QuickTime Streaming Server Modules

November 24, 1999
Technical Publications

Contents

Figures, Tables, and Listings

About This Manual

This manual describes Version 1.0 of the programming interface for creating QuickTime Streaming Server modules, which allow developers to easily add new functionality to the QuickTime Streaming Server. This version of the programming interface is compatible with QuickTime Streaming Server Version 2.0.

Conventions Used in This Manual

The Courier 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. ▲

For more information

The following sources provide additional information that may be of interest to developers of QuickTime Streaming Server modules:

- RFC 2326, Real Time Streaming Protocol (RTSP), available at http://www.landfield.com/rfcs/rfc2326.html and other locations on the Internet
- RFC 1889, RTP: A Transport Protocol for Real-Time Applications, available at http://www.landfield.com/rfcs/rfc1889.html and other locations on the Internet
- RFC 2327, SDP: Session Description Protocol, available at http://www.landfield.com/rfcs/rfc2327.html and other locations on the Internet

See http://developer.apple.com/techpubs/quicktime for QuickTime developer documentation.

The source code for the QuickTime Streaming Server is available at http://www.publicsource.apple.com/projects/streaming.

About QuickTime Streaming Server Modules

This manual describes Version 1.0 of the programming interface for creating QuickTime Streaming Server (QTSS) modules. This version of the programming interface is compatible with QuickTime Streaming Server Version 2.0.

QTSS is an open-source, standards-based streaming server that runs on top of several UNIX implementations, including Mac OS Server, 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 a QTSS server that is located behind a firewall
- authorizes users using name and password information stored in a database

- 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 user's QuickTime Streaming Server preferences
- monitors and report statistical information in real time
- tracks pay-per-view accounting information

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 entry point, as described in the section "Main Routine" (page 12).

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 `/usr/sbin/QTSSModules` directory. 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.

Module Requirements

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. For instructions on how to do this, see "Compiling a QTSS Module into the Server" (page 10).

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 12).

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 21).)

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.

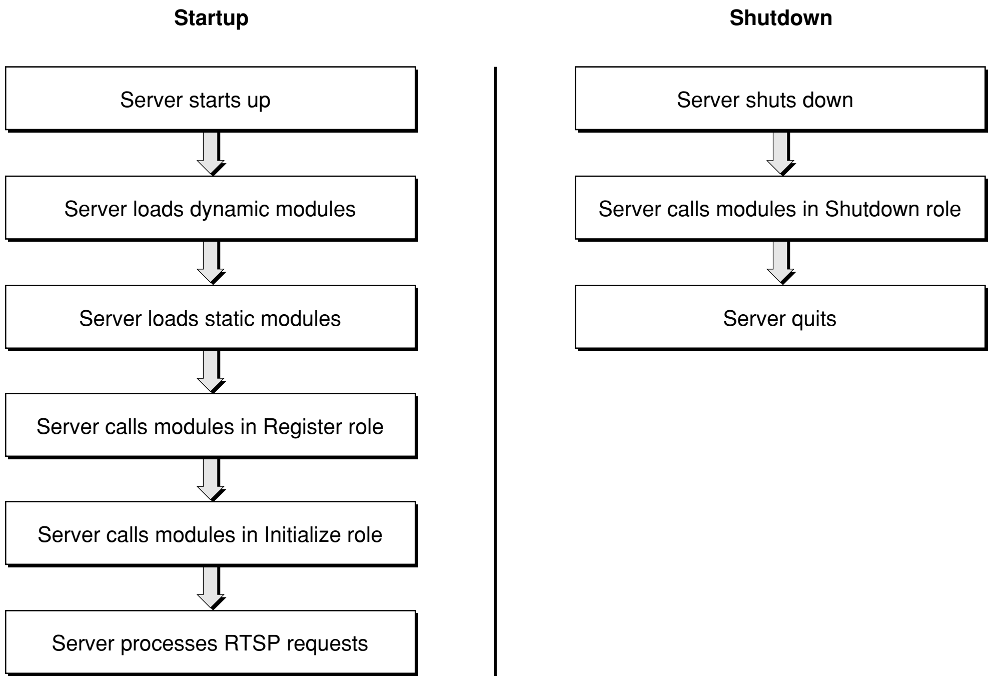
Overview of QuickTime Streaming Server Operations

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.

Server Startup and Shutdown

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

Figure 1-1 QuickTime Streaming Server startup and shutdown


Startup
Server starts up
Server loads dynamic modules
Server loads static modules
Server calls modules in Register role
Server calls modules in Initialize role
Server processes RTSP requests
Shutdown
Server shuts down
Server calls modules in Shutdown role
Server quits

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` (page 50) 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.

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 1-2.

Figure 1-2 Sample RTSP request

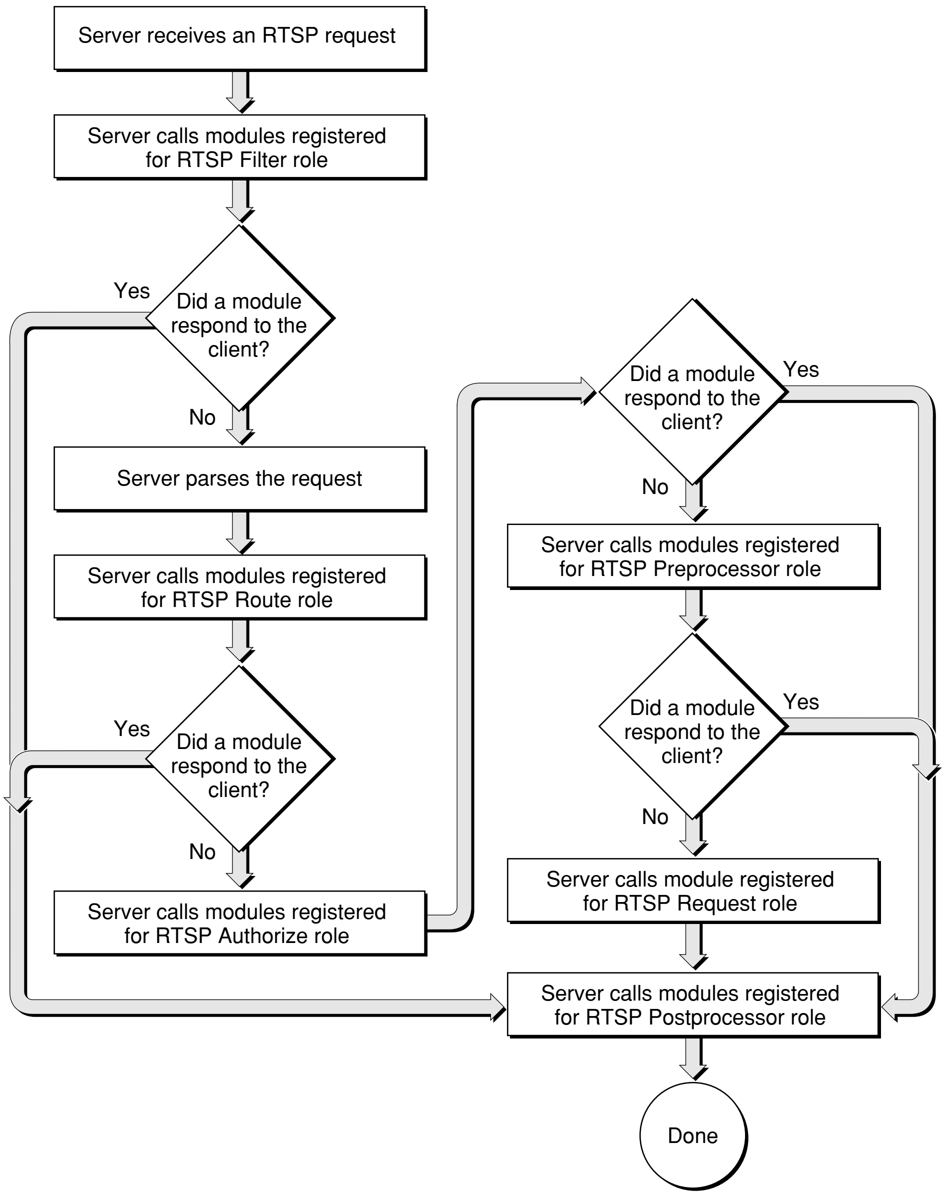
```
DESCRIBE rtsp://streaming.site.com/foo.mov RTSP/1.0
CSeq:  1
Accept: application/sdp
User-agent: QTS/1.0
```

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 1-3.

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

Figure 1-3 Summary of RTSP request processing


Server receives an RTSP request
Server calls modules registered for RTSP Filter role
Did a module respond to the client?
Yes
No
Server parses the request
Server calls modules registered for RTSP Route role
Did a module respond to the client?
Yes
No
Server calls modules registered for RTSP Authorize role
Did a module respond to the client?
Yes
No
Server calls modules registered for RTSP Preprocessor role
Did a module respond to the client?
Yes
No
Server calls module registered for RTSP Request role
Server calls modules registered for RTSP Postprocessor role
Done

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 calls 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 the 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 Authorize role for each module that has registered for that role. The RTSP Authorize role has the option of denying the request based on the name and password (stored as attributes in the RTSP request object) provided by the client.

IMPORTANT

Any module handling the RTSP Authorize role that responds to the client causes the server to skip other modules that have registered for the RTSP Authorize role, skip modules that have registered for other RTSP roles, and immediately calls the RTSP Postprocessor role of the responding module. ▲

After all RTSP Authorize 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` (page 59) or `QTSS_WriteV` (page 60) to send data to the client. To send a standard response, the module can call `QTSS_SendStandardRTSPResponse` (page 66), or `QTSS_AppendRTSPHeader` (page 64) and `QTSS_SendRTSPHeaders` (page 65).

IMPORTANT

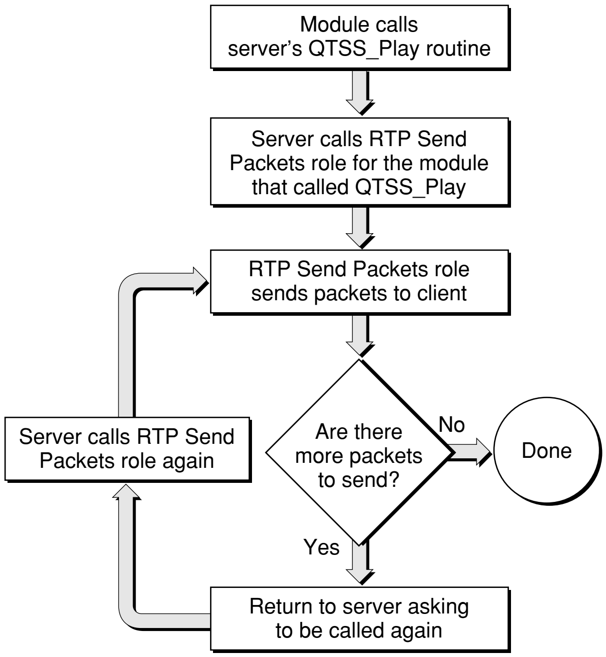
Any module handling the RTSP Preprocessor role that responds to the client causes the server to skip other modules that have registered for the RTSP Preprocessor role, skip modules that have registered for other RTSP roles, and immediately calls the RTSP Postprocessor role of the responding module. ▲

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` (page 69), which causes that module to be invoked in the RTP Send Packets role, as shown in Figure 1-4.

Figure 1-4 Summary of the RTSP Preprocessor and RTSP Request roles


Module calls server's QTSS_Play routine
Server calls RTP Send Packets role for the module that called QTSS_Play
RTP Send Packets role sends packets to client
Are there more packets to send?
No
Done
Yes
Server calls RTP Send Packets role again
Return to server asking to be called again

The RTP Send Packets role calls `QTSS_Write` (page 59) or `QTSS_WriteV` (page 60) 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 I/O. Except when responding to RTSP requests, the `QTSS_Write`, and `QTSS_WriteV` stream callback routines described in the section "QTSS Callback Routines" (page 49) return the error `QTSS_WouldBlock` if the requested I/O would block.
- 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.

To make time as clear as possible, 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. Any time value, unless otherwise noted, is reported in milliseconds from the server's internal clock. This clock has no relation to real time and is provides relative time since the server started up.

To get the current value of the server's clock, call `QTSS_Milliseconds` (page 53). To convert a time obtained from the server's clock to the current time, call `QTSS_MilliSecsTo1970Secs` (page 53).

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 one or more values of type `QTSSObject` to the module. In general, the server uses objects to exchange information with modules. For more information about objects, see "QTSS Objects" (page 38).

Table 1-1 lists the roles that the QuickTime Streaming Server Version 2.0 supports.

Table 1-1 Module roles

Name	Constant	Task
Register role	`QTSS_Register_Role`	Register the roles the module supports
Initialize role	`QTSS_Initialize_Role`	Perform tasks that initialize the module
Shutdown role	`QTSS_Shutdown_Role`	Perform cleanup tasks
Reread Preferences role	`QTSS_RereadPrefs_Role`	Reread the modules's preferences
Error Log role	`QTSS_ErrorLog_Role`	Log errors
RTSP Filter role	`QTSS_RTSPFilter_Role`	Make changes to the contents of RTSP requests
RTSP Route role	`QTSS_RTSPRoute_Role`	Routes requests from the client to the appropriate folder
RTSP Authorize role	`QTSS_RTSPAuthorize_Role`	Accepts or denies client requests based on authentication information provided by the client
RTSP Preprocessor role	`QTSS_RTSPPreProcessor_Role`	Processes requests from the client before the server processes them
RTSP Request role	`QTSS_RTSPRequest_Role`	Processes a request from the client if no other role responds the request

continued

Table 1-1 Module roles (continued)

Name	Constant	Task
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

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` (page 59) or `QTSS_WriteV` (page 60).

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` (page 50) to tell the server the roles they support.

Modules also use the Register role to call `QTSS_AddService` (page 62) to register services and to call `QTSS_AddAttribute` (page 54) to add attributes to QTSS objects. (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_TextMessagesObject inMessages;
    QTSS_StreamRef          inErrorLogStream;
} QTSS_Initialize_Params;
```

Field descriptions

inServer	A QTSS_ServerObject containing the server's global attributes. For a description of each attribute, see the section "QTSS_ServerObject" (page 87).
inPrefs	A QTSS_PrefsObject containing the server's preferences. For a description of each attribute, see the section "QTSS_PrefsObject" (page 72).
inMessages	A QTSS_TextMessagesObject that a module can use for providing localized text strings.
inErrorLogStream	A QTSS_StreamRef that a module can use to write to the server's error log.

A module that wants to be called in the Initialize role must in its Register role call QTSS_AddRole (page 50) 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 (page 50) 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 45).

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 (page 50) 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;
```

Field descriptions

`inVerbosity`	Specifies the verbosity level of this error message. Modules should use the `inflags` parameter of `QTSS_Write` (page 59) 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` (page 50) 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 13).

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. Processi ng 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;
```

Field descriptions

`inRTSPSession`	The `QTSS_RTSPSessionObject` for this RTSP session. See the section "QTSS_RTSPSessionObject" (page 85) for information about RTSP session object attributes.
`inRTSPRequest`	The `QTSS_RTSPRequestObject` for this RTSP request. When called in the RTSP Filter role, only the `qtssRTSPReqFullRequest` attribute has a value. See the section "QTSS_RTSPRequestObject" (page 82) for information about RTSP request object attributes.
`outNewRequest`	A pointer to a location in memory.

The module calls `QTSS_GetValuePtr` (page 57) 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` (page 52) 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 34).

A module that wants to be called in the RTSP Filter role must in its Register role call `QTSS_AddRole` (page 50) 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;
```

Field descriptions

`inRTSPSession`	The `QTSS_RTSPSessionObject` for this RTSP session. See the section "QTSS_RTSPSessionObject" (page 85) for information about RTSP session object attributes.
`inRTSPRequest`	The `QTSS_RTSPRequestObject` for this RTSP request. In the Route role and all subsequent RTSP roles, all of the attributes are filled in. See the section "QTSS_RTSPRequestObject" (page 82) for information about RTSP request object attributes.

`inRTSPHeaders`	The `QTSS_RTSPHeaderObject` for the RTSP headers. See the section "QTSS_RTSPHeaderObject" (page 81) for information about RTSP header object attributes.
`inClientSession`	The `QTSS_ClientSessionObject` for the client session. See the section "QTSS_ClientSessionObject" (page 76) 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 changing the `qtssRTSPReqRootDir` attribute of `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 34).

A module that wants to be called in the RTSP Route role must in its Register role call `QTSS_AddRole` (page 50) and specify `QTSS_RTSPRoute_Role` as the role.

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

RTSP Authorize Role

The server calls the RTSP Authorize role after the server has called all modules that have registered for the RTSP Route role. By default, the server allows all requests. It is the responsibility of a module handling this role to deny any particular request.

When an RTSP Authorize role denies the request, the server calls no other modules for the RTSP Authorize role and sends a 401 authorization error to the client. After receiving a 401 authorization error, the client is expected to respond with proper authorization information or fail to connect.

When called, the RTSP Authorize 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;
```

Field descriptions

`inRTSPSession`	The `QTSS_RTSPSessionObject` for this RTSP session. See the section "QTSS_RTSPSessionObject" (page 85) for information about RTSP session object attributes.
`inRTSPRequest`	The `QTSS_RTSPRequestObject` for this RTSP request with a value for each attribute. See the section "QTSS_RTSPRequestObject" (page 82) for information about RTSP request object attributes.
`inRTSPHeaders`	The `QTSS_RTSPHeaderObject` for the RTSP headers. See the section "QTSS_RTSPHeaderObject" (page 81) for information about RTSP header object attributes.
`inClientSession`	The `QTSS_ClientSessionObject` for the client session. See the section "QTSS_ClientSessionObject" (page 76) for information about client session object attributes.

A module handling the RTSP Authorize role can use a combination of authorization attributes as well as session and server attributes to evaluate the request. The retrieval and checking of server-stored passwords, users, files and directories is the module's responsibility.

Modules typically use the following attributes in the `QTSS_RTSPRequestObject` to determine whether to allow or deny an RTSP request: `qtssRTSPReqLocalPath`, `qtssRTSPReqUserName`, and `qtssRTSPReqUserPassword`. When set, these attributes contain 8-bit ASCII values. The server automatically handles the parsing and decoding of name and password values sent by the client.

An RTSP Authorize role may be called in the following contexts:

- The client requests a file but does not provide a name or a password. The RTSP Authorize role is called and finds that the `qtssRTSPReqUserName` and `qtssRTSPReqUserPassword` attributes of the `QTSS_RTSPRequestObject` are empty. The RTSP Authorize role may choose to deny the request by setting the `qtssRTSPReqUserAllowed` attribute of the `QTSS_RTSPRequestObject` to `false`. The server sends an error response of "401 authorization required" to the client.

- The client requests a file and provides a name and a password. The RTSP Authorize role is called and it evaluates the name and password. If the password is incorrect for the name or if the name is not authorized to access the file as specified by the `qtssRTSPReqLocalPath` attribute, the RTSP Authorize role sets the `qtssRTSPReqUserAllowed` attribute to `false`. The server sends an error response of "401 authorization required" to the client.
- The client requests a file and provides a name and a password. The RTSP Authorize role is called and it evaluates the name and password. If the password is correct for the name and if the name is authorized to access the file, the RTSP Authorize role allows the default setting of the `qtssRTSPReqUserAllowed` attribute to remain `true` and the server continues to process the request.

See "Setting Attribute Values" (page 41) for sample code that gets the value of the `qtssRTSPReqUserName` and `qtssRTSPReqUserPassword` attributes and sets the `qtssRTSPReqUserAllowed` attribute to `false`.

When denying a request, the RTSP Authorize role may also want to set the `qtssRTSPReqURLRealm` attribute of the object `QTSS_RTSPRequestObject` so that the client can fill in the realm-string when it displays the following message:

```
Please enter name and password for realm-string on DNS-server-name
```

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

A module that wants to be called in the RTSP Authorize role must in its Register role call `QTSS_AddRole` (page 50) and specify `QTSS_RTSPAuthorize_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 Authorize 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_ClientSessionObject    inClientSession;
} QTSS_StandardRTSP_Params;
```

Field descriptions

`inRTSPSession`	The `QTSS_RTSPSessionObject` for this RTSP session. See the section "QTSS_RTSPSessionObject" (page 85) for information about RTSP session object attributes.
`inRTSPRequest`	The `QTSS_RTSPRequestObject` for this RTSP request with a value for each attribute. See the section "QTSS_RTSPRequestObject" (page 82) for information about RTSP request object attributes.
`inRTSPHeaders`	The `QTSS_RTSPHeaderObject` for the RTSP headers. See the section "QTSS_RTSPHeaderObject" (page 81) for information about RTSP header object attributes.
`inClientSession`	The `QTSS_ClientSessionObject` for the client session. See the section "QTSS_ClientSessionObject" (page 76) 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` (page 66), `QTSS_Write` (page 59), or `QTSS_WriteV` (page 60), or by calling `QTSS_AppendRTSPHeader` (page 64) and `QTSS_SendRTSPHeaders` (page 65). If this module is also responsible for generating RTP packets for this client session, it should call `QTSS_AddRTPStream` (page 68) to add streams to the client session, and `QTSS_Play` (page 69), 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` (page 50) 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 module 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;
```

Field descriptions

`inRTSPSession`	The `QTSS_RTSPSessionObject` for this RTSP session. See the section "QTSS_RTSPSessionObject" (page 85) for information about RTSP session object attributes.
`inRTSPRequest`	The `QTSS_RTSPRequestObject` for this RTSP request with a value for each attribute. See the section "QTSS_RTSPRequestObject" (page 82) for information about RTSP request object attributes.
`inRTSPHeaders`	The `QTSS_RTSPHeaderObject` for the RTSP headers. See the section "QTSS_RTSPHeaderObject" (page 81) for information about RTSP header object attributes.
`inClientSession`	The `QTSS_ClientSessionObject` for the client session. See the section "QTSS_ClientSessionObject" (page 76) 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` (page 64) and `QTSS_AppendRTSPHeader` (page 64), by calling `QTSS_SendStandardRTSPResponse` (page 66), or by calling `QTSS_Write` (page 59) or `QTSS_WriteV` (page 60).
- Preparing the `QTSS_ClientSessionObject` for streaming by using the RTP callbacks, such as `QTSS_AddRTPStream` (page 68) and `QTSS_Play` (page 69). 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` (page 50) 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;
```

Field descriptions

`inRTSPSession`	The `QTSS_RTSPSessionObject` for this RTSP session. See the section "QTSS_RTSPSessionObject" (page 85) for information about RTSP session object attributes.
`inRTSPRequest`	The `QTSS_RTSPRequestObject` for this RTSP request with a value for each attribute. See the section "QTSS_RTSPRequestObject" (page 82) for information about RTSP request object attributes.
`inRTSPHeaders`	The `QTSS_RTSPHeaderObject` for the RTSP headers. See the section "QTSS_RTSPHeaderObject" (page 81) for information about RTSP header object attributes.
`inClientSession`	The `QTSS_ClientSessionObject` for the client session. See the section "QTSS_ClientSessionObject" (page 76) 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` (page 50) 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` (page 69). 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;
    SInt64                      outNextPacketTime;
} QTSS_RTPSendPackets_Params;
```

`inClientSession`	The `QTSS_ClientSessionObject` for the client session. See the section "QTSS_ClientSessionObject" (page 76) 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_AddRole` (page 50) for that client session. The module calls `QTSS_Write` (page 59) or `QTSS_WriteV` (page 60) 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` (page 50) 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;
```

Field descriptions

`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` 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` (page 50) 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;
```

Field descriptions

`inRTPStream`	The `QTSS_RTPStreamObject` of the RTP stream that this RTCP packet belongs to. See the section "QTSS_RTPStreamObject" (page 77) for information about RTP stream object attributes.
`inClientSession`	The `QTSS_ClientSessionObject` for the client session. See the section "QTSS_ClientSessionObject" (page 76) 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` (page 50) and specify `QTSS_RTCPProcess_Role` as the role.

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

QTSS Objects

A QTSS object is a collection of attributes whose values QTSS modules read and, in some cases, write. QTSS objects provide a way for modules to get data from the server and to provide data to the server.

The QuickTime Streaming Server defines several object types. For each of the objects that the server defines, there is a set of predefined attributes. For example, the object `QTSS_RTSPRequestObject` has a URL attribute that a module can read to obtain the URL associated with a particular RTSP request.

The server defines several object types to describe client sessions and streams, RTSP headers, sessions, and requests, global server information, server preferences, and error messages:

- `qtssRTPStreamObjectType` — Attributes associated with an individual RTP stream, such as 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` (page 68). 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.
- `qtssClientSessionObjectType` — Attributes associated with a client session, where a client session is defined as a single client streaming presentation.
- `qtssRTSPSessionObjectType` — Attributes associated with an RTSP client-server connection. An RTSP session object (`QTSS_RTSPSessionObject`) is an instance of this object type that exists as long as the RTSP client is connected to the server.
- `qtssRTSPRequestObjectType` — Attributes associated with an individual RTSP request. An RTSP request object (`QTSS_RTSPRequestObject`) is an instance of this object type that exists from the time the server receives a complete RTSP request from a client until the time that the response has been 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 on a single connection.
- `qtssRTSPHeaderObjectType` — All of the RTSP request headers associated with an individual RTSP request. The names of the built-in attributes in this object are the names of RTSP headers and their values correspond directly to the names of RTSP headers. For example, if a module wants to read the value of a session header in an RTSP request, it would read the value of the `Session` attribute in an RTSP header reference (`QTSS_RTSPHeaderObject`).
- `qtssServerObjectType` —Global server attributes, such as server statistics. There is a single instance of this object type.
- `qtssPrefsObjectType` —Attributes for the server's internal preference storage system. On Mac OS X, the attribute values for this object are stored in a NetInfo database. On other platforms, the attribute values come from a text configuration file named `QTSS.conf`. Attribute names correspond to keywords that identify preferences. For example, a module could add an attribute to the `QTSS_Prefs` object named `num_outstanding_doodads`. Retrieving the value of this attribute from the global preferences object retrieves the current value of this preference from the NetInfo database or the `QTSS.conf` file. There is a single instance of this object type.

- `qtssTextMessageObjectType` — Contains attributes whose values are intended for display to the user or are returned to the client. To make localization easier, the values are text strings.

Note
Modules can also add new attributes to any object. See `QTSS_AddAttribute` (page 54) for information on adding attributes to objects. ◆

Getting Attribute Values

Modules use attributes stored in objects to exchange information with the server, so they frequently get and set attribute values. Some attributes are preemptive safe and their values can be obtained at any time by calling `QTSS_GetValuePtr` (page 57), which returns a pointer to the server's internal copy of the attribute value. Other attributes are not preemptive safe and their values must be obtained by calling `QTSS_GetValue` (page 56), which copies the attribute value into a buffer provided by the module.

Note
A module can obtain the value of any attribute by calling `QTSS_GetValue`, but whenever modules get the value of preemptive safe attributes, they should call `QTSS_GetValuePtr` because it is faster than `QTSS_GetValue`. ◆

The sample code in Listing 1-1 calls `QTSS_GetValue` (page 56) to get the value of the `qtssRTPSvrCurConn` attribute, which is not preemptive safe, from the object `QTSS_ServerObject`.

Listing 1-1 Sample code that calls 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);
    // Retreive 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 1-2 calls `QTSS_GetValuePtr` (page 57), 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 1-2 Sample code that calls 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;
}
```

Setting Attribute Values

The sample code in Listing 1-3 calls `QTSS_GetValue` (page 56) to get the values of the `qtssRTSPReqUserName`, the `qtssRTSPReqUserPassword`, and the `qtssRTSPReqLocalPath` attribute from the object `QTSS_RTSPRequestObject`. If the user is not authorized, this function denies the request by setting the `qtssRTSPReqUserAllowed` attribute to `false`.

Listing 1-3 Sample code that calls QTSS_SetValue

```
// First get the user name using QTSS_GetValuePtr
// theUserName & theUserNameLen will get set by QTSS_GetValuePtr below

char* theUserName = NULL;
UInt32 theUserNameLen = 0;

QTSS_Error theErr = QTSS_GetValuePtr(inParams->inRTSPRequest, qtssRTSPReqUserName, 0,
    &theUserName, &theUserNameLen);

// Check for any errors
if (theErr != QTSS_NoErr) return;

// Get the user password by calling QTSS_GetValuePtr.

char* theUserPassword = NULL;
UInt32 theUserPasswordLen = 0;

theErr = QTSS_GetValuePtr(inParams->inRTSPRequest, qtssRTSPReqUserPassword, 0,
    &theUserPassword, &theUserPasswordLen);

// Check for any errors.
if (theErr != QTSS_NoErr) return;

// Get the full path to the requested file.
char* theFullPath = NULL;
UInt32 theFullPathLen = 0;

theErr = QTSS_GetValuePtr(inParams->inRTSPRequest, qtssRTSPReqLocalPath, 0,
    &theFullPath, &theFullPathLen);

// Check for any errors.
if (theErr != QTSS_NoErr) return;
```

```
// Check the name, password, and file. (Code not provided)
// If the check determines the user is not authorized, deny the request.
Bool16 allow = false;
theErr = QTSS_SetValue (inParams->inRTSPRequest, qtssRTSPReqUserAllowed,
    0, (void *) &allow, sizeof(allow) );
```

Adding Attributes to QTSS Object Types

Any module can add an attribute to a QTSS object type by calling the `QTSS_AddAttribute` (page 54) callback routine from its Register role.

Note
Using one or more added attributes 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` (page 58), `QTSS_GetValue` (page 56), and `QTSS_GetValuePtr` (page 57).

The sample code in Listing 1-4 calls `QTSS_AddAttribute` (page 54) to add an attribute to the object `QTSS_ClientSessionObject`.

Listing 1-4 Sample code that calls QTSS_AddAttribute

```
QTSS_Error MyRegisterRoleFunction()
{
    // Add the attribute. The third parameter is an optional attribute retrieval
    // function that can be set to NULL.

    QTSS_Error theErr = QTSS_AddAttribute(qtssClientSessionObjectType,
                            sExampleAttributeName, NULL);

    // 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,
                QTSSSampleModuleExampleAttribute, &theID);

    // Store the attribute ID in a global so for later use. Attribute IDs do not
    // change while the server is running.

    gMyExampleAttrID = theID;
}
```

QTSS Streams

The QTSS programming interface provides QTSS stream references as a generalized stream abstraction. QTSS stream references are usually used for communicating with the client. For example, in all RTSP roles modules receive an object of type `QTSS_RTSPRequestObject` having 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.

QTSS stream references are generalized enough to be used in many other situations. For example, modules receive a QTSS stream reference for the error log, which modules can use when writing errors in the error log.

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

`inErrorLog`	Allows binary data to be written 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 25).
`qtssRTSPSesStreamRef`	Represents a stream for writing data to an RTSP client. The server may encounter flow control conditions when sending data to the RTSP client, so modules should be prepared to handle `QTSS_WouldBlock` errors when writing to this stream type. This stream reference is an attribute of the object `QTSS_RTSPSessionObject`.

`qtssRTSPReqStreamRef`
: Represents a stream for writing data to an RTSP client. This stream type is identical to the `qtssRTSPSessionRef` stream except that data written to this stream type is buffered in memory by the stream 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. This stream reference is an attribute of the object `QTSS_RTSPRequestObject`.

`qtssRTPStrStreamRef`
: Represents a stream 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. Modules should be prepared to handle `QTSS_WouldBlock` errors that may be returned when writing to this stream type. Data written to this stream is not buffered by the server, so this stream is useful for sending long RTSP responses to the client. This stream reference is an attribute of the object `QTSS_RTPStreamObject`.

Note
All stream references are asynchronous. ◆

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 62) 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` (page 62) 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` (page 63) and providing the name of the service as a parameter. With the service's ID, the module calls `QTSS_DoService` (page 63) to cause the service to run. Here is an example:

```
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` (page 63). Then call `QTSS_DoService` (page 63) to run the service.

C H A P T E R 2

QuickTime Streaming Server Module Reference

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

QTSS Callback Routines

This section describes the QTSS callback routines that modules call to obtain information from the server, to allocate and deallocate memory, to get and set attribute values, and to manage client and RTSP sessions.

The QTSS callback routines are described in these sections:

- "QTSS Utility Callback Routines" (page 49)
- "QTSS Attribute Callback Routines" (page 53)
- "Stream Callback Routines" (page 59)
- "Service Callback Routines" (page 62)
- "RTSP Header Callback Routines" (page 64)
- "RTP Callback Routines" (page 67)

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 50) to tell the server that the module wants to be called for a specific role.

- `QTSS_Milliseconds` (page 53) to get the current value of the server's internal timer.
- `QTSS_MilliSecsTo1970Secs` (page 53) to convert a value returned by `QTSS_Milliseconds` to the current time.
- `QTSS_New` (page 52) to allocate memory.
- `QTSS_Delete` (page 52) to dispose of memory allocated by `QTSS_New`.

QTSS_AddRole

Adds a role.

```
QTSS_Error QTSS_AddRole(QTSS_Role inRole);
```

`inRole`	On input, a value of type `QTSS_Role` 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 Initialize role.

For this version of the server, you can add the roles listed in Table 2-1:

Table 2-1 Role constants

`QTSS_ErrorLog_Role`	Called when an error occurs
`QTSS_Initialize_Role`	Called at server startup after the Register role to initialize the module
`QTSS_RTSPFilter_Role`	Called to filter RTSP requests before the server parses them
`QTSS_RTSPRoute_Role`	Called to change the root folder for handling an RTSP request
`QTSS_RTSPAuthorize_Role`	Called to authorize RTSP requests
`QTSS_RTSPPreProcessor_Role`	Called to process RTSP requests. Modules can respond to the request by sending packets to the client
`QTSS_RTSPRequest_Role`	Called to process an RTSP request and send a response to the client if no module responds to the client in the RTSP Preprocessor role
`QTSS_RTSPPostProcessor_Role`	Called to post-process RTSP requests
`QTSS_RTPSendPackets_Role`	Called to send RTP packets to the client
`QTSS_ClientSessionClosing_Role`	Called to inform the module that a client session is closing
`QTSS_RTCPProcess_Role`	Called to process all RTCP packets sent to the server by the client
`QTSS_Shutdown_Role`	Called when the server shuts down

QTSS_New

Allocates memory.

```
void* QTSS_New(
                    FourCharCode inMemoryIdentifier,
                    UInt32 inSize);
```

`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	A result code. Possible values are `QTSS_NoErr`.

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 52).

QTSS_Delete

Deletes memory.

```
void* QTSS_Delete(void* inMemory);
```

`inMemory`	On input, a pointer to an arbitrary value that specifies in bytes the amount of memory to be deleted.
result	A result code. Possible values are `QTSS_NoErr`.

DISCUSSION

The `QTSS_Delete` callback routine deletes memory that was previously allocated by `QTSS_New` (page 52).

QTSS_Milliseconds

Gets the current value of the server's internal timer.

```
SInt64 QTSS_Milliseconds();
```

result	The value of the server's internal timer in milliseconds since the server started up.

DISCUSSION

The `QTSS_Milliseconds` callback routine gets the current value of the server's internal timer since the server started up. Unless otherwise noted, all millisecond values that the server provides in attributes are obtained from this timer.

QTSS_MilliSecsTo1970Secs

Converts milliseconds to seconds since 1970.

```
time_t QTSS_MilliSecsTo1970Secs(SInt64 inQTSS_Milliseconds);
```

`inQTSS_Milliseconds`	On input, a value of type `SInt64` obtained by calling `QTSS_Milliseconds`.
result	A value of type `time_t` containing the converted value.

DISCUSSION

The `QTSS_MilliSecsto1970Secs` callback routine converts a value obtained by calling `QTSS_Milliseconds` (page 53) to the number of seconds since 1970.

QTSS Attribute Callback Routines

Modules call the following routines to work with attributes:

- `QTSS_AddAttribute` (page 54) to add an attribute to an object type.

- `QTSS_IDForAttr` (page 55) to get the ID for an attribute name.
- `QTSS_GetValue` (page 56) to get the value of an attribute.
- `QTSS_GetValuePtr` (page 57) to get a pointer to an attribute value.
- `QTSS_SetValue` (page 58) to set the value of an attribute.

QTSS_AddAttribute

Adds an attribute to an object type.

```
QTSS_Error QTSS_AddAttribute(
                QTSS_ObjectType inType,
                const char* inAttributeName,
                QTSS_AttrFunctionPtr inFunctionPtr);
```

`inType`	On input, a value of type `QTSS_ObjectType` that specifies the type of object to which the attribute is to be added. For possible values, see the section "QTSS Objects" (page 38).
`inAttributeName`	On input, a pointer to a byte array that specifies the name of the attribute that is to be added.
`inFunctionPtr`	On input, a pointer to a module-implemented routine that is called whenever a module calls `QTSS_GetValue` (page 56) or `QTSS_GetValuePtr` (page 57) to get the value of the added attribute, or `NULL`.
result	A result code. Possible values are `QTSS_NoErr`, `QTSS_OutOfState` if `QTSS_AddAttribute` is called from a role other than the Register role, and `QTSS_BadArgument` if the specified object type does not exist, the attribute name is too long, or a parameter is not specified.

DISCUSSION

The `QTSS_AddAttribute` callback routine adds a new attribute to all objects of the type specified by the `inType` parameter.

The `QTSS_AddAttribute` callback can only be called from the Register role.

The module-implemented routine must conform to the following type definition:

```
typedef void* (*QTSS_AttrFunctionPtr)(QTSS_Object, UInt32* );
```

On input, the `QTSS_Object` for the specified attribute is passed in.

The module-implemented routine has two ways to return the attribute value to the caller:

- Return the attribute body as the return value and the attribute length in the second parameter, which returns the attribute to the caller but does not update the object. Each subsequent time that the attribute value is requested, the routine is invoked again.
- Call `QTSS_SetValue` (page 58) to set the attribute in the object. Once the attribute is set in the object, your routine will not be called again for this object.

QTSS_IDForAttr

Gets the ID of an attribute.

```
QTSS_Error QTSS_IDForAttr(
                    QTSS_ObjectType inType,
                    const char* inAttributeName,
                    QTSS_AttributeID* outID);
```

`inType`	On input, a value of type `QTSS_ObjectType` that specifies the type of object for which the ID is to be obtained. For possible values, see the section "QTSS Objects" (page 38).
`inAttributeName`	On input, a pointer to a byte array that specifies the name of the attribute whose ID is to be obtained.
`outID`	On input, a pointer to a value of type `QTSS_AttributeID`. On output, `outID` contains 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 attribute in the specified object type. You can use the ID to obtain the value of the attribute by calling `QTSS_GetValue` (page 56) or `QTSS_GetValuePtr` (page 57).

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);
```

`inObject`	On input, a value of type `QTSS_Object` that specifies the object that contains the attribute whose value is to be obtained.
`inID`	On input, a value of type `QTSS_AttributeID` that specifies 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, `ioBuffer` contains the value of the attribute specified by `inID`. If the buffer is too small to contain the value, `ioBuffer` is empty.
`ioLen`	On input, a pointer to a value of type `UInt32` that specifies the length of `ioBuffer`. On output, `ioLen` contains 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` of the index specified by inIndex does not exist, `QTSS_NotEnoughSpace` if the attribute value is longer than the value specified by `ioLen`.

DISCUSSION

The `QTSS_GetValue` callback routine copies the value of the specified attribute into the provided buffer.

You must call `QTSS_GetValue` to get the value of any attribute that is not preemptive safe. When getting the value of a preemptive safe attribute, you should always call `QTSS_GetValuePtr` (page 57) because `QTSS_GetValuePtr` is the most efficient function and less likely to encounter an error condition.

QTSS_GetValuePtr

Gets a pointer to an attribute value.

```
QTSS_Error QTSS_GetValuePtr (
                    QTSS_Object inObject,
                    QTSS_AttributeID inID,
                    UInt32 inIndex,
                    void** outBuffer,
                    UInt32* outLen);
```

`inObject`	On input, a value of type `QTSS_Object` that specifies the object containing the attribute whose value is to be obtained.
`inID`	On input, a value of type `QTSS_AttributeID` that specifies the ID of an attribute.
`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.
`outBuffer`	On input, a pointer to an address in memory. On output, `outBuffer` points to the value of the attribute specified by `inID`.
`outLen`	On output, a pointer to a value of type `UInt32` that contains 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, and `QTSS_BadIndex` if the index specified by `inIndex` does not exist.

DISCUSSION

The `QTSS_GetValuePtr` callback routine gets a pointer to an attribute value. When getting the value of an attribute that is preemptive safe, you should always call `QTSS_GetValuePtr` because it is faster, more efficient, and less likely to generate an error.

Note
This `QTSS_GetValuePtr` callback cannot be used to get the value of attributes that are not preemptive safe. To get the value of an attribute that is not preemptive safe, call `QTSS_GetValue` (page 56). ◆

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);
```

`inObject`	On input, a value of type `QTSS_Object` that specifies the object containing the attribute whose value is to be set.
`inID`	On input, a value of type `QTSS_AttributeID` 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, and `QTSS_ReadOnly` if the attribute is read-only.

DISCUSSION

The `QTSS_SetValue` callback routine sets the value of the specified attribute.

Stream Callback Routines

This section describes the callback routines that modules call to perform I/O on streams. The routine are

- `QTSS_Write` (page 59) to write data to the client.
- `QTSS_WriteV` (page 60) to write data the client using an `iovec` structure.
- `QTSS_Flush` (page 61) to write data that may have been buffered.

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_Write

Writes data to a stream.

```
QTSS_Error QTSS_Write(
                 QTSS_StreamRef inRef,
                 void* inBuffer,
                 UInt32 inLen,
                 UInt32* outLenWritten,
                 UInt32 inFlags);
```

`inRef`	On input, a value of type `QTSS_StreamRef` 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 relevent when writing to an RTP stream reference and tell the server whether the data written should be sent over the RTP channel (`qtssWriteFlagsIsRTP`) or 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);
```

`inRef`	On input, a value of type `QTSS_StreamRef` 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_Flush

Forces an immediate write operation.

```
QTSS_Error QTSS_Flush(QTSS_StreamRef inRef);
```

`inRef`	On input, a value of type `QTSS_StreamRef` 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.

Service Callback Routines

Modules use the callback routines described in this section to register and invoke services. The service callback routines are:

- QTSS_AddService (page 62) to add a service that other modules can call.
- QTSS_IDForService (page 63) to get the ID of a service.
- QTSS_DoService (page 63) to call a service provided by another module or by the server.

QTSS_AddService

Adds a service.

```
QTSS_Error QTSS_AddService(
                    const char* inServiceName,
                    QTSS_ServiceFunctionPtr inFunctionPtr);
```

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.

Note
The QTSS_AddService callback can only be called from the Register role. ◆

QTSS_IDForService

Resolves a service name to a service ID.

```
QTSS_Error QTSS_IDForService(
                    const char* inTag,
                    QTSS_ServiceID* outID);
```

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. 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 63) to invoke the service that serviceID represents.

QTSS_DoService

Invokes a service.

```
QTSS_Error QTSS_DoService(
                    QTSS_ServiceID inID,
                    QTSS_ServiceFunctionArgsPtr inArgs);
```

`inID`	On input, a value of type `QTSS_ServiceID` that specifies the service that is to be invoked. Call `QTSS_IDForAttr` (page 55) 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`.

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. The routines are

- `QTSS_AppendRTSPHeader` (page 64) to append information to an RTSP header.
- `QTSS_AppendRTSPHeader` (page 64) to send an RTSP header
- `QTSS_SendStandardRTSPResponse` (page 66) to send an RTSP response to a client.

QTSS_AppendRTSPHeader

Appends information to an RTSP header.

```
QTSS_Error QTSS_AppendRTSPHeader(
                    QTSS_RTSPRequestObject inRef,
                    QTSS_RTSPHeader inHeader,
                    const char* inValue,
                    UInt32 inValueLen);
```

`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 you call `QTSS_AppendRTSPHeader`, call `QTSS_SendRTSPHeaders` (page 65) to send the entire header.

QTSS_SendRTSPHeaders

Sends an RTSP header.

```
QTSS_Error QTSS_SendRTSPHeaders(QTSS_RTSPRequestOjbect inRef);
```

`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);
```

`inRTSPRequest`	On input, a value of type `QTSS_RTSPRequestObject` for the RTSP stream.
`inRTPInfo`	On input, a value of type `QTSS_Object` that identifies the QTSS object type.
`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 an RTP-Info header.
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 written depends on the method.

The following enumeration defines a constant for the `inFlags` parameter:

```
enum
{
    qtssPlayRespWriteTrackInfo = 0x00000001
};
```

Table 2-2 describes the data returned by each method that the `QTSS_SendStandardRTSPResponse` callback supports.

Table 2-2 `QTSS_SendStandardRTSPResponse` method responses

Method	Response	Object
`DESCRIBE`	Writes status line, CSeq, SessionID, and connection headers as determined by the request. Writes a content-base header with the provided URL as the content base. Writes `application/sdp` as the content-type header.	`QTSS_ClientSessionObject`
`SETUP`	Writes status line, CSeq, SessionID, and connection headers as determined by the request. Writes a transport header. If the connection is over UDP, the transport header includes client and server ports.	`QTSS_ClientSessionObject`
`PLAY`	Writes status line, CSeq, SessionID, and connection headers as determined by the request. Set `inFlags` to `qtssPlayRespWriteTrackInfo` if you want the server to append the seq number, a timestamp, and SSRC information into an RTP-Info header.	`QTSS_ClientSessionObject`
`PAUSE`	Writes status line, CSeq, and connection headers as determined by the request.	`QTSS_ClientSessionObject`
`TEARDOWN`	Writes status line, CSeq, SessionID, and connection headers as determined by the request.	`QTSS_ClientSessionObject`

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.

The RTP callback routines are

- QTSS_AddRTPStream (page 68), which is called by a module to enable the sending of RTP packets to a client. Only one module can call QTSS_AddRTPStream for any particular session.
- QTSS_Play (page 69), which is called by a module to start the playing of streams for a client session.
- QTSS_Pause (page 70), which is called by a module pause the playing of streams for a client session
- QTSS_Teardown (page 71), which is called by a module to close a client session.

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);
```

inClientRequest	On input, a value of type QTSS_ClientSessionObject that identifies 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 that specifies stream options. See the Discussion section for possible values.
result	A result code. Possible values are QTSS_NoErr, QTSS_RequestFailed if the QTSS_RTPStreamObject 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.

The following enumeration defines possible values for the inFlags parameter:

```
enum
{
    qtssASFlagsAllowDestination = 0x00000001,
    qtssASFlagsForceInterleave = 0x00000002
};
typedef UInt32 QTSS_AddStreamFlags;
```

To start playing a stream, call QTSS_Play (page 69).

QTSS_Play

Starts playing streams associated with a client session.

```
QTSS_Error QTSS_Play(
                     QTSS_ClientSessionObject inClientSession,
                     QTSS_RTSPRequestObject inRTSPRequest,
                     QTSS_PlayFlags inPlayFlags);
```

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 68).
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. After calling `QTSS_Play`, the module is invoked in the RTP Send Packets role.

Before calling `QTSS_Play`, the module should set the following attributes of the object `QTSS_RTPStreamObject` for this RTP stream:

- The `qtssRTPStrFirstSeqNumber` attribute, 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.
- The `qtssRTPStrFirstTimestamp` attribute, 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.
- The `qtssRTPStrTimescale` attribute, which should be set to the timescale for the track.

Call `QTSS_Pause` (page 70) to pause playing or call `QTSS_Teardown` (page 71) to close the client session.

Note
The module that called `QTSS_AddRTPStream` (page 68) is the only module that can call `QTSS_Play`. ◆

QTSS_Pause

Pauses a stream that is playing.

```
QTSS_Error QTSS_Pause(QTSS_ClientSessionObject inClientSession);
```

`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.

Note
The module that called `QTSS_AddRTPStream` (page 68) is the only module that can call `QTSS_Pause`. ◆

QTSS_Teardown

Closes a client session.

```
QTSS_Error QTSS_Teardown(QTSS_ClientSessionObject inClientSession);
```

`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. Calling `QTSS_Teardown` causes the calling module to be invoked in the Client Session Closing role for the session identified by the `inClientSession` parameter.

Note
The module that called `QTSS_AddRTPStream` (page 68) is the only module that can call `QTSS_Teardown`. ◆

QTSS Data Types

This section describes QTSS objects and their attributes.

QTSS_PrefsObject

A `QTSS_PrefsObject` is the collection of attributes that contain server preferences. Table 2-3 lists the attributes of the object `QTSS_PrefsObject`. These attributes are valid in all methods.

Note
None of the attributes for the object `QTSS_PrefsObject` are preemptive safe, so they can only be read by calling `QTSS_GetValue` (page 56). ◆

Table 2-3 Attributes of the object `QTSS_PrefsObject`

Attribute Name and Content	Read/Write	Data Type
`qtssPrefsRTSPTimeout`	Read	`UInt32`
RTSP timeout in seconds sent to the client.		
`qtssPrefsRealRTSPTimeout`	Read	`UInt32`
The amount of time in seconds the server will wait before disconnecting idle RTSP clients. Zero means that there is no timeout.		
`qtssPrefsRTPTimeout`	Read	`UInt32`
The amount of time in seconds the server will wait before disconnecting idle RTP clients. Zero means there is no timeout.		

continued

Table 2-3 Attributes of the object `QTSS_PrefsObject` (continued)

Attribute Name and Content	Read/Write	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.	Read	`SInt32`
`qtssPrefsMaximumBandwidth` The maximum amount of bandwidth the server is allowed to serve in K bits. A value of –1means the amount is unlimited.	Read	`SInt32`
`qtssPrefsMovieFolder` The path to the root movie folder.	Read	`char*`
`qtssPrefsRTSPIPAddr` The IP address in dotted-decimal format the server should accept RTSP connections on. A value of 0.0.0.0 means all addresses on the machine.	Read	`char*`
`qtssPrefsBreakOnAssert` If `true`, the server will enter the debugger when an assert fails	Read	`Bool16`
`qtssPrefsAutoRestart` If `true`, the server automatically restarts itself if it crashes.	Read	`Bool16`
`qtssPrefsTotalBytesUpdate` The interval in seconds between updates of the server's total bytes and current bandwidth statistics.	Read	`UInt32`
`qtssPrefsAvgBandwidthUpdate` The interval in seconds between computations of the server's average bandwidth.	Read	`UInt32`
`qtssPrefsSafePlayDuration` See `QTSS.conf`.	Read	`UInt32`

continued

Table 2-3 Attributes of the object QTSS_PrefsObject (continued)

Attribute Name and Content	Read/Write	Data Type
qtssPrefsModuleFolder The path to the module folder.	Read	char*
The built-in error log module that loads before all other modules uses the following attributes:		
qtssPrefsErrorLogName The name of the error log file.	Read	char*
qtssPrefsErrorLogDir The path to the directory containing the error log file.	Read	char*
qtssPrefsErrorLogDir The path to the directory containing the error log file.	Read	char*
qtssPrefsErrorRollInterval The interval in days between rolling the error log file.	Read	UInt32
qtssPrefsMaxErrorLogSize The maximum size in bytes of the error log.	Read	UInt32
qtssPrefsErrorLogVerbosity The maximum verbosity level of messages the error logger logs.	Read	UInt32
qtssPrefsScreenLogging If the error logger should echo messages to the screen, this attribute is true.	Read	Bool16
qtssPrefsErrorLogEnabled If error logging is enabled, this attribute is true.	Read	Bool16
qtssPrefsTCPMinThinDelayToleranceInMSec The maximum delay in milliseconds for minimum (bframe) thinning bandwidth.	Read	SInt32
qtssPrefsTCPMaxThinDelayToleranceInMSec The maximum delay in milliseconds for maximum (keyframe) thinning bandwidth.	Read	SInt32

continued

Table 2-3 Attributes of the object `QTSS_PrefsObject` (continued)

Attribute Name and Content	Read/Write	Data Type
`qtssPrefsTCPVideoDelayToleranceInMSec` The maximum delay in milliseconds before dropping video packets.	Read	`SInt32`
`qtssPrefsMinTCPBufferSizeInBytes` The minimum size in bytes that the TCP socket send buffer can be set to when streaming over TCP.	Read	`UInt32`
`qtssPrefsMaxTCPBufferSizeInBytes` The maximum size in bytes that the TCP socket send buffer can be set to when streaming over TCP.	Read	`UInt32`
`qtssPrefsTCPSecondsToBuffer` The size of the TCP send buffer is scaled to based on the movie's bit rate when streaming over TCP. The buffer will store all the data that is sent in the specified amount of time.	Read	`Float32`
`qtssPrefsDoReportHTTPConnectionAddress` 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.	Read	`Bool16`
`qtssPrefsAuthorizationEnabled` If authorization modules are to be enabled, this attribute is `true`; otherwise, this attribute is `false`.	Read	`Bool16`
`qtssPrefsDefaultAuthorizationRealm` The default authorization realm. The value of this attribute is "QT Streaming Server".	Read	`char*`

QTSS_ClientSessionObject

A `QTSS_ClientSessionObject` is the collection of attributes that describe client sessions.Table 2-4 lists the attributes for the object `QTSS_ClientSessionObject`. These attributes are valid for all roles that receive a value of type `QTSS_ClientSessionObject` in the structure the server passes to them.

Note
All of the attributes for the object `QTSS_ClientSessionObject` are preemptive safe, so they can be read by calling `QTSS_GetValue` (page 56) or `QTSS_GetValuePtr` (page 57). ◆

Table 2-4 Attributes of the object `QTSS_ClientSessionObject`

Attribute Name and Content	Read/Write	Data Type
`qtssCliSesStreamObjects` Iterated attribute containing all RTP stream references (`QTSS_RTPStreamObject`) belonging to this session.	Read	`QTSS_RTPStreamObject`
`qtssCliSesCreateTimeInMsec` The time in milliseconds that the session was created.	Read	`SInt64`
`qtssCliSesFirstPlayTimeInMsec` The time in milliseconds at which `QTSS_Play` was first called.	Read	`SInt64`
`qtssCliSesPlayTimeInMsec` The time in milliseconds at which `QTSS_Play` was most recently called.	Read	`SInt64`
`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.	Read	`SInt64`
`qtssCliSesRTPBytesSent` The number of RTP bytes sent for this session.	Read	`SInt32`

continued

Table 2-4 Attributes of the object `QTSS_ClientSessionObject (continued)`

Attribute Name and Content	Read/Write	Data Type
`qtssCliSesState` The state of this session. Possible values are `qtssPausedState` and `qtssPlayingState`.	Read	`QTSS_RTPSessionState`
`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.	Read	`char*`
`qtssCliSesMovieDurationInSecs` Duration of the movie for this session in seconds. The value is zero unless set by a module.	Read/write	`Float64`
`qtssCliSesMovieSizeInBytes` Movie size in bytes. The value is zero unless set by a module.	Read/write	`UInt64`
`qtssCliSesMovieAverageBitRate` The average bits per second based on total RTP bits/movie duration. The value is zero unless set by a module.	Read/write	`UInt32`

QTSS_RTPStreamObject

A `QTSS_RTPStreamObject` is the collection of attributes that describe a particular RTP stream. Table 2-5 lists the attributes for the object `QTSS_RTPStreamObject`. These attributes are valid for all roles that receive a value of type `QTSS_RTPStreamObject` in the structure the server passes to them.

Note
All of the attributes for the object `QTSS_RTPStreamObject` are preemptive safe, so they can be read by calling `QTSS_GetValue` (page 56) or `QTSS_GetValuePtr` (page 57). ◆

Table 2-5 Attributes of the object `QTSS_RTPStreamObject`

Attribute Name and Content	Read/Write	Data Type
`qtssRTPStrTrackID` Unique ID that identifies each RTP stream.	Read/write	`UInt32`
`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.	Read	`UInt32`
`qtssRTPStrPayloadName` Name of the media for this stream. This attribute is empty unless a module explicitly sets it.	Read/write	`char*`
`qtssRTPStrPayloadType` Payload type of the media for this stream. The value of this attribute is `qtssUnknownPayloadType` unless a module sets it `qtssVideoPayloadType` or `qtssAudioPayloadType`.	Read/write	`QTSS_RTPCodecType`
`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` (page 69). The server uses this attribute to generate a proper RTSP PLAY response.	Read/write	`SInt16`

continued

Table 2-5 Attributes of the object `QTSS_RTPStreamObject` (continued)

Attribute Name and Content	Read/Write	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` (page 69). The server uses this attribute to generate a proper RTSP PLAY response.	Read/write	`SInt32`
`qtssRTPStrTimescale` Timescale for the track. If known, this must be set before calling `QTSS_Play` (page 69).	Read/write	`SInt32`
`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.	Read	`Float32`
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.		
`qtssRTPStrFractionLostPackets` The fraction of packets that have been lost for this stream.	Read	`UInt32`
`qtssRTPStrTotalLostPackets` The total number of packets that have been lost for this stream.	Read	`UInt32`
`qtssRTPStrJitter` Cumulative jitter for this stream.	Read	`UInt32`
`qtssRTPStrRecvBitRate` Average bit rate received by the client in bits per second.	Read	`UInt32`
`qtssRTPStrAvgLateMilliseconds` Average in milliseconds of packets that the client received late.	Read	`UInt16`

continued

Table 2-5 Attributes of the object `QTSS_RTPStreamObject` (continued)

Attribute Name and Content	Read/Write	Data Type
`qtssRTPStrPercentPacketsLost` Fixed percentage of lost packets for this stream.	Read	`UInt16`
`qtssRTPStrAvgBufDelayInMsec` Average buffer delay in milliseconds.	Read	`UInt16`
`qtssRTPStrGettingBetter` A non-zero value if the client reports that the stream is getting better.	Read	`UInt16`
`qtssRTPStrGettingWorse` A non-zero value if the client reports that the stream is getting worse.	Read	`UInt16`
`qtssRTPStrNumEyes` Number of clients connected to this stream.	Read	`UInt32`
`qtssRTPStrNumEyesActive` Number of clients playing this stream.	Read	`UInt32`
`qtssRTPStrNumEyesPaused` Number of clients connected but currently paused.	Read	`UInt32`
`qtssRTPStrTotPacketsRecv` Total packets received by the client.	Read	`UInt32`
`qtssRTPStrTotPacketsDropped` Total packets dropped by the client.	Read	`UInt16`
`qtssRTPStrTotPacketsLost` Total packets lost.	Read	`UInt16`
`qtssRTPStrClientBufFill` How full the client buffer is in tenths of a second.	Read	`UInt16`
`qtssRTPStrFrameRate` The current frame rate in frames per second.	Read	`UInt16`
`qtssRTPStrExpFrameRate` The expected frame rate in frames per second.	Read	`UInt16`

continued

Table 2-5 Attributes of the object `QTSS_RTPStreamObject` (continued)

Attribute Name and Content	Read/Write	Data Type
`qtssRTPStrAudioDryCount` Number of times the audio has run dry.	Read	`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`.	Read	`Bool16`
`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.	Read	`QTSS_StreamRef`
`qtssRTPStrCurrentPacketDelay` Delivery delay in milliseconds for the most recently written packet as measured by a module. Values less than zero indicate the number of milliseconds by which the packet was delivered early. Values greater than zero indicate the number of milliseconds by which the packet was delivered late. Modules should update this attribute before calling `QTSS_Write` or `QTSS_WriteV` with an RTP packet.	Read/write	`SInt32`

QTSS_RTSPHeaderObject

A `QTSS_RTSPHeaderObject` is the collection of attributes that contain all of the header information sent by the client in an RSTP 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` (page 56) or `QTSS_GetValuePtr` (page 57).

QTSS_RTSPRequestObject

A `QTSS_RTSPRequestObject` is the collection of attributes that describe a particular RTSP request.Table 2-6 lists the attributes of the object `QTSS_RTSPRequestObject`.

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.

Note
All of the attributes for the object `QTSS_RTSPRequestObject` are preemptive safe, so they can be read by calling `QTSS_GetValue` (page 56) or `QTSS_GetValuePtr` (page 57). ◆

Table 2-6 Attributes of the object `QTSS_RTSPRequestObject`

Attribute Name and Content	Read/Write	Data Type
`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`.	Read	`char*`
`qtssRTSPReqMethodStr` The RTSP method of this request.	Read	`char*`
`qtssRTSPReqFilePath` URI for this request, converted to a local file system path.	Read	`char*`

continued

Table 2-6 Attributes of the object `QTSS_RTSPRequestObject` (continued)

Attribute Name and Content	Read/Write	Data Type
`qtssRTSPReqURI` URI for this request.	Read	`char*`
`qtssRTSPReqFilePathTrunc` Same as `qtssRTSPReqFilePath`, but without the last element of the path.	Read	`char*`
`qtssRTSPReqFileName` All characters after the last path separator in the file system path.	Read	`char*`
`qtssRTSPReqFileDigit` If the URI ends with one or more digits, this attribute points to those digits.	Read	`char*`
`qtssRTSPReqAbsoluteURL` The full URL starting with "rtsp://".	Read	`char*`
`qtssRTSPReqTruncAbsoluteURL` The URL without last element of the path.	Read	`char*`
`qtssRTSPReqMethod` The RTSP method as a value of type `QTSS_RTSPMethod`.	Read	`QTSS_RTSPMethod`
`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.	Read/write	`QTSS_RTSPStatusCode`
`qtssRTSPReqStartTime` The start time specified in the Range header of the PLAY request.	Read	`Float64`
`qtssRTSPReqStopTime` The stop time specified in the Range header of the PLAY request.	Read	`Float64`

continued

Table 2-6 Attributes of the object QTSS_RTSPRequestObject (continued)

Attribute Name and Content	Read/Write	Data Type
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.	Read/write	Bool16
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.	Read/write	char*
qtssRTSPReqRealStatusCode Same as the qtssRTSPReqStatusCode attribute but translated from a QTSS_RTSPStatusCode to an actual RTSP status code.	Read	UInt32
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.	Read	QTSS_StreamRef
qtssRTSPReqUserName The decoded user name, if provided by the RTSP request.	Read	char*
qtssRTSPReqUserPassword The decoded user password, if provided by the RTSP request.	Read	char*

continued

Table 2-6 Attributes of the object `QTSS_RTSPRequestObject` (continued)

Attribute Name and Content	Read/Write	Data Type
`qtssRTSPReqUserAllowed` Whether the user is authorized for this request. By default, the value of this attribute is `true`. Set this value to `false` if the request is denied. If the specified file is missing or invalid, maintain the default setting and allow the server to handle the error.	Read/write	`Bool16`
`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 "QT Streaming Server".	Read/write	`char *`
`qtssRTSPReqLocalPath`	Read/write	`char *`

QTSS_RTSPSessionObject

A `QTSS_RTSPSessionObject` is the collection of attributes that describe a particular RTSP session.

Table 2-7 lists the attributes for the object `QTSS_RTSPSessionObject`. These attributes are valid for all roles that receive a value of type `QTSS_RTSPSessionObject` in the structure the server passes to them.

Table 2-7 Attributes of the object `QTSS_RTSPSessionObject`

Attribute Name and Content	Read/Write	Data Type
`qtssRTSPSesID` An ID that uniquely identifies each RTSP session since the server started up.	Read	`UInt32`
`qtssRTSPSesLocalAddr` Local IP address for this RTSP session.	Read	`UInt32`
`qtssRTSPSesLocalAddrStr` Local IP address for the RTSP session in dotted-decimal format.	Read	`char*`
`qtssRTSPSesLocalDNS` DNS name that corresponds to the local IP address for this RTSP session.	Read	`char*`
`qtssRTSPSesRemoteAddr` The IP address of the client.	Read	`UInt32`
`qtssRTSPSesRemoteAddrStr` The IP address of the client in dotted-decimal format.	Read	`UInt32`

continued

Table 2-7 Attributes of the object `QTSS_RTSPSessionObject` (continued)

Attribute Name and Content	Read/Write	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.	Read	`QTSS_EventContextRef`
`qtssRTSPSesType` The RTSP session type. Possible values are `qtssRTSPSession`, `qtssRTSPHTTPSession` (an HTTP tunnelled RTSP session), and `qtssRTSPHTTPInputSession`. Sessions of type `qtssRTSPHTTPInputSession` are usually very short lived.	Read	`char*`
`qtssRTSPSesStreamRef` A `QTSS_StreamRef` used for sending data to the RTSP client.	Read	`QTSS_StreamRef`

QTSS_ServerObject

A `QTSS_ServerObject` is the collection of attributes that describe a particular QuickTime Streaming Server. Table 2-8 lists the attributes of the object `QTSS_ServerObject`. These attributes are valid for all roles that receive a value of type `QTSS_ServerObject` in the structure the server passes to them.

Some of these attributes are not preemptive safe, as noted in Table 2-8.

Table 2-8 Attributes of the object `QTSS_ServerObject`

Attribute Name and Content	Read/Write	Data Type
The following attributes are preemptive safe and can be read by `QTSS_GetValue` or `QTSS_GetValuePtr`:		
`qtssServerAPIVersion`	Read	`UInt32`
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.		
`qtssSvrDefaultDNSName`	Read	`char*`
The "default" DNS name of the server.		
`qtssSvrDefaultIPAddr`	Read	`UInt32`
The "default" IP address of the server.		
`qtssSvrServerName`	Read	`char*`
The name of the server.		
`qtssSvrServerVersion`	Read	`char*`
The version of the server.		
`qtssSvrServerBuildDate`	Read	`char*`
Date that the server was built.		
`qtssSvrRTSPPorts`	Read	`char*`
An indexed attribute containing all the ports the server is listening on.		
`qtssSvrRTSPServerHeader`	Read	`char*`
The Server header that the server uses when responding to RTSP clients.		

continued

Table 2-8 Attributes of the object `QTSS_ServerObject` (continued)

Attribute Name and Content	**Read/Write**	**Data Type**
The following attributes are not preemptive safe and cannot be read by `QTSS_GetValuePtr`:		

`qtssSvrState`
The current state of the server. Possible values are
`qtssStartingUpState`
`qtssRunningState`
`qtssRefusingConnectionsState`
`qtssFatalErrorState`
`qtssShuttingDownState`
Modules can set the server state. If a module sets the server state, the server responds accordingly.
Setting the server state to `qtssRefusingConnectionsState` 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.
	Read/write	`QTSS_ServerState`

`qtssSvrIsOutOfDescriptors` If the server has run out of file descriptors, this attribute is `true`; otherwise, this attribute is `false`.	Read	`Bool16`
`qtssRTSPCurrentSessionCount` The number of clients that are currently connected over standard RTSP.	Read	`UInt32`
`qtssRTSPHTTPCurrentSessionCount` The number of clients that are currently connected over RTSP/HTTP.	Read	`UInt32`
`qtssRTPSvrCurConn` The number of clients currently connected to the RTP server.	Read	`UInt32`

continued

Table 2-8 Attributes of the object `QTSS_ServerObject` (continued)

Attribute Name and Content	**Read/Write**	**Data Type**
`tssRTPSvrTotalConn` Total number of clients that have connected to the RTP server since the server started up.	Read	`UInt32`
`qtssRTPSvrNumUDPSockets` The number of UDP sockets the server is using.	Read	`UInt32`
`qtssRTPSvrCurBandwidth` Current bandwidth being output by the RTP server in bits per second.	Read	`UInt32`
`qtssRTPSvrTotalBytes` Total number of bytes output since the RTP server started up.	Read	`UInt64`
`qtssRTPSvrAvgBandwidth` Average bandwidth output by the RTP server in bits per second.	Read	`UInt32`
`qtssRTPSvrCurPackets` Current packets per second being output by the RTP server.	Read	`UInt32`
`qtssRTPSvrTotalPackets` Total number of bytes output since the RTP server started up.	Read	`UInt64`

Index