UInt32, so provide a UInt32 for the result.
    UInt32 theNumConnections = 0;
    // Pass in the size of the attribute value.
    UInt32 theLength = sizeof(theNumConnections);
    // Retrieve the value.
    QTSS_Error theErr = QTSS_GetValue(inServerObject, qtssRTPSvrCurConn, 0,
        &theNumConnections, &theLength);
    // Check for errors. If the length is not what was expected, return 0.
    if ((theErr != QTSS_NoErr) || (theLength != sizeof(theNumConnections))
        return 0;
    return theNumConnections;
}
```

The sample code in Listing 2-2 (page 135) calls `QTSS_GetValuePtr`, which is the preferred way to get the value of preemptive-safe attributes. In this example, value of the `qtssRTSPReqMethod` attribute is obtained from the object `QTSS_RTSPRequestObject`.

Listing 2-2   Getting the value of an attribute by calling QTSS_GetValuePtr

```
QTSS_RTSPMethod MyGetRTSPRequestMethod(QTSS_RTSPRequestObject inRTSPRequestObject)
{
    QTSS_RTSPMethod* theMethod = NULL;
    UInt32 theLen = 0;

QTSS_Error theErr = QTSS_GetValuePtr(inRTSPRequestObject, qtssRTSPReqMethod, 0,
    (void**)&theMethod, &theLen);
if ((theErr != QTSS_NoErr) || (theLen != sizeof(QTSS_RTSPMethod))
    return -1;  // Return a -1 if there is an error, which is not a valid
                // QTSS_RTSPMethod index
else
```

```
      return *theMethod;

}
```

You can obtain the value any attribute by calling `QTSS_GetValueAsString`, which gets the attribute's value as a C string. Calling `QTSS_GetValueAsString` is convenient when you don't know the type of data the attribute contains. In Listing 2-3 (page 136), the value of the `qtssRTPSvrCurConn` attribute is obtained as a string from the `QTSS_ServerObject`.

Listing 2-3   Getting the value of an attribute by calling QTSS_GetValueAsString

```
void MyPrintNumCurrentConnections(QTSS_ServerObject inServerObject)
{
    // Provide a string pointer for the result
    char* theCurConnString = NULL;
    // Retrieve the value as a string.
    QTSS_Error theErr = QTSS_GetValueAsString(inServerObject, qtssRTPSvrCurConn,
0,   &theCurConnString);
    if (theErr != QTSS_NoErr) return;
    // Print out the result. Because the value was returned as a string, use
    // %s in the printf format.
    ::printf("Number of currently connected clients: %s\n", theCurConnString);
    // QTSS_GetValueAsString allocates memory, so reclaim the memory by calling
QTSS_Delete.
    QTSS_Delete(theCurConnString);
}
```

Setting Attribute Values

Two QTSS callback routines are available for setting the value of an attribute: `QTSS_SetValue` and `QTSS_SetValuePtr`.

The sample code in Listing 2-4 (page 136) would be found handling the Route role. It calls `QTSS_GetValuePtr` to get the value of the `qtssRTSPReqFilePath`. If the path matches a certain string, the function sets a new request root directory by calling `QTSS_SetValue` to set the `qtssRTSPReqRootDir` attribute to a new path.

Listing 2-4   Setting the value of an attribute by calling QTSS_SetValue

```
// First get the file path for this request using QTSS_GetValuePtr
```

```
char* theFilePath = NULL;

UInt32 theFilePathLen = 0;

QTSS_Error theErr = QTSS_GetValuePtr(inParams->inRTSPRequest, qtssRTSPReqFilePath,
   0, &theFilePath,
                &theFilePathLen);


// Check for any errors

if (theErr != QTSS_NoErr) return;

// See if this path is a match. If it is, use QTSS_SetValue to set the root directory
   for this request.

if ((theFilePathLen == sStaticFilePathLen) &&
                 (::strncmp(theFilePath, sStaticFilePath, theFilePathLen) ==
0))
{
    theErr = QTS_SetValue(inParams->inRTSPRequest, qtssRTSPReqRootDir, 0,
sNewRootDirString,
          sNewRootDirStringLen);

    if (theErr != QTSS_NoErr) return;

}
```

Listing 2-5 (page 137) demonstrates the use of the `QTSS_SetValuePtr` callback. The `QTSS_SetValuePtr` callback associates an attribute with the value of a module's variable. This code sample modifies the `QTSS_ServerObject` object nonatomically, so it calls `QTSS_LockObject` to prevent other threads from accessing the attributes of the `QTSS_ServerObject` before the value has been set.

Then the code sample calls `QTSS_CreateObjectValue` to create a `QTSS_ConnectedUserObject` object as the value of the `qtssSvrConnectedUsers` attribute of the `QTSS_ServerObject` object. Then the code sample calls `QTSS_SetValuePtr` to set the value of the `qtssConnectionBytesSent` attribute of the `QTSS_ConnectedUserObject` object to the module's `fBytesSent` variable. Thereafter, when any module gets the value of the `qtssConnectionBytesSent` attribute, it will get the current value of the module's `fBytesSent` variable.

After calling `QTSS_SetValuePtr`, the code sample calls `QTSS_UnlockObject` to unlock the `QTSS_ServerObject` object.

Listing 2-5   Setting the value of an attribute by calling QTSS_SetValuePtr

```
UInt32 index;

QTSS_LockObject(sServer);
```

```
QTSS_CreateObjectValue(sServer, qtssSvrConnectedUsers,  qtssConnectedUserTypeObject,
 &index, &fQTSSObject);


QTSS_CreateObjectValue(sServer, qtssSvrConnectedUsers,  qtssConnectedUserObjectType,
 &index, &fQTSSObject);


QTSS_SetValuePtr(fQTSSObject, qtssConnectionBytesSent, &fBytesSent,
sizeof(fBytesSent));


QTSS_UnlockObject(sServer);
```

Adding Attributes

Any module can add an attribute to a QTSS object type by calling the `QTSS_AddStaticAttribute` callback
routine from its Register role. Modules can also call `QTSS_AddInstanceAttribute` from any role to add an
attribute to an instance of an object.

> **Note:** Adding one or more attributes to an object type or to an instance of an object is the most
> efficient and the recommended way for modules to store data that is specific to a particular session.

Once added, the new attribute is included in every object of that type that the server creates and its value can
be set and obtained by calling that same callback routines that set and obtain the value of the server's built-in
attributes: `QTSS_SetValue`, `QTSS_SetValuePtr, QTSS_GetValue`, and `QTSS_GetValuePtr`.

> **Note:** If you are adding attributes to an object that your module created, you must first lock the
> object by calling `QTSS_LockObject. When` all of the attributes have been added, call
> `QTSS_UnlockObject` to unlock the object.

The sample code in Listing 2-6 (page 138) calls `QTSS_AddStaticAttribute` to add an attribute to the object
`QTSS_ClientSessionObject`.

Listing 2-6   Adding a static attribute

```
QTSS_Error MyRegisterRoleFunction()
{
    // Add the static attribute. The third parameter is always NULL.
```

```
    QTSS_Error theErr = QTSS_AddStaticAttribute(qtssClientSessionObjectType,
                            "MySampleAttribute", NULL, qtssAttrDataTypeUInt32);
    // Retrieve the ID for this attribute. This ID can be passed into QTSS_GetValue,
    // QTSS_SetValue, and QTSS_GetValuePtr.
    QTSS_AttributeID theID;
    theErr = QTSS_IDForAttr(qtssClientSessionObjectType, MySampleAttribute",
 &theID);
    // Store the attribute ID in a global for later use. Attribute IDs do not
    // change while the server is running.
    gMyExampleAttrID = theID;
}
```

Note: Attribute permissions for an added attribute (static or instance) are automatically set to readable, writable, and preemptive safe.

Using Files

QTSS supports file system modules so that QTSS can transparently and easily work with custom file systems. For example, a QTSS file system module can allow a QTSS module to read a custom networked file system or a custom database. Support for reading files consists of the following:

- QTSS file system callback routines that any module can use to open, read, and close files. Calling the file system callback routines is described in the section "Reading Files Using Callback Routines" (page 139). The QTSS file system callback routines allow QTSS to easily work with many different file system types. A QTSS module that uses the file system callbacks for reading all files can transparently use whatever file system is deployed on a server.

- File system roles for which modules that implement file systems register. These roles provide a bridge between QTSS and a specific file system. The file system roles are described in the section "Implementing a QTSS File System Module" (page 141). You could, for example, write a file system module that interfaces QTSS to a custom database or a custom networked file system.

Reading Files Using Callback Routines

In QTSS, a file is represented by a QTSS stream, so you can use existing QTSS stream callback routines to read files. The callback routines that are available for working with files are:

- `QTSS_OpenFileObject`, which is called to open a file in the local operating system. This call is one of two callback routines that is only used when working with files.

- `QTSS_CloseFileObject`, which is called to close a file that was opened by a previous call to `QTSS_OpenFileObject`. This call is one of two callback routines that is only used when working with files.

- `QTSS_Read`, which is called to read data from a file object's stream that was created by a previous call to `QTSS_OpenFileObject`.

- `QTSS_Seek`, which is called to set the current position of a file object's stream.

- `QTSS_Advise`, which is called to tell a file system module that a specified section of one of its streams will be read soon.

- `QTSS_RequestEvent`, which is called to tell a file system module that the calling module wants to be notified when one of the events in the specified event mask occurs. The events are when a stream becomes readable and when a stream becomes writable.

In QTSS, a file is `QTSS_Object` that has its own object type, `QTSS_FileObject`, that allows you to use standard QTSS callbacks (`QTSS_GetValue`, `QTSS_GetValueAsString`, and `QTSS_GetValuePtr`) to get meta information about a file, such as its length and modification date. You can use standard QTSS callbacks to store any amount of file system meta information with the file object. For example, a module working with a POSIX file system would want to add an attribute to the file object that stores the POSIX file system descriptor. A file object also has a QTSS stream reference that can be used when calling QTSS stream routines that work with files, such as `QTSS_Read`.

The sample code in `Listing 3–7` (page 140) shows how to open a file, determine the file's length, read the entire file, close the file, and return the data it contains.

Listing 2-7   Reading a file

```
QTSS_Error ReadEntireFile(char* inPath, void** outData, UInt32* outDataLen)

{


    QTSS_Object theFileObject = NULL;

    QTSS_Error theErr = QTSS_OpenFileObject(inPath, qtssOpenFileNoFlags,
&theFileObject);

    if (theErr != QTSS_NoErr)

        return theErr; // The file wasn't found or it couldn't be opened.


    // The file is open. Find out how long it is.
```

| 2009-06-01 | © 2002, 2009 Apple Inc. All Rights Reserved.

140

```
    UInt64* theLength = NULL;

    UInt32 theParamLen = 0;

    theErr = QTSS_GetValuePtr(theFileObject, qtssFlObjLength, 0,  (void**)&theLength,
  &theParamLen);


    if (theErr != QTSS_NoErr)

        return theErr;

    if (theParamLen != sizeof(UInt64))

        return QTSS_RequestFailed;;


    // Allocate memory for the file data.

    *outData = new char[*theLength + 1];

    *outDataLen = *theLength;


    // Read the data

    UInt32 recvLen = 0;

    theErr = QTSS_Read(theFileObject, *outData, *outDataLen, &recvLen);


    if ((theErr != QTSS_NoErr) || (recvLen != *outDataLen))

    {

        delete *outData;

        return theErr;

    }


    // Close the file.

    (void)QTSS_CloseFileObject(theFileObject);

}
```

Implementing a QTSS File System Module

A file system module provides a way for QTSS modules to read files in a specific file system regardless of that file system's type. Typically, a file system module handles a subset of paths in a file system, but it may handle all paths on the system. If a file system module handles only a certain subset of paths, it usually handles all paths inside a certain root path. For example, a module handling files stored in a certain database may only respond to paths that begin with `/Local/database_root/`.

Implementing a QTSS file system module begins with registering for one of the following roles:

- Open File Preprocess role, which the server calls in response to a module (or the server) that calls the `QTSS_OpenFileObject` callback routine to open a file. If the module does not handle files of the specified type, the module immediately returns `QTSS_FileNotFound`. If the module handles the files of the specified type, it opens the file, updates a file object provided by the server and returns `QTSS_NoErr`. If an error occurs during this setup period, the module returns `QTSS_RequestFailed`. Once the module returns `QTSS_NoErr`, it should be prepared to handle the Advise File, Read File, Request Event File and Close File roles for the opened file. The server calls each module registered in the Open File Preprocess role until one of the called modules returns `QTSS_NoErr` or `QTSS_RequestFailed`.

- Open File role, which the server calls in response to a module (or the server) that calls the `QTSS_OpenFileObject` callback routine for which all modules handling the Open File Preprocess role return `QTSS_FileNotFound`. Only one module can register for the Open File role. Like modules called for the Open File Preprocess role, the module called for the Open File role must determine whether it can handle the specified file. It it can, it opens the file, updates the file object provided by the server and returns `QTSS_NoErr`. If an error occurs during the setup process or if the module cannot handle the specified file, the module returns `QTSS_RequestFailed` or `QTSS_FileNotFound`, respectively.

A file system module should register in the Open File Preprocess role if it handles a subset of files available on the system. For instance, a file system module that serves files out of a database may only handle files rooted at a certain path. All other paths should fall through to other modules that handle other paths.

A file system module should register in the Open File role if it implements the default file system on a system. For instance, on a UNIX system the module handling the Open File Role would probably provide an interface between the server and the standard POSIX file system.

Once a module returns `QTSS_NoErr` from either the Open File Role or the Open File Preprocess role, it is responsible for the newly opened file. It should be prepared to handle the following roles on behalf of that file:

- Advise File role, which is called in response to a module (or the server) calling the `QTSS_Advise` callback for a file object. The `QTSS_Advise` callback is made to inform the file system module that a specific region of the file will be needed soon.

- Read File role, which is called in response to a module (or the server) calling the `QTSS_Read` callback for a file object. It is the responsibility of a file system module handling this role to make a best-effort attempt to fill the buffer provided by the caller with the appropriate file data.

- Request Event File role, which is called in response to a module (or the server) calling the `QTSS_RequestEvent` callback on a file object.

- Close File role, which is called in response to a module (or the server) calling the `QTSS_Close` callback on a file object. The module should clean up any file-system and module-specific data structures for this file. This role is always the last role a file system module will be invoked in for a given file object.

> **Note:** Modules do not need to explicitly register for the Advise File, Read File, Request Event File or Close File roles in order to handle them. Instead, returning `QTSS_NoErr` or `QTSS_RequestFailed` from one of the open file roles constitutes taking ownership for a specific file object, and therefore means that the module has implicitly registered for those roles.

File System Module Roles

This section describes the file system module roles. The roles are:

- "Open File Preprocess Role" (page 143) which is called to process requests to open files.

- "Open File Role" (page 144) which is the default role that is called when none of the modules registered for the Open File Preprocess role opens the specified file.

- "Advise File Role" (page 145) which is called to tell a file system module about the caller's I/O preferences.

- "Read File Role" (page 146) which is called to read a file.

- "Close File Role" (page 147) which is called to close a file.

- "Request Event File Role" (page 148) which is called to request notification when a file becomes available for reading or writing.

Open File Preprocess Role

The server calls the Open File Preprocess role in response to a module that calls the `QTSS_OpenFileObject` callback routine to open a file. It is the responsibility of a module handling this role to determine whether it handles the type of file specified to be opened. If it does and if the file exists, the module opens the file, updates the file object provided by the server, and returns `QTSS_NoErr`.

When called, an Open File Preprocess role receives a `QTSS_OpenFile_Params` structure, which is defined as follows:

```
typedef struct
{
    char*               inPath;
    QTSS_OpenFileFlags  inFlags;
    QTSS_Object         inFileObject;
} QTSS_OpenFile_Params;s
```

`inPath`
    A pointer to a null-terminated C string containing the full path to the file that is to be opened.

`inFlags`

> Open flags specifying whether the module that called `QTSS_OpenFileObject` can handle asynchronous read operations (`qtssOpenFileAsync`) or expects to read the file in order from beginning to end (`qtssOpenFileReadAhead`).

`inFileObject`

> A QTSS object that the module updates if it can open the file specified by `inPath`.

If the file is a file the module handles, the module should do whatever work is necessary to open and set up the file. It can use `inFileObject` to store any module-specific information for that file. In addition, the module should set the value of the file object's `qtssFlObjLenth` and `qtssFlObjModDate` attributes.

If the file is a file the module handles but an error occurs while attempting to set up the file, the module should return `QTSS_RequestFailed`.

If every module registered for the Open File Preprocess role returns `QTSS_FileNotFound`, the server calls the one module that is registered in the Open File role.

A module that wants to be called in the Open File Preprocess role must in its Register role call `QTSS_AddRole` and specify `QTSS_OpenFilePreprocess_Role` as the role. Modules that register for this role must also handle the following roles, but they do not need to explicitly register for them: Advise File, Read File, Request Event File, and Close File.

Open File Role

The server calls the module registered for the Open File role when all modules registered for the Open File Preprocess role have been called and have returned `QTSS_FileNotFound`. Only one module can be registered for the Open File role, and that module is the first module that registers for this role when QTSS starts up.

Like modules called for the Open File Preprocess role, it is the responsibility of a module handling the Open File role to determine whether it handles the type of file specified to be opened. If it does and if the file exists, the module opens the file, updates the file object provided by the server, and returns `QTSS_NoErr`.

When called, the module receives a `QTSS_OpenFile_Params` structure, which is defined as follows:

```
typedef struct
{
    char*               inPath;
    QTSS_OpenFileFlags  inFlags;
    QTSS_Object         inFileObject;
} QTSS_OpenFile_Params;
```

`inPath`

> A pointer to a null-terminated C string containing the full path to the file that is to be opened.

`inFlags`

> Open flags specifying whether the module that called `QTSS_OpenFileObject` can handle asynchronous read operations (`qtssOpenFileAsync`) or expects to read the file in order from beginning to end (`qtssOpenFileReadAhead`).

`inFileObject`

> A QTSS object that the module updates if it can open the file specified by `inPath`.

If the file is a file the module handles, the module should do whatever work is necessary to open and set up the file. It can use `inFileObject` to store any module-specific information for that file. In addition, the module should set the value of the file object's `qtssFlObjLength` and `qtssFlObjModDate` attributes.

If the file is a file the module handles but an error occurs while attempting to set up the file, the module should return `QTSS_RequestFailed`.

A module that wants to be called in the Open File role must in its Register role call `QTSS_AddRole` and specify `QTSS_OpenFile_Role` as the role. Modules that register for this role must also handle the following roles, but they do not need to explicitly register for them: Advise File, Read File, Request Event File, and Close File.

Advise File Role

The server calls modules for the Advise File role in response to a module (or the server) calling the `QTSS_Advise` callback routine for a file object in order to inform the file system module that the calling module will soon read the specified section of the file.

When called, an Advise File role receives a `QTSS_AdviseFile_Params` structure, which is defined as follows:

```
typedef struct
{
    QTSS_Object         inFileObject;
    UInt64              inPosition;
    UInt32              inSize;
} QTSS_AdviseFile_Params;
```

`inFileObject`

> The file object for the opened file. The file system module uses the file object to determine the file for which the `QTSS_Advise` callback routine was called.

145

`inPosition`

The offset in bytes from the beginning of the file that represents the beginning of the section that is soon to be read.

`inSize`

The number of bytes that are soon to be read.

The file system module is not required to do anything while handling this role, but it may take this opportunity to read the specified section of the file.

File system modules do not need to explicitly register for this role.

Modules should always return `QTSS_NoErr` when they finish handling this role.

Read File Role

The server calls modules for the Read File role in response to a module (or the server) calling the `QTSS_Read` callback routine for a file object in order to read the specified file.

When called, a Read File role receives a `QTSS_ReadFile_Params` structure, which is defined as follows:

```
typedef struct
{   QTSS_Object inFileObject;

    UInt64      inFilePosition;

    void*       ioBuffer;

    UInt32      inBufLen;

    UInt32*     outLenRead;

} QTSS_ReadFile_Params;
```

`inFileObject`

The file object for the file that is to be read. The file system module uses the file object to determine the file for which the `QTSS_Read` callback routine was called.

`inFilePosition`

The offset in bytes from the beginning of the file that represents the beginning of the section that is to be read. The server maintains the file position as an attribute of the file object, so the file system module does not have to cache the file position internally and can obtain the position at any time.

`ioBuffer`

A pointer to the buffer in which the file system module is to place the data that is read.

`ioBufLen`

> The length of the buffer pointed to by `ioBuffer`.

`outLenRead`

> The number of bytes actually read.

The file system module should make a best-effort attempt to fill the buffer pointed to by `ioBuffer` with data from the file that is being read starting with the position specified by `inFilePosition`.

If the file was opened with the `qtssOpenFileAsync` flag, the module should return `QTSS_WouldBlock` if reading the data will cause the thread to block. Otherwise, the module should block the thread until all of the data has become available. When the buffer pointed to by `ioBuffer` is full or the end of file has been reached, the file system module should set `outLenRead` to the number of bytes read and return `QTSS_NoErr`.

If the read fails for any reason, the file system module handling this role should return `QTSS_RequestFailed`.

File system modules do not need to explicitly register for this role.

Close File Role

The server calls modules for the Close File role in response to a module (or the server) calling the `QTSS_CloseFile` callback routine for a file object in order to close a file that has been opened.

When called, a Close File role receives a `QTSS_CloseFile_Params` structure, which is defined as follows:

```
typedef struct
{
    QTSS_Object        inFileObject;
} QTSS_CloseFile_Params;
```

`inFileObject`

> The file object for the file that is to be closed. The file system module uses the file object to determine the file for which the `QTSS_Close` callback routine was called.

A module handling this role should dispose of any data structures that it has created for the file that is to be closed.

This role is always the last role for which a file system module will be invoked for any given file object.

File system modules do not need to explicitly register for this role.

Modules should always return QTSS_NoErr when they finish handling this role.

Request Event File Role

The server calls modules for the Request Event File role in response to a module (or the server) calling the QTSS_RequestEvent callback routine. If a module or the server calls the QTSS_OpenFileObject callback routine and specifies the qtssOpenFileAsync flag, the file system module handling that file object may return QTSS_WouldBlock from its Read File role. When that occurs, the caller of QTSS_Read may call QTSS_RequestEvent callback to tell the server that the caller of QTSS_Read wants to be notified when the data becomes available for reading.

When called, a Request Event File role receives a QTSS_RequestEventFile_Params structure, which is defined as follows:

```
typedef struct
{
    QTSS_Object      inFileObject;
    QTSS_EventType   inEventMask;
} QTSS_RequestEventFile_Params;
```

inFileObject
> The file object for the file for which notifications are requested. The file system module uses the file object to determine the file for which the QTSS_RequestEvent callback routine was called.

inEventMask
> A mask specifying the type of events for which notification is requested. Possible values are QTSS_ReadableEvent and QTSS_WriteableEvent.

If the file system that the file system module is implementing supports notification, the file system module should do whatever setup is necessary to receive an event for the file for which the QTSS_RequestEvent callback routine was called. When the file becomes readable, the file system module should call the QTSS_SignalStream callback routine and pass the stream reference for this file object (which can be obtained through the file object's qtssFlObjStream attribute). Calling the QTSS_SignalStream callback routine tells the server that the caller of QTSS_RequestEvent should be notified that the file is now readable.

File system modules do not need to explicitly register for this role.

Modules should always return QTSS_NoErr when they finish handling this role.

| 2009-06-01 | © 2002, 2009 Apple Inc. All Rights Reserved.

148

Sample Code for the Open File Role

The sample code in `Listing 3–8` (page 149) handles the Open File role, but it could also be used to handle the Open File Preprocess role. This code uses the POSIX file system layer as the file system and does not support asynchronous I/O.

Listing 2-8   Handling the Open File Role

```
QTSS_Error OpenFile(QTSS_OpenFile_Params* inParams)
{
    // Use the POSIX open call to attempt to open the specified file.
    // If it doesn't exist, return QTSS_FileNotFound

    int theFile = open(inParams->inPath, O_RDONLY);
    if (theFile == -1)
        return QTSS_FileNotFound;

    // Use the POSIX stat call to get the length and the modification date
    // of the file. This information must be set in the QTSS_FileObject
    // by every file system module.

    UInt64 theLength = 0;
    time_t theModDate = 0;
    struct stat theStatStruct;
    if (::fstat(fFile, &theStatStruct) >= 0)
    {
        theLength = buf.st_size;
        theModDate = buf.st_mtime;
    }
    else
    {
        ::close(theFile);
        return QTSS_RequestFailed; // Stat failed
    }

    // Set the file length and the modification date attributes of this file
    // object before returning
```

| 2009-06-01 | © 2002, 2009 Apple Inc. All Rights Reserved.

149

```
    (void)QTSS_SetValue(inParams->inFileObject, qtssFlObjLength, 0, &theLength,
sizeof(theLength));

    (void)QTSS_SetValue(inParams->inFileObject, qtssFlObjModDate, 0, &theModDate,
  sizeof(theModDate));


    // Place the file reference in a custom attribute in the QTSS_FileObject.
    // This way, we can easily get the file reference in other role handlers,
    // such as the QTSS_ReadFile_Role and the QTSS_CloseFile_Role.

    QTSS_Error theErr = QTSS_SetValue(inParams->inFileObject, sFileRefAttr, 0,
&theFile, sizeof(theFileSource));


    if (theErr != QTSS_NoErr)
    {
        ::close(theFile);
        return QTSS_RequestFailed;
    }


    return QTSS_NoErr;
}
```

Implementing Asynchronous Notifications

If a module, or the server, calls the QTSS_OpenFileObject and specifies the qtssOpenFileAsync flag, the file system module handling that file object may return QTSS_WouldBlock from its QTSS_ReadFile_Role handler. Once that happens, the caller of QTSS_Read may want to be notified when the requested data becomes available for reading. This is possible by calling the QTSS_RequestEvent callback, which tells the server that the caller would like to be notified when data is available to be read from the file.

Not all file systems support notification mechanisms, and if they do, the notification mechanisms are particular to each file system architecture. Therefore, whether a file system module supports notifications is at the discretion of the developer of the file system module. In general it is better for a file system module to support asynchronous notifications and not block in QTSS_ReadFile_Role because blocking on one file operation may disrupt service for many of the server's clients.

Two facilities allow file system modules to implement notifications:

- `QTSS_RequestEventFile_Role`, which is called in response to a module (or the server) calling the `QTSS_RequestEvent` callback on a file object. Modules do not need to explicitly register for this role. If a module doesn't implement asynchronous notifications, it should return `QTSS_RequestFailed` from this role. If a module does implement asynchronous notifications, it should do whatever setup is necessary to receive an event for this file when the file becomes readable.

- `QTSS_SendEventToStream` callback, called by a file system module when a file does become readable. Calling `QTSS_SendEventToStream` tells the server that the caller of `QTSS_RequestEvent` should be notified that the file is now readable.

Using the Admin Protocol

You can use the Admin protocol to communicate with QTSS. The Admin Protocol relies on the URI mechanism defined by RFC 2396 for specifying a container entity using a path and on the request and response mechanism for the Hypertext Transfer Protocol defined in RFC 1945.

The server's internal data is mapped to a hierarchical tree of element arrays. Each element is a named type including a container type for retrieval of sub-node elements.

The server state machine and database can be accessed through a regular expression. The Admin Protocol abstracts the QTSS module API to handle data access and in some cases to provide data access triggers for execution of server functions.

Server streaming threads are blocked while the Admin Protocol accesses the server's internal data. To minimize blocking, the Admin Protocol allows scoped access to the server's data structures by allowing specific URL paths to any element.

The Admin Protocol uses the HTTP GET as the request and response method. At the end of each response, the session between client and server is closed. The Admin Protocol also supports the Authorization request header field as described in RFC 1945, section 10.2.

Access to Server Data

The Admin Protocol uses URIs to specify the location of server data. The following URI references the top level of the server's hierarchical data tree using a simple HTTP GET request.

```
GET /modules/admin
```

Request Syntax

A valid request is an absolute reference followed by the server URI. An absolute reference is a path beginning with a forward slash character (/). A path represents the server's virtual hierarchical data structure of containers and is expressed as a URL.

Here is the request syntax:

[*absolute URL*]?[`parameters=`"*values*"]+[`command=`"*value*"]+["`option`"="*value*"]

The following rules govern URIs:

- */path* is an absolute reference.
- *path*/∗ is defined as all elements contained in the "path" container.
- An asterisk (∗) in the current URL location causes each element in that location to be iterated.
- A question mark (?) indicates that options follow. Options are specified as `name=`"*value*" pairs delimited by the plus (+) character.
- Space and tab characters are treated as stop characters.
- Values can be enclosed by the double quotation characters ("). Enclosing double quotation characters is required for values that contain spaces and tabs.
- These characters cannot be used: period (`.`), two periods (`..`), and semicolon (`;`).

Here is an example of a request:

```
GET /modules/admin/server/qtssSvrClientSessions?parameters=rt+command=get
```

Request Functionality

Requests can contain an array iterator, a name lookup, a recursive tree walk, and a filtered response. All functions can execute in a single URI query.

Here is a request that gets the stream time scale and stream payload name for every stream in every session:

```
GET /modules/admin/server/qtssSvrClientSessions/*/qtssCliSesStreamObjects?
parameters=r+command=get+filter1=qtssRTPStrTimescale+filter2=qtssRTPStrPayloadName
```

where

- ∗ iterates the array of sessions

- `r` in `parameter=rt` specifies a recursive walk and t specifies that data types are to be included in the result
- `filter=qtssRTPStrTimescale` specifies that the stream time scale is to be returned
- `filter2=qtssRTPStrPayloadName` specifies that the stream payload is to be returned

This request gets all server module names and their descriptions:

```
GET /modules/admin/server/qtssSvrModuleObjects?

parameters=r+command=get+filter2=qtssModDesc+filter1=qtssModName
```

The following example does a recursive search and gets all server attributes and their data types:

```
GET /modules/admin/server/?parameters=rt
```

> **Note:** Repeated recursive searches should be avoided because they impact server performance.

The following examples return server attributes and their paths:

```
GET /modules/admin/server/*
GET /modules/admin/server/qtssSvrPreferences/*
```

Data References

All elements are arrays. Single element arrays may be referenced in any of the following ways:

- *path* / *element*
- *path* / *element* /
- *path* / *element* / *
- *path* / *element* / 1

The references listed above are all evaluated as the same request.

Request Options

URIs that do not include a question mark (?) default to a GET request option.

URIs that include a question mark (?) must be followed by a "`command=`*command-option*" request option, where *command-option* is `GET`, `SET`, `ADD`, or `DEL`. URIs may also be followed by a "`parameters=`*parameter-option*" that refines the action of the command option.

Request options are not case-sensitive, but request option values are case-sensitive.

The Admin Protocol ignores any request option that it does not recognize as well any request options that a command does not require.

Command Options

The Admin Protocol recognizes the following command options:

- `GET`, described in the section "GET Command Option" (page 154)

- `SET`, described in the section "SET Command Option" (page 155)

- `DEL`, described in the section "DEL Command Option" (page 155)

- `ADD`, described in the section "ADD Command Option" (page 155)

Any unknown command option is reported as an error.

The effect of a command option may be modified by in the inclusion of one or more of the following modifiers:

- `value` — used to specify a value

- `type` — used to specify a data type

- `name` — used to specify an element name

GET Command Option

The `GET` command option gets the data identified by the URI. It is the default command option. For that reason, it does not have to be specified, as shown in the following example:

```
GET /modules/admin/example_count
```

The `GET` command does not require any request options. If any request options were specified, they would be ignored.

| 2009-06-01 | © 2002, 2009 Apple Inc. All Rights Reserved.

154

SET Command Option

The SET command option sets the data identified by the URI. No value checking is performed. Conversion between the text value and the actual value is type-specific. Here are two examples of the `SET command option`:

```
GET /modules/admin/example_count?command=SET+value=5

GET /modules/admin/maxcount?command=SET+value=5+type=SInt32
```

If the `type` option is included in the command, type checking of the server element type and the set type is performed. If the types do not match, an error is returned and the command fails.

DEL Command Option

The DEL command option deletes the element referenced by the URL and any data it contains. Here is an example:

```
GET /modules/admin/maxcount?command=DEL
```

ADD Command Option

The ADD command option adds the data specified by the URI to the specified element.

If the end of the URL is an element, the ADD command performs an add to the array of elements referenced by the element name. The following example adds 6 to `example_count` if the data type of example_count is `SInt16`:

```
GET /modules/admin/example_count?command=ADD+value=6+type=SInt16
```

If the element at the end of the URL is a `QTSS_Object` container, the ADD command option adds the element to the container. The following example adds 5 to the element whose name is `maxcount` if the data type of `maxcount` is `SInt16`:

```
GET /modules/admin/?command=ADD+value=5+name=maxcount+type=SInt16
```

Parameter Options

Parameter options are single characters without delimiters that appear after the URL.

The Admin Protocol recognizes the following parameter options:

- r — Walk downward in the hierarchy starting at end of the URL. Recursion should be avoided if "*" iterators or direct URL access to elements can be used instead.

- v — Return the full path in *name*.

- a — Return the access type.

- t — Return the data type of *value*.

- d — Return debugging information if an error occurs.

- c — Return the count of elements in the path.

Here is an example that uses the r and t parameter options to recursively get the data type of all `qtssSvrClientSessions`:

```
GET /modules/admin/server/qtssSvrClientSessions?parameters=rt+command=get
```

Attribute Access Types

The following access types are used to control access to server data:

- r — Read access type

- w — Write access type

- p — Preemptive safe access type

Data Types

Data types can be any server-allowed text value. New data types can be defined and returned by the server, so data types are not limited to the basic set listed here:

UInt8	SInt16	UInt64	Float64	char
SInt8	UInt32	SInt64	Bool8	QTSS_Object
UInt16	SInt32	Float32	Bool16	void_pointer

Values of type `QTSS_Object`, pointers, and unknown data types always converted to a host-ordered string of hexadecimal values. Here is an example of a hexadecimal value result:

```
unknown_pointer=halogen; type=void_pointer
```

Server Responses

This section describes the data that is returned in response to a request. The information on response data is organized in the following sections:

Unauthorized Response

Here is an example of an unauthorized response:

```
HTTP/1.1 401 Unauthorized

WWW-Authenticate: Basic realm="QTSS/modules/admin"

Server: QTSS

Connection: Close

Content-Type: text/plain
```

OK Response

Here is an example of an "OK" response:

```
HTTP/1.0 200 OK

Server: QTSS/4.0 [v408]-MacOSX

Connection: Close

Content-Type: text/plain

Container="/"

admin/

error:(0)
```

All OK responses end with `error:(0)`.

Response Data

All entity references in response data follow this form:

[*NAME*=*VALUE*]**;**[*attribute*="*value*"]**,**[*attribute*="*value*"]

where brackets ([ ]) indicate that the enclosed response data is optional. Therefore, the response data may take the following forms:

NAME=*VALUE*

NAME=*VALUE* **;** *attribute*="*value*"

NAME=*VALUE* **;** *attribute*="*value*" **,** *attribute*="*value*"

All container references follow this form:

[*NAME* /]**;**[*attribute*="*value*"]**,**[*attribute*="*value*"]

where brackets ([ ]) indicate that the enclosed response data is optional. Therefore, response data may take the following forms:

NAME /

NAME / **;** *attribute*="*value*"

NAME **;** *attribute*="*value*" **,** *attribute*="*value*"

The order of appearance of container references and the container's entity references are important. This is especially true when the response is a recursive walk of a container hierarchy.

Each new level in the hierarchy must begin with a `Container=` reference. Each container list of elements must be a complete list of the contained elements and any containers. The appearance of a `Container=` reference indicates the end of a previous container's contents and the beginning of a new container.

This example shows how each new container is identified with a unique path:

```
Container="/level1/"
field1="value"
field2="value"
level2a/
level2b/
Container="/level1/level2a/"
field1="value"
```

```
level3a/

level3b/

Container="/level1/level2a/level3a"

field1="value"

Container="/level1/level2a/level3b"

Container="/level1/level2b/"

field1="value"

level3a/

Container="/level1/level2b/level3a/"

field1="value"
```

Array Values

For arrays of elements, a numerical value represents the index. Arrays are containers. Here is an example:

```
Container="/level1/"

field1="value"

field2="value"

array1/

Container="/level1/array1/"

1=value

2=value
```

Array elements may be containers, as shown in this example:

```
Container="/level1/array1/"

1/

2/

3/


Container="/level1/array1/1/"

field1="value"

field2="value"

Container="/level1/array1/2/"

Container="/level1/array1/3/"

field1="value"
```

Response Root

The root for responses is `/admin`.

Errors in Responses

For each response, the error state for the request is reported at the end of the data. Here are some examples:

`Error:(0)` indicates that no error occurred

`Error:(404)` indicates that no data was found

The number enclosed by parentheses is an HTTP error code followed by an error string when debugging is turned on using the "`parameters=d`" query option. Here is an example:

```
error:(404);reason="No data found"
```

Request and Response Examples

An easy way to make requests is to use a web browser and a URL like this:

`http://`*IP-address*`:554/modules/admin/?parameters=a+command=get`

The following example uses basic authentication and shows the HTTP response headers:

Request: `GET /modules/admin?parameters=a+command=get`

Authorization: Basic QWXtaW5pT3RXYXRvcjXkZWZhdWx0

Response:

```
HTTP/1.0 200 OK
Server: QTSS/4.0 [v408]—MacOSX
Connection: Close
Content—Type: text/plain


Container="/"
admin/;a=r
error:(0)
```

The following recursive request gets the value of each element in `/modules/admin`:

```
GET /modules/admin?command=get+parameters=r
```

The following recursive request returns the access type and data type for the value of each element in `/modules/admin`:

```
GET /modules/admin?command=get+parameters=rat
```

The following request gets the elements in `/modules/admin`. Note that the `GET` command option is not required because request options are not present.

```
GET /modules/admin/*
```

A request like the following can be used to monitor the session list:

```
GET /modules/admin/server/qtssSvrClientSessions/*
```

The response is a list of unique `qtssSvrClientSessions` session IDs. Here is an example::

```
Container="/admin/server/qtssSvrClientSessions/"
12/
2/
4/
8/
error:(0)
```

The following request gets the indexes for the `qtssCliSesStreamObjects` object, which is an indexed array of streams:

```
GET /modules/admin/server/qtssSvrClientSessions/*/qtssCliSesStreamObjects/*
```

The response might look like this:

```
Container="/admin/server/qtssSvrClientSessions/3/qtssCliSesStreamObjects/"
0/
1/
error:(0)
```

Here is another request:

```
GET /modules/admin/server/qtssSvrClientSessions/3/qtssCliSesStreamObjects/0/*
```

And here is a typical response:

```
qtssRTPStrTrackID="4"
qtssRTPStrSSRC="683618521"
qtssRTPStrPayloadName="X-QT/600"
qtssRTPStrPayloadType="1"
qtssRTPStrFirstSeqNumber="-7111"
qtssRTPStrFirstTimestamp="433634204"
qtssRTPStrTimescale="600"
qtssRTPStrQualityLevel="0"
qtssRTPStrNumQualityLevels="3"
qtssRTPStrBufferDelayInSecs="3.000000"
qtssRTPStrFractionLostPackets="0"
qtssRTPStrTotalLostPackets="52"
qtssRTPStrJitter="0"
qtssRTPStrRecvBitRate="1526072"
qtssRTPStrAvgLateMilliseconds="501"
qtssRTPStrPercentPacketsLost="0"
qtssRTPStrAvgBufDelayInMsec="30"
qtssRTPStrGettingBetter="0"
qtssRTPStrGettingWorse="0"
qtssRTPStrNumEyes="0"
qtssRTPStrNumEyesActive="0"
qtssRTPStrNumEyesPaused="0"
qtssRTPStrTotPacketsRecv="6763"
qtssRTPStrTotPacketsDropped="0"
qtssRTPStrTotPacketsLost="0"
qtssRTPStrClientBufFill="0"
qtssRTPStrFrameRate="0"
qtssRTPStrExpFrameRate="3903"
qtssRTPStrAudioDryCount="0"
```

```
qtssRTPStrIsTCP="false"

qtssRTPStrStreamRef="18861508"

qtssRTPStrCurrentPacketDelay="-2"

qtssRTPStrTransportType="0"

qtssRTPStrStalePacketsDropped="0"

qtssRTPStrTimeFlowControlLifted="974373815109"

qtssRTPStrCurrentAckTimeout="0"

qtssRTPStrCurPacketsLostInRTCPInterval="52"

qtssRTPStrPacketCountInRTCPInterval="689"

QTSSReflectorModuleStreamCookie=(null)

qtssNextSeqNum=(null)

qtssSeqNumOffset=(null)

QTSSSplitterModuleStreamCookie=(null)

QTSSFlowControlModuleLossAboveTol="0"

QTSSFlowControlModuleLossBelowTol="3"

QTSSFlowControlModuleGettingWorses="0"

error:(0)
```

Here is an request that returns the IP addresses of connected clients:

```
GET /modules/admin/server/qtssSvrClientSessions/*/qtssCliRTSPSessRemoteAddrStr
```

And here is a typical response:

```
Container="/admin/server/qtssSvrClientSessions/5/
"qtssCliRTSPSessRemoteAddrStr=17.221.40.1

Container="/admin/server/qtssSvrClientSessions/6/
"qtssCliRTSPSessRemoteAddrStr=17.221.40.2

Container="/admin/server/qtssSvrClientSessions/8/
"qtssCliRTSPSessRemoteAddrStr=17.221.40.3

Container="/admin/server/qtssSvrClientSessions/14/
"qtssCliRTSPSessRemoteAddrStr=17.221.40.4

error:(0)
```

Changing Server Settings

To change a server setting, the entity name and the value to be set are specified in the request body. If a match is made on the URL base and entity name at the current container level and if the setting is writable, the value is set.

```
base = /base/container

name = value

/base/container/name="value"
```

Getting and Setting Preferences

Preferences paths are useful for getting and setting a server or module preference. Setting a preference causes the preference's new value to be flushed to the server's XML preference file. The new value takes effect immediately.

Server preferences are stored in `/modules/admin/server/qtssSvrPreferences`. Module preferences are stored in `/modules/admin/server/qtssSvrModuleObjects/*/qtssModPrefs/`.

The elements defined in the `qtssSvrPreferences` object can only be modified — they cannot be deleted.

The elements defined in `qtssModPrefs` can be added to, deleted, and modified.

A module or the server can automatically restore some deleted elements if the elements are needed by a module or the server. When applied to a `qtssModPrefs` element, the `ADD`, `DEL`, and `SET` commands cause the streaming server's XML preference file to be rewritten.

Getting and Changing the Server's State

The `qtssSvrState` attribute controls the server's state. The path is `/modules/admin/server/qtssSvrState`. It can be modified as a `UInt32` with the following values.

```
qtssStartingUpState        = 0,

qtssRunningState           = 1,

qtssRefusingConnectionsState  = 2,

qtssFatalErrorState        = 3,

qtssShuttingDownState      = 4,

qtssIdleState              = 5
```

QTSS Callback Routines

This section describes the QTSS callback routines that modules call to obtain information from the server, allocate and deallocate memory, create objects, get and set attribute values, and manage client and RTSP sessions.

Callbacks by Task

QTSS Utility Callback Routines

Modules call the following callback routines to register for roles, allocate and deallocate memory, get the value of the server's internal timer, and to convert a value from the internal timer to the current time.

QTSS_AddRole (page 170)

> Adds a role.

QTSS_New (page 189)

> Allocates memory.

QTSS_Delete (page 178)

> Deletes memory.

QTSS_Milliseconds (page 188)

> Gets the current value of the server's internal clock.

QTSS_MilliSecsTo1970Secs (page 188)

> Converts a value obtained from the server's internal clock to the current time.

QTSS Object Callback Routines

Modules call the object callback routines to create, lock, and unlock o bjects.

QTSS_CreateObjectType (page 176)

> Creates an object type.

QTSS_CreateObjectValue (page 177)

> Creates a new object that is the value of another object's attribute.

QTSS_LockObject (page 187)

>   Locks an object.

QTSS_UnLockObject (page 203)

>   Unlocks an object.

QTSS Attribute Callback Routines

Modules call the attribute callback routines to work with attributes.

QTSS_AddInstanceAttribute (page 169)

>   Adds an instance attribute to the instance of an object.

QTSS_AddStaticAttribute (page 173)

>   Adds a static attribute to an object type.

QTSS_GetAttrInfoByID (page 179)

>   Uses an attribute ID to get information about an attribute.

QTSS_GetAttrInfoByIndex (page 180)

>   Gets information about all of an object's attributes by iteration.

QTSS_GetAttrInfoByName (page 181)

>   Uses an attribute's name to get information about an attribute.

QTSS_GetNumAttributes (page 182)

>   Gets a count of an object's attributes.

QTSS_GetValue (page 182)

>   Copies the value of an attribute into a buffer.

QTSS_GetValueAsString (page 183)

>   Gets the value of an attribute as a C string.

QTSS_GetValuePtr (page 184)

>   Gets a pointer to an attribute's value.

QTSS_IDForAttr (page 186)

>   Gets the ID of a static attribute.

QTSS_RemoveInstanceAttribute (page 192)

>   Remove an instance attribute from the instance of an object.

QTSS_RemoveValue (page 193)

>   Removes the specified value from an attribute.

QTSS_SetValue (page 197)

> Sets the value of an attribute.

QTSS_SetValuePtr (page 198)

> Sets an existing variable as the value of an attribute.

QTSS_StringToValue (page 200)

> Converts an attribute data type in C string format to a value in `QTSS_AttrDataType` format.

QTSS_TypeStringToType (page 201)

> Gets the attribute data type of a data type string that is in C string format.

QTSS_TypeToTypeString (page 202)

> Gets the name in C string format of an attribute data type.

QTSS_ValueToString (page 203)

> Converts an attribute data type in `QTSS_AttrDataType` format to a value in C string format.

Stream Callback Routines

This section describes the callback routines that modules call to perform I/O on streams. Internally, the server performs I/O asynchronously, so QTSS stream callback routines do not block and, unless otherwise noted, return the error `QTSS_WouldBlock` if data cannot be written.

QTSS_Advis (page 174)

> Advises that the specified section of the stream will soon be read.

QTSS_Read (page 192)

> Reads data from a stream.

QTSS_Seek (page 195)

> Sets the position of a stream.

QTSS_RequestEvent (page 194)

> Requests notification of specified events.

QTSS_SignalStream (page 199)

> Notifies the recipient of events that a stream has become available for I/O.

QTSS_Write (page 204)

> Writes data to a stream.

QTSS_WriteV (page 205)

> Writes data to a stream using an iovec structure.

QTSS_Flush (page 179)

> Forces an immediate write operation.

| 2009-06-01 | © 2002, 2009 Apple Inc. All Rights Reserved.

167

File System Callback Routines

Modules use the callback routines described in this section to open and close a file object.

QTSS_OpenFileObject (page 189)

> Opens a file.

QTSS_CloseFileObject (page 175)

> Closes a file.

Service Callback Routines

Modules use the callback routines described in this section to register and invoke services.

QTSS_AddService (page 172)

> Adds a service.

QTSS_IDForService (page 186)

> Resolves a service name to a service ID.

QTSS_DoService (page 178)

> Invokes a service.

RTSP Header Callback Routines

As a convenience to modules that want to send RTSP responses, the server provides the utilities described in this section for formatting RTSP responses properly.

QTSS_AppendRTSPHeader (page 175)

> Appends information to an RTSP header.

QTSS_SendRTSPHeaders (page 195)

> Sends an RTSP header.

QTSS_SendStandardRTSPResponse (page 196)

> Sends an RTSP response to a client.

RTP Callback Routines

QTSS modules can generate and send RTP packets in response to an RTSP request. Typically RTP packets are sent in response to a SETUP request from the client. Currently, only one module can generate packets for a particular session.

QTSS_AddRTPStream (page 171)

> Enables a module to send RTP packets to a client.

QTSS_Play (page 190)

> Starts playing streams associated with a client session.

QTSS_Pause (page 190)

> Pauses a stream that is playing.

QTSS_Teardown (page 201)

> Closes a client session.

Callbacks

QTSS_AddInstanceAttribute

Adds an instance attribute to the instance of an object.

```
QTSS_Error QTSS_AddInstanceAttribute(
    QTSS_Object inObject,
    char* inAttrName,
    void* inUnused,
    QTSS_AttrDataType inAttrDataType);
```

Parameters

inObject

> On input, a value of type QTSS_Object (page 207) that specifies the object to which the instance attribute is to be added.

inAttrName

> On input, a pointer to a byte array that specifies the name of the attribute that is to be added.

inUnused

> Always NULL.

`QTSS_AttrDataType`

> On input, a value of type `QTSS_AttrDataType` (page 211) that specifies the data type of the attribute that is being added.

`result`

> A result code. Possible values are `QTSS_NoErr`, `QTSS_OutOfState` if `QTSS_AddInstanceAttribute` is called from a role other than the Register role, `QTSS_BadArgument` if the specified object type does not exist, the attribute name is too long, or a parameter is not specified, and `QTSS_AttrNameExists` if an attribute of the specified name already exists.

Discussion

The `QTSS_AddInstanceAttribute` callback routine adds an attribute to the instance of an object as specified by the `inObject` parameter. This callback can only be called from the Register role.

When adding attributes to an object that a module as created, you must lock the object first by calling `QTSS_LockObject` (page 187). Add the attributes and then call `QTSS_UnLockObject` (page 203).

All added instance attributes have values that are implicitly readable, writable, and preemptive safe, so their values can be obtained by calling `QTSS_GetValueAsString` (page 183) and `QTSS_GetValuePtr` (page 184). You can also call `QTSS_GetValue` (page 182) to get the value of an added static attribute, but doing so is less efficient.

Adding static attributes is more efficient than adding instance attributes, so adding static attributes instead of adding instance attributes is strongly recommended.

Typically, a module adds an instance attribute and sets its value by calling `QTSS_SetValue` (page 197) when it is first installed to add its default preferences to its module preferences object. On subsequent runs of the server, the preferences will already exist in the module's module preferences object, so the module only needs to call `QTSS_GetValue` (page 182), `QTSS_GetValueAsString` (page 183), or `QTSS_GetValuePtr` (page 184) to get the value. Calling `QTSS_GetValuePtr` is the most efficient and recommended way to get the value of an attribute. Calling `QTSS_GetValue` is less efficient than calling `QTSS_GetValuePtr`, and calling `QTSS_GetValueAsString` is less efficient than calling `QTSS_GetValue`.

Call `QTSS_RemoveValue` (page 193) to remove the value of an added attribute.

Unlike static attributes, instance attributes can be removed. To remove an instance attribute from the instance of an object, call `QTSS_RemoveInstanceAttribute` (page 192).

QTSS_AddRole

Adds a role.

```
QTSS_Error QTSS_AddRole(QTSS_Role inRole);
```

Parameters

`inRole`

On input, a value of type `QTSS_Role` (page 209) that specifies the role that is to be added.

`result`

A result code. Possible values are `QTSS_NoErr`, `QTSS_OutOfState` if `QTSS_AddRole` is called from a role other than the Register role, `QTSS_RequestFailed` if the module is registering for the RTSP Request role and a module is already registered for that role, and `QTSS_BadArgument` if the specified role does not exist.

Discussion

The `QTSS_AddRole` callback routine tells the server that your module can be called for the role specified by `inRole`.

The `QTSS_AddRole` callback can only be called from a module's Register role. For this version of the server, you can add the following roles: `QTSS_ClientSessionClosing_Role`, `QTSS_ErrorLog_Role`, `QTSS_Initialize_Role`, `QTSS_OpenFilePreprocess_Role`, `QTSS_OpenFile_Role`, `QTSS_RTSPFilter_Role`, `QTSS_RTSPRoute_Role`, `QTSS_RTSPPreProcessor_Role`, `QTSS_RTSPRequest_Role`, `QTSS_RTSPPostProcessor_Role`, `QTSS_RTPSendPackets_Role`, `QTSS_RTCPProcess_Role`, `QTSS_Shutdown_Role`.

QTSS_AddRTPStream

Enables a module to send RTP packets to a client.

```
QTSS_Error QTSS_AddRTPStream(
    QTSS_ClientSessionObject inClientSession,
    QTSS_RTSPRequestObject inRTSPRequest,
    QTSS_RTPStreamObject* outStream,
    QTSS_AddStreamFlags inFlags);
```

Parameters

`inClientRequest`

On input, a value of type `QTSS_ClientSessionObject` identifying the client session for which the sending of RTP packets is to be enabled.

inRTSPRequest

>   On input, a value of type `QTSS_RTSPRequestObject`.

outStream

>   On output, a pointer to a value of type `QTSS_RTPStreamObject`, containing the newly created stream.

inFlags

>   On input, a value of type `QTSS_AddStreamFlags` (page 213) that specifies stream options.

result

>   A result code. Possible values are `QTSS_NoErr, QTSS_RequestFailed` if the `QTSS_RTPSteamObject` couldn't be created, and `QTSS_BadArgument` if a parameter is invalid.

Discussion

The `QTSS_AddRTSPStream` callback routine enables a module to send RTP packets to a client in response to an RTSP request. Call `QTSS_AddRTSPStream` multiple times in order to add more than one stream to the session.

To start playing a stream, call `QTSS_Play` (page 190).

QTSS_AddService

Adds a service.

```
QTSS_Error QTSS_AddService(
    const char* inServiceName,
    QTSS_ServiceFunctionPtr inFunctionPtr);
```

Parameters

inServiceName

>   On input, a pointer to a string containing the name of the service that is being added.

inFunctionPtr

>   On input, a pointer to the module that provides the service that is being added.

result

>   A result code. Possible values include `QTSS_NoErr`, `QTSS_OutOfState` if `QTSS_AddService` is not called from the Register role, and `QTSS_BadArgument` if `inServiceName` is too long or if a parameter is `NULL`.

Discussion

The `QTSS_AddService` callback routine makes the specified service available for other modules to call.

This callback can only be called from the Register role.

QTSS_AddStaticAttribute

Adds a static attribute to an object type.

```
QTSS_Error QTSS_AddStaticAttribute(
    QTSS_ObjectType inObjectType,
    const char* inAttributeName,
    void* inUnused,
    QTSS_AttrDataType inAttrDataType);
```

Parameters

inType

On input, a value of type `QTSS_ObjectType` (page 208) that specifies the type of object to which the attribute is to be added.

inAttributeName

On input, a pointer to a byte array that specifies the name of the attribute that is to be added.

inUnused

Always `NULL`.

QTSS_AttrDataType

On input, a value of type `QTSS_AttrDataType` (page 211) that specifies the data type of the attribute that is being added.

result

A result code. Possible values are `QTSS_NoErr`, `QTSS_OutOfState` if `QTSS_AddStaticAttribute` is called from a role other than the Register role, `QTSS_BadArgument` if the specified object type does not exist, the attribute name is too long, or a parameter is not specified, and `QTSS_AttrNameExists` if an attribute of the specified name already exists.

Discussion

The `QTSS_AddStaticAttribute` callback routine adds the specified attribute to all objects of the type specified by the `inType` parameter. This callback can only be called from the Register role. Once added, static attributes cannot be removed while the server is running.

When adding attributes to an object that a module as created, you must lock the object first by calling QTSS_LockObject (page 187). Add the attributes and then call QTSS_UnLockObject (page 203).

Adding static attributes is more efficient than adding instance attributes, so adding static attributes instead of instance attributes is strongly recommended.

The values of all added static attributes are implicitly readable, writable, and preemptive safe. Call QTSS_SetValue (page 197) or QTSS_SetValuePtr (page 198) to set the value of an added attribute.

Call QTSS_GetValuePtr (page 184), QTSS_GetValue (page 182) or QTSS_GetValueAsString (page 183) to get the value of a static attribute that has been added. Calling QTSS_GetValuePtr is the most efficient and recommended way to get the value of an attribute. Calling QTSS_GetValue is less efficient than calling QTSS_GetValuePtr, and calling QTSS_GetValueAsString is less efficient than calling QTS_GetValue.

Call QTSS_RemoveValue (page 193) to remove the value of an added static attribute.

QTSS_Advis

Advises that the specified section of the stream will soon be read.

```
QTSS_Error QTSS_Advise(
    QTSS_StreamRef inRef,
    UInt64 inPosition,
    UInt32 inAdviseSize);
```

Parameters
inRef

On input, a value of type QTSS_StreamRef (page 209) obtained by calling QTSS_OpenFileObject (page 189) that specifies the stream.

inPosition

On input, the offset in bytes from the beginning of the stream that marks the beginning of the advise section.

inAdviseSize

On input, the size in bytes of the advise section.

result

A result code. Possible values include QTSS_NoErr, QTSS_BadArgument if a parameter is invalid, and QTSS_RequestFailed.

Discussion

The `QTSS_Advise` callback routine tells a file system module that the specified section of a stream will be read soon. The file system module may read ahead in order to respond more quickly to future calls to `QTSS_Read` for the specified stream.

QTSS_AppendRTSPHeader

Appends information to an RTSP header.

```
QTSS_Error QTSS_AppendRTSPHeader(
    QTSS_RTSPRequestObject inRef,
    QTSS_RTSPHeader inHeader,
    const char* inValue,
    UInt32 inValueLen);
```

Parameters

`inRef`

On input, a value of type `QTSS_RTSPRequestObject` for the RTSP stream.

`inHeader`

On input, a value of type `QTSS_RTSPHeader`.

`inValue`

On input, a pointer to a byte array containing the header that is to be appended.

`inValueLen`

On input, a value of type `UInt32` containing the length of valid data pointed to by `inValue`.

`result`

A result code. Possible values are `QTSS_NoErr` and `QTSS_BadArgument` if a parameter is invalid.

Discussion

The `QTSS_AppendRTSPHeader` callback routine appends headers to an RTSP header. After calling `QTSS_AppendRTSPHeader`, call `QTSS_SendRTSPHeaders` (page 195) to send the entire header.

QTSS_CloseFileObject

Closes a file.

```
QTSS_Error QTSS_CloseFileObject(QTSS_Object inFileObject);
```

Parameters

`inFileObject`

> On input, a value of type `QTSS_Object` (page 207) that represents the file that is to be closed.

`result`

> A result code. Possible values include `QTSS_NoErr` and `QTSS_BadArgument` if a parameter is invalid.

Discussion

The `QTSS_CloseFileObject` callback routine closes the specified file.

QTSS_CreateObjectType

Creates an object type.

```
QTSS_Error QTSS_CreateObjectType(QTSS_ObjectType* outType);
```

Parameters

`outType`

> On input, a pointer to a value of type `QTSS_ObjectType` (page 208).

`result`

> A result code. Possible values are `QTSS_NoErr`, `QTSS_FailedRequest` too many object types already exist, and `QTSS_OutOfState` if `QTSS_CreateObjectType` an attribute of the specified name already exists.

Discussion

The `QTSS_CreateObjectType` callback routine creates a new object type and provides a pointer to it. Static attributes can be added to the object type by calling `QTSS_AddStaticAttribute` (page 173). Instance attributes can be added to instances of objects of the new object type.

The `QTSS_AddStaticAttribute` callback can only be called from the Register role. Call `QTSS_SetValue` (page 197) to set the value of an added attribute and `QTSS_RemoveValue` (page 193) to remove the value of an added attribute.

This callback may only be called from the Register role.

QTSS_CreateObjectValue

Creates a new object that is the value of another object's attribute.

```
QTSS_Error QTSS_CreateObjectValue(
    QTSS_Object inObject,
    QTSS_AttributeID inID,
    QTSS_ObjectType inType,
    UInt32* outIndex,
    QTSS_Object* outCreatedObject);
```

Parameters

`inObject`

On input, a pointer to a value of type `QTSS_ObjectType` (page 208) that specifies the object having an attribute whose value will be the created object.

`inID`

On input, a value of type `QTSS_AttributeID` (page 207) that specifies the attribute ID of the attribute whose value will be the created object.

`inType`

On input, a value of type `QTSS_ObjectType` (page 208) that specifies the object type of the object that is to be created.

`outIndex`

On output, a pointer to a value of type `UInt32` that contains the index of the created object.

`outCreatedObject`

On output, a pointer to a value of type `QTSS_ObjectType` (page 208)s that is the new object.

`result`

A result code. Possible values are `QTSS_NoErr`, `QTSS_BadArgument` if any parameter is invalid, and `QTSS_ReadOnly` if the attribute specified by `inID` is a read-only attribute.

Discussion

The `QTSS_CreateObjectValue` callback routine creates an object that is the value of an existing object's attribute. The object specified by inObject is the "parent" object.

If the object specified by inObject is later locked by calling `QTSS_LockObject` (page 187), the object pointed to by outCreatedObject is also locked.

QTSS_Delete

Deletes memory.

```
void* QTSS_Delete(void* inMemory);
```

Parameters

`inMemory`

On input, a pointer to an arbitrary value that specifies in bytes the amount of memory to be deleted.

`result`

None.

Discussion

The `QTSS_Delete` callback routine deletes memory that was previously allocated by `QTSS_New` (page 189).

QTSS_DoService

Invokes a service.

```
QTSS_Error QTSS_DoService(
    QTSS_ServiceID inID,
    QTSS_ServiceFunctionArgsPtr inArgs);
```

Parameters

`inID`

On input, a value of type `QTSS_ServiceID` (page 209) that specifies the service that is to be invoked. Call `QTSS_IDForAttr` (page 186) to get the service ID of the service you want to invoke.

`inArgs`

On input, a value of type `QTSS_ServiceFunctionArgsPtr` that points to the arguments that are to be passed to the service.

`result`

A result code returned by the service or `QTSS_IllegalService` if `inID` is invalid.

Discussion

The `QTSS_DoService` callback routine invokes the service specified by `inID`.

QTSS_Flush

Forces an immediate write operation.

```
QTSS_Error QTSS_Flush(QTSS_StreamRef inRef);
```

Parameters

`inRef`

On input, a value of type `QTSS_StreamRef` (page 209) that specifies the stream for which buffered data is to be written.

`result`

A result code. Possible values include `QTSS_NoErr`, `QTSS_BadArgument` if a parameter is `NULL`, and `QTSS_WouldBlock` if the stream cannot be flushed completely at this time.

Discussion

The `QTSS_Flush` callback routine forces the stream to immediately write any data that has been buffered. Some QTSS stream references, such as `QTSSRequestRef`, buffer data before sending it.

QTSS_GetAttrInfoByID

Uses an attribute ID to get information about an attribute.

```
QTSS_Error QTSS_GetAttrInfoByID(
    QTSS_Object inObject,
    QTSS_AttributeID inAttrID,
    QTSS_AttrInfoObject* outAttrInfoObject);
```

Parameters

`inObject`

On input, a value of type `QTSS_Object` (page 207) that specifies the object having the attribute for which information is to be obtained.

`inAttrID`

On input, a value of type `QTSS_AttributeID` (page 207) that specifies the attribute for which information is to be obtained.

outAttrInfoObject

> On output, a pointer to a value of type `QTSS_AttrInfoObject` that can be used to get information about the attribute specified by `inAttrID`.

result

> A result code. Possible values are `QTSS_NoErr`, `QTSS_BadArgument` if the specified object does not exist, and `QTSS_AttrDoesntExist` if the attribute doesn't exist.

Discussion

The `QTSS_GetAttrInfoByID` callback routine uses an attribute ID to get an `QTSS_AttrInfoObject` that can be used to get the attribute's name, data type, permissions for reading and writing the attribute's value, and whether getting the attribute's value is preemptive safe.

QTSS_GetAttrInfoByIndex

Gets information about all of an object's attributes by iteration.

```
QTSS_Error QTSS_GetAttrInfoByIndex(
    QTSS_Object inObject,
    UInt32 inIndex,
    QTSS_AttrInfoObject* outAttrInfoObject);
```

Parameters

inObject

> On input, a value of type `QTSS_Object` (page 207) that specifies the object having the attribute for which information is to be obtained.

inIndex

> On input, a value of type `UInt32` that specifies the index of the attribute for which information is to be obtained. Start by setting `inIndex` to zero. For the next call to `QTSS_GetAttrInfoByIndex`, increment `inIndex` by one to get information for the next attribute. Call `QTSS_GetNumAttributes` (page 182) to get the number of attributes that `inObject` has.

outAttrInfoObject

> On output, a pointer to a value of type `QTSS_AttrInfoObject` that can be used to get information about the attribute specified by `inAttrName`.

result

> A result code. Possible values are `QTSS_NoErr`, `QTSS_BadArgument` if the specified object does not exist, and `QTSS_AttrDoesntExist` if the attribute doesn't exist.

Discussion

The `QTSS_GetAttrInfoByIndex` callback routine uses an index to get an `QTSS_AttrInfoObject` that can be used to get the attribute's name and ID, data type, permissions for reading and write the attribute's value, and whether getting the attribute's value is preemptive safe

The `QTSS_GetAttrInfoByIndex` callback routine returns a `QTSS_AttrInfoObject` for both static and instance attributes.

QTSS_GetAttrInfoByName

Uses an attribute's name to get information about an attribute.

```
QTSS_Error QTSS_GetAttrInfoByName(
    QTSS_Object inObject,
    char* inAttrName,
    QTSS_AttrInfoObject* outAttrInfoObject);
```

Parameters

`inObject`

On input, a value of type `QTSS_Object` (page 207) that specifies the object having the attribute for which information is to be obtained.

`inAttrName`

On input, a pointer to a C string containing the name of the attribute for which information is to be obtained.

`outAttrInfoObject`

On output, a pointer to a value of type `QTSS_AttrInfoObject` that can be used to get information about the attribute specified by `inAttrName`.

`result`

A result code. Possible values are `QTSS_NoErr`, `QTSS_BadArgument` if the specified object does not exist, and `QTSS_AttrDoesntExist` if the attribute doesn't exist.

Discussion

The `QTSS_GetAttrInfoByName` callback routine uses an attribute name to get an `QTSS_AttrInfoObject` that can be used to get the attribute's ID, its data type, and permissions for reading and writing the attribute's value, and whether getting the attribute's value is preemptive safe.

The `QTSS_GetAttrInfoByName` callback routine returns a `QTSS_AttrInfoObject` for both static and instance attributes.

QTSS_GetNumAttributes

Gets a count of an object's attributes.

```
QTSS_Error QTSS_GetNumAttributes(
    QTSS_Object inObject,
    UInt32* outNumAttributes);
```

Parameters

`inObject`

On input, a value of type `QTSS_Object` (page 207) that specifies the object whose attributes are to be counted.

`outNumAttributes`

On output, a pointer to a value of type `UInt32` that contains the count of the object's attributes.

`result`

A result code. Possible values are `QTSS_NoErr` and `QTSS_BadArgument` if the specified object does not exist.

Discussion

The `QTSS_GetNumAttributes` callback routine gets the number of attributes for the object specified by `inObject`. Having the number of attributes lets you know how often to call `QTSS_GetAttrInfoByIndex` (page 180) when getting information about each of an object's attributes.

QTSS_GetValue

Copies the value of an attribute into a buffer.

```
QTSS_Error QTSS_GetValue (
    QTSS_Object inObject,
    QTSS_AttributeID inID,
    UInt32 inIndex,
    void* ioBuffer,
    UInt32* ioLen);
```

Parameters

`inObject`

On input, a value of type `QTSS_Object` (page 207) specifying the object that contains the attribute whose value is to be obtained.

`inID`

On input, a value of type `QTSS_AttributeID` (page 207) specifying the ID of the attribute whose value is to be obtained.

`inIndex`

On input, a value of type `UInt32` that specifies which attribute value to get (if the attribute can have multiple values) or zero for single-value attributes.

`ioBuffer`

On input, a pointer to a buffer. On output, the buffer pointed to by `ioBuffer` contains the value of the attribute specified by `inID`. If the buffer is too small to contain the value, the buffer is empty.

`ioLen`

On input, a pointer to a value of type `UInt32` specifying the length of the buffer pointed to by `ioBuffer`. On output, `ioLen` points to a value that is the length of the valid data in `ioBuffer`.

`result`

A result code. Possible values include `QTSS_NoErr`, `QTSS_BadArgument` if a parameter is invalid, `QTSS_BadIndex` if the index specified by `inIndex` does not exist, `QTSS_NotEnoughSpace` if the attribute value is longer than the value pointed to by `ioLen`, and `QTSS_AttrDoesntExist` if the attribute doesn't exist.

Discussion

The `QTSS_GetValue` callback routine copies the value of the specified attribute into the provided buffer.

Calling `QTSS_GetValue` is slower and less efficient than calling `QTSS_GetValuePtr` (page 184).

QTSS_GetValueAsString

Gets the value of an attribute as a C string.

```
QTSS_Error QTSS_GetValueAsString (
    QTSS_Object inObject,
    QTSS_AttributeID inID,
    UInt32 inIndex,
    char** outString);
```

Parameters

`inObject`

On input, a value of type `QTSS_Object` (page 207) specifying the object that contains the attribute whose value is to be obtained.

`inID`

On input, a value of type `QTSS_AttributeID` (page 207) specifying the ID of the attribute whose value is to be obtained.

`inIndex`

On input, a value of type `UInt32` specifying which attribute value to get (if the attribute can have multiple values) or zero for single-value attributes.

`outString`

On input, a pointer to an address in memory. On output, `outString` points to the value of the attribute specified by `inID` in string format.

`result`

A result code. Possible values include `QTSS_NoErr`, `QTSS_BadArgument` if a parameter is invalid, and `QTSS_BadIndex` if the index specified by `inIndex` does not exist.

Discussion

The `QTSS_GetValueAsString` callback routine gets the value of the specified attribute, converts it to C string format, and stores it at the location in memory pointed to by the `outString` parameter.

When you no longer need `outString`, call `QTSS_Delete` (page 178) to free the memory that has been allocated for it.

The `QTSS_GetValueAsString` callback routine can be called to get the value of preemptive safe attributes as well as attributes that are not preemptive safe. However, calling `QTSS_GetValueAsString` is less efficient than calling `QTSS_GetValue` (page 182), and calling `QTSS_GetValue` is less efficient than calling `QTSS_GetValuePtr` (page 184).

Calling `QTSS_GetValue` is the recommended way to get the value of an attribute that is not preemptive safe and calling `QTSS_GetValuePtr` is the recommended way to get the value of an attribute that is preemptive safe.

QTSS_GetValuePtr

Gets a pointer to an attribute's value.

```
QTSS_Error QTSS_GetValuePtr (
    QTSS_Object inObject,
    QTSS_AttributeID inID,
    UInt32 inIndex,
    void** outBuffer,
    UInt32* outLen);
```

Parameters

inObject

> On input, a value of type QTSS_Object (page 207) specifying the object containing the attribute whose value is to be obtained.

inID

> On input, a value of type QTSS_AttributeID (page 207) specifying the ID of an attribute.

inIndex

> On input, a value of type UInt32 specifying which attribute value to get (if the attribute can have multiple values) or zero for single-value attributes.

outBuffer

> On input, a pointer to an address in memory. On output, outBuffer points to the value of the attribute.

outLen

> On output, a pointer to a value of type UInt32 specifying the number of valid bytes pointed to by outBuffer.

result

> A result code. Possible values include QTSS_NoErr, QTSS_NotPreemptiveSafe if inID is an attribute that is not preemptive safe, QTSS_BadArgument if a parameter is invalid, QTSS_BadIndex if the index specified by inIndex does not exist, and QTSS_AttrDoesntExist if the attribute doesn't exist.

Discussion

The QTSS_GetValuePtr callback routine gets a pointer to an attribute's value. Calling QTSS_GetValuePtr is the fastest and most efficient way to get the value of an attribute, and it is less likely to generate an error.

Before calling QTSS_GetValuePtr to get the value of an attribute that is not preemptive safe, you must lock the object by calling QTSS_LockObject (page 187). After getting the value, unlock the object by calling QTSS_UnLockObject (page 203).

If you don't want to lock and unlock the object to get the value of an attribute that is not preemptive safe, get the value by calling QTSS_GetValue (page 182) or QTSS_GetValueAsString (page 183).

QTSS_IDForAttr

Gets the ID of a static attribute.

```
QTSS_Error QTSS_IDForAttr(
    QTSS_ObjectType inType,
    const char* inAttributeName,
    QTSS_AttributeID* outID);
```

Parameters

`inType`

> On input, a value of type `QTSS_ObjectType` (page 208) specifying the type of object for which the ID is to be obtained.

`inAttributeName`

> On input, a pointer to a byte array specifying the name of the attribute whose ID is to be obtained.

`outID`

> On input, a pointer to a value of type `QTSS_AttributeID` (page 207). On output, `outID` points to the ID of the attribute specified by `inAttributeName`.

`result`

> A result code. Possible values are `QTSS_NoErr` and `QTSS_BadArgument` if a parameter is invalid.

Discussion

The `QTSS_IDForAttr` callback routine obtains the attribute ID for the specified static attribute in the specified object type. The attribute ID is used to when calling `QTSS_GetValue` (page 182), `QTSS_GetValueAsString` (page 183), and `QTSS_GetValuePtr` (page 184) get the attribute's value.

To get the ID of an instance attribute, call `QTSS_GetAttrInfoByName` (page 181) or `QTSS_GetAttrInfoByIndex` (page 180).

QTSS_IDForService

Resolves a service name to a service ID.

```
QTSS_Error QTSS_IDForService(
```

```
    const char* inTag,
    QTSS_ServiceID* outID);
```

Parameters

`inTag`

> On input, a pointer to a string containing the name of the service that is to be resolved.

`outID`

> On input, a pointer to a value of type `QTSS_ServiceID` (page 209). On output, `QTSS_ServiceID` contains the ID of the service specified by `inTag`.

`result`

> A result code. Possible values are `QTSS_NoErr` and `QTSS_BadArgument` if a parameter is invalid.

Discussion

The `QTSS_IDForService` callback routine returns in the `outID` parameter the service ID of the service specified by the `inTag` parameter. You can use the service ID to call `QTSS_DoService` (page 178) to invoke the service that `serviceID` represents.

QTSS_LockObject

Locks an object.

```
QTSS_Error QTSS_LockObject(QTSS_Object inObject);
```

Parameters

`inObject`

> On input, a value of type `QTSS_Object` (page 207) that specifies the object that is to be locked.

`result`

> A result code. Possible values are `QTSS_NoErr` and `QTSS_BadArgument` if the specified object instance does not exist.

Discussion

The `QTSS_LockObject` callback routine locks the specified object so that accesses to the object's attributes from other threads will block. Call `QTSS_LockObject` before performing non-atomic updates on a variable that is pointed to by an attribute—as set by calling `QTSS_SetValuePtr` (page 198)—or before getting the value of a non-preemptive safe attribute.

Call `QTSS_UnLockObject` (page 203) to unlock the object.

Objects created by `QTSS_CreateObjectValue` (page 177) are locked when the parent object is locked.

QTSS_Milliseconds

Gets the current value of the server's internal clock.

```
QTSS_TimeVal QTSS_Milliseconds();
```

Parameters

`result`

> The value of the server's internal clock in milliseconds since midnight January 1, 1970.

Discussion

The `QTSS_Milliseconds` callback routine gets the current value of the server's internal clock since midnight January 1, 1970. Unless otherwise noted, all millisecond values that the server provides in attributes are obtained from this clock.

QTSS_MilliSecsTo1970Secs

Converts a value obtained from the server's internal clock to the current time.

```
time_t QTSS_MilliSecsTo1970Secs(QTSS_TimeVal inQTSS_Milliseconds);
```

Parameters

`inQTSS_Milliseconds`

> On input, a value of type `QTSS_TimeVal` obtained by calling `QTSS_Milliseconds()`.

`result`

> A value of type `time_t` containing the current time.

Discussion

The `QTSS_MilliSecsto1970Secs` callback routine converts a value obtained by calling `QTSS_Milliseconds` (page 188) to the current time.

QTSS_New

Allocates memory.

```
void* QTSS_New(
    FourCharCode inMemoryIdentifier,
    UInt32 inSize);
```

Parameters

`inMemoryIdentifier`

On input, a value of type `FourCharCode` that will be associated with this memory allocation. The server can track the allocated memory to make debugging memory leaks easier.

`inSize`

On input, a value of type `UInt32` that specifies in bytes the amount of memory to be allocated.

`result`

None.

Discussion

The `QTSS_New` callback routine allocates memory. QTSS modules should call `QTSS_New` whenever it needs to allocate memory dynamically.

To delete the memory that `QTSS_New` allocates, call `QTSS_Delete` (page 178).

QTSS_OpenFileObject

Opens a file.

```
QTSS_Error QTSS_OpenFileObject(
    char* inPath,
    QTSS_OpenFileFlags inFlags,
    QTSS_Object* outFileObject);
```

Parameters

`inPath`

On input, a pointer to a null-terminated C string containing the full path to the file in the local file system that is to be opened.

`inFlags`

On input, a value of type `QTSS_OpenFileFlags` (page 214) specifying flags that describe how the file is to be opened.

`outFileObject`

On output, a pointer to a value of type `QTSS_Object` (page 207) in which the file object for the opened file is to be placed.

`result`

A result code. Possible values include `QTSS_NoErr`, `QTSS_BadArgument` if a parameter is invalid, and `QTSS_FileNotFound` if the specified file does not exist.

Discussion

The `QTSS_OpenFileObject` callback routine opens the specified file and returns a file object for it. One of the attributes of the file object is a stream reference that is passed to QTSS stream callback routines to read and write data to the file and to perform other file operations.

QTSS_Pause

Pauses a stream that is playing.

```
QTSS_Error QTSS_Pause(QTSS_ClientSessionObject inClientSession);
```

Parameters

`inClientSession`

On input, a value of type `QTSS_ClientSessionObject` that identifies the client session that is to be paused.

`result`

A result code. Possible values are `QTSS_NoErr` and `QTSS_BadArgument` if a parameter is invalid.

Discussion

The `QTSS_Pause` callback routine pauses playing for a stream. The module that called `QTSS_AddRTPStream` (page 171) is the only module that can call `QTSS_Pause`.

QTSS_Play

Starts playing streams associated with a client session.

```
QTSS_Error QTSS_Play(
    QTSS_ClientSessionObject inClientSession,
    QTSS_RTSPRequestObject inRTSPRequest,
    QTSS_PlayFlags inPlayFlags);
```

Parameters

`inClientSession`

> On input, a value of type QTSS_ClientSessionObject that identifies the client session for which the sending of RTP packets was enabled by previously calling `QTSS_AddRTPStream` (page 171).

`inRTSPRequest`

> On input, a value of type `QTSS_RequestObject`.

`inPlayFlags`

> On input, a value of type `QTSS_PlayFlags`. Set `inPlayFlags` to the constant `qtssPlaySendRTCP` to cause the server to generate RTCP sender reports automatically while playing. Otherwise, the module is responsible for generating sender reports that specify play characteristics.

`result`

> A result code. Possible values are `QTSS_NoErr` and `QTSS_BadArgument` if a parameter is invalid, and `QTSS_RequestFailed` if no streams have been added to the session.

Discussion

The `QTSS_Play` callback routine starts playing streams associated with the specified client session.

The module that called `QTSS_AddRTPStream` (page 171) is the only module that can call `QTSS_Play`.

Before calling `QTSS_Play`, the module should set the following attributes of the `QTSS_RTPStreamObject` object for this RTP stream:

- `qtssRTPStrFirstSeqNumber`, which should be set to the sequence number of the first packet after the last PLAY request was issued. The server uses the sequence number to generate a proper RTSP PLAY response.

- `qtssRTPStrFirstTimestamp`, which should be set to the timestamp of the first RTP packet generated for this stream after the last PLAY request was issued. The server uses the timestamp to generate a proper RTSP PLAY response.

- `qtssRTPStrTimescale`, which should be set to the timescale for the track.

After calling `QTSS_Play`, the module is invoked in the RTP Send Packets role.

Call `QTSS_Pause` (page 190) to pause playing or call `QTSS_Teardown` (page 201) to close the client session.

QTSS_Read

Reads data from a stream.

```
QTSS_Error QTSS_Read(
    QTSS_StreamRef inRef,
    void* ioBuffer,
    UInt32 inBufLen,
    UInt32* outLengthRead);
```

Parameters

`inRef`

> On input, a value of type `QTSS_StreamRef` (page 209) that specifies the stream from which data is to be read. Call `QTSS_OpenFileObject` to obtain a stream reference for the file you want to read.

`ioBuffer`

> On input, a pointer to a buffer in which data that is read is to be placed.

`inBufLen`

> On input, a value of type `UInt32` that specifies the length of the buffer pointed to by `ioBuffer`.

`outLenRead`

> On output, a pointer to a value of type `UInt32` that contains the number of bytes that were read.

`result`

> A result code. Possible values include `QTSS_NoErr`, `QTSS_BadArgument` if a parameter is invalid, `QTSS_WouldBlock` if the read operation would block, or `QTSS_RequestFailed` if the read operation failed.

Discussion

The `QTSS_Read` callback routine reads a buffer of data from a stream.

QTSS_RemoveInstanceAttribute

Remove an instance attribute from the instance of an object.

```
QTSS_Error QTSS_RemoveInstanceAttribute(
```

```
   QTSS_Object inObject,
   QTSS_AttributeID inID);
```

Parameters

`inObject`

On input, a value of type `QTSS_Object` (page 207) that specifies the object from which the instance attribute is to be removed.

`inID`

On input, a value of type `QTSS_AttributeID` (page 207) that specifies the ID of the attribute that is to be removed.

`result`

A result code. Possible values are `QTSS_NoErr`, `QTSS_BadArgument` if the specified object instance does not exist, and `QTSS_AttrDoesntExist` if the attribute doesn't exist.

Discussion

The `QTSS_RemoveInstanceAttribute` callback routine removes the attribute specified by the `inID` parameter from the instance of an object specified by the `inObject` parameter.

The `QTSS_RemoveInstanceAttribute` callback can be called from any role.

QTSS_RemoveValue

Removes the specified value from an attribute.

```
QTSS_Error QTSS_RemoveValue (
   QTSS_Object inObject,
   QTSS_AttributeID inID,
   UInt32 inIndex);
```

Parameters

`inObject`

On input, a value of type `QTSS_Object` (page 207) having an attribute whose value is to be removed.

`inValueLen`

On input, a value of type `QTSS_AttributeID` (page 207) containing the attribute ID of the attribute whose value is to be removed.

`inIndex`

On input, a value of type `UInt32` that specifies the attribute value that is to be removed. Attribute value indexes are numbered starting from zero.

`result`

A result code. Possible values include `QTSS_NoErr`, `QTSS_BadArgument` if `InObject`, `inID`, or `inIndex` do not contain valid values, `QTSS_ReadOnly` if the attribute is read-only, and `QTSS_BadIndex` if the specified index does not exist.

Discussion

The `QTSS_RemoveValue` callback routine removes the value of the specified attribute. After the value is removed, the attribute values are renumbered.

QTSS_RequestEvent

Requests notification of specified events.

```
QTSS_Error QTSS_RequestEvent(
    QTSS_StreamRef inStream,
    QTSS_EventType inEventMask);
```

Parameters

`inStream`

On input, a value of type `QTSS_StreamRef` (page 209) that specifies the stream for which event notifications are requested.

`inEventMask`

On input, a value of type `QTSS_EventType` (page 213) specifying a mask that represents the events for which notifications are requested.

`result`

A result code. Possible values include `QTSS_NoErr`, `QTSS_BadArgument` if a parameter is invalid, and `QTSS_RequestFailed` if the call failed.

Discussion

The `QTSS_RequestEvent` callback requests that the caller be notified when the specified events occur on the specified stream. After calling `QTSS_RequestEvent`, the calling module should return as soon as possible from its current module role. The server preserves the calling module's current state and, when the event occurs, calls the module in the role the module was in when it called `QTSS_RequestEvent`.

QTSS_Seek

Sets the position of a stream.

```
QTSS_Error QTSS_Seek(
    QTSS_StreamRef inRef,
    UInt64 inNewPosition);
```

Parameters

inRef

On input, a value of type QTSS_StreamRef (page 209) QTSS_StreamRef that specifies the stream whose position is to be set. Call QTSS_OpenFileObject to obtain stream reference.

inNewPosition

On input, the offset in bytes from the start of the stream to which the position is to be set.

result

A result code. Possible values include QTSS_NoErr, QTSS_BadArgument if a parameter is invalid, and QTSS_RequestFailed if the seek operation failed.

Discussion

The QTSS_Seek callback routine sets the stream position to the value specified by inNewPosition.

QTSS_SendRTSPHeaders

Sends an RTSP header.

```
QTSS_Error QTSS_SendRTSPHeaders(QTSS_RTSPRequestOjbect inRef);
```

Parameters

inRef

On input, a value of type QTSS_RTSPRequestObject for the RTSP stream.

result

A result code. Possible values are QTSS_NoErr and QTSS_BadArgument if a parameter is invalid.

Discussion

The QTSS_SendRTSPHeaders callback routine sends an RTSP header. When a module calls QTSS_SendRTSPHeaders, the server sends a proper RTSP status line, using the request's current status code. The server also sends the proper CSeq header, session ID header, and connection header.

QTSS_SendStandardRTSPResponse

Sends an RTSP response to a client.

```
QTSS_Error QTSS_SendStandardRTSPResponse(
    QTSS_RTSPRequestObject inRTSPRequest,
    QTSS_Object inRTPInfo,
    UInt32 inFlags);
```

Parameters

inRTSPRequest

On input, a value of type QTSS_RTSPRequestObject for the RTSP stream.

inRTPInfo

On input, a value of type QTSS_Object (page 207). This parameter is a QTSS_ClientSessionObject or a QTSS_RTPStreamObject, depending the response that is sent.

inFlags

On input, a value of type UInt32. Set inFlags to qtssPlayRespWriteTrackInfo if you want the server to append the seq number, a timestamp, and SSRC information to RTP-Info headers.

result

A result code. Possible values include QTSS_NoErr and QTSS_BadArgument if a parameter is invalid.

Discussion

The QTSS_SendStandardRTSPResponse callback routine writes a standard response to the stream specified by the inRTSPRequest parameter. The actual response that is sent depends on the method.

The following enumeration defines the qtssPlayRespWriteTrackInfo constant for the inFlags parameter:

```
enum
{
    qtssPlayRespWriteTrackInfo = 0x00000001
};
```

This function supports the following response methods:

- DESCRIBE. This response method writes status line, CSeq, SessionID, Connection headers as determined by the request. Writes a Content-Base header with the content base being the URL provided. Writes a Content-Type header of `application/sdp`. The `inRTPInfo` parameter must be a `QTSS_ClientSessionObject`.

- ANNOUNCE. This response method writes status line, CSeq, and Connection headers as determined by the request. The `inRTPInfo` parameter must be a `QTSS_ClientSessionObject`.

- SETUP. This response method writes status line, CSeq, SessionID, Connection headers as determined by the request. Writes a Transport header with client and server ports (if the connection is over UDP). The `inRTPInfo` parameter must be a `QTSS_RTPStreamObject`.

- PLAY. This response method writes status line, CSeq, SessionID, Connection headers as determined by the request. The `inRTPInfo` parameter must be a `QTSS_ClientSessionObject`. Set the `inFlags` parameter to `qtssPlayRespWriteTrackInfo` to specify that you want the server to append the sequence number, timestamp, and SSRC information to the RTP-Info header.

- PAUSE. This response method writes status line, CSeq, SessionID, Connection headers as determined by the request. The `inRTPInfo` parameter must be a `QTSS_ClientSessionObject`.

- TEARDOWN. This response method writes status line, CSeq, SessionID, Connection headers as determined by the request. The `inRTPInfo` parameter must be a `QTSS_ClientSessionObject`.

QTSS_SetValue

Sets the value of an attribute.

```
QTSS_Error QTSS_SetValue (
    QTSS_Object inObject,
    QTSS_AttributeID inID,
    UInt32 inIndex,
    const void* inBuffer,
    UInt32 inLen);
```

Parameters

`inObject`

On input, a value of type `QTSS_Object` (page 207) that specifies the object containing the attribute whose value is to be set.

`inID`

On input, a value of type `QTSS_AttributeID` (page 207) that specifies the ID of the attribute whose value is to be set.

`inIndex`

On input, a value of type `UInt32` that specifies which attribute value to set (if the attribute can have multiple values) or zero for single-value attributes.

`inBuffer`

On input, a pointer to a buffer containing the value that is to be set. When `QTSS_SetValue` returns, you can dispose of `inBuffer`.

`inLen`

On input, a pointer to a value of type `UInt32` that specifies the length of valid data in `inBuffer`.

`result`

A result code. Possible values are `QTSS_NoErr`, `QTSS_BadIndex` if the index specified by `inIndex` does not exist, `QTSS_BadArgument` if a parameter is invalid, `QTSS_ReadOnly` if the attribute is read-only, and `QTSS_AttrDoesntExist` if the attribute doesn't exist.

Discussion

The `QTSS_SetValue` callback routine explicitly sets the value of the specified attribute. Another way to set the value of an attribute is to call `QTSS_SetValuePtr` (page 198).

QTSS_SetValuePtr

Sets an existing variable as the value of an attribute.

```
QTSS_Error QTSS_SetValue (
    QTSS_Object inObject,
    QTSS_AttributeID inID,
    const void* inBuffer,
    UInt32 inLen);
```

Parameters

`inObject`

On input, a value of type `QTSS_Object` (page 207) that specifies the object containing the attribute whose value is to be set.

`inID`

On input, a value of type `QTSS_AttributeID` (page 207) that specifies the ID of the attribute whose value is to be set.

`inBuffer`

On input, a pointer to a buffer containing the value that is to be set.

`inLen`

On input, a pointer to a value of type `UInt32` that specifies the length of valid data in `inBuffer`.

`result`

A result code. Possible values are `QTSS_NoErr`, `QTSS_BadArgument` if a parameter is invalid, and `QTSS_ReadOnly` if the attribute is a read-only attribute.

Discussion

The `QTSS_SetValuePtr` callback routine allows modules to set an attribute that its value is the value of a module's variable. This callback is an alternative to the `QTSS_SetValue` (page 197) callback.

After calling `QTSS_SetValuePtr`, the module must insure that the buffer pointed to by `inBuffer` exists as long as the attribute specified by `inID` exists.

If the buffer pointed to by `inBuffer` is not updated atomically, updating the value of inBuffer should be protected by calling `QTSS_LockObject` (page 187) before an update.callback

QTSS_SignalStream

Notifies the recipient of events that a stream has become available for I/O.

```
QTSS_Error QTSS_RequestEvent(
    QTSS_StreamRef inStream,
    QTSS_EventType inEventMask);
```

Parameters

`inStream`

On input, a value of type `QTSS_StreamRef` (page 209) specifying the stream that has become available for I/O.

`inEventMask`

On input, a value of type `QTSS_EventType` (page 213) containing a mask that represents whether the stream has become available for reading, writing, or both.

```
result
```

A result code. Possible values include `QTSS_NoErr`, `QTSS_BadArgument` if a parameter is invalid, `QTSS_OutOfState` if this callback is made from a role that does not allow asynchronous events, and `QTSS_RequestFailed` if the call failed.

Discussion

The `QTSS_SignalStream` callback routine tells the server that the stream represented by inStream has become available for I/O. Currently only file system modules have reason to call `QTSS_SignalStream`.

QTSS_StringToValue

Converts an attribute data type in C string format to a value in `QTSS_AttrDataType` format.

```
QTSS_Error QTSS_StringToValue(
    const char* inValueAsString,
    const QTSS_AttrDataType inType,
    void* ioBuffer,
    UInt32* ioBufSize);
```

Parameters

```
inValueAsString
```

On input, a pointer to a character array containing the value that is to be converted.

```
inType
```

On input, a value of type `QTSS_AttrDataType` (page 211) that specifies the attribute data type to which the value pointed to by `inValueAsString` is to be converted.

```
ioBuffer
```

On input, a pointer to a buffer. On output, the buffer contains the attribute data type to which inValueAsString has been converted. The calling module must allocate `ioBuffer` before calling `QTSS_StringToValue`.

```
ioBufSize
```

On input, a pointer to a value of type `UInt32 that specifies the length of the buffer pointed to by ioBuffer.` On output, `ioBufSize` points to the length of data in `ioBuffer`.

```
result
```

A result code. Possible values are `QTSS_NoErr`, `QTSS_BadArgument` if `inValueAsString` or `inType` do not contain valid values, and `QTSS_NotEnoughSpace` if the buffer pointed to by `ioBuffer` is too small to contain the converted value.

Discussion

The QTSS_StringToValue callback routine converts an attribute data type that is in C string format to a value that is in QTSS_AttrDataType format.

When the memory allocated for the buffer pointed to by ioBuffer is no longer needed, you should deallocate the memory.

QTSS_Teardown

Closes a client session.

```
QTSS_Error QTSS_Teardown(QTSS_ClientSessionObject inClientSession);
```

Parameters

inClientSession

On input, a value of type QTSS_ClientSessionObject that identifies the client session that is to be closed.

result

A result code. Possible values are QTSS_NoErr and QTSS_BadArgument if a parameter is invalid.

Discussion

The QTSS_Teardown callback routine closes a client session.

The module that called QTSS_AddRTPStream (page 171) is the only module that can call QTSS_Teardown.

Calling QTSS_Teardown causes the calling module to be invoked in the Client Session Closing role for the session identified by the inClientSession parameter.

QTSS_TypeStringToType

Gets the attribute data type of a data type string that is in C string format.

```
QTSS_Error QTSS_TypeStringToType(
    const char* inTypeString,
    QTSS_AttrDataType* outType);
```

201

Parameters

`inTypeString`

On input, a pointer to a character array containing the attribute data type in C string format.

`outType`

On output, a pointer to a value of type QTSS_AttrDataType (page 211) containing the attribute data type.

`result`

A result code. Possible values are `QTSS_NoErr` and `QTSS_BadArgument` if `inTypeString` does not contain a value for which an attribute data type can be returned.

Discussion

The `QTSS_TypeStringToType` callback routine gets the attribute data type of a data type string that is in C string format.

QTSS_TypeToTypeString

Gets the name in C string format of an attribute data type.

```
QTSS_Error QTSS_TypeToTypeString(
    const QTSS_AttrDataType inType,
    char** outTypeString);
```

Parameters

`inType`

On input, a pointer to a value of type QTSS_AttrDataType (page 211) containing the attribute data type that is to be returned in C string format.

`outType`

On input, a pointer to an address in memory. On output, `outType` points to a C string containing the attribute data type.

`result`

A result code. Possible values are `QTSS_NoErr` and `QTSS_BadArgument` if `inType` does not contain a valid attribute data type.

Discussion

The `QTSS_TypeToTypeString` callback routine gets the name in C string format of a value that is in `QTSS_AttrDataType` format.

QTSS_UnLockObject

Unlocks an object.

```
QTSS_Error QTSS_UnLockObject(QTSS_Object inObject);
```

Parameters

`inObject`

On input, a value of type `QTSS_Object` (page 207) that is to be unlocked.

`result`

A result code. Possible values are `QTSS_NoErr` and `QTSS_BadArgument` if the specified object is not a valid object.

Discussion

The `QTSS_UnLockObject` callback routine unlocks an object that was previously locked by `QTSS_LockObject` (page 187).

QTSS_ValueToString

Converts an attribute data type in `QTSS_AttrDataType` format to a value in C string format.

```
QTSS_Error QTSS_ValueToString(
    const void* inValue,
    const UInt32 inValueLen,
    const QTSS_AttrDataType inType,
    char** outString);
```

Parameters

`inValue`

On input, a pointer to a buffer containing the value that is to be converted from `QTSS_AttrDataType` format.

`inValueLen`

On input, a value of type `UInt32` that specifies the length of the value pointed to by `inValue`.

inType

On input, a value of type QTSS_AttrDataType (page 211) that specifies the attribute data type of the value pointed by inValue.

outString

On output, a pointer to a location in memory containing the attribute data type in C string format.

result

A result code. Possible values are QTSS_NoErr and QTSS_BadArgument if inValue, inValueLen, or inType do not contain valid values.

Discussion

The QTSS_ValueToString callback routine converts an attribute data type in QTSS_AttrDataType format to a value in C string format.

QTSS_Write

Writes data to a stream.

```
QTSS_Error QTSS_Write(
    QTSS_StreamRef inRef,
    void* inBuffer,
    UInt32 inLen,
    UInt32* outLenWritten,
    UInt32 inFlags);
```

Parameters

inRef

On input, a value of type QTSS_StreamRef (page 209) that specifies the stream to which data is to be written.

inBuffer

On input, a pointer to a buffer containing the data that is to be written.

inLen

On input, a value of type UInt32 that specifies the length of the data in the buffer pointed to by ioBuffer.

outLenWritten

On output, a pointer to a value of type UInt32 that contains the number of bytes that were written.

`inFlags`

On input, a value of type `UInt32`. See the Discussion section for possible values.

`result`

A result code. Possible values include `QTSS_NoErr`, `QTSS_BadArgument` if a parameter is invalid, `QTSS_NotConnected` if the stream receiver is no longer connected, and `QTSS_WouldBlock` if the stream cannot be completely flushed at this time.

Discussion

The `QTSS_Write` callback routine writes a buffer of data to a stream.

The following enumeration defines constants for the `inFlags` parameter:

```
enum
{
    qtssWriteFlagsIsRTP = 0x00000001,
    qtssWriteFlagsIsRTCP= 0x00000002
};
```

These flags are relevant when writing to an RTP stream reference and tell the server whether the data written should be sent over the RTP channel (`qtssWriteFlagsIsRTP`) or over the RTCP channel of the specified RTP stream (`qtssWriteFlagsIsRTCP`).

QTSS_WriteV

Writes data to a stream using an iovec structure.

```
QTSS_Error QTSS_WriteV(
    QTSS_StreamRef inRef,
    iovec* inVec,
    UInt32 inNumVectors,
    UInt32 inTotalLength,
    UInt32* outLenWritten);
```

Parameters

`inRef`

On input, a value of type `QTSS_StreamRef` (page 209) that specifies the stream to which data is to be written.

inVec

On input, a pointer to an `iovec` structure. The first member of the `iovec` structure must be empty.

inNumVectors

On input, a value of type `UInt32` that specifies the number of vectors.

inTotalLength

On input, a value of type `UInt32` specifying the total length of `inVec`.

outLenWritten

On output, a pointer to a value of type `UInt32` containing the number of bytes that were written.

result

A result code. Possible values include `QTSS_NoErr`, `QTSS_BadArgument` if a parameter is `NULL`, and `QTSS_WouldBlock` if the write operation would block.

Discussion

The `QTSS_WriteV` callback routine writes a data to a stream using an `iovec` structure in a way that is similar to the POSIX `writev` call.

QTSS Data Types

QTSS_AttributeID

A *QTSS_AttributeID is a signed 32-bit integer that uniquely identifies an attribute.*

```
typedef SInt32 QTSS_AttributeID;
```

QTSS_Object

A *QTSS_Object is a pointer to a value that identifies a particular object. The QTSS_Object is defined as*

```
typedef void* QTSS_Object;
```

Discussion

The QTSS_Object is used to define other QTSS objects:

```
typedef QTSS_Object QTSS_RTPStreamObject;
typedef QTSS_Object QTSS_RTSPSessionObject;
typedef QTSS_Object QTSS_RTSPRequestObject;
typedef QTSS_Object QTSS_RTSPHeaderObject;
typedef QTSS_Object QTSS_ClientSessionObject;
typedef QTSS_Object QTSS_ConnectedUserObject;
typedef QTSS_Object QTSS_ServerObject;
typedef QTSS_Object QTSS_PrefsObject;
typedef QTSS_Object QTSS_TextMessagesObject;
typedef QTSS_Object QTSS_FileObject;
typedef QTSS_Object QTSS_ModuleObject;
typedef QTSS_Object QTSS_ModulePrefsObject;
typedef QTSS_Object QTSS_AttrInfoObject;
typedef QTSS_Object QTSS_UserProfileObject;
```

QTSS_ObjectType

A `QTSS_ObjectType` is a value of type `UInt32` that identifies a particular QTSS object type.

```
typedef UInt32 QTSS_ObjectType;
```

Discussion

Constants for the following QTSS object types are defined:

- `qtssAttrInfoObjectType` — The attribute information object type. Objects of this type have attributes that describe an attribute.

- `qtssClientSessionObjectType` — The client session object type. Objects of this type have attributes that describe a client session.

- `qtssConnectedUsderObjectType` — The connected user object type. Objects of this type have attributes that described connections other than those described by qtssClientSessionObjectType objects.

- `qtssFileObjectType` — The file object type. Objects of this type have attributes that describe an open file.

- `qtssModuleObjectType` — The module object type. Objects of this type have attributes that describe a QTSS module.

- `qtssModulePrefsObjectType` — The module preferences object type. Objects of this type have attributes that describe module preferences.

- `qtssPrefsObjectType` — The preferences object type. Objects of this type have attributes that describe the server's preferences.

- `qtssRTPStreamObjectType` — The RTPS stream object type. Objects of this type have attributes that describe an RTP stream.

- `qtssRTSPHeaderObjectType` — The RTSP header object type. Objects of this type have attributes that contain all of the RTSP headers associated with an individual RTSP request.

- `qtssRTSPRequestObjectTYPE` — The RTSP request object type. Objects of this type have attributes that describe a particular RTSP request.

- `qtssRTSPSessionObjectType` — The RTSP session object type. Objects of this type have attributes that describe an RTSP client-server connection.

- `qtssServerObjectType` — The server object type. Objects of this type have attributes that contain global server information, such as server statistics.

- `qtssTextMessageOjbectType` — The text messages object type. Objects of this type have attributes that contain messages intended for display to the user.

- `qtssUserProfileObjectType` — The user profile object type. Objects of this type have attributes that contain information about a user, such as name, password, the groups the user is a member of, and the user's authentication realm.

QTSS_Role

A value of type `QTSS_Role` is an unsigned 32-bit integer used to store module roles. It is defined as

```
typedef UInt32 QTSS_Role;
```

QTSS_ServiceID

A `QTSS_ServiceID` is a signed 32-bit integer that uniquely identifies a service. It is defined as

```
typedef SInt32 QTSS_ServiceID;
```

QTSS_StreamRef

A value of type `QTSS_StreamRef` is a pointer to a value that identifies a particular stream. It is defined as

```
typedef void* QTSS_StreamRef;
```

Discussion
The `QTSS_StreamRef` is used to define other stream references:

```
typedef QTSS_StreamRef  QTSS_ErrorLogStream;
typedef QTSS_StreamRef  QTSS_FileStream;
typedef QTSS_StreamRef  QTSS_RTSPSessionStream;
typedef QTSS_StreamRef  QTSS_RTSPRequestStream;
typedef QTSS_StreamRef  QTSS_RTPStreamStream;
typedef QTSS_StreamRef  QTSS_SocketStr
```

QTSS_TimeVal

A value of type `QTSS_TimeVal` is a signed 64-bit integer used to store time values. It is defined as

```
typedef SInt64 QTSS_TimeVal;
```

QTSS Constants

QTSS_AttrDataType

Each QTSS attribute has an associated data type. The `QTSS_AttrDataType` *enumeration defines values for attribute data types. Having an attribute's data type helps the server and modules handle an attribute value without having specific knowledge about the attribute.*

```
typedef UInt32 QTSS_AttrDataType;
enum
{
    qtssAttrDataTypeUnknown     = 0,
    qtssAttrDataTypeCharArray   = 1,
    qtssAttrDataTypeBool16      = 2,
    qtssAttrDataTypeSInt16      = 3,
    qtssAttrDataTypeUInt16      = 4,
    qtssAttrDataTypeSInt32      = 5,
    qtssAttrDataTypeUInt32      = 6,
    qtssAttrDataTypeSInt64      = 7,
    qtssAttrDataTypeUInt64      = 8,
    qtssAttrDataTypeQTSS_Object = 9,
    qtssAttrDataTypeQTSS_StreamRef= 10,
    qtssAttrDataTypeFloat32     = 11,
    qtssAttrDataTypeFloat64     = 12,
    qtssAttrDataTypeVoidPointer = 13,
    qtssAttrDataTypeTimeVal     = 14,
    qtssAttrDataTypeNumTypes    = 15
};
```

Constants

qtssAttrDataTypeUnknown

qtssAttrDataTypeCharArray

qtssAttrDataTypeBool16

qtssAttrDataTypeSInt16

qtssAttrDataTypeUInt16

qtssAttrDataTypeSInt32

qtssAttrDataTypeUInt32

qtssAttrDataTypeSInt64

qtssAttrDataTypeQTSS_Object

qtssAttrDataTypeQTSS_StreamRef

qtssAttrDataTypeFloat32

qtssAttrDataTypeFloat64

qtssAttrDataTypeVoidPointer

qtssAttrDataTypeTimeVal

qtssAttrDataTypeNumTypes

QTSS_AttrPermission

The `QTSS_AttrPermission` *data type is an enumeration that defines values used to indicate whether an attribute is readable, writable, or preemptive safe. The data type of the* `qtssAttrPermissions` *attribute of the* `QTSS_AttrInfoObject` *object type is of type* `QTSS_AttrPermission`.

```
typedef UInt32 QTSS_AttrPermission;
enum
{
    qtssAttrModeRead    = 1,
    qtssAttrModeWrite   = 2,
    qtssAttrModePreempSafe= 4
};
```

Constants

qtssAttrModeRead

qtssAttrModeWrite

qtssAttrModePrempSafe

Discussion

Once set, attribute permissions cannot be changed.

QTSS_AddStreamFlags

The `QTSS_AddStreamFlags` enumeration defines flags that specify stream options when adding RTP streams.

```
enum
{
    qtssASFlagsAllowDestination    = 0x00000001,
    qtssASFlagsForceInterleave     = 0x00000002
};
typedef UInt32 QTSS_AddStreamFlags;
```

Constants

qtssASFlagsAllowDestination

qtssASFlagsForceInterleave

QTSS_CliSesTeardownReason

The `QTSS_CliSesTeardownReason` enumeration defines values that describe why a session is closing. The `QTSS_RTPSessionState` enumeration is defined as

```
enum
{
    qtssCliSesTearDownClientRequest = 0,
    qtssCliSesTearDownUnsupportedMedia = 1,
    qtssCliSesTearDownServerShutdown = 2,
    qtssCliSesTearDownServerInternalErr = 3
};
typedef UInt32 QTSS_CliSesTeardownReason;
```

Constants

qtssCliSesTearDownClientRequest

qtssCliSesTearDownUnsupportedMedia

qtssCliSesTearDownServerShutdown

qtssCliSesTearDownServerInternalErr

QTSS_EventType

A `QTSS_EventType` is an unsigned 32-bit integer whose value uniquely identifies stream I/O events.

```
enum
{
```

```
    QTSS_ReadableEvent              = 1,
    QTSS_WriteableEvent             = 2
};
typedef UInt32 QTSS_EventType;
```

Constants

QTSS_ReadableEvent

QTSS_WriteableEvent

QTSS_OpenFileFlags

A `QTSS_OpenFileFlags` is an unsigned 32-bit integer whose value describes how a file is to be opened.

```
enum
{
    qtssOpenFileNoFlags = 0,
    qtssOpenFileAsync   = 1,
    qtssOpenFileReadAhead= 2
};
typedef UInt32 QTSS_OpenFileFlags;
```

Constants

qtssOpenFileNoFlags

qtssOpenFileAsync

qtssOpenReadAhead

QTSS_RTPPayloadType

The `QTSS_RTPPayloadType` enumeration defines values that a module uses to specify the stream's payload type when it adds an RTP stream to a client session. The enumeration is defined as

```
enum
{
    qtssUnknownPayloadType = 0,
    qtssVideoPayloadType = 1,
    qtssAudioPayloadType = 2
};
typedef UInt32 QTSS_RTPPayloadType;
```

Constants

```
qtssUnknownPayloadType
qtssVideoPayloadType
qtssAudioPayloadType
```

QTSS_RTPNetworkMode

The `QTSS_RTPNetworkMode` enumeration defines values that describe the RTP network mode. These values are set as the value of the `qtssRTPStrNetworkMode` and `qtssRTSPReqNetworkMode` attributes of objects of type `qtssRTPStreamObjectType` and `qtssRTSPRequestObjectType`, respectively. The `QTSS_RTPNetworkMode` enumeration is defined as

```
enum
{
    qtssRTPNetworkModeDefault = 0,
    qtssRTPNetworkModeMulticast = 1,
    qtssRTPNetworkModeUnicast= 2
};
typedef UInt32 QTSS_RTPNetworkModes;
```

Constants

```
qtssRTPNetworkModeDefault
qtssRTPNetworkModeMulticast
qtssRTPNetworkModeUnicast
```

QTSS_RTPSessionState

The `QTSS_RTPSessionState` enumeration defines values that identify the state of an RTP session. The `QTSS_RTPSessionState` enumeration is defined as

```
enum
{
    qtssPausedState = 0,
    qtssPlayingState = 1
};
typedef UInt32 QTSS_RTPSessionState;
```

Constants

qtssPausedState

qtssPlayingState

QTSS_RTPTransportType

The `QTSS_RTPTransportType` enumeration defines values for RTP transports. The enumeration is defined as

```
enum
{
    qtssRTPTransportTypeUDP      = 0,
    qtssRTPTransportTypeReliableUDP= 1,
    qtssRTPTransportTypeTCP      = 2
};
typedef UInt32 QTSS_RTPTransportType;
```

Constants

qtssRTPTransportTypeUDP

qtssRTPTransportTypeReliableUDP

qtssRTPTransportTypeTCP

QTSS_RTSPSessionType

The `QTSS_RTSPSessionType` enumeration defines values that specify RTSP session types. The enumeration is defined as

```
enum
{
    qtssRTSPSession          = 0,
    qtssRTSPHTTPSession      = 1,
    qtssRTSPHTTPInputSession= 2
};
typedef UInt32 QTSS_RTSPSessionType;
```

Constants

```
qtssRTSPSession

qtssRTSPHTTPSession

qtssRTSPHTTPInputSession
```

Discussion

These session types are usually very short lived.

QTSS_ServerState

The `QTSS_ServerState` *enumeration defines values that describe the server's state. Modules can set the server's state by setting the value of the* `qtssSvrState` *attribute in the* `QTSS_ServerObject` *object. The enumeration is defined as*

```
enum
{
    qtssStartingUpState     = 0,
    qtssRunningState        = 1,
    qtssRefusingConnectionsState= 2,
    qtssFatalErrorState     = 3,
    qtssShuttingDownState   = 4,
    qtssIdleState           = 5
};
typedef UInt32 QTSS_ServerState;
```

Constants

```
qtssStartingUpState
```

```
qtssRunningState
```

```
qtssRefusingConnectionsState
```

```
qtssFatalErrorState
```

```
qtssShuttingDownState
```

```
qtssIdleState
```

Document Revision History

This table describes the changes to *QuickTime Streaming Server Modules Programming Guide* .

Date	Notes
2009-06-01	Marked as legacy.
2005-04-29	Updated for consistency with version 5.0 of the programming interface for creating QuickTime Streaming Server (QTSS) modules.

218

Index