



---

# QuickTime Streaming Server Modules



© 2001 Apple Computer, Inc.



Apple Computer, Inc.  
© 1999-2001 Apple Computer, Inc.  
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Computer, Inc., except to make a backup copy of any documentation provided on CD-ROM.

The Apple logo is a trademark of Apple Computer, Inc. Use of the “keyboard” Apple logo (Option-Shift-K) for commercial purposes without the prior written consent of Apple may constitute trademark infringement and unfair competition in violation of federal and state laws.

No licenses, express or implied, are granted with respect to any of the technology described in this book. Apple retains all intellectual property rights associated with the technology described in this book. This book is intended to assist application developers to develop applications only for Apple-labeled or Apple-licensed computers.

Every effort has been made to ensure that the information in this manual is accurate. Apple is not responsible for typographical errors.

Apple Computer, Inc.  
1 Infinite Loop  
Cupertino, CA 95014  
408-996-1010

Apple, the Apple logo, and Macintosh are trademarks of Apple Computer, Inc., registered in the United States and other countries.

Adobe, Acrobat, and PostScript are trademarks of Adobe Systems Incorporated or its subsidiaries and may be registered in certain jurisdictions.

Helvetica and Palatino are registered trademarks of Linotype-Hell AG and/or its subsidiaries.

ITC Zapf Dingbats is a registered trademark of International Typeface Corporation.

QuickView™ is licensed from Altura Software, Inc.

Simultaneously published in the United States and Canada.

Even though Apple has reviewed this manual, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS MANUAL, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS MANUAL IS SOLD “AS IS,” AND YOU, THE PURCHASER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS MANUAL, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

# Contents

Figures, Tables, and Listings   vii

## Preface   **About This Manual   9**

---

What's New Since Version 2.0   9  
Conventions Used in This Manual   12  
For More Information   13

## Chapter 1   **About QuickTime Streaming Server Modules   15**

---

Building a QuickTime Streaming Server Module   16  
    Compiling a QTSS Module into the Server   16  
    Building a QTSS Module as a Code Fragment   17  
Module Requirements   17  
    Main Routine   17  
    Dispatch Routine   18  
Overview of QuickTime Streaming Server Operations   19  
    Server Startup and Shutdown   19  
    RTSP Request Processing   21  
Runtime Environment for QTSS Modules   25  
    Server Time   26  
Naming Conventions   27  
Module Roles   27  
    Register Role   29  
    Initialize Role   30  
    Shutdown Role   31  
    Reread Preferences Role   31  
    Error Log Role   32  
RTSP Roles   33  
    RTSP Filter Role   33  
    RTSP Route Role   34  
    RTSP Preprocessor Role   36  
    RTSP Request Role   37

RTSP Postprocessor Role	38
RTP Roles	39
RTP Send Packets Role	40
Client Session Closing Role	40
RTCP Process Role	41
QTSS Objects	43
Getting Attribute Values	45
Setting Attribute Values	47
Adding Attributes	48
QTSS Streams	50
QTSS Services	52
Built-in Services	54
Using Files	55
Reading Files Using Callback Routines	55
Implementing a QTSS File System Module	57
File System Module Roles	59
Sample Code for the Open File Role	65
Implementing Asynchronous Notifications	67
Using QTSS Web Admin	68
CGIs, Template HTML Files, and Special Tags	68
ECHODATA	69
GETDATA	70
GETVALUE	70
MAKEARRAY	71
HASVALUE	71
IFVALUEEQUALS	72
CONVERTTOLOCALTIME	72
ACTIONONDATA	72
FORMATFLOAT	73
CONVERTMSECTIMETOSTR	74
MODIFYDATA	74
PRINTFILE	75
PRINHTHTMLFORMATFILE	75
PROCESSFILE	76
HTMLIZE	76
Monitoring Server Status and Modifying Server Settings	77
Customizing Web Admin	78
Admin Protocol	79

Request and Response Methods	79
Session State	79
Supported Request Header Features	79
Server Data Access	79
Request Syntax	80
Query Functionality	80
Data References	81
Query Options	81
Command Options	82
Parameter Options	83
Access Types	83
Data Types	84
Responses	84
Changing Server Settings	90
Special Paths	90

---

## Chapter 2    QuickTime Streaming Server Module Reference    93

---

QTSS Callback Routines	93
QTSS Utility Callback Routines	93
QTSS Attribute Callback Routines	97
Stream Callback Routines	114
File System Callback Routines	121
Service Callback Routines	123
RTSP Header Callback Routines	125
RTP Callback Routines	129
QTSS Data Types	133
QTSS Objects	133
Other QTSS Data Types	163

## Index    173

---



# Figures, Tables, and Listings

Chapter 1	About QuickTime Streaming Server Modules	15
<b>Figure 1-1</b>	QuickTime Streaming Server startup and shutdown	20
<b>Figure 1-2</b>	Sample RTSP request	21
<b>Figure 1-3</b>	Summary of RTSP request processing	22
<b>Figure 1-4</b>	Summary of the RTSP Preprocessor and RTSP Request roles	25
<b>Table 1-1</b>	Module roles	28
<b>Table 1-2</b>	Streams and appropriate callback routines	52
<b>Listing 1-1</b>	Sample code that calls QTSS_GetValue	46
<b>Listing 1-2</b>	Sample code that calls QTSS_GetValuePtr	46
<b>Listing 1-3</b>	Sample code for getting the value of the qtssRTPSvrCurConn attribute	47
<b>Listing 1-4</b>	Sample code for setting the value of the qtssRTSPReqRootDir attribute	48
<b>Listing 1-5</b>	Sample code for adding a static attribute	49
<b>Listing 1-6</b>	Sample code for starting a service	54
<b>Listing 1-7</b>	Sample code for reading an entire file	56
<b>Listing 1-8</b>	Sample code for handling the Open File role	65
Chapter 2	QuickTime Streaming Server Module Reference	93
<b>Table 2-1</b>	Role constants	95
<b>Table 2-2</b>	QTSS_SendStandardRTSPResponse method responses	128
<b>Table 2-3</b>	Attributes of the object type QTSS_AttrInfoObject	134
<b>Table 2-4</b>	Attributes of the object QTSS_ClientSessionObject	135
<b>Table 2-5</b>	Attributes of the object QTSS_FileObject	138
<b>Table 2-6</b>	Attributes of the QTSS_ModuleObject object	140
<b>Table 2-7</b>	Attributes of the object QTSS_PrefsObject	141
<b>Table 2-8</b>	Attributes of the object QTSS_RTPStreamObject	149
<b>Table 2-9</b>	Attributes of the object QTSS_RTSPRequestObject	153
<b>Table 2-10</b>	Attributes of the object QTSS_RTSPSessionObject	157
<b>Table 2-11</b>	Attributes of the object QTSS_ServerObject	159





# About This Manual

---

This manual describes version 3.0 of the programming interface for creating QuickTime Streaming Server modules. The QTSS programming interface provides an easy way for developers to add new functionality to the QuickTime Streaming Server. This version of the programming interface is compatible with QuickTime Streaming Server version 3.0.

## What's New Since Version 2.0

---

Since version 2.0 of the programming interface for QTSS the following features have been added:

- Callbacks allowing third-party developers to write QTSS modules that manipulate file systems and databases. Modules that read media files should always use these callbacks for manipulating files. The file system callbacks isolate QTSS modules from file system specific details, so it is a good practice in general for third-party QTSS modules to use the file system callbacks even when working with non-media files.
- Support for instance attributes. For example, you can add an instance attribute to a single RTSP session instead of adding that attribute to all RTSP sessions. Instance attributes can be added to an instance of a particular object and are present only for that particular instance of the object. In previous versions of the QTSS programming interface, when a module added an attribute to an object type, the attribute was present in all instances of the object type. In QTSS version 3.0, that type of attribute is called a static attribute. A module can add static attributes to object types only from its Register role; a module can add and remove instance attributes from any role. All built-in attributes are static attributes. New callbacks have been added to the programming interface to support static and instance attributes:
  - `QTSS_AddStaticAttribute`, which supersedes `QTSS_AddAttribute`. For compatibility, `QTSS_AddAttribute` remains in the programming interface, but you should call `QTSS_AddStaticAttribute` instead.
  - `QTSS_AddInstanceAttribute`

- QTSS\_RemoveInstanceAttribute

- All attributes now have a data type, which gives the server and modules enough information about an attribute to handle the attribute properly without having specific knowledge of the attribute. Because each attribute has an associated data type, the server can format any value as a string and convert any string to a value. The following callbacks have been added to support attribute data types:

- QTSS\_GetValueAsString
- QTSS\_TypeStringToType
- QTSS\_TypeToTypeString
- QTSS\_StringToValue
- QTSS\_ValueToString

- A new object type, `qtssModuleObjectType`, is a collection of attributes that store information about the module, including

- `qtssModName`, which contains the module's name.
- `qtssModDesc`, which contains a text description of what the module does
- `qtssModVersion`, which contains the module's version number
- `qtssModRoles`, which contains a list of all the roles for which the module has registered.
- `qtssModPrefs`, which contains a `QTSS_ModulePrefsObject`, whose attributes contain the preferences for this module. All attributes of a `QTSS_ModulePrefsObject` object are instance attributes. All modifications to objects of this type are persistent between invocations of the server because the server writes the contents of these attributes to a configuration file and reads them when it restarts. The single instance of the `qtssPrefsObjectType`, which formerly stored module preferences, now only stores core server preferences.

Each module receives its own module object in the module's Initialize role. Modules can get information about other loaded modules by accessing the value of their module object attributes through the new `qtssSvrModuleObjects` attribute of the server object (`qtssServerObjectType`).

- A new object type, `qtssAttrInfoObjectType`, which is a collection of attributes that describes an attribute: the attribute's name, its attribute ID, its data type, and its permissions (readable, writable, and whether it is preemptive safe). There is one attribute information object for each attribute. The following callbacks have been added to make it easy to get the attribute information object for static and instance attributes:

- QTSS\_GetAttrInfoByName
- QTSS\_GetAttrInfoByID
- QTSS\_GetAttrInfoByIndex

■ **New attributes for the QTSS\_ClientSessionObject object type:**

- qtssCliSesReqQueryString, which contains the text following a '?' in the URL) from the request that created this client session
- qtssCliSesCurrentBitRate, which contains the current bit rate in bits per second of all the streams on this session.
- qtssCliSesPacketLossPercent, which is the current percentage of packet loss.
- qtssCliSesTimeConnectedInMsec, which is the time in milliseconds that the client has been connected.
- qtssCliSesCounterID, which is a unique, nonrepeating ID for this session.

■ **New attribute for the QTSS\_RTPStreamObject object type:**

- qtssRTPStrTransportType, which is the transport that is being used for this stream.

■ **New attributes for the QTSS\_RTSPRequestObject object type:**

- qtssRTSPReqContentLen, which contains the length of the incoming RTSP request body.
- qtssRTSPReqSpeed, which contains the value of the speed header, converted to a value of type Float32.
- qtssRTSPReqLateTolerance, which contains the value of the late-tolerance field of the x-RTP-Options header, or -1 if not present.

■ **New attributes for the QTSS\_ServerObject object type:**

- qtssSvrModuleObjects, which consists of a module object containing information about each module
- qtssSvrStartupTime, which contains the time the server started up.
- qtssSvrGMTOffsetInHrs, which contains the server time zone (an offset from GMT in hours).
- qtssSvrDefaultIPAddrStr, which contains the “default” IP address of the server as a string.
- qtssSvrPreferences, which is an object representing each the server's preferences.

- `qtssSvrMessages`, which is an object containing the server's error messages.
- `qtssSvrCurrentTimeMilliseconds`, which contains the server's current time; retrieving this attribute is equivalent to calling `QTSS_Milliseconds`.
- `qtssSvrCPULoadPercent`, which contains as a percentage the server's current CPU usage.
- The `QTSS_TimeVal` data type has been added. The server's internal clock now counts the milliseconds that have elapsed since midnight on January 1, 1970 instead of since the server was started. Affected callback routines are.
  - `QTSS_Milliseconds()`, which now returns a value of type `QTSS_TimeVal`
  - `QTSS_MilliSecsTo1970Secs()`, which now takes a parameter of type `QTSS_TimeVal`

Attributes that were formerly of type `SInt64` are now of type `QTSS_TimeVal`. Affected attributes are

  - `qtssRTPStrTimeFlowControlLifted`
  - `qtssClisCreateTimeInMsec`
  - `qtssClisFirstPlayTimeInMsec`
  - `qtssClisPlayTimeInMsec`
  - `qtssClisAdjustedPlayTimeInMsec`
  - `qtssRTSPReqIfModSinceDate`

One structure is affected by this change: `QTSS_RTSPSendPackets_Params`.
- The `QTSS_RTSPAuthorize_Role` is deprecated. Modules that use the `QTSS_RTSPAuthorize_Role` that existed in previous versions of the QTSS programming interface will not work. You should update your source code and binaries to remove the `QTSS_RTSPAuthorize_Role` as soon as possible.
- QTSS Web Admin, which allows an administrator to use a browser to monitor the server's status, change server settings, and create and manage playlists, has been added.
- The Admin Protocol, which Web Admin uses to communicate with QTSS.

## 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>.

# P R E F A C E

# About QuickTime Streaming Server Modules

---

This document describes Version 3.0 of the programming interface for creating QuickTime Streaming Server (QTSS) modules. This version of the programming interface is compatible with QuickTime Streaming Server Version 3.0.

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

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

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

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

- acts as an RTSP proxy, which would be useful for a streaming clients located behind a firewall
- supports virtual hosting, allowing a single server to serve multiple domains from multiple document roots.

- logs statistical information for particular RTSP and client sessions
- supports additional ways of storing content, such as storing movies in databases
- configures 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 17).



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/local/sbin/StreamingServerModules` 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.

## About QuickTime Streaming Server Modules

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 16).

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 18).

**IMPORTANT**

For code fragment modules, the main routine must be named *MyModule\_Main* where *MyModule* is the name of the file that contains the module. ▲

## Dispatch Routine

---

Every QTSS module must provide a dispatch routine. The server calls the dispatch routine when it invokes a module for a specific task, passing to the dispatch routine the name of the task and a task-specific parameter block. (The programming interface uses the term “role” to describe specific tasks. For information about roles, see “Module Roles” (page 27).)

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

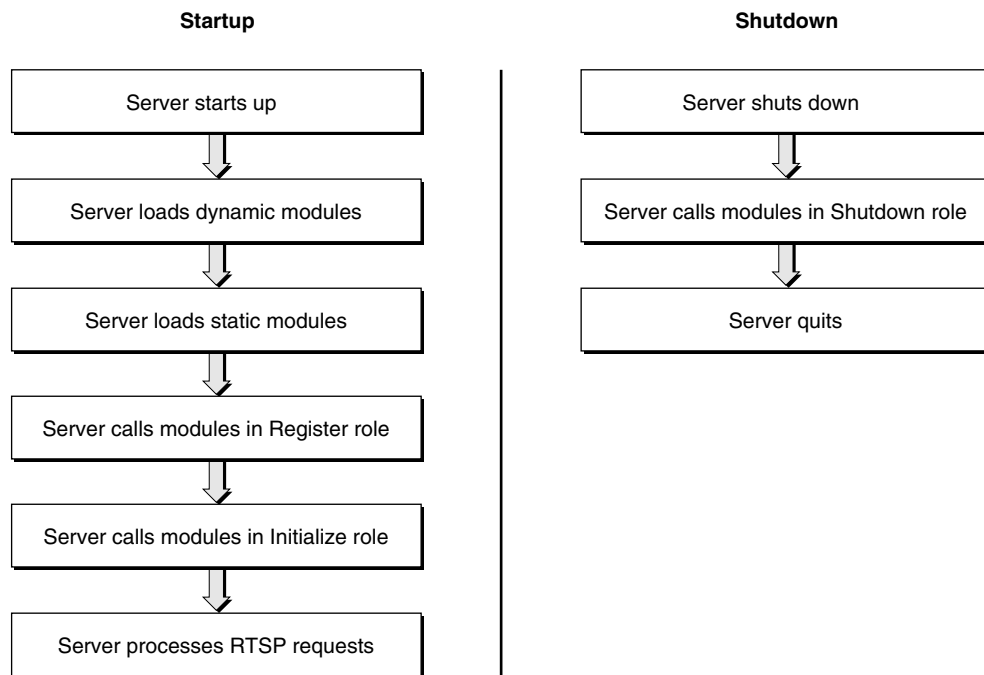
---

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

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 94) 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

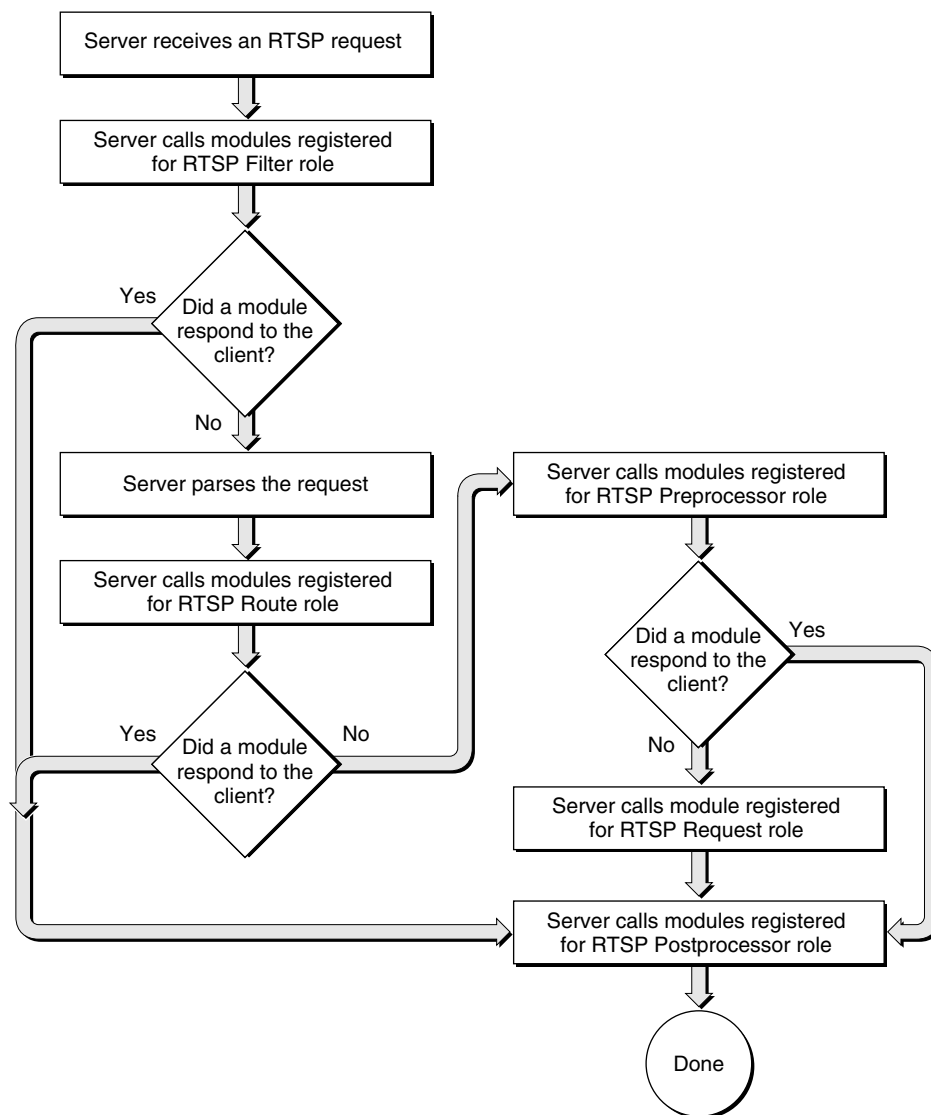
```
DESCRIBE rtsp://streaming.site.com/foo.mov RTSP/1.0
CSeq: 1
Accept: application/sdp
User-agent: QTS/1.0
```

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

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

## About QuickTime Streaming Server Modules

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 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 119) or `QTSS_WriteV` (page 120) to send data to the client. To send a standard response, the module can call `QTSS_SendStandardRTSPResponse` (page 127), or `QTSS_AppendRTSPHeader` (page 126) and `QTSS_SendRTSPHeaders` (page 126).

**IMPORTANT**

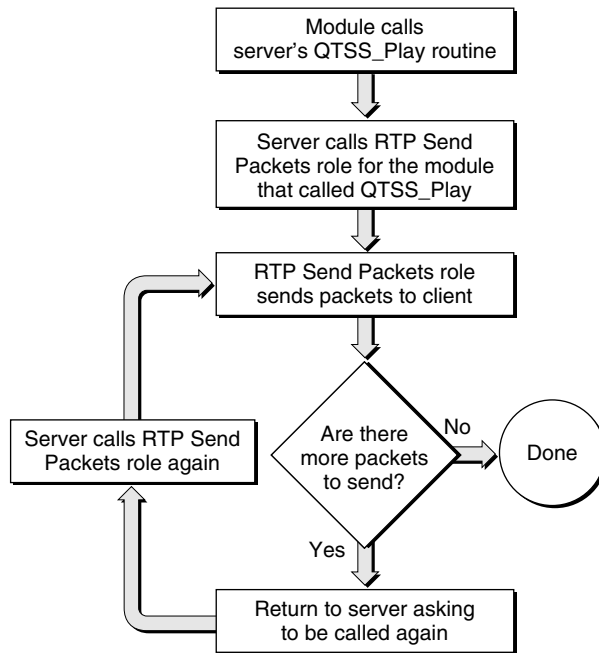
Any module handling the RTSP Preprocessor role that responds to the client causes the server to skip other modules that have registered for the RTSP Preprocessor role, skip modules that have registered for other RTSP roles, and immediately calls the RTSP Postprocessor role of the responding module. ▲

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 130), 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

The RTP Send Packets role calls `QTSS_Write` (page 119) or `QTSS_WriteV` (page 120) to send data to the client over the RTP session. When the RTP Send Packets role has sent some packets, it returns to the server and specifies the time that is to elapse before the server calls the module's RTP Send Packets role again. This cycle repeats until all of the packets for the media have been sent or until the client requests that the client session be paused or torn down.

## Runtime Environment for QTSS Modules

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

The QuickTime Streaming Server is fully multi-threaded, so QTSS modules must be prepared to be preempted. Global data structures and critical sections

in code should be protected with mutexes. Unless otherwise noted, assume that preemption can occur at any time.

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

QTSS modules should adhere to the following rules:

- Perform tasks and return control to the server as quickly as possible. Returning quickly allows the server to load balance among a large number of clients.
- Be prepared for `QTSS_WouldBlock` errors when performing stream I/O. The `QTSS_Write`, `QTSS_WriteV`, and `QTSS_Read` callback routines described in the section “QTSS Callback Routines” (page 93) return `QTSS_WouldBlock` if the requested I/O would block. For more information about streams, see “QTSS Streams” (page 50).
- Avoid using synchronous I/O wherever possible. An I/O operation that blocks may affect streaming quality for other clients.

## Server Time

---

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

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

To get the current value of the server’s clock, call `QTSS_Milliseconds` (page 97) or get the value of the `qtssSvrCurrentTimeMilliseconds` attribute of the server object (`QTSS_ServerObject`). To convert a time obtained from the server’s clock to the current time, call `QTSS_MilliSecsTo1970Secs` (page 97).

## 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 43).

Table 1-1 lists the roles that this version of the QuickTime Streaming Server supports.

**Table 1-1**      Module roles

Name	Constant	Task
Register role	QTSS_Register_Role	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 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
RTSP Postprocessor role	QTSS_RTSPPostProcessor_Role	Performs tasks, such as logging statistical information, after a request has been responded to
RTP Send Packets role	QTSS_RTPSendPackets_Role	Sends packets
Client Session Closing role	QTSS_ClientSessionClosing_Role	Performs tasks when a client session closes
RTCP Process role	QTSS_RTCPProcess_Role	Processes RTCP receiver reports
Open File Preprocess role	QTSS_OpenFilePreProcess_Role	Processes requests to open files

*continued*

**Table 1-1** Module roles (continued)

Name	Constant	Task
Open File role	QTSS_OpenFile_Role	Processes requests to open files that are not handled by the Open File Preprocess role
Advise File role	QTSS_AdviseFile_Role	Responds when a module (or the server) calls the QTSS_Advise callback for a file object
Read File role	QTSS_ReadFile_Role	Reads a file
Request Event File role	QTSS_RequestEventFile_Role	Handles requests for notification of when a file becomes available for reading
Close File role	QTSS_CloseFile_Role	Closes a file that was previously opened

For detailed information about the file roles, see “Implementing a QTSS File System Module” (page 57).

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 119) or QTSS\_WriteV (page 120).

#### 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 94) to tell the server the roles they support.

## About QuickTime Streaming Server Modules

Modules also use the Register role to call `QTSS_AddService` (page 123) to register services and to call `QTSS_AddStaticAttribute` (page 98) to add static attributes to QTSS object types. (QTSS objects are collections of attributes, each having a value.)

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

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

## Initialize Role

---

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

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

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

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

### Field descriptions

<code>inServer</code>	A <code>QTSS_ServerObject</code> object containing the server's global attributes and an attribute that contains information about all of the modules in the running server. For a description
-----------------------	--

## About QuickTime Streaming Server Modules

	of each attribute, see the section “QTSS_ServerObject” (page 158).
<code>inPrefs</code>	A <code>QTSS_PrefsObject</code> object containing the server’s preferences. For a description of each attribute, see the section “QTSS_ModuleObject” (page 139).
<code>inMessages</code>	A <code>QTSS_TextMessagesObject</code> object that a module can use for providing localized text strings.
<code>inErrorLogStream</code>	A <code>QTSS_ErrorLogStream</code> stream reference that a module can use to write to the server’s error log. Writing to this stream causes the module to be invoked in its Error Log role.
<code>inModule</code>	A <code>QTSS_ModuleObject</code> object that a module can use to store information about itself, including its name, version number, and a description of what the module does.

A module that wants to be called in the Initialize role must in its Register role call `QTSS_AddRole` (page 94) 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 94) 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

## About QuickTime Streaming Server Modules

a `SIGHUP` signal or when a module calls the Reread Preferences service described in the section “QTSS Services” (page 52).

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 94) 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 119) 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.



## About QuickTime Streaming Server Modules

A module that wants to be called in the Error Log role must in its Register role call `QTSS_AddRole` (page 94) 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 19).

### Note

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

## RTSP Filter Role

---

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

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

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

**Field descriptions**

<code>inRTSPSession</code>	The <code>QTSS_RTSPSessionObject</code> for this RTSP session. See the section “ <code>QTSS_RTSPSessionObject</code> ” (page 156) for information about RTSP session object attributes.
<code>inRTSPRequest</code>	The <code>QTSS_RTSPRequestObject</code> for this RTSP request. When called in the RTSP Filter role, only the <code>qtssRTSPReqFullRequest</code> attribute has a value. See the section “ <code>QTSS_RTSPRequestObject</code> ” (page 153) for information about RTSP request object attributes.
<code>outNewRequest</code>	A pointer to a location in memory.

The module calls `QTSS_GetValuePtr` (page 108) 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 96) 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 38).

A module that wants to be called in the RTSP Filter role must in its Register role call `QTSS_AddRole` (page 94) 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:

## About QuickTime Streaming Server Modules

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

**Field descriptions**

<code>inRTSPSession</code>	The <code>QTSS_RTSPSessionObject</code> for this RTSP session. See the section “ <code>QTSS_RTSPSessionObject</code> ” (page 156) for information about RTSP session object attributes.
<code>inRTSPRequest</code>	The <code>QTSS_RTSPRequestObject</code> for this RTSP request. In the Route role and all subsequent RTSP roles, all of the attributes are filled in. See the section “ <code>QTSS_RTSPRequestObject</code> ” (page 153) for information about RTSP request object attributes.
<code>inRTSPHeaders</code>	The <code>QTSS_RTSPHeaderObject</code> for the RTSP headers. See the section “ <code>QTSS_RTSPHeaderObject</code> ” (page 152) for information about RTSP header object attributes.
<code>inClientSession</code>	The <code>QTSS_ClientSessionObject</code> for the client session. See the section “ <code>QTSS_ClientSessionObject</code> ” (page 135) 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 38).

## About QuickTime Streaming Server Modules

A module that wants to be called in the RTSP Route role must in its Register role call `QTSS_AddRole` (page 94) and specify `QTSS_RTSPRoute_Role` as the role.

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

## RTSP Preprocessor Role

---

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

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

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

### Field descriptions

<code>inRTSPSession</code>	The <code>QTSS_RTSPSessionObject</code> for this RTSP session. See the section “ <code>QTSS_RTSPSessionObject</code> ” (page 156) for information about RTSP session object attributes.
<code>inRTSPRequest</code>	The <code>QTSS_RTSPRequestObject</code> for this RTSP request with a value for each attribute. See the section “ <code>QTSS_RTSPRequestObject</code> ” (page 153) for information about RTSP request object attributes.
<code>inRTSPHeaders</code>	The <code>QTSS_RTSPHeaderObject</code> for the RTSP headers. See the section “ <code>QTSS_RTSPHeaderObject</code> ” (page 152) for information about RTSP header object attributes.
<code>inClientSession</code>	The <code>QTSS_ClientSessionObject</code> for the client session. See the section “ <code>QTSS_ClientSessionObject</code> ” (page 135) 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`.

## About QuickTime Streaming Server Modules

If the request matches, the module handling the RTSP Preprocessor role responds to the request by calling `QTSS_SendStandardRTSPResponse` (page 127), `QTSS_Write` (page 119), or `QTSS_WriteV` (page 120), or by calling `QTSS_AppendRTSPHeader` (page 126) and `QTSS_SendRTSPHeaders` (page 126). If this module is also responsible for generating RTP packets for this client session, it should call `QTSS_AddRTPStream` (page 129) to add streams to the client session, and `QTSS_Play` (page 130), 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 94) 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

<code>inRTSPSession</code>	The <code>QTSS_RTSPSessionObject</code> for this RTSP session. See the section “ <code>QTSS_RTSPSessionObject</code> ” (page 156) for information about RTSP session object attributes.
----------------------------	---

## About QuickTime Streaming Server Modules

<code>inRTSPRequest</code>	The <code>QTSS_RTSPRequestObject</code> for this RTSP request with a value for each attribute. See the section “ <code>QTSS_RTSPRequestObject</code> ” (page 153) for information about RTSP request object attributes.
<code>inRTSPHeaders</code>	The <code>QTSS_RTSPHeaderObject</code> for the RTSP headers. See the section “ <code>QTSS_RTSPHeaderObject</code> ” (page 152) for information about RTSP header object attributes.
<code>inClientSession</code>	The <code>QTSS_ClientSessionObject</code> for the client session. See the section “ <code>QTSS_ClientSessionObject</code> ” (page 135) 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 126) and `QTSS_SendRTSPHeaders` (page 126), by calling `QTSS_SendStandardRTSPResponse` (page 127), or by calling `QTSS_Write` (page 119) or `QTSS_WriteV` (page 120).
- Preparing the `QTSS_ClientSessionObject` for streaming by using the RTP callbacks, such as `QTSS_AddRTPStream` (page 129) and `QTSS_Play` (page 130). 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 94) 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.

## About QuickTime Streaming Server Modules

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**

<code>inRTSPSession</code>	The <code>QTSS_RTSPSessionObject</code> for this RTSP session. See the section “ <code>QTSS_RTSPSessionObject</code> ” (page 156) for information about RTSP session object attributes.
<code>inRTSPRequest</code>	The <code>QTSS_RTSPRequestObject</code> for this RTSP request with a value for each attribute. See the section “ <code>QTSS_RTSPRequestObject</code> ” (page 153) for information about RTSP request object attributes.
<code>inRTSPHeaders</code>	The <code>QTSS_RTSPHeaderObject</code> for the RTSP headers. See the section “ <code>QTSS_RTSPHeaderObject</code> ” (page 152) for information about RTSP header object attributes.
<code>inClientSession</code>	The <code>QTSS_ClientSessionObject</code> for the client session. See the section “ <code>QTSS_ClientSessionObject</code> ” (page 135) 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 94) 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 130). It is the responsibility of the RTP Send Packets role to send media data to the client and tell the server when the module's RTP Send Packets role should be called again.

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

```
typedef struct
{
    QTSS_ClientSessionObject    inClientSession;
    Sint64                     inCurrentTime;
    QTSS_TimeVal                outNextPacketTime;
} QTSS_RTPSendPackets_Params;
```

`inClientSession`     The `QTSS_ClientSessionObject` for the client session. See the section “`QTSS_ClientSessionObject`” (page 135) 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 94) for that client session. The module calls `QTSS_Write` (page 119) or `QTSS_WriteV` (page 120) 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 94) 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.



## About QuickTime Streaming Server Modules

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**

<code>inReason</code>	The reason why the session is closing. The session may be closing because the client sent an RTSP teardown ( <code>qtssCliSesClosClientTeardown</code> ), because this session has timed out ( <code>qtssCliSesClosTimeout</code> ), or because the client disconnected without issuing a teardown ( <code>qtssCliSesClosClientDisconnect</code> ).
<code>inClientSession</code>	The <code>QTSS_ClientSessionObject</code> 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 94) 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.

## About QuickTime Streaming Server Modules

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**

<code>inRTPStream</code>	The <code>QTSS_RTPStreamObject</code> of the RTP stream that this RTCP packet belongs to. See the section “ <code>QTSS_RTPStreamObject</code> ” (page 148) for information about RTP stream object attributes.
<code>inClientSession</code>	The <code>QTSS_ClientSessionObject</code> for the client session. See the section “ <code>QTSS_ClientSessionObject</code> ” (page 135) for information about client session object attributes.
<code>inRTCPPacketData</code>	A pointer to a buffer containing the packets that are to be processed.
<code>inRTCPPacketDataLen</code>	The length of valid data in the buffer pointed to by <code>inRTCPPacketData</code> .

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 94) and specify `QTSS_RTCPProcess_Role` as the role.

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

## QTSS Objects

---

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

There are two attribute types:

- **static attributes.** Static attributes are present in all instances of an object type. A module can add static attributes to objects from its Register role only. All of the server's built-in attributes are static attributes. For information about adding static attributes to object types, see `QTSS_AddStaticAttribute` (page 98).
- **instance attributes.** Instance attributes are added to a specific instance of any object type. A module can use any role to add an instance attribute to an object and can also remove instance attributes that it has added to an object. For information about adding instance attributes to objects, see `QTSS_AddInstanceAttribute` (page 99).

### Note

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

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:

- `qtssAttrInfoObjectType` — Consists of attributes whose values describe an attribute: the attribute's name, attribute ID, data type, and permissions for reading and writing the attribute's value. There is one `QTSS_AttrInfoObject` for every attribute.
- `qtssModuleObjectType` — Consists of attributes whose values describe a particular QTSS module, such as its name and version number and a description of what the module does. For each module that it loads, the

server creates a module object and passes it to the module in its Initialize role. Modules can get information about other loaded modules by accessing the `qtssSvrModuleObject` attribute of the object type `qtssServerObjectType`. In addition to the attributes that store the module's name, version number and description, the object type `qtssModuleObjectType` object type has a module preferences attribute. The module preferences attribute itself is an object whose attributes store the module's preferences as instance attributes. All modifications to the module preferences object are persistent between invocations of the server because the contents of each module's module preferences object are written to the server's configuration file, which is read when the server starts up.

- `qtssRTPStreamObjectType` — Consists of 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 129). 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` — Consists of attributes associated with a client session, where a client session is defined as a single client streaming presentation.
- `qtssRTSPSessionObjectType` — Consists of 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` — Consists of 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` — Consists of attributes containing 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

## About QuickTime Streaming Server Modules

would read the value of the `Session` attribute in an RTSP header reference (`QTSS_RTSPHeaderObject`).

- `qtssServerObjectType` — Consists of global server attributes, such as server statistics. There is a single instance of this object type for each QuickTime Streaming Server.
- `qtssPrefsObjectType` — Consists of attributes that describe the server's internal preference storage system. The attribute values for this object are stored in the server's configuration file named `streamingserver.xml`. There is a single instance of this object type. In previous versions of QTSS, module preferences were stored in this object, but with QTSS version 3.0, module preferences are stored in the module's module object type (`qtssModuleObjectType`).
- `qtssTextMessageObjectType` — Consists of 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.

## 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 108), 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 106), 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 106) to get the value of the `qtssRTSPSvrCurConn` 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)
{
    // qtssRTSPSvrCurConn is a UInt32, so provide a UInt32 for the result.
    UInt32 theNumConnections = 0;

    // Pass in the size of the attribute value.
    UInt32 theLength = sizeof(theNumConnections);

    // Retrieve the value.
    QTSS_Error theErr = QTSS_GetValue(inServerObject, qtssRTSPSvrCurConn, 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 108), 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);
}

```

## About QuickTime Streaming Server Modules

```

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;
}

```

With QTSS version 3.0, you can obtain the value any attribute by calling `QTSS_GetValueAsString` (page 107), which gets the attribute's value as a C string. Calling `QTSS_GetValueAsString` is convenient when you don't know the type of data the attribute contains. In Listing 1-3, the value of the `qtssRTSPSvrCurConn` attribute is obtained as a string from the `QTSS_ServerObject`.

---

**Listing 1-3** Sample code for getting the value of the `qtssRTSPSvrCurConn` attribute

```

void MyPrintNumCurrentConnections(QTSS_ServerObject inServerObject)
{
    // Provide a string pointer for the result
    char* theCurConnString = NULL;

    // Retrieve the value as a string.
    QTSS_Error theErr = QTSS_GetValueAsString(inServerObject, qtssRTSPSvrCurConn, 0,
    &theCurConnString);

    if (theErr != QTSS_NoErr) return;

    // Print out the result. Because the value was returned as a string, use %s in the
    printf format.
    ::printf("Number of currently connected clients: %s\n", theCurConnString);

    // QTSS_GetValueAsString allocates memory, so reclaim the memory by calling
    QTSS_Delete.
    QTSS_Delete(theCurConnString);
}

```

## Setting Attribute Values

---

The sample code in Listing 1-4 would be found handling the Route role. It calls `QTSS_GetValuePtr` to get the value of the `qtssRTSPReqFilePath`. If the path

matches a certain string, the function sets a new request root directory by setting the `qtssRTSPReqRootDir` attribute to a new path.

---

**Listing 1-4** Sample code for setting the value of the `qtssRTSPReqRootDir` attribute

```
// First get the file path for this request using QTSS_GetValuePtr
char* theFilePath = NULL;
UInt32 theFilePathLen = 0;

QTSS_Error theErr = QTSS_GetValuePtr(inParams->inRTSPRequest, qtssRTSPReqFilePath, 0,
&theFilePath,
                                &theFilePathLen);

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

// See if this path is a match. If it is, use QTSS_SetValue to set the root directory
for this request.
if ((theFilePathLen == sStaticFilePathLen) &&
    (::strncmp(theFilePath, sStaticFilePath, theFilePathLen) == 0))
{
    theErr = QTSS_SetValue(inParams->inRTSPRequest, qtssRTSPReqRootDir, 0,
sNewRootDirString,
                        sNewRootDirStringLen);

    if (theErr != QTSS_NoErr) return;
}
```

## Adding Attributes

---

Any module can add an attribute to a QTSS object type by calling the `QTSS_AddStaticAttribute` (page 98) callback routine from its Register role. Modules can also call `QTSS_AddInstanceAttribute` (page 99) from any role to add an attribute to an instance of an object.



**Note**

Adding one or more attributes to an object type or to an instance of an object is the most efficient and the recommended way for modules to store data that is specific to a particular session. ♦

Once added, the new attribute is included in every object of that type that the server creates and its value can be set and obtained by calling that same callback routines that set and obtain the value of the server's built-in attributes: `QTSS_SetValue` (page 109), `QTSS_GetValue` (page 106), and `QTSS_GetValuePtr` (page 108).

The sample code in Listing 1-5 calls `QTSS_AddStaticAttribute` (page 98) to add an attribute to the object `QTSS_ClientSessionObject`.

**Listing 1-5** Sample code for adding a static attribute

```
QTSS_Error MyRegisterRoleFunction()
{
    // Add the static attribute. The third parameter is always NULL.
    QTSS_Error theErr = QTSS_AddStaticAttribute(qtssClientSessionObjectType,
        "MySampleAttribute", NULL, qtssAttrDataTypeUInt32);

    // Retrieve the ID for this attribute. This ID can be passed into QTSS_GetValue,
    // QTSS_SetValue, and QTSS_GetValuePtr.
    QTSS_AttributeID theID;
    theErr = QTSS_IDForAttr(qtssClientSessionObjectType, "MySampleAttribute", &theID);

    // Store the attribute ID in a global for later use. Attribute IDs do not
    // change while the server is running.
    gMyExampleAttrID = theID;
}
```

**Note**

Attribute permissions for an added attribute (static or instance) are automatically set to read, write, and preemptive safe. ♦

## QTSS Streams

---

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

Unless otherwise noted, all streams are asynchronous. When using the asynchronous QTSS file system callbacks, modules should be prepared to receive the `QTSS_WouldBlock` result code, subject to the restrictions and rules of each stream type described in this section. The `QTSS_WouldBlock` error is returned from a stream callback when completing the requested operation would require the current thread to block. For instance, `QTSS_Write` on a socket will return `QTSS_WouldBlock` if the socket is currently subject to flow control. For information on threading and asynchronous I/O, see the section “Runtime Environment for QTSS Modules” (page 25).

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

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

- `QTSS_ErrorLogStream` Used for writing binary data to the server’s error log.  
There is a single instance of this stream type, which is passed to each module in the Initialize role. When data is written to this stream, modules that have registered for the Error Log role are invoked. For information about this role, see the section “Error Log Role” (page 32). All operations on this stream type are synchronous.
- `QTSS_FileStream` Represents a file and is obtained by making the `QTSS_OpenFileStream` callback. If the file stream is opened

## About QuickTime Streaming Server Modules

with the `qtssFileStreamAsync` flag, callers should expect to receive a result code of `QTSS_WouldBlock` when they call `QTSS_Read`, `QTSS_Write`, and `QTSS_WriteV`.

`QTSS_RTSPSessionStream`

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

`QTSS_RTSPRequestStream`

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

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

`QTSS_SocketStream` Represents a socket. This stream type allows modules to use the QTSS stream event mechanism (`QTSS_RequestEvent`) for raw socket I/O. (In fact, the `QTSS_RequestEvent` callback

is the only stream callback available for this type of stream.) Modules should read sockets asynchronously and should use the operating system's socket function to read from and write to sockets. When those routines reach a blocking condition, the module can call `QTSS_RequestEvent` to be notified when the blocking condition has cleared.

Table 1-2 uses an "X" to summarize the I/O-related callback routines that are appropriate for each type of stream.

**Table 1-2** Streams and appropriate callback routines

<b>Stream Type</b>	<b>Read</b>	<b>Seek</b>	<b>Flush</b>	<b>Advise</b>	<b>Write</b>	<b>WriteV</b>	<b>Request Event</b>	<b>Signal Stream</b>
File Stream	X	X		X			X	X
Error Log					X			
Socket Stream							X	
RTSP Session Stream	X		X		X	X	X	
RTSP Request Stream	X		X		X	X	X	
RTP Stream	X		X		X	X		

## QTSS Services

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

## About QuickTime Streaming Server Modules

Modules use the callback routines described in the section “Service Callback Routines” (page 123) 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 123) 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 124) and providing the name of the service as a parameter. With the service’s ID, the module calls `QTSS_DoService` (page 125) to cause the service to run, as shown in Listing 1-6.

---

**Listing 1-6**      Sample code for starting a service

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

    MyServiceParamBlock theParamBlock;

    // Initialize service-specific parameters in the parameter block.
    theParamBlock.myArgument = xxx;
    QTSS_ServiceID theServiceID = qtssIllegalServiceID;

    // Get the service ID by providing the name of the service.
    QTSS_Error theErr = QTSS_IDForService('MyService', &theServiceID);

    if (theErr != QTSS_NoErr)
        return; // The service isn't available.

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

## Built-in Services

---

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

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

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

To invoke a built-in service, retrieve the service ID of the service by calling `QTSS_IDForService` (page 124). Then call `QTSS_DoService` (page 125) to run the service.

## Using Files

---

This version of QTSS supports file system modules so that QTSS can transparently and easily work with custom file systems. For example, a QTSS file system module can allow a QTSS module to read a custom networked file system or a custom database. Support for reading files consists of the following:

- QTSS file system callback routines that any module can use to open, read, and close files. Calling the file system callback routines is described in the section “Reading Files Using Callback Routines” (page 55). The QTSS file system callback routines allow QTSS to easily work with many different file system types. A QTSS module that uses the file system callbacks for reading all files can transparently use whatever file system is deployed on a server.
- File system roles for which modules that implement file systems register. These roles provide a bridge between QTSS and a specific file system. The file system roles are described in the section “Implementing a QTSS File System Module” (page 57). You could, for example, write a file system module that interfaces QTSS to a custom database or a custom networked file system.

### Reading Files Using Callback Routines

---

In QTSS, a file is represented by a QTSS stream, so you can use existing QTSS stream callback routines to read files. The callback routines that are available for working with files are:

- `QTSS_OpenFileObject`, which is called to open a file in the local operating system. This call is one of two callback routines that is only used when working with files.
- `QTSS_CloseFileObject`, which is called to close a file that was opened by a previous call to `QTSS_OpenFileObject`. This call is one of two callback routines that is only used when working with files.
- `QTSS_Read`, which is called to read data from a file object’s stream that was created by a previous call to `QTSS_OpenFileObject`.
- `QTSS_Seek`, which is called to set the current position of a file object’s stream.

- `QTSS_Advise`, which is called to tell a file system module that a specified section of one of its streams will be read soon.
- `QTSS_RequestEvent`, which is called to tell a file system module that the calling module wants to be notified when one of the events in the specified event mask occurs. The events are when a stream becomes readable and when a stream becomes writable.

In QTSS, a file is `QTSS_Object` that has its own object type, `QTSS_FileObject`, that allows you to use standard QTSS callbacks (`QTSS_GetValue` and `QTSS_GetValuePtr`) to get meta information about a file, such as its length and modification date. You can use standard QTSS callbacks to store any amount of file system meta information with the file object. For example, a module working with a POSIX file system would want to add an attribute to the file object that stores the POSIX file system descriptor.

A file object also contains a QTSS stream reference that can be used when calling QTSS stream routines that work with files, such as `QTSS_Read`. For more information, see the section “QTSS Streams” (page 50). For a list of the attributes of a QTSS file object, see the section `QTSS_FileObject` (page 138).

The sample code in Listing 1-7 shows how to open a file, determine the file’s length, read the entire file, close the file, and return the data it contains.

---

**Listing 1-7**      Sample code for reading an entire file

```
QTSS_Error ReadEntireFile(char* inPath, void** outData, UInt32* outDataLen)
{
    QTSS_Object theFileObject = NULL;
    QTSS_Error theErr = QTSS_OpenFileObject(inPath, qtssOpenFileNoFlags,
    &theFileObject);
    if (theErr != QTSS_NoErr)
        return theErr; // The file wasn't found or it couldn't be opened.

    // The file is open. Find out how long it is.
    UInt64* theLength = NULL;
    UInt32 theParamLen = 0;
    theErr = QTSS_GetValuePtr(theFileObject, qtssFlObjLength, 0, (void**)&theLength,
    &theParamLen);
```



```

if (theErr != QTSS_NoErr)
    return theErr;

if (theParamLen != sizeof(UInt64))
    return QTSS_RequestFailed;;

// Allocate memory for the file data.
*outData = new char[*theLength + 1];
*outDataLen = *theLength;

// Read the data
UInt32 recvLen = 0;
theErr = QTSS_Read(theFileObject, *outData, *outDataLen, &recvLen);

if ((theErr != QTSS_NoErr) || (recvLen != *outDataLen))
{
    delete *outData;
    return theErr;
}

// Close the file.
(void)QTSS_CloseFileObject(theFileObject);
}

```

## Implementing a QTSS File System Module

---

A file system module provides a way for QTSS modules to read files in a specific file system regardless of that file system's type. Typically, a file system module handles a subset of paths in a file system, but it may handle all paths on the system. If a file system module handles only a certain subset of paths, it usually handles all paths inside a certain root path. For example, a module handling files stored in a certain database may only respond to paths that begin with `/Local/database_root/`.

Implementing a QTSS file system module begins with registering for one of the following roles:

- **Open File Preprocess** role, which the server calls in response to a module (or the server) that calls the `QTSS_OpenFileObject` callback routine to open a file. **If the module does not handle files of the specified type, the module immediately returns `QTSS_FileNotFound`. If the module**

handles the files of the specified type, it opens the file, updates a file object provided by the server and returns `QTSS_NoErr`. If an error occurs during this setup period, the module returns `QTSS_RequestFailed`. Once the module returns `QTSS_NoErr`, it should be prepared to handle the Advise File, Read File, Request Event File and Close File roles for the opened file. The server calls each module registered in the Open File Preprocess role until one of the called modules returns `QTSS_NoErr` or `QTSS_RequestFailed`.

- **Open File role**, which the server calls in response to a module (or the server) that calls the `QTSS_OpenFileObject` callback routine for which all modules handling the Open File Preprocess role return `QTSS_FileNotFound`. Only one module can register for the Open File role. Like modules called for the Open File Preprocess role, the module called for the Open File role must determine whether it can handle the specified file. If it can, it opens the file, updates the file object provided by the server and returns `QTSS_NoErr`. If an error occurs during the setup process or if the module cannot handle the specified file, the module returns `QTSS_RequestFailed` or `QTSS_FileNotFound`, respectively.

A file system module should register in the Open File Preprocess role if it handles a subset of files available on the system. For instance, a file system module that serves files out of a database may only handle files rooted at a certain path. All other paths should fall through to other modules that handle other paths.

A file system module should register in the Open File role if it implements the default file system on a system. For instance, on a UNIX system the module handling the Open File Role would probably provide an interface between the server and the standard POSIX file system.

Once a module returns `QTSS_NoErr` from either the Open File Role or the Open File Preprocess role, it is responsible for the newly opened file. It should be prepared to handle the following roles on behalf of that file:

- **Advise File role**, which is called in response to a module (or the server) calling the `QTSS_Advise` callback for a file object. The `QTSS_Advise` callback is made to inform the file system module that a specific region of the file will be needed soon.
- **Read File role**, which is called in response to a module (or the server) calling the `QTSS_Read` callback for a file object. It is the responsibility of a file system module handling this role to make a best-effort attempt to fill the buffer provided by the caller with the appropriate file data.

## About QuickTime Streaming Server Modules

- Request Event File role, which is called in response to a module (or the server) calling the `QTSS_RequestEvent` callback on a file object.
- Close File role, which is called in response to a module (or the server) calling the `QTSS_Close` callback on a file object. The module should clean up any file-system and module-specific data structures for this file. This role is always the last role a file system module will be invoked in for a given file object.

**Note**

Modules do not need to explicitly register for the Advise File, Read File, Request Event File or Close File roles in order to handle them. Instead, returning `QTSS_NoErr` or `QTSS_RequestFailed` from one of the open file roles constitutes taking ownership for a specific file object, and therefore means that the module has implicitly registered for those roles. ♦

**File System Module Roles**

---

This section describes the file system module roles. The roles are:

- “Open File Preprocess Role” (page 59) which is called to process requests to open files.
- “Open File Role” (page 61) which is the default role that is called when none of the modules registered for the Open File Preprocess role opens the specified file.
- “Advise File Role” (page 62) which is called to tell a file system module about the caller’s I/O preferences.
- “Read File Role” (page 62) which is called to read a file.
- “Close File Role” (page 64) which is called to close a file.
- “Request Event File Role” (page 64) which is called to request notification when a file becomes available for reading or writing.

**Open File Preprocess Role**

---

The server calls the Open File Preprocess role in response to a module that calls the `QTSS_OpenFileObject` callback routine to open a file. It is the responsibility of a module handling this role to determine whether it handles the type of file

## About QuickTime Streaming Server Modules

specified to be opened. If it does and if the file exists, the module opens the file, updates the file object provided by the server, and returns `QTSS_NoErr`.

When called, an Open File Preprocess role receives a `QTSS_OpenFile_Params` structure, which is defined as follows:

```
typedef struct
{
    char*                inPath;
    QTSS_OpenFileFlags  inFlags;
    QTSS_Object          inFileObject;
} QTSS_OpenFile_Params;
```

**Field descriptions**

<code>inPath</code>	A pointer to a null-terminated C string containing the full path to the file that is to be opened.
<code>inFlags</code>	Open flags specifying whether the module that called <code>QTSS_OpenFileObject</code> can handle asynchronous read operations ( <code>qtssOpenFileAsync</code> ) or expects to read the file in order from beginning to end ( <code>qtssOpenFileReadAhead</code> ).
<code>inFileObject</code>	A QTSS object that the module updates if it can open the file specified by <code>inPath</code> .

If the file is a file the module handles, the module should do whatever work is necessary to open and set up the file. It can use `inFileObject` to store any module-specific information for that file. In addition, the module should set the value of the file object's `qtssF1ObjLength` and `qtssF1ObjModDate` attributes.

If the file is a file the module handles but an error occurs while attempting to set up the file, the module should return `QTSS_RequestFailed`.

If every module registered for the Open File Preprocess role returns `QTSS_FileNotFound`, the server calls the one module that is registered in the Open File role.

A module that wants to be called in the Open File Preprocess role must in its Register role call `QTSS_AddRole` (page 94) and specify `QTSS_OpenFilePreprocess_Role` as the role. Modules that register for this role must also handle the following roles, but they do not need to explicitly register for them: Advise File, Read File, Request Event File, and Close File.

**Open File Role**

The server calls the module registered for the Open File role when all modules registered for the Open File Preprocess role have been called and have returned `QTSS_FileNotFound`. Only one module can be registered for the Open File role, and that module is the first module that registers for this role when QTSS starts up.

Like modules called for the Open File Preprocess role, it is the responsibility of a module handling the Open File role to determine whether it handles the type of file specified to be opened. If it does and if the file exists, the module opens the file, updates the file object provided by the server, and returns `QTSS_NoErr`.

When called, the module receives a `QTSS_OpenFile_Params` structure, which is defined as follows:

```
typedef struct
{
    char*                inPath;
    QTSS_OpenFileFlags  inFlags;
    QTSS_Object          inFileObject;
} QTSS_OpenFile_Params;
```

**Field descriptions**

<code>inPath</code>	A pointer to a null-terminated C string containing the full path to the file that is to be opened.
<code>inFlags</code>	Open flags specifying whether the module that called <code>QTSS_OpenFileObject</code> can handle asynchronous read operations ( <code>qtssOpenFileAsync</code> ) or expects to read the file in order from beginning to end ( <code>qtssOpenFileReadAhead</code> ).
<code>inFileObject</code>	A QTSS object that the module updates if it can open the file specified by <code>inPath</code> .

If the file is a file the module handles, the module should do whatever work is necessary to open and set up the file. It can use `inFileObject` to store any module-specific information for that file. In addition, the module should set the value of the file object's `qtssFileObjLength` and `qtssFileObjModDate` attributes.

If the file is a file the module handles but an error occurs while attempting to set up the file, the module should return `QTSS_RequestFailed`.

A module that wants to be called in the Open File role must in its Register role call `QTSS_AddRole` (page 94) and specify `QTSS_OpenFile_Role` as the role. Modules that register for this role must also handle the following roles, but they do not

## About QuickTime Streaming Server Modules

need to explicitly register for them: Advise File, Read File, Request Event File, and Close File.

### Advise File Role

---

The server calls modules for the Advise File role in response to a module (or the server) calling the `QTSS_Advise` callback routine for a file object in order to inform the file system module that the calling module will soon read the specified section of the file.

When called, an Advise File role receives a `QTSS_AdviseFile_Params` structure, which is defined as follows:

```
typedef struct
{
    QTSS_Object      inFileObject;
    UInt64           inPosition;
    UInt32           inSize;
} QTSS_AdviseFile_Params;
```

#### Field descriptions

<code>inFileObject</code>	The file object for the opened file. The file system module uses the file object to determine the file for which the <code>QTSS_Advise</code> callback routine was called.
<code>inPosition</code>	The offset in bytes from the beginning of the file that represents the beginning of the section that is soon to be read.
<code>inSize</code>	The number of bytes that are soon to be read.

The file system module is not required to do anything while handling this role, but it may take this opportunity to read the specified section of the file.

File system modules do not need to explicitly register for this role.

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

### Read File Role

---

The server calls modules for the Read File role in response to a module (or the server) calling the `QTSS_Read` callback routine for a file object in order to read the specified file.

## About QuickTime Streaming Server Modules

When called, a Read File role receives a `QTSS_ReadFile_Params` structure, which is defined as follows:

```
typedef struct
{
    QTSS_Object inFileObject;
    UInt64      inFilePosition;
    void*       ioBuffer;
    UInt32      inBufLen;
    UInt32*     outLenRead;
} QTSS_ReadFile_Params;
```

**Field descriptions**

<code>inFileObject</code>	The file object for the file that is to be read. The file system module uses the file object to determine the file for which the <code>QTSS_Read</code> callback routine was called.
<code>inFilePosition</code>	The offset in bytes from the beginning of the file that represents the beginning of the section that is to be read. The server maintains the file position as an attribute of the file object, so the file system module does not have to cache the file position internally and can obtain the position at any time.
<code>ioBuffer</code>	A pointer to the buffer in which the file system module is to place the data that is read.
<code>inBufLen</code>	The length of the buffer pointed to by <code>ioBuffer</code> .
<code>outLenRead</code>	The number of bytes actually read.

The file system module should make a best-effort attempt to fill the buffer pointed to by `ioBuffer` with data from the file that is being read starting with the position specified by `inFilePosition`.

If the file was opened with the `qtssOpenFileAsync` flag, the module should return `QTSS_WouldBlock` if reading the data will cause the thread to block. Otherwise, the module should block the thread until all of the data has become available. When the buffer pointed to by `ioBuffer` is full or the end of file has been reached, the file system module should set `outLenRead` to the number of bytes read and return `QTSS_NoErr`.

If the read fails for any reason, the file system module handling this role should return `QTSS_RequestFailed`.

File system modules do not need to explicitly register for this role.

**Close File Role**

---

The server calls modules for the Close File role in response to a module (or the server) calling the `QTSS_CloseFile` callback routine for a file object in order to close a file that has been opened.

When called, a Close File role receives a `QTSS_CloseFile_Params` structure, which is defined as follows:

```
typedef struct
{
    QTSS_Object      inFileObject;
} QTSS_CloseFile_Params;
```

**Field descriptions**

<code>inFileObject</code>	The file object for the file that is to be closed. The file system module uses the file object to determine the file for which the <code>QTSS_Close</code> callback routine was called.
---------------------------	---

A module handling this role should dispose of any data structures that it has created for the file that is to be closed.

This role is always the last role for which a file system module will be invoked for any given file object.

File system modules do not need to explicitly register for this role.

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

**Request Event File Role**

---

The server calls modules for the Request Event File role in response to a module (or the server) calling the `QTSS_RequestEvent` callback routine. If a module or the server calls the `QTSS_OpenFileObject` callback routine and specifies the `qtssOpenFileAsync` flag, the file system module handling that file object may return `QTSS_WouldBlock` from its Read File role. When that occurs, the caller of `QTSS_Read` may call `QTSS_RequestEvent` callback to tell the server that the caller of `QTSS_Read` wants to be notified when the data becomes available for reading.

When called, a Request Event File role receives a `QTSS_RequestEventFile_Params` structure, which is defined as follows:



## About QuickTime Streaming Server Modules

```
typedef struct
{
    QTSS_Object      inFileObject;
    QTSS_EventType   inEventMask;
} QTSS_RequestEventFile_Params;
```

**Field descriptions**

<code>inFileObject</code>	The file object for the file for which notifications are requested. The file system module uses the file object to determine the file for which the <code>QTSS_RequestEvent</code> callback routine was called.
<code>inEventMask</code>	A mask specifying the type of events for which notification is requested. Possible values are <code>QTSS_ReadableEvent</code> and <code>QTSS_WriteableEvent</code> .

If the file system that the file system module is implementing supports notification, the file system module should do whatever setup is necessary to receive an event for the file for which the `QTSS_RequestEvent` callback routine was called. When the file becomes readable, the file system module should call the `QTSS_SignalStream` callback routine and pass the stream reference for this file object (which can be obtained through the file object's `qtssFLObjStream` attribute). Calling the `QTSS_SignalStream` callback routine tells the server that the caller of `QTSS_RequestEvent` should be notified that the file is now readable.

File system modules do not need to explicitly register for this role.

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

**Sample Code for the Open File Role**

The sample code in Listing 1-8 handles the Open File role, but it could also be used to handle the Open File Preprocess role. This code uses the POSIX file system layer as the file system and does not support asynchronous I/O.

**Listing 1-8** Sample code for handling the Open File role

```
QTSS_Error OpenFile(QTSS_OpenFile_Params* inParams)
{
    // Use the POSIX open call to attempt to open the specified file.
    // If it doesn't exist, return QTSS_FileNotFound
```

## About QuickTime Streaming Server Modules

```

    int theFile = open(inParams->inPath, O_RDONLY);
    if (theFile == -1)
        return QTSS_FileNotFound;

    // Use the POSIX stat call to get the length and the modification
date
    // of the file. This information must be set in the QTSS_FileObject
    // by every file system module.

    UInt64 theLength = 0;
    time_t theModDate = 0;
    struct stat theStatStruct;

    if (::fstat(fFile, &theStatStruct) >= 0)
    {
        theLength = buf.st_size;
        theModDate = buf.st_mtime;
    }
    else
    {
        ::close(theFile);
        return QTSS_RequestFailed; // Stat failed
    }

    // Set the file length and the modification date attributes of this
file
    // object before returning

    (void)QTSS_SetValue(inParams->inFileObject, qtssFIObjLength, 0,
&theLength, sizeof(theLength));
    (void)QTSS_SetValue(inParams->inFileObject, qtssFIObjModDate, 0,
&theModDate, sizeof(theModDate));

    // Place the file reference in a custom attribute in the
QTSS_FileObject.
    // This way, we can easily get the file reference in other role
handlers,
    // such as the QTSS_ReadFile_Role and the QTSS_CloseFile_Role.

```

## About QuickTime Streaming Server Modules

```

    QTSS_Error theErr = QTSS_SetValue(inParams->inFileObject,
    sFileRefAttr, 0,
        &theFile, sizeof(theFileSource));

    if (theErr != QTSS_NoErr)
    {
        ::close(theFile);
        return QTSS_RequestFailed;
    }

    return QTSS_NoErr;
}

```

## Implementing Asynchronous Notifications

---

If a module, or the server, calls the `QTSS_OpenFileObject` and specifies the `qtssOpenFileAsync` flag, the file system module handling that file object may return `QTSS_WouldBlock` from its `QTSS_ReadFile_Role` handler. Once that happens, the caller of `QTSS_Read` may want to be notified when the requested data becomes available for reading. This is possible by calling the `QTSS_RequestEvent` callback, which tells the server that the caller would like to be notified when data is available to be read from the file.

Not all file systems support notification mechanisms, and if they do, the notification mechanisms are particular to each file system architecture. Therefore, whether a file system module supports notifications is at the discretion of the developer of the file system module. In general it is better for a file system module to support asynchronous notifications and not block in `QTSS_Read_File_Role` because blocking on one file operation may disrupt service for many of the server's clients.

Two facilities allow file system modules to implement notifications:

- `QTSS_RequestEventFile_Role`, which is called in response to a module (or the server) calling the `QTSS_RequestEvent` callback on a file object. Modules do not need to explicitly register for this role. If a module doesn't implement asynchronous notifications, it should return `QTSS_RequestFailed` from this role. If a module does implement asynchronous notifications, it should do whatever setup is necessary to receive an event for this file when the file becomes readable.
- `QTSS_SendEventToStream` callback, called by a file system module when a file does become readable. Calling `QTSS_SendEventToStream` tells the server

that the caller of `QTSS_RequestEvent` should be notified that the file is now readable.

## Using QTSS Web Admin

---

QTSS Web Admin is a simple web server that uses the Admin Protocol to communicate with QTSS. QTSS administrators can use a web browser (Netscape 4.5 or higher or Internet Explorer 4.5 or higher) to connect to the Web Admin server to see the status of QTSS, to view the server's access and error logs, to change server settings, and to manage QTSS playlists.

The Web Admin server provides a library of Perl functions that are used to communicate with QTSS and to process data received from QTSS into the required format.

## CGIs, Template HTML Files, and Special Tags

---

To display data in a web browser, the Web Admin uses two Common Gateway Interfaces (CGIs) (`parse.cgi` and `parsewithinput.cgi`) and some template HTML files. The template HTML files contain regular HTML tags as well as special tags. When one of the CGIs parses a template HTML file, the special tags direct the CGI to perform special processing for the special tag while the regular HTML tags are left unchanged. The special tags work like server-side includes.

When the CGI finds a special tag, the CGI performs the function (found in `adminprotocol-lib.pl`) that corresponds to the tag and replaces the tag with the function's results. Sometimes the special tag tells the CGI to perform an operation and to save the results in a placeholder for later use. Then a second special tag can process the saved result if needed and display the result in the page.

The Perl function `ParseFile` in `adminprotocol-lib.pl` processes all tags. If you know Perl, you can easily add more tags.

All template html files are passed as arguments to the `parse.cgi` script (or the `parsewithinput.cgi` script for tags that have more advanced input values). The CGIs call the Perl functions from `adminprotocol-lib.pl` and send the appropriate CGI headers back. Most of the CGI headers are encapsulated in the `cgi-lib.pl` library. This library is easily extensible.

## About QuickTime Streaming Server Modules

The QTSS Web Admin server doesn't do authentication of its own, so the CGIs pipe the authentication challenge from QTSS to the browser, and the authorization back from the browser to QTSS. Thus a CGI should send an HTTP 401 unauthorized header back to the client if QTSS responds to an Admin Protocol request with an unauthorized header.

The following sections describe some of the tags that Web Admin supports. The tags are

- "ECHODATA" (page 69)
- "GETDATA" (page 70)
- "GETVALUE" (page 70)
- "MAKEARRAY" (page 71)
- "HASVALUE" (page 71)
- "IFVALUEEQUALS" (page 72)
- "CONVERTTOLOCALTIME" (page 72)
- "ACTIONONDATA" (page 72)
- "FORMATFLOAT" (page 73)
- "CONVERTMSECTIMETOSTR" (page 74)
- "MODIFYDATA" (page 74)
- "PRINTFILE" (page 75)
- "PRINTHTMLFORMATFILE" (page 75)
- "PROCESSFILE" (page 76)
- "HTMLIZE" (page 76)

## ECHODATA

---

The function for the `ECHODATA` special tag uses the Admin Protocol to get the value of the specified QTSS attribute from QTSS and parses the received contents to extract the value. The tag is replaced by this value in the resulting HTML.

**Usage:** `<%ECHODATA parameter%>`

where *parameter* is a partial path to the attribute. The string `"/modules/admin/"` is prepended to *parameter* when the tag is processed.

**Example:** `<%%ECHODATA /server/qtssSvrDefaultDNSName%%>`

This example replaces the `ECHODATA` tag with the value of the `/modules/admin/server/qtssSvrDefaultDNSName` attribute, which might be of the form “foo.bar.com”.

## GETDATA

---

The function for the `GETDATA` special tag uses the Admin Protocol to get the value of the specified attribute and stores the value in the specified variable for later use by another special tag. The tag itself is removed from the resulting HTML.

**Usage:** `<%% GETDATA variable parameter%%>`

where *variable* is the name of the variable in which the attribute is to be stored and *parameter* is a partial path to the attribute. The string “/modules/admin/” is prepended to *parameter* when the tag is processed.

**Example:** `<%%GETDATA rtspport /server/qtssSvrPreferences/rtsp_port/*%%>`

This example stores the contents of the `/modules/admin/server/qtssSvrPreferences/rtsp_port/*` attributes in the `rtspport` variable without processing the contents.

## GETVALUE

---

The function for the `GETVALUE` special tag uses the Admin Protocol to get the value of the specified attribute, processes the value, and stores the result in the specified variable for later use by another special tag. The tag itself is removed from the resulting HTML.

**Usage:** `<%% GETVALUE variable parameter%%>`

where *variable* is the name of the variable in which the processed attribute value is to be stored and *parameter* is a partial path to the attribute. The string “/modules/admin/” is prepended to *parameter* when the tag is processed.

**Example:** `<%%GETVALUE authscheme /server/qtssSvrPreferences/authentication_scheme%%>`

This example gets the value of the `/modules/admin/server/qtssSvrPreferences/authentication_scheme` attribute, processes the value, and stores the resulting value in the `authscheme` variable.

## MAKEARRAY

---

The function for the `MAKEARRAY` special tag processes the data that is accessed by *parameter* by looking for *container* and stores the elements in the array specified by *variable*. Later, *variable* can be used to reference the data stored in the array. The string `"/modules/admin/"` is prepended to *container* when the tag is processed.

**Usage:** `<%% MAKEARRAY variable container parameter%%>`

where *variable* is used to access the array that contains the values parsed out of the parameter. The container is used to look for the array data in the data accessed by parameter. Here *e* parameter is the accessor for data that has already been retrieved from the server.

**Example:** `<%%MAKEARRAY ports /server/qtssSvrPreferences/rtsp_port/rtspport%%>`

This example gets all of the values of the multivalued attribute `/server/qtssSvrPreferences/rtsp_port/` from data that was previously retrieved from the server and stored in `rtspport`. The reference to the resulting array of values is stored and can be accessed using the `ports` variable.

## HASVALUE

---

The function for the `HASVALUE` special tag searches for the value in the array specified the *parameter*. If the value is found, `"1"` is stored in the variable specified by *variable*; otherwise, `"0"` is stored.

**Usage:** `<%% HASVALUE variable parameter 'value' [num | alpha]%%>`

where *variable* is used to store the Boolean value that represents the result of testing whether the array specified by *parameter* contains the value specified by *value*. Single quotes must enclose *value*. If *value* is numeric, it must be followed by `num`, and if *value* is a string, it must be followed by `alpha`.

**Example:** `<%%HASVALUE has80 ports '80' num%%>`

This example searches for the numeric value 80 in the `ports` array, which was previously obtained from the server through the use of the `MAKEARRAY` special tag. If found, `has80` is set to 1; otherwise `has80` is set to zero.

## IFVALUEEQUALS

---

The function for the `IFVALUEEQUALS` special tag compares a value with a string. If they are equal, *variable* is set to `true`; otherwise, *variable* is set to `false`.

**Usage:** `<%% IFVALUEEQUALS variable parameter 'compareStr' %>`

where *parameter* contains the value that is compared with *compareStr*. The result of the operation is stored in *variable*, which you can use to format an HTML tag later in the page.

**Example:** `<%%IFVALUEEQUALS isBasic authscheme 'basic' %>`

The value of `authscheme` may have been retrieved earlier using the `GETVALUE` tag as in this example:

```
<%%GETVALUE authscheme /server/qtssSvrPreferences/
authentication_scheme %>
```

If the value of `authscheme` is `basic`, `IFVALUEEQUALS` sets `isBasic` to `true`. Otherwise, it sets `isBasic` to `false`.

## CONVERTTOLOCALTIME

---

The function for the `CONVERTTOLOCALTIME` special tag converts an elapsed time value (such as the server time) to the standard HTTP Date format. The tag is replaced by the HTTP Date string in the page.

**Usage:** `<%% CONVERTTOLOCALTIME parameter %>`

where *parameter* contains the elapsed time that is to be converted. The value stored in *parameter* must be in milliseconds elapsed from midnight on January 1, 1970. It may have been obtained by the `GETVALUE` special tag and may have been modified.

**Example:** `<%%CONVERTTOLOCALTIME curTime %>`

The parameter `curTime` contains a time value stored in milliseconds elapsed from midnight on January 1, 1970. It may have been retrieved using:

```
<%%GETVALUE curTime /server/qtssSvrCurrentTimeMilliseconds %>
```

## ACTIONONDATA

---

The function for the `ACTIONONDATA` special tag performs an arithmetic operation on two values.

**Usage:** `<%% ACTIONONDATA variable parameter1 parameter2 'operation' %>`



## About QuickTime Streaming Server Modules

where *parameter1* and *parameter2* contain values on which the operation is performed. The result is stored in *variable*.

The values stored in *parameter1* and *parameter2* were retrieved from QTSS using one of the tags described earlier, and possibly modified. The arithmetic operation is performed on the two values and the result is stored in *variable* for later use. Nothing is displayed in place of the tag.

**Example:** `<%%ACTIONONDATA diffTimeInmSec curTime startTime '-'%%>`

The parameters `curTime` and `startTime` already have values that may have been retrieved using the `GETVALUE` tag as in the following example:

```
<%%GETVALUE startTime /server/qtssSvrStartupTime%%>
<%%GETVALUE curTime /server/qtssSvrCurrentTimeMilliseconds%%>
```

This example subtracts `startTime` from `curTime` and stores the result in `diffTimeInmSec`.

## FORMATFLOAT

---

The function for the `FORMATFLOAT` special tag formats a value as a floating point number.

**Usage:** `<%% FORMATFLOAT parameter%%>`

where *parameter* contains a floating point number that was retrieved from QTSS. The result is displayed on the page and replaces the tag.

The value stored in *parameter* was retrieved from QTSS using one of the special tags described earlier and may have been modified. The value is formatted as a 3.2f, which results in a minimum total field of three digits, of which the last two digits hold the decimal part.

**Example:** `<%%FORMATFLOAT loadpercent%%>`

The parameter `loadpercent` is displayed as a floating point number with the above formatting, and maybe rounded to the nearest decimal if necessary. The value in `loadpercent` may have been retrieved using `GETVALUE`, as in the following example:

```
<%%GETVALUE loadpercent /server/qtssSvrCPULoadPercent%%>
```

## CONVERTMSECTIMETOSTR

---

The function for the `CONVERTMSECTIMETOSTR` special tag converts a time value in milliseconds to a string that displays the time in days, hours, minutes, and seconds. The value 95780003 would be displayed as "1 day, 2 hrs, 36 min, 20 sec".

**Usage:** `<%% CONVERTMSECTIMETOSTR parameter%%>`

where *parameter* contains a elapsed time value in milliseconds that was retrieved from QTSS using one of the tags described earlier and that may have been modified.

**Example:** `<%%CONVERTMSECTIMETOSTR diffTimeInmSec%%>`

The parameter `diffTimeInmSec` is displayed as a string formatted into days, hours, minutes, and seconds.

## MODIFYDATA

---

The function for the `MODIFYDATA` special tag modifies the value of a parameter if a condition is true.

**Usage:** `<%% MODIFYDATA parameter 'condition' 'operation'%%>`

where *parameter* contains a value that is checked for the condition. If the condition is true, the operation is performed, otherwise the value is left unchanged. The resulting value is displayed in place of the tag.

The value stored in *parameter* was retrieved from QTSS using one of the tags described earlier and may have been modified.

Example: `<%%MODIFYDATA maxThroughput '> 9999' '/' 1024'%%>`

The example causes the result of performing the following operation to be displayed on the page:

```
If (maxThroughput > 9999) {
    Result = maxThroughput / 1024;
}
else {
    Result = maxThroughput;
}
```

## PRINTFIL

---

The function for the `PRINTFIL` special tag opens the file specified by *parameter2* in the directory specified by *parameter1* and displays the contents on the page.

**Usage:** `<%% PRINTFIL parameter1 parameter2 %%>`

where *parameter1* contains the path to the directory and *parameter2* contains the filename. The tag is replaced by the contents of the file.

**Example:** `<%% PRINTFIL logdirpath logfilename %%>`

The contents of the file `logfilename` in the directory `logdirpath` are read and displayed in place of the tag. If the file cannot be read due to some error, the text “Server is not Running. Cannot display file” replaces the tag.

## PRINTHTMLFORMATFILE

---

The function for the `PRINTHTMLFORMATFILE` special tag processes a QTSS Error Log file. It appends “.log” to the filename specified by *parameter2*, opens the file of that name that resides in the directory specified by *parameter1*, and displays the contents on the page.

**Usage:** `<%% PRINTHTMLFORMATFILE parameter1 parameter2 %%>`

where *parameter1* contains the path to a directory and *parameter2* contains the filename.

Lines that start with a ‘#’ are comments and are displayed in bold text (that is, enclosed by `<B>` and `</B>` HTML tags. Lines that do not start with a ‘#’ are not displayed with bold text. Each line is followed by a line break (`<BR>`).

**Example:** `<%%PRINTHTMLFORMATFILE dir file%%>`

The contents of the file named `file.log` in the directory `dir` are read and displayed as described above. The two parameters, `dir` and `file`, may have been retrieved from the server using the following `GETVALUE` tags:

```
<%%GETVALUE dir /server/qtssSvrPreferences/error_logfile_dir%%>
<%%GETVALUE file /server/qtssSvrPreferences/error_logfile_name%%>
```

If the file cannot be read due to an error, the tag is replaced by a blank line.

## PROCESSFILE

---

The function for the `PRINTHTMLFORMATFILE` special tag processes a QTSS Access log file.

**Usage:** `<%% PROCESSFILE variable parameter1 parameter2 columnIndex%%>`

where *parameter1* contains the path to the directory and *parameter2* contains the filename. The function appends “.log” to the filename specified by *parameter2*, opens the file of that name that resides in the directory specified by *parameter1*, reads the column of data specified by *columnIndex*. It then counts the number of occurrences of each value in the column and stores the result in *variable* as a hash map with the values as the key and the number of occurrences of each as the value.

**Example:** `<%%PROCESSFILE accessdata dir file 4%%>`

The contents of the file `file.log` in the directory `dir` are parsed. The values in column 4 for each record are counted for the number of occurrences of each value. The two parameters, `dir` and `file`, may have been retrieved from the server using the `GETVALUE` tag as in the following example:

```
<%%GETVALUE dir /server/qtssSvrModuleObjects/QTSSAccessLogModule/qtssModPrefs/
request_logfile_dir%%>
<%%GETVALUE file /server/qtssSvrModuleObjects/QTSSAccessLogModule/qtssModPrefs/
request_logfile_name%%>
```

## HTMLIZE

---

The function for the `PRINTHTMLFORMATFILE` special tag converts certain characters to appropriate HTML representation.

**Usage:** `<%% HTMLIZE parameter%%>`

where *parameter* contains the text that has characters that need to be converted to values that can be displayed in a page.

The text in *parameter* is parsed and the characters `&`, `<`, and `>` are converted to `&amp`, `&lt`, and `&gt` respectively.

**Example:** `<%%HTMLIZE version%%>`

The text in *version* is parsed and any `&`, `<`, or `>` characters are converted to HTML representation. The text may have been retrieved using the `GETVALUE` tag as in the following example:

```
<%%GETVALUE version /server/qtssRTSPSvrServerVersion%%>
```

## Monitoring Server Status and Modifying Server Settings

---

QTSS Web Admin makes use of HTML forms to provide a way for the administrator to monitor the server's status and modify server settings. When a form is submitted, a CGI is executed that uses Perl functions to communicate with QTSS to perform the desired task.

The QTSS Web Admin provides the following HTML forms for monitoring server status:

- **Connected Users**, which for each connected user displays the IP address, bit rate, bytes sent, percentage of packet loss, the length of time for which the user has been connected, and the media file the user is receiving. This form also allows the administrator to change the number of connected users that the form displays, the page's update interval, and the column by which the information is sorted.
- **Server Snapshot**, which displays the following information:
  - ☐ The time at which QTSS was started
  - ☐ The elapsed time that QTSS has been running
  - ☐ The QuickTime Streaming Server's DNS name
  - ☐ The QuickTime Streaming Server's current time
  - ☐ The QuickTime Streaming Server's version number and the version number of its programming interface
  - ☐ The percentage of the system's CPU load that is being used by QTSS
  - ☐ The current number of connections
  - ☐ The QuickTime Streaming Server's current throughput
  - ☐ The total number of bytes QTSS has served
  - ☐ The total number of connections QTSS has served
  - ☐ Current server settings for the maximum number of connections, maximum throughput, movie folder path as well as the server's RTSP IP address and whether streaming on port 80 is enabled.
  - ☐ Current settings for whether access and error logging is enabled as well as the log file size and interval at which access and log error logs are rolled.

The QTSS Web Admin provides the following HTML forms for monitoring server logs:

- **Access History**, which displays the media files that have been requested and the number of requests for that media file.

- Error Log, which displays the contents of the error log.

The QTSS Web Admin provides the following HTML forms for modifying server settings:

- General Settings, which allows the administrator to change the following settings:
  - ☐ Movies directory
  - ☐ Authentication scheme (basic or digest)
  - ☐ Streaming on Port 80 (enabled or disabled)
  - ☐ Maximum number of connections
  - ☐ Maximum throughput in Mbps
  - ☐ Whether to start QTSS at system startup
  - ☐ QTSS administrator password
- Log Settings, which allows the administrator to change the following settings:
  - ☐ Whether error logging is enabled or disabled
  - ☐ How often to roll the error log file
  - ☐ Whether access logging is enabled or disabled
  - ☐ How often to roll the access log file
- Playlist Settings, which allows the administrator to create, modify, and delete playlists and stop, start, and pause playlists.

## Customizing Web Admin

---

QTSS Web Admin uses a scalable architecture that can be easily customized by anyone who has an understanding of HTML, JavaScript, and CGI scripting. Some knowledge of the Admin Protocol is necessary to add or modify pages that administer QTSS.

For example, you could add more tabs to the main menu bar (which currently has tabs for Status, Setting, and Logs) modifying `nav.htm`. The HTML and JavaScript is extensible so that more submenus can be added.

## Admin Protocol

---

The Admin Protocol relies on the URI mechanism as defined by RFC 2396 for specifying a container entity using a path and HTTP 1.0 RFC 1945 for specifying the request and response mechanisms.

The server's internal data is mapped to a hierarchical tree of element arrays. Each element is a named type including a container type for retrieval of sub-node elements.

The server state machine and database can be accessed through a regular expression. The Admin Protocol abstracts the QTSS module API to handle data access and in some cases to provide data access triggers for execution of server functions.

Four basic functions provide all of the administrative functions used by the server: add, set, del, and get.

Server streaming threads are blocked while the Admin Protocol accesses the server's internal data. To minimize blocking, the Admin Protocol allows scoped access to the server's data structures by allowing specific URL paths to any element.

## Request and Response Methods

---

HTTP GET is the current request and response method.

## Session State

---

The session is closed at the end of each HTTP request response.

## Supported Request Header Features

---

Authorization.

## Server Data Access

---

All data on the server is specified using a URI. The following URI example references the top level of the server's hierarchical data tree using a simple HTTP GET request.

```
GET /modules/admin
```

## Request Syntax

---

A valid request is an absolute reference followed by the server URI. An absolute reference is a path beginning with a forward slash character (/). A path represents a server's virtual hierarchical data structure of containers and is expressed as a URL.

Here the syntax for a request:

```
[absolute URL]?[parameters="values"]+[command="value"]+["option"="value"]
```

Here is an example:

```
GET /modules/admin/server/qtssSvrClientSessions?parameters=rt+command=get
```

\* causes each element in current URL location to be iterated.

*path*/\* is defined as all elements contained in the "path" container.

? indicates that options follow. Options are specified as *name*="value" pairs delimited by the plus (+) symbol.

Space and tab characters are stop characters.

Values can be enclosed by double quotation characters (" "). Enclosing double quotation characters are required for values that contain spaces and tabs.

These symbols are not supported in requests: period (.), two periods (..), and semicolon (;).

## Query Functionality

---

Queries can contain an array iterator, a name lookup, a recursive tree walk, and a filtered response. All functions can execute in a single URI query.

Here is an example of a query that gets the stream time scale and stream payload name from every stream in every session:

```
GET /modules/admin/server/qtssSvrClientSessions/*/
qtssCliSesStreamObjects?
parameters=r+command=get+filter1=qtssRTPStrTimescale+filter2=qtssRTPStrPa
yloadName
```

where

\* iterates the array of sessions

r in parameters=rt specifies a recursive walk and t specifies that data types are to be included in the result



## About QuickTime Streaming Server Modules

`filter1=qtssRTPStrTimescale` specifies that the stream time scale is to be returned

`filter2=qtssRTPStrPayloadName` specifies that the stream payload is to be returned

Here is an example of a query that gets all server module names and their descriptions:

```
GET/modules/admin/server/qtssSvrModuleObjects?
parameters=r+command=get+filter2=qtssModDesc+filter1=qtssModName
```

The following example does a full recursive search and gets all server attributes and their data types:

```
GET /modules/admin/server/?parameters=rt
```

Repeated recursive searches should be avoided because they impact server performance.

The following examples discover server attributes and their paths:

```
GET /modules/admin/server/*
```

```
GET /modules/admin/server/qtssSvrPreferences/*
```

## Data References

---

All elements are arrays. Single element arrays may be referenced in any of the following ways:

- *path/element*
- *path/element/*
- *path/element/\**
- *path/element/1*

The references listed above are all evaluated as the same query.

## Query Options

---

URIs that do not include a ? designator default to a Get request.

URIs that include a ? designator must have a "`command=command-option`" query option, where *command-option* is GET, SET, ADD, or DEL.

Query options are not case-sensitive. Except for command options, query option values are case-sensitive.

The following query options are ignored:

- unknown query options
- query options that a command does not require

### Command Options

---

The following options to the `command` query option are recognized:

- GET
- SET
- DEL
- ADD

Unknown commands are reported as errors.

#### The GET Command Option

---

The `GET` command option gets the data identified by the URI and does not require other query options.

**Example:** `GET /modules/admin/example_count`

The `SET` command option sets the data identified by the URI. No value checking is performed. Conversion between the text value and the actual value is type specific.

**Example:** `GET /modules/admin/example_count?command=SET+value=5`

If the `type` option is included in the command, type checking of the server element type and the set type is performed. If the types do not match, an error is returned and the command fails.

**Example:** `GET /modules/admin/maxcount?command=SET+value=5+type=SInt32`

#### The DEL Command Option

---

The `DEL` command deletes the element referenced by the URL and any data it contains. Here is an example:

`GET /modules/admin/maxcount?command=DEL`

#### The ADD Command Option

---

The `ADD` command adds the data specified by the URI to the specified element.

## About QuickTime Streaming Server Modules

If the element at the end of the URL is an element, the `ADD` command performs an add to the array of elements referenced by the element name. In this case, the following query options are required:

- `value`
- `type`

Here is an example:

```
GET /modules/admin/example_count?command=ADD+value=6+type=SInt16
```

If the element at the end of the URL is a `QTSS_Object` container, the `ADD` command option adds the element to the container. In this case, the following query options are required:

- `value`
- `type`
- `name`

Here is an example:

```
GET /modules/admin/?command=ADD+value=5+name=maxcount+type=SInt16
```

## Parameter Options

---

Parameter options are single characters without delimiters and follow the URL.

The following parameter options are recognized:

- `r` (for *recurse*). Walks downward in the hierarchy starting at end of the URL. Recursion should be avoided if `***` iterators or direct URL access to elements can be used.
- `v` (for *“verbose”*) — returns the full path in *name*.
- `a` (for *“access”*) — returns the access type.
- `t` (for *“type”*) — returns the data type of *value*.
- `d` (for *“debug”*) — returns debugging information if an error occurs.
- `c` (for *“count”*) — returns the count of elements in the path.

Here is an example:

```
GET /modules/admin/server/qtssSvrClientSessions?parameters=rt+command=get
```

## Access Types

---

The following access types are supported:

## About QuickTime Streaming Server Modules

- r (for “read”)
- w (for “write”)
- p (for “pre-emptive safe”)

## Data Types

---

Data types can be any server-allowed text value. New data types can be defined and returned by the server, so data types are not limited to the basic set listed here:

UInt8	SInt16	UInt64	Float64	CharArray
SInt8	UInt32	SInt64	Bool8	QTSS_Object
UInt16	SInt32	Float32	Bool16	void_pointer

Values of type `QTSS_Object`, pointers, and unknown data types always converted to a host-ordered string of hexadecimal values. Here is an example of a hexadecimal value result:

```
unknown_pointer=halogen; type=void_pointer
```

## Responses

---

This section describes the data that is returned in response to a request. The information on response data is organized in

- “Unauthorized Response” (page 84)
- “OK Response” (page 85)
- “Response Data” (page 85)
- “Array Values” (page 86)
- “Root Value” (page 87)
- “Errors in Responses” (page 87)

### Unauthorized Response

---

Here is an example of an unauthorized response:

### About QuickTime Streaming Server Modules

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Basic realm="QTSS/modules/admin"
Server: QTSS
Connection: Close
Content-Type: text/plain
```

#### OK Response

---

Here is an example of a “OK” response:

```
HTTP/1.0 200 OK
Server: QTSS/3.0 [v252]-Linux
Connection: Close
Content-Type: text/plain

Container="/"
admin/
error:(0)
```

All OK response end with `error:(0)`.

#### Response Data

---

In response data, all entity references follow this form:

```
[NAME=VALUE];[attribute="value"],[attribute="value"]
```

where brackets ([ ]) indicate that the enclosed response data is optional. Therefore, the response data may take the following forms:

```
NAME=VALUE
```

```
NAME=VALUE;attribute="value"
```

```
NAME=VALUE;attribute="value",attribute="value"
```

All container references follow this form:

```
[NAME/];[attribute="value"],[attribute="value"]
```

where brackets ([ ]) indicate that the enclosed response data is optional. Therefore, the response data may take the following forms:

```
NAME/
```

```
NAME/;attribute="value"
```

## About QuickTime Streaming Server Modules

*NAME;attribute="value",attribute="value"*

The order of appearance of container references and the container's entity references is important. This is especially true when the response is a recursive walk of a container hierarchy.

Each new level in the hierarchy must begin with `Container="reference"`. Each Container list of elements must be a complete list of the contained elements and any containers. The appearance of `Container="reference"` indicates the end of a previous container's contents and the beginning of a new container.

This example shows how each new container is identified with a unique path:

```
Container="/level1/"
field1="value"
field2="value"
level2a/
level2b/
Container="/level1/level2a/"
field1="value"
level3a/
level3b/
Container="/level1/level2a/level3a"
field1="value"
Container="/level1/level2a/level3b"
Container="/level1/level2b/"
field1="value"
level3a/
Container="/level1/level2b/level3a/"
field1="value"
```

### Array Values

---

For arrays of elements, a numerical value represents the index. Arrays are containers. Here is an example:

```
Container="/level1/"
field1="value"
field2="value"
array1/
Container="/level1/array1/"
1=value
2=value
```

### About QuickTime Streaming Server Modules

Array elements may be containers, as shown in this example:

```
Container="/level1/array1/"
1/
2/
3/

Container="/level1/array1/1/"
field1="value"
field2="value"
Container="/level1/array1/2/"
Container="/level1/array1/3/"
field1="value"
```

---

#### Root Value

The root for responses is `/admin`.

---

#### Errors in Responses

For each response, the error state for the request is reported at the end of the data. Here are some examples:

Error:(0) indicates that no error occurred

Error:(404) indicates that no data was found

The number enclosed by parentheses is an HTTP error code followed by an error string when debugging is turned on using the "parameters=d" query option. Here is an example:

```
error:(404);reason="No data found"
```

---

#### Request and Response Examples

An easy way to make requests is to use a web browser and this type of URL:

```
http://IP-address:554/modules/admin/?parameters=a+command=get
```

The following example uses basic authentication and shows the HTTP response headers.

**Request:** GET /modules/admin?parameters=a+command=get

**Authorization:** Basic QWxtaW5pT3RXYXRvcjXkZWZhdWx0

**Response:**

### About QuickTime Streaming Server Modules

```
HTTP/1.0 200 OK
Server: QTSS/3.0 [v252]-Linux
Connection: Close
Content-Type: text/plain
```

```
Container="/"
admin/;a=r
error:(0)
```

The following recursive request gets all values:

```
GET /modules/admin?command=get+parameters=r
```

The following recursive request returns the access and data type for each value:

```
GET /modules/admin?command=get+parameters=rat
```

The following request gets the elements in /modules/admin. Note that the command option is not required because query options are not present.

```
GET /modules/admin/*
```

A request like the following can be used to monitor the session list:

**Request:** GET /modules/admin/server/qtssSvrClientSessions/\*

**Response:**

```
Container="/admin/server/qtssSvrClientSessions/"
12/
2/
4/
8/
error:(0)
```

The response is a list of unique qtssSvrClientSessions session IDs.

The following request gets the indexes for qtssCliSesStreamObjects, which are an indexed array of streams:

**Request:** GET /modules/admin/server/qtssSvrClientSessions/\*/qtssCliSesStreamObjects/\*

**Response:**

```
Container="/admin/server/qtssSvrClientSessions/3/qtssCliSesStreamObjects/"
"
0/
1/
error:(0)
```



## About QuickTime Streaming Server Modules

**Request:** GET /modules/admin/server/qtssSvrClientSessions/3/  
qtssCliSesStreamObjects/0/\*

**Response:**

```
qtssRTPStrTrackID="4"
qtssRTPStrSSRC="683618521"
qtssRTPStrPayloadName="X-QT/600"
qtssRTPStrPayloadType="1"
qtssRTPStrFirstSeqNumber="-7111"
qtssRTPStrFirstTimestamp="433634204"
qtssRTPStrTimescale="600"
qtssRTPStrQualityLevel="0"
qtssRTPStrNumQualityLevels="3"
qtssRTPStrBufferDelayInSecs="3.000000"
qtssRTPStrFractionLostPackets="0"
qtssRTPStrTotalLostPackets="52"
qtssRTPStrJitter="0"
qtssRTPStrRecvBitRate="1526072"
qtssRTPStrAvgLateMilliseconds="501"
qtssRTPStrPercentPacketsLost="0"
qtssRTPStrAvgBufDelayInMsec="30"
qtssRTPStrGettingBetter="0"
qtssRTPStrGettingWorse="0"
qtssRTPStrNumEyes="0"
qtssRTPStrNumEyesActive="0"
qtssRTPStrNumEyesPaused="0"
qtssRTPStrTotPacketsRecv="6763"
qtssRTPStrTotPacketsDropped="0"
qtssRTPStrTotPacketsLost="0"
qtssRTPStrClientBufFill="0"
qtssRTPStrFrameRate="0"
qtssRTPStrExpFrameRate="3903"
qtssRTPStrAudioDryCount="0"
qtssRTPStrIsTCP="false"
qtssRTPStrStreamRef="18861508"
qtssRTPStrCurrentPacketDelay="-2"
qtssRTPStrTransportType="0"
qtssRTPStrStalePacketsDropped="0"
qtssRTPStrTimeFlowControlLifted="974373815109"
qtssRTPStrCurrentAckTimeout="0"
qtssRTPStrCurPacketsLostInRTCPInterval="52"
qtssRTPStrPacketCountInRTCPInterval="689"
```

## About QuickTime Streaming Server Modules

```

QTSSReflectorModuleStreamCookie=(null)
qtssNextSeqNum=(null)
qtssSeqNumOffset=(null)
QTSSSplitterModuleStreamCookie=(null)
QTSSFlowControlModuleLossAboveTol="0"
QTSSFlowControlModuleLossBelowTol="3"
QTSSFlowControlModuleGettingWorses="0"
error:(0)

```

Here is an example that returns the IP addresses of connected clients:

**Request:** /modules/admin/server/qtssSvrClientSessions/\*/  
qtssCliRTSPSessRemoteAddrStr

**Response:**

```

Container="/admin/server/qtssSvrClientSessions/5/"
"qtssCliRTSPSessRemoteAddrStr=17.221.40.1
Container="/admin/server/qtssSvrClientSessions/6/"
"qtssCliRTSPSessRemoteAddrStr=17.221.40.2
Container="/admin/server/qtssSvrClientSessions/8/"
"qtssCliRTSPSessRemoteAddrStr=17.221.40.3
Container="/admin/server/qtssSvrClientSessions/14/"
"qtssCliRTSPSessRemoteAddrStr=17.221.40.4
error:(0)

```

## Changing Server Settings

---

To change a server setting, the entity name and the value to be set are specified in the request body. If a match is made on the URL base and entity name at the current container level, and if the setting is writable, the value is set.

```

base = /base/container
name = value
/base/container/name="value"

```

## Special Paths

---

The special paths described in this section are useful for getting and setting preferences and for getting and setting the server's state. The paths are described in

- "Preferences Paths" (page 91)

## About QuickTime Streaming Server Modules

## ■ “Server State Path” (page 91)

**Preferences Paths**

---

Setting a server or module preference causes the preference’s new value to be flushed to the server’s XML preference file. The new value takes effect immediately.

Server preferences are stored in `/modules/admin/server/qtssSvrPreferences`.

Module preferences are stored in `/modules/admin/server/qtssSvrModuleObjects/*/qtssModPrefs/`.

The elements defined in `qtssSvrPreferences` can only be modified — they cannot be deleted.

The elements defined in `qtssModPrefs` can be added to, deleted, and modified.

A module or the server can automatically restore some deleted elements if the elements are needed by a module or the server. When applied to a `qtssModPrefs` element, the `ADD`, `DEL`, and `SET` commands cause the streaming server’s `.xml` file to be rewritten.

**Server State Path**

---

The `qtssSvrState` attribute controls the server’s state. The path is `/modules/admin/server/qtssSvrState`. It can be modified as a `UInt32` with the following values.

```
qtssStartingUpState    = 0,
qtssRunningState       = 1,
qtssRefusingConnectionsState = 2,
qtssFatalErrorState    = 3,
qtssShuttingDownState  = 4,
qtssIdleState          = 5
```

## CHAPTER 1

### About QuickTime Streaming Server Modules

# 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 93)
- “QTSS Attribute Callback Routines” (page 97)
- “Stream Callback Routines” (page 114)
- “Service Callback Routines” (page 123)
- “RTSP Header Callback Routines” (page 125)
- “RTP Callback Routines” (page 129)

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

- `QTSS_Milliseconds` (page 97) to get the current value of the server's internal timer.
- `QTSS_MilliSecsTo1970Secs` (page 97) to convert a value returned by `QTSS_Milliseconds` to the current time.
- `QTSS_New` (page 96) to allocate memory.
- `QTSS_Delete` (page 96) 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` (page 167) that specifies the role that is to be added.

*result*            A result code. Possible values are `QTSS_NoErr`, `QTSS_OutOfState` if `QTSS_AddRole` is called from a role other than the Register role, `QTSS_RequestFailed` if the module is registering for the RTSP Request role and a module is already registered for that role, and `QTSS_BadArgument` if the specified role does not exist.

### DISCUSSION

The `QTSS_AddRole` callback routine tells the server that your module can be called for the role specified by `inRole`.

The `QTSS_AddRole` callback can only be called from a module's Register role.

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

**Table 2-1**      Role constants

Role Constant	Description
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_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_RTSPSendPackets_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_OpenFilePreprocess_Role	Called to process requests to open files.
QTSS_OpenFile_Role	Called to open a file when all modules that have registered for the QTSS_OpenFilePreprocess_Role have returned QTSS_FileNotFound

## 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*

None.

### DISCUSSION

The `QTSS_New` callback routine allocates memory. QTSS modules should call `QTSS_New` whenever it needs to allocate memory dynamically.

To delete the memory that `QTSS_New` allocates, call `QTSS_Delete` (page 96).

## 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*

None.

### DISCUSSION

The `QTSS_Delete` callback routine deletes memory that was previously allocated by `QTSS_New` (page 96).



## QTSS\_Milliseconds

---

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

```
QTSS_TimeVal QTSS_Milliseconds();
```

*result*                The value of the server's internal clock in milliseconds since midnight January 1, 1970.

### DISCUSSION

The `QTSS_Milliseconds` callback routine gets the current value of the server's internal clock since midnight January 1, 1970. Unless otherwise noted, all millisecond values that the server provides in attributes are obtained from this clock.

## QTSS\_MilliSecsTo1970Secs

---

Converts a value obtained from the server's internal clock to the current time.

```
time_t QTSS_MilliSecsTo1970Secs(QTSS_TimeVal inQTSS_Milliseconds);
```

```
inQTSS_Milliseconds
```

On input, a value of type `QTSS_TimeVal` obtained by calling `QTSS_Milliseconds()`.

*result*                A value of type `time_t` containing the current time.

### DISCUSSION

The `QTSS_MilliSecsTo1970Secs` callback routine converts a value obtained by calling `QTSS_Milliseconds` (page 97) to the current time.

## QTSS Attribute Callback Routines

---

Modules call the following routines to work with attributes:

- `QTSS_AddStaticAttribute` (page 98) to add an attribute to an object type.

- `QTSS_AddStaticAttribute` (page 98) to add a static attribute to an object type.
- `QTSS_AddInstanceAttribute` (page 99) to add an instance attribute to an object.
- `QTSS_RemoveInstanceAttribute` (page 101) to remove an instance attribute from an object.
- `QTSS_IDForAttr` (page 105) to get the ID of an attribute.
- `QTSS_GetValue` (page 106) to get the value of an attribute.
- `QTSS_GetValuePtr` (page 108) to get a pointer to an attribute value.
- `QTSS_SetValue` (page 109) to set the value of an attribute.

## QTSS\_AddStaticAttribute

---

Adds a static attribute to an object type.

```
QTSS_Error QTSS_AddStaticAttribute(
    QTSS_ObjectType inObjectType,
    const char* inAttributeName,
    void* inUnused,
    QTSS_AttrDataType inAttrDataType);
```

**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 43).

**inAttributeName** On input, a pointer to a byte array that specifies the name of the attribute that is to be added.

**inUnused** Always `NULL`.

**QTSS\_AttrDataType** On input, a value of type `QTSS_AttrDataType` (page 164) that specifies the data type of the attribute that is being added.

**result** A result code. Possible values are `QTSS_NoErr`, `QTSS_OutOfState` if `QTSS_AddStaticAttribute` is called from a role other than the Register role, `QTSS_BadArgument` if the specified object type does not exist, the attribute name is too long, or a parameter is not specified, and `QTSS_AttrNameExists` if an attribute of the specified name already exists.

## DISCUSSION

The `QTSS_AddStaticAttribute` callback routine adds the specified attribute to all objects of the type specified by the `inType` parameter. The values of all added static attributes are implicitly readable, writable, and preemptive safe, so their values can be obtained by calling `QTSS_GetValuePtr` (page 108), `QTSS_GetValue` (page 106) or `QTSS_GetValueAsString` (page 107).

**Note**

Calling `QTSS_GetValuePtr` is the most efficient and recommended way to get the value of an attribute. Calling `QTSS_GetValueAsString` is even less efficient than calling `QTSS_GetValue`. ♦

The `QTSS_AddStaticAttribute` callback can only be called from the Register role. Call `QTSS_SetValue` (page 109) to set the value of an added attribute and `QTSS_RemoveValue` (page 113) to remove the value of an added attribute.

Once added, static attributes cannot be removed while the server is running.

**Note**

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

## QTSS\_AddInstanceAttribute

---

Adds an instance attribute to the instance of an object.

```
QTSS_Error QTSS_AddInstanceAttribute(
    QTSS_Object inObject,
    char* inAttrName,
    void* inUnused,
    QTSS_AttrDataType inAttrDataType);
```

<code>inObject</code>	On input, a value of type <code>QTSS_Object</code> (page 167) that specifies the object to which the instance attribute is to be added.
<code>inAttrName</code>	On input, a pointer to a byte array that specifies the name of the attribute that is to be added.

*inUnused* Always NULL.

*QTSS\_AttrDataType*

On input, a value of type `QTSS_AttrDataType` (page 164) that specifies the data type of the attribute that is being added.

*result*

A result code. Possible values are `QTSS_NoErr`, `QTSS_OutOfState` if `QTSS_AddStaticAttribute` is called from a role other than the Register role, `QTSS_BadArgument` if the specified object type does not exist, the attribute name is too long, or a parameter is not specified, and `QTSS_AttrNameExists` if an attribute of the specified name already exists.

## DISCUSSION

The `QTSS_AddInstanceAttribute` callback routine adds an attribute to the instance of an object as specified by the `inObject` parameter. All added instance attributes have values that are implicitly readable, writable, and preemptive safe, so their values can be obtained by calling `QTSS_GetValueAsString` (page 107) and `QTSS_GetValuePtr` (page 108). You can also call `QTSS_GetValue` (page 106) to get the value of an added static attribute, but doing so is less efficient.

The `QTSS_AddInstanceAttribute` callback can be called from any role. Typically, a module adds an instance attribute and sets its value by calling `QTSS_SetValue` (page 109) when it is first installed to add its default preferences to its module preferences object. On subsequent runs of the server, the preferences will already exist in the module's module preferences object, so the module only needs to call `QTSS_GetValue` (page 106), `QTSS_GetValueAsString` (page 107), or `QTSS_GetValuePtr` (page 108) to get the value.

### Note

Calling `QTSS_GetValuePtr` is the most efficient and recommended way to get the value of an attribute. Calling `QTSS_GetValueAsString` is even less efficient than calling `QTSS_GetValue`. ♦

Call `QTSS_RemoveValue` (page 113) to remove the value of an added attribute.

Unlike static attributes, instance attributes can be removed. To remove an instance attribute from the instance of an object, call `QTSS_RemoveInstanceAttribute` (page 101).

**Note**

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

## QTSS\_RemoveInstanceAttribute

---

Remove an instance attribute from the instance of an object.

```
QTSS_Error QTSS_RemoveInstanceAttribute(  
    QTSS_Object inObject,  
    QTSS_AttributeID inID);
```

<i>inObject</i>	On input, a value of type <code>QTSS_Object</code> (page 167) that specifies the object from which the instance attribute is to be removed.
<i>inID</i>	On input, a value of type <code>QTSS_AttributeID</code> (page 165) that specifies the ID of the attribute that is to be removed.
<i>result</i>	A result code. Possible values are <code>QTSS_NoErr</code> , <code>QTSS_BadArgument</code> if the specified object instance does not exist, and <code>QTSS_AttrDoesntExist</code> if the attribute doesn't exist.

### DISCUSSION

The `QTSS_RemoveInstanceAttribute` callback routine removes the attribute specified by the `inID` parameter from the instance of an object specified by the `inObject` parameter.

The `QTSS_RemoveInstanceAttribute` callback can be called from any role.

## QTSS\_GetAttrInfoByID

---

Uses an attribute ID to get information about an attribute.

```

QTSS_Error QTSS_GetAttrInfoByID(
    QTSS_Object inObject,
    QTSS_AttributeID inAttrID,
    QTSS_AttrInfoObject* outAttrInfoObject);

```

*inObject*      On input, a value of type `QTSS_Object` (page 167) that specifies the object having the attribute for which information is to be obtained.

*inAttrID*      On input, a value of type `QTSS_AttributeID` (page 165) that specifies the attribute for which information is to be obtained.

*outAttrInfoObject*      On output, a pointer to a value of type `QTSS_AttrInfoObject` (page 134) that can be used to get information about the attribute specified by *inAttrID*.

*result*      A result code. Possible values are `QTSS_NoErr`, `QTSS_BadArgument` if the specified object does not exist, and `QTSS_AttrDoesntExist` if the attribute doesn't exist.

### DISCUSSION

The `QTSS_GetAttrInfoByID` callback routine uses an attribute ID to get an `QTSS_AttrInfoObject` (page 134) that can be used to get the attribute's name, its data type, permissions for reading and writing the attribute's value, and whether getting the attribute's value is preemptive safe.

## QTSS\_GetAttrInfoByName

---

Uses an attribute's name to get information about an attribute.

```

QTSS_Error QTSS_GetAttrInfoByName(
    QTSS_Object inObject,
    char* inAttrName,
    QTSS_AttrInfoObject* outAttrInfoObject);

```

## QuickTime Streaming Server Module Reference

<code>inObject</code>	On input, a value of type <code>QTSS_Object</code> (page 167) that specifies the object having the attribute for which information is to be obtained.
<code>inAttrName</code>	On input, a pointer to a C string containing the name of the attribute for which information is to be obtained.
<code>outAttrInfoObject</code>	On output, a pointer to a value of type <code>QTSS_AttrInfoObject</code> (page 134) that can be used to get information about the attribute specified by <code>inAttrName</code> .
<i>result</i>	A result code. Possible values are <code>QTSS_NoErr</code> , <code>QTSS_BadArgument</code> if the specified object does not exist, and <code>QTSS_AttrDoesntExist</code> if the attribute doesn't exist.

## DISCUSSION

The `QTSS_GetAttrInfoByName` callback routine uses an attribute ID to get an `QTSS_AttrInfoObject` (page 134) that can be used to get the attribute's ID, its data type, and permissions for reading and writing the attribute's value, and whether getting the attribute's value is preemptive safe.

The `QTSS_GetAttrInfoByName` callback routine returns a `QTSS_AttrInfoObject` for both static and instance attributes.

## QTSS\_GetAttrInfoByIndex

---

Gets information about all of an object's attributes by iteration.

```
QTSS_Error QTSS_GetAttrInfoByIndex(
    QTSS_Object inObject,
    UInt32 inIndex,
    QTSS_AttrInfoObject* outAttrInfoObject);
```

<code>inObject</code>	On input, a value of type <code>QTSS_Object</code> (page 167) that specifies the object having the attribute for which information is to be obtained.
-----------------------	---

<i>inIndex</i>	On input, a value of type <code>UInt32</code> that specifies the index of the attribute for which information is to be obtained. Start by setting <i>inIndex</i> to zero. For the next call to <code>QTSS_GetAttrInfoByIndex</code> , increment <i>inIndex</i> by one to get information for the next attribute. Call <code>QTSS_GetNumAttributes</code> (page 104) to get the number of attributes that <i>inObject</i> has.
<i>outAttrInfoObject</i>	On output, a pointer to a value of type <code>QTSS_AttrInfoObject</code> (page 134) that can be used to get information about the attribute specified by <i>inAttrName</i> .
<i>result</i>	A result code. Possible values are <code>QTSS_NoErr</code> , <code>QTSS_BadArgument</code> if the specified object does not exist, and <code>QTSS_AttrDoesntExist</code> if the attribute doesn't exist.

## DISCUSSION

The `QTSS_GetAttrInfoByIndex` callback routine uses an attribute ID to get an `QTSS_AttrInfoObject` (page 134) that can be used to get the attribute's name and ID, its data type and permissions for reading and write the attribute's value.

The `QTSS_GetAttrInfoByIndex` callback routine returns a `QTSS_AttrInfoObject` for both static and instance attributes.

## QTSS\_GetNumAttributes

---

Gets a count of an object's attributes.

```
QTSS_Error QTSS_GetNumAttributes(
    QTSS_Object inObject,
    UInt32* outNumAttributes);
```

*inObject* On input, a value of type `QTSS_Object` (page 167) that specifies the object whose attributes are to be counted.

*outNumAttributes* On output, a pointer to a value of type `UInt32` that contains the count of the object's attributes. **(Reviewer's: starting at zero or one?).**



*result*                    A result code. Possible values are `QTSS_NoErr` and `QTSS_BadArgument` if the specified object does not exist.

## DISCUSSION

The `QTSS_GetNumAttributes` callback routine gets the number of attributes for the object specified by `inObject`. Having the number of attributes lets you know how often to call `QTSS_GetAttrInfoByIndex` (page 103) when getting information about each of an object's attributes.

## QTSS\_IDForAttr

---

Gets the ID of a static 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 43).

*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` (page 165). 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 static attribute in the specified object type. You can use the ID to obtain the value of the attribute by calling `QTSS_GetValue` (page 106) or `QTSS_GetValuePtr` (page 108).

To get the ID of an instance attribute, call the `QTSS_GetAttrInfoByName` (page 102) or the `QTSS_GetAttrInfoByIndex` (page 103) callback.

## 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);
```

<i>inObject</i>	On input, a value of type <code>QTSS_Object</code> (page 167) that specifies the object that contains the attribute whose value is to be obtained.
<i>inID</i>	On input, a value of type <code>QTSS_AttributeID</code> (page 165) that specifies the ID of the attribute whose value is to be obtained.
<i>inIndex</i>	On input, a value of type <code>UInt32</code> that specifies which attribute value to get (if the attribute can have multiple values) or zero for single-value attributes.
<i>ioBuffer</i>	On input, a pointer to a buffer. On output, <i>ioBuffer</i> contains the value of the attribute specified by <i>inID</i> . If the buffer is too small to contain the value, <i>ioBuffer</i> is empty.
<i>ioLen</i>	On input, a pointer to a value of type <code>UInt32</code> that specifies the length of <i>ioBuffer</i> . On output, <i>ioLen</i> contains the length of the valid data in <i>ioBuffer</i> .
<i>result</i>	A result code. Possible values include <code>QTSS_NoErr</code> , <code>QTSS_BadArgument</code> if a parameter is invalid, <code>QTSS_BadIndex</code> if the index specified by <i>inIndex</i> does not exist, <code>QTSS_NotEnoughSpace</code> if the attribute value is longer than the value specified by <i>ioLen</i> , and <code>QTSS_AttrDoesntExist</code> if the attribute doesn't exist.

## 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 108) because `QTSS_GetValuePtr` is the most efficient function and less likely to encounter an error condition.

## QTSS\_GetValueAsString

---

Gets the value of an attribute as a C string.

```
QTSS_Error QTSS_GetValueAsString (
    QTSS_Object inObject,
    QTSS_AttributeID inID,
    UInt32 inIndex,
    char** outString);
```

<i>inObject</i>	On input, a value of type <code>QTSS_Object</code> (page 167) that specifies the object that contains the attribute whose value is to be obtained.
<i>inID</i>	On input, a value of type <code>QTSS_AttributeID</code> (page 165) that specifies the ID of the attribute whose value is to be obtained.
<i>inIndex</i>	On input, a value of type <code>UInt32</code> that specifies which attribute value to get (if the attribute can have multiple values) or zero for single-value attributes.
<i>outString</i>	On input, a pointer to an address in memory. On output, <i>outString</i> points to the value of the attribute specified by <i>inID</i> in string format.
<i>result</i>	A result code. Possible values include <code>QTSS_NoErr</code> , <code>QTSS_BadArgument</code> if a parameter is invalid, and <code>QTSS_BadIndex</code> if the index specified by <i>inIndex</i> does not exist.

## DISCUSSION

The `QTSS_GetValueAsString` callback routine gets the value of the specified attribute converts it to C string format and stores it at the location in memory pointed to by the `outString` parameter.

When you no longer need `outString`, call `QTSS_Delete` to free the memory that has been allocated for it.

The `QTSS_GetValueAsString` callback routine can be called to get the value of any attribute regardless of whether the getting the attribute's value is preemptive safe.

**Note**

Calling `QTSS_GetValueAsString` is less efficient than calling `QTSS_GetValue` (page 106), which is less efficient than calling `QTSS_GetValuePtr` (page 108). Calling `QTSS_GetValue` is the recommended way to get the value of an attribute that is not preemptive safe and calling `QTSS_GetValuePtr` is the recommended way to get the value of an attribute that is preemptive safe. ♦

**QTSS\_GetValuePtr**

---

Gets a pointer to an attribute 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` (page 167) that specifies the object containing the attribute whose value is to be obtained.

`inID`            On input, a value of type `QTSS_AttributeID` (page 165) that specifies the ID of an attribute.

## QuickTime Streaming Server Module Reference

<i>inIndex</i>	On input, a value of type <code>UInt32</code> that specifies which attribute value to get (if the attribute can have multiple values) or zero for single-value attributes.
<i>outBuffer</i>	On input, a pointer to an address in memory. On output, <i>outBuffer</i> points to the value of the attribute specified by <i>inID</i> .
<i>outLen</i>	On output, a pointer to a value of type <code>UInt32</code> that contains the number of valid bytes pointed to by <i>outBuffer</i> .
<i>result</i>	A result code. Possible values include <code>QTSS_NoErr</code> , <code>QTSS_NotPreemptiveSafe</code> if <i>inID</i> is an attribute that is not preemptive safe, <code>QTSS_BadArgument</code> if a parameter is invalid, <code>QTSS_BadIndex</code> if the index specified by <i>inIndex</i> does not exist, and <code>QTSS_AttrDoesntExist</code> if the attribute doesn't 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 an attribute that is not preemptive safe. To get the value of an attribute that is not preemptive safe, call `QTSS_GetValue` (page 106) or `QTSS_GetValueAsString` (page 107). ♦

**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);
```

<i>inObject</i>	On input, a value of type <code>QTSS_Object</code> (page 167) that specifies the object containing the attribute whose value is to be set.
<i>inID</i>	On input, a value of type <code>QTSS_AttributeID</code> (page 165) that specifies the ID of the attribute whose value is to be set.
<i>inIndex</i>	On input, a value of type <code>UInt32</code> that specifies which attribute value to set (if the attribute can have multiple values) or zero for single-value attributes.
<i>inBuffer</i>	On input, a pointer to a buffer containing the value that is to be set. When <code>QTSS_SetValue</code> returns, you can dispose of <i>inBuffer</i> .
<i>inLen</i>	On input, a pointer to a value of type <code>UInt32</code> that specifies the length of valid data in <i>inBuffer</i> .
<i>result</i>	A result code. Possible values are <code>QTSS_NoErr</code> , <code>QTSS_BadIndex</code> if the index specified by <i>inIndex</i> does not exist, <code>QTSS_BadArgument</code> if a parameter is invalid, <code>QTSS_ReadOnly</code> if the attribute is read-only, and <code>QTSS_AttrDoesntExist</code> if the attribute doesn't exist.

**DISCUSSION**

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

## QTSS\_TypeStringToType

---

Gets the attribute data type of a data type string that is in C string format.

```
QTSS_Error QTSS_TypeStringToType(
    const char* inTypeString,
    QTSS_AttrDataType* outType);
```

<i>inTypeString</i>	On input, a pointer to a character array containing the attribute data type in C string format.
<i>outType</i>	On output, a pointer to a value of type <code>QTSS_AttrDataType</code> (page 164) containing the attribute data type.

*result*                    A result code. Possible values are QTSS\_NoErr and QTSS\_BadArgument if `inTypeString` does not contain a value for which an attribute data type can be returned.

## DISCUSSION

The `QTSS_TypeStringToType` callback routine gets the attribute data type of a data type string that is in C string format.

## QTSS\_TypeToTypeString

---

Gets the name in C string format of an attribute data type.

```
QTSS_Error QTSS_TypeToTypeString(
    const QTSS_AttrDataType inType,
    char** outTypeString);
```

*inType*                    On input, a pointer to a value of type `QTSS_AttrDataType` (page 164) containing the attribute data type that is to be returned in C string format.

*outType*                   On input, a pointer to an address in memory. On output, `outType` points to a C string containing the attribute data type.

*result*                    A result code. Possible values are QTSS\_NoErr and QTSS\_BadArgument if `inType` does not contain a valid attribute data type.

## DISCUSSION

The `QTSS_TypeToTypeString` callback routine gets the name in C string format of a value that is in `QTSS_AttrDataType` format.

## QTSS\_StringToValue

---

Converts an attribute data type in C string format to a value in QTSS\_AttrDataType format.

```
QTSS_Error QTSS_StringToValue(
    const char* inValueAsString,
    const QTSS_AttrDataType inType,
    void* ioBuffer,
    UInt32* ioBufSize);
```

**inValueAsString** On input, a pointer to a character array containing the value that is to be converted.

**inType** On input, a value of type QTSS\_AttrDataType (page 164) that specifies the attribute data type to which the value pointed to by inValueAsString is to be converted.

**ioBuffer** On input, a pointer to a buffer. On output, the buffer contains the attribute data type to which inValueAsString has been converted. The calling module must allocate ioBuffer before calling QTSS\_StringToValue.

**ioBufSize** On input, a pointer to a value of type UInt32 that specifies the length of the buffer pointed to by ioBuffer. On output, ioBufSize points to the length of data in ioBuffer.

**result** A result code. Possible values are QTSS\_NoErr, QTSS\_BadArgument if inValueAsString or inType do not contain valid values, and QTSS\_NotEnoughSpace if the buffer pointed to by ioBuffer is too small to contain the converted value.

### DISCUSSION

The QTSS\_StringToValue callback routine converts an attribute data type that is in C string format to a value that is in QTSS\_AttrDataType format.

When the memory allocated for the buffer pointed to by ioBuffer is no longer needed, you should deallocate the memory.



## QTSS\_ValueToString

---

Converts an attribute data type in QTSS\_AttrDataType format to a value in C string format.

```
QTSS_Error QTSS_ValueToString(
    const void* inValue,
    const UInt32 inValueLen,
    const QTSS_AttrDataType inType,
    char** outString);
```

<i>inValue</i>	On input, a pointer to a buffer containing the value that is to be converted from QTSS_AttrDataType format.
<i>inValueLen</i>	On input, a value of type UInt32 that specifies the length of the value pointed to by <i>inValue</i> .
<i>inType</i>	On input, a value of type QTSS_AttrDataType (page 164) that specifies the attribute data type of the value pointed to by <i>inValue</i> .
<i>outString</i>	On output, a pointer to a location in memory containing the attribute data type in C string format.
<i>result</i>	A result code. Possible values are QTSS_NoErr and QTSS_BadArgument if <i>inValue</i> , <i>inValueLen</i> , or <i>inType</i> do not contain valid values.

### DISCUSSION

The QTSS\_ValueToString callback routine converts an attribute data type in QTSS\_AttrDataType format to a value in C string format.

## QTSS\_RemoveValue

---

Removes the specified value from an attribute.

```
QTSS_Error QTSS_RemoveValue (
    QTSS_Object inObject,
    QTSS_AttributeID inID,
    UInt32 inIndex);
```

<i>inObject</i>	On input, a value of type <code>QTSS_Object</code> having an attribute whose value is to be removed.
<i>inValueLen</i>	On input, a value of type <code>QTSS_AttributeID</code> containing the attribute ID of the attribute whose value is to be removed.
<i>inIndex</i>	On input, a value of type <code>UInt32</code> that specifies the attribute value that is to be removed. Attribute value indexes are numbered starting from zero.
<i>result</i>	A result code. Possible values include <code>QTSS_NoErr</code> , <code>QTSS_BadArgument</code> if <i>inObject</i> , <i>inVID</i> , or <i>inIndex</i> do not contain valid values, <code>QTSS_ReadOnly</code> if the attribute is read-only, and <code>QTSS_BadIndex</code> if the specified index does not exist.

## DISCUSSION

The `QTSS_RemoveValue` callback routine removes the value of the specified attribute. After the value is removed, the attribute values are renumbered.

## Stream Callback Routines

---

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

- `QTSS_Advise` (page 115) to advise that the specified section of a stream will soon be read.
- `QTSS_Read` (page 115) to read data from a stream.
- `QTSS_Seek` (page 116) to set the position of a stream.
- `QTSS_RequestEvent` (page 117) to request to notification of when a stream becomes readable or writable.
- `QTSS_Write` (page 119) to write data to a stream.
- `QTSS_WriteV` (page 120) to write data to a stream using an `iovec` structure.
- `QTSS_Flush` (page 121) 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\_Advise

---

Advises that the specified section of the stream will soon be read.

```
QTSS_Error QTSS_Advise(QTSS_StreamRef inRef,
                      UInt64 inPosition,
                      UInt32 inAdviseSize);
```

<i>inRef</i>	On input, a value of type <code>QTSS_StreamRef</code> obtained by calling <code>QTSS_OpenFileObject</code> that specifies the stream.
<i>inPosition</i>	On input, the offset in bytes from the beginning of the stream that marks the beginning of the advise section.
<i>inAdviseSize</i>	On input, the size in bytes of the advise section.
<i>result</i>	A result code. Possible values include <code>QTSS_NoErr</code> , <code>QTSS_BadArgument</code> if a parameter is invalid, and <code>QTSS_RequestFailed</code> .

### DISCUSSION

The `QTSS_Advise` callback routine tells a file system module that the specified section of a stream will be read soon. The file system module may read ahead in order to respond more quickly to future calls to `QTSS_Read` for the specified stream.

## QTSS\_Read

---

Reads data from a stream.

```
QTSS_Error QTSS_Read(QTSS_StreamRef inRef,
                    void* ioBuffer,
                    UInt32 inBufLen,
                    UInt32* outLengthRead);
```

<i>inRef</i>	On input, a value of type <code>QTSS_StreamRef</code> that specifies the stream from which data is to be read. Call <code>QTSS_OpenFileObject</code> to obtain a stream reference for the file you want to read.
--------------	--

<i>ioBuffer</i>	On input, a pointer to a buffer in which data that is read is to be placed.
<i>inBufLen</i>	On input, a value of type <code>UInt32</code> that specifies the length of the buffer pointed to by <i>ioBuffer</i> .
<i>outLenRead</i>	On output, a pointer to a value of type <code>UInt32</code> that contains the number of bytes that were read.
<i>result</i>	A result code. Possible values include <code>QTSS_NoErr</code> , <code>QTSS_BadArgument</code> if a parameter is invalid, <code>QTSS_WouldBlock</code> if the read operation would block, or <code>QTSS_RequestFailed</code> if the read operation failed.

**DISCUSSION**

The `QTSS_Read` callback routine reads a buffer of data from a stream.

**QTSS\_Seek**

---

Sets the position of a stream.

```
QTSS_Error QTSS_Seek(QTSS_StreamRef inRef,
                    UInt64 inNewPosition);
```

<i>inRef</i>	On input, a value of type <code>QTSS_StreamRef</code> that specifies the stream whose position is to be set. Call <code>QTSS_OpenFileObject</code> to obtain stream reference.
<i>inNewPosition</i>	On input, the offset in bytes from the start of the stream to which the position is to be set.
<i>result</i>	A result code. Possible values include <code>QTSS_NoErr</code> , <code>QTSS_BadArgument</code> if a parameter is invalid, and <code>QTSS_RequestFailed</code> if the seek operation failed.

**DISCUSSION**

The `QTSS_Seek` callback routine sets the stream position to the value specified by *inNewPosition*.

## QTSS\_RequestEvent

---

Requests notification of specified events.

```
QTSS_Error QTSS_RequestEvent(QTSS_StreamRef inStream,
                             QTSS_EventType inEventMask);
```

<i>inStream</i>	On input, a value of type <code>QTSS_StreamRef</code> that specifies the stream for which event notifications are requested.
<i>inEventMask</i>	On input, a mask that represents the events for which notifications are requested. For possible values, see the Discussion section.
<i>result</i>	A result code. Possible values include <code>QTSS_NoErr</code> , <code>QTSS_BadArgument</code> if a parameter is invalid, and <code>QTSS_RequestFailed</code> if the call failed.

### DISCUSSION

The `QTSS_RequestEvent` callback requests that the caller be notified when the specified events occur on the specified stream. After calling `QTSS_RequestEvent`, the calling module should return as soon as possible from its current module role. The server preserves the calling module's current state and, when the event occurs, calls the module in the role the module was in when it called `QTSS_RequestEvent`.

The following enumeration defines values for the `inEventMask` parameter:

```
enum
{
    QTSS_ReadableEvent = 1,
    QTSS_WriteableEvent = 2
};
typedef UInt32 QTSS_EventType;
```

## QTSS\_SignalStream

---

Tells the server that a stream has become available for I/O.

```
QTSS_Error QTSS_RequestEvent(QTSS_StreamRef inStream,
                             QTSS_EventType inEventMask);
```

<i>inStream</i>	On input, a value of type <code>QTSS_StreamRef</code> that specifies the stream that has become available for I/O.
<i>inEventMask</i>	On input, a mask that represents whether the stream has become available for reading ( <code>QTSS_ReadableEvent</code> ) or writing ( <code>QTSS_WritableEvent</code> ).
<i>result</i>	A result code. Possible values include <code>QTSS_NoErr</code> , <code>QTSS_BadArgument</code> if a parameter is invalid, <code>QTSS_OutOfState</code> if this callback is made from a role that does not allow asynchronous events, and <code>QTSS_RequestFailed</code> if the call failed.

### DISCUSSION

The `QTSS_SignalStream` callback routine tells the server that the stream represented by `inStream` has become available for I/O. Currently only file system modules have reason to call `QTSS_SignalStream`.

The following enumeration defines constants for the `inEventMask` parameter:

```
enum
{
    QTSS_ReadableEvent= 1,
    QTSS_WritableEvent= 2
};
typedef UInt32 QTSS_EventType;
```

## QTSS\_Write

---

Writes data to a stream.

```

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

```

<i>inRef</i>	On input, a value of type <code>QTSS_StreamRef</code> that specifies the stream to which data is to be written.
<i>inBuffer</i>	On input, a pointer to a buffer containing the data that is to be written.
<i>inLen</i>	On input, a value of type <code>UInt32</code> that specifies the length of the data in the buffer pointed to by <code>inBuffer</code> .
<i>outLenWritten</i>	On output, a pointer to a value of type <code>UInt32</code> that contains the number of bytes that were written.
<i>inFlags</i>	On input, a value of type <code>UInt32</code> . See the Discussion section for possible values.
<i>result</i>	A result code. Possible values include <code>QTSS_NoErr</code> , <code>QTSS_BadArgument</code> if a parameter is invalid, <code>QTSS_NotConnected</code> if the stream receiver is no longer connected, and <code>QTSS_WouldBlock</code> if the stream cannot be completely flushed at this time.

### DISCUSSION

The `QTSS_Write` callback routine writes a buffer of data to a stream.

The following enumeration defines constants for the `inFlags` parameter:

```

enum
{
    qtssWriteFlagsIsRTP = 0x00000001,
    qtssWriteFlagsIsRTCP= 0x00000002
};

```

These flags are relevant when writing to an RTP stream reference and tell the server whether the data written should be sent over the RTP channel (`qtssWriteFlagsIsRTP`) or 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);
```

<code>inRef</code>	On input, a value of type <code>QTSS_StreamRef</code> that specifies the stream to which data is to be written.
<code>inVec</code>	On input, a pointer to an <code>iovec</code> structure. The first member of the <code>iovec</code> structure must be empty.
<code>inNumVectors</code>	On input, a value of type <code>UInt32</code> that specifies the number of vectors.
<code>inTotalLength</code>	On input, a value of type <code>UInt32</code> specifying the total length of <code>inVec</code> .
<code>outLenWritten</code>	On output, a pointer to a value of type <code>UInt32</code> containing the number of bytes that were written.
<i>result</i>	A result code. Possible values include <code>QTSS_NoErr</code> , <code>QTSS_BadArgument</code> if a parameter is <code>NULL</code> , and <code>QTSS_WouldBlock</code> 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 `writv` 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.

## File System Callback Routines

---

Modules use the callback routines described in this section to open and close a file object. The files system callback routines are:

- `QTSS_OpenFileObject` (page 121)
- `QTSS_CloseFileObject` (page 123)

## QTSS\_OpenFileObject

---

Opens a file.

```
QTSS_Error QTSS_OpenFileObject(
    char* inPath,
    QTSS_OpenFileFlags inFlags,
    QTSS_Object* outFileObject);
```

## QuickTime Streaming Server Module Reference

<i>inPath</i>	On input, a pointer to a null-terminated C string containing the full path to the file in the local file system that is to be opened.
<i>inFlags</i>	On input, a value of type <code>QTSS_OpenFileFlags</code> specifying flags that describe how the file is to be opened. For possible values, see the Discussion section below.
<i>outFileObject</i>	On output, a pointer to a value of type <code>QTSS_Object</code> (page 167) in which the file object for the opened file is to be placed.
<i>result</i>	A result code. Possible values include <code>QTSS_NoErr</code> , <code>QTSS_BadArgument</code> if a parameter is invalid, and <code>QTSS_FileNotFound</code> if the specified file does not exist.

## DISCUSSION

The `QTSS_OpenFileObject` callback routine opens the specified file and returns a file object for it. One of the attributes of the file object is a stream reference that is passed to QTSS stream callback routines to read and write data to the file and to perform other file operations.

The following enumeration defines constants for the `inFlags` parameter:

```
enum
{
    qtssOpenFileNoFlags = 0,
    qtssOpenFileAsync = 1,
    qtssOpenFileReadAhead = 2
};
typedef UInt32 QTSS_OpenFileFlags;
```

**Constant descriptions**

`qtssOpenFileNoFlags` No flags are specified.

`qtssOpenFileAsync` The file stream will be read asynchronously. Reads may return `QTSS_WouldBlock`. Modules that open files with `qtssOpenFileAsync` should call `QTSS_RequestEvent` to be notified when data is available for reading.

`qtssOpenFileReadAhead` The file stream will be read in order from beginning to end.

## QTSS\_CloseFileObject

---

Closes a file.

```
QTSS_Error QTSS_CloseFileObject(QTSS_Object inFileObject);
```

*inFileObject*    On input, a value of type `QTSS_Object` (page 167) that represents the file that is to be closed.

*result*            A result code. Possible values include `QTSS_NoErr` and `QTSS_BadArgument` if a parameter is invalid.

### DISCUSSION

The `QTSS_CloseFileObject` callback routine closes the specified file.

## 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 123) to add a service that other modules can call.
- `QTSS_IDForService` (page 124) to get the ID of a service.
- `QTSS_DoService` (page 125) 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 125) to invoke the service that `serviceID` represents.

## QTSS\_DoService

---

Invokes a service.

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

<i>inID</i>	On input, a value of type <code>QTSS_ServiceID</code> that specifies the service that is to be invoked. Call <code>QTSS_IDForAttr</code> (page 105) to get the service ID of the service you want to invoke.
<i>inArgs</i>	On input, a value of type <code>QTSS_ServiceFunctionArgsPtr</code> that points to the arguments that are to be passed to the service.
<i>result</i>	A result code returned by the service or <code>QTSS_IllegalService</code> if <i>inID</i> 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 126) to append information to an RTSP header.
- `QTSS_AppendRTSPHeader` (page 126) to send an RTSP header
- `QTSS_SendStandardRTSPResponse` (page 127) 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);
```

<i>inRef</i>	On input, a value of type <code>QTSS_RTSPRequestObject</code> for the RTSP stream.
<i>inHeader</i>	On input, a value of type <code>QTSS_RTSPHeader</code> .
<i>inValue</i>	On input, a pointer to a byte array containing the header that is to be appended.
<i>inValueLen</i>	On input, a value of type <code>UInt32</code> containing the length of valid data pointed to by <i>inValue</i> .
<i>result</i>	A result code. Possible values are <code>QTSS_NoErr</code> and <code>QTSS_BadArgument</code> 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 126) to send the entire header.

## QTSS\_SendRTSPHeaders

---

Sends an RTSP header.

```
QTSS_Error QTSS_SendRTSPHeaders(QTSS_RTSPRequestObject inRef);
```

<i>inRef</i>	On input, a value of type <code>QTSS_RTSPRequestObject</code> for the RTSP stream.
<i>result</i>	A result code. Possible values are <code>QTSS_NoErr</code> and <code>QTSS_BadArgument</code> 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` (page 167) 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.QTSS_ClientSessionObject (page 135)  Writes a content-base header with the provided URL as the content base. Writes application/sdp as the content-type header.	QTSS_ClientSessionObject (page 135)
ANNOUNCE	Writes status line, Cseq, and connection headers as determined by the request	QTSS_ClientSessionObject (page 135)
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 (page 135)
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 (page 135)
PAUSE	Writes status line, CSeq, and connection headers as determined by the request.	QTSS_ClientSessionObject (page 135)
TEARDOWN	Writes status line, CSeq, SessionID, and connection headers as determined by the request.	QTSS_ClientSessionObject (page 135)



## 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 129), 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 130), which is called by a module to start the playing of streams for a client session.
- `QTSS_Pause` (page 132), which is called by a module pause the playing of streams for a client session
- `QTSS_Teardown` (page 132), 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 130).

## 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_AddRTSPStream` (page 129).

*inRTSPRequest* On input, a value of type `QTSS_RequestObject`.

## QuickTime Streaming Server Module Reference

<i>inPlayFlags</i>	On input, a value of type <code>QTSS_PlayFlags</code> . Set <code>inPlayFlags</code> to the constant <code>qtssPlaySendRTCP</code> 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.
<i>result</i>	A result code. Possible values are <code>QTSS_NoErr</code> and <code>QTSS_BadArgument</code> if a parameter is invalid, and <code>QTSS_RequestFailed</code> 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 132) to pause playing or call `QTSS_Teardown` (page 132) to close the client session.

**Note**

The module that called `QTSS_AddRTPStream` (page 129) 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 129) 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 129) is the only module that can call `QTSS_Teardown`. ♦

## QTSS Data Types

---

This section describes QTSS data types. The data types are organized into the following sections:

- “QTSS Objects” (page 133)
- “Other QTSS Data Types” (page 163)

## QTSS Objects

---

This section describes QTSS objects and their attributes. The objects are

- `QTSS_AttrInfoObject` (page 134), which consists of attributes that describe an attribute, such as the attributes name and attribute ID
- `QTSS_ClientSessionObject` (page 135), which consists of attributes that describe a client session
- `QTSS_FileObject` (page 138), which consists of attributes that describe a file that a module has opened
- `QTSS_ModuleObject` (page 139), which consists of attributes that describe a loaded QTSS module, such as its name and version number, a description of what it does, and a list of the roles for which the module is registered
- `QTSS_ModulePrefsObject` (page 140), which consists of attributes containing a module’s preferences
- `QTSS_PrefsObject` (page 141), which consists of attributes that contain server preferences
- `QTSS_RTPStreamObject` (page 148), which consists of attributes that describe a particular RTP stream
- `QTSS_RTSPHeaderObject` (page 152), which consists of attributes that contain all of the header information sent by the client in an RSTP request

- `QTSS_RTSPRequestObject` (page 153), which consists of attributes that describe an RTSP request
- `QTSS_RTSPSessionObject` (page 156), which consists of attributes that describe an RTSP session
- `QTSS_ServerObject` (page 158), which consists of attributes that describe a particular QuickTime Streaming Server.

## QTSS\_AttrInfoObject

---

A `QTSS_AttrInfoObject` consists of attributes that describe an attribute. Table 2-3 lists the attributes for the object type `QTSS_ClientSessionObject`.

**Note**

All `QTSS_AttrInfoObject` attributes are preemptive safe, so they can be read by calling `QTSS_GetValue` (page 106) or `QTSS_GetValuePtr` (page 108). ♦

**Table 2-3** Attributes of the object type `QTSS_AttrInfoObject`

---

Attribute Name and Content	Read/write	Data Type
<code>qtssAttrName</code> The attribute's name.	Read	char array
<code>qtssAttrID</code> The attribute's identifier.	Read	<code>QTSS_AttributeID</code> (page 165)
<code>qtssAttrDataType</code> The attribute's data type.	Read	<code>QTSS_AttrDataType</code> (page 164)
<code>qtssAttrPermissions</code> Permissions for reading and writing the attribute's value, and whether getting the attribute's value is preemptive safe.	Read	<code>QTSS_AttrPermission</code> (page 166)

## 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 106) or `QTSS_GetValuePtr` (page 108). ♦

**Table 2-4** Attributes of the object `QTSS_ClientSessionObject`

---

Attribute Name and Content	Read/Write	Data Type
<code>qtssCliSesStreamObjects</code> Iterated attribute containing all RTP stream references ( <code>QTSS_RTPStreamObject</code> ) belonging to this session.	Read	<code>QTSS_RTPStreamObject</code> (page 148)
<code>qtssCliSesCreateTimeInMsec</code> The time in milliseconds that the session was created.	Read	<code>QTSS_TimeVal</code> (page 170)
<code>qtssCliSesFirstPlayTimeInMsec</code> The time in milliseconds at which <code>QTSS_Play</code> was first called.	Read	<code>QTSS_TimeVal</code> (page 170)
<code>qtssCliSesPlayTimeInMsec</code> The time in milliseconds at which <code>QTSS_Play</code> was most recently called.	Read	<code>QTSS_TimeVal</code> (page 170)
<code>qtssCliSesAdjustedPlayTimeInMsec</code> 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	<code>QTSS_TimeVal</code> (page 170)
<code>qtssCliSesRTPBytesSent</code> The number of RTP bytes sent for this session.	Read	<code>SInt32</code>

*continued*

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

Attribute Name and Content	Read/Write	Data Type
<code>qtssCliSesRTTPacketsSent</code> The number of RTP packets sent for this session.	Read	SInt32
<code>qtssCliSesState</code> The state of this session. Possible values are <code>qtssPausedState</code> and <code>qtssPlayingState</code> .	Read	QTSS_RTPSessionState
<code>qtssCliSesPresentationURL</code> 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 array
<code>qtssCliSesMovieDurationInSecs</code> Duration of the movie for this session in seconds. The value is zero unless set by a module.	Read/write	Float64
<code>qtssCliSesMovieSizeInBytes</code> Movie size in bytes. The value is zero unless set by a module.	Read/write	UInt64
<code>qtssCliSesMovieAverageBitRate</code> The average bits per second based on total RTP bits/movie duration. The value is zero unless set by a module.	Read/write	UInt32
<code>qtssCliSesFullURL</code> The full presentation URL for this session. Same as the <code>qtssCliSesPresentationURL</code> attribute but includes <code>rtsp://domain_name</code> prefix.	Read	char array
<code>qtssCliSesHostName</code> The host name for this session. The <i>domain_name</i> portion of the <code>qtssCliSesFullURL</code> attribute.	Read	char array
<code>qtssCliRTSPSessRemoteAddrStr</code> The IP address of client in dotted decimal format.	Read	char array

*continued*



**Table 2-4** Attributes of the object QTSS\_ClientSessionObject (continued)

Attribute Name and Content	Read/Write	Data Type
qtssCliRTSPSessLocalDNS The DNS name of local IP address for this RTSP connection.	Read	char array
qtssCliRTSPSessLocalAddrStr The local IP address for this RTSP connection in dotted decimal format.	Read	char array
qtssCliRTSPSesUserName The name of the user from the most recent request.	Read	char array
qtssCliRTSPSesURLRealm The realm from the most recent request.	Read	char array
qtssCliRTSPReqRealStatusCode The status from the most recent request. (Same as the qtssRTSPReqRTSPReqRealStatusCode attribute.)	Read	UInt32
qtssCliTeardownReason The teardown reason. If not requested by the client, the reason for the disconnection must be set by the module that calls QTSS_Teardown.	Read/write	QTSS_CliSesTeardownReason (page 166)
qtssCliSesReqQueryString The query string from the request that created this client session.	Read	char array
qtssCliRTSPReqRespMsg The error message sent to the client for the most recent request if the response was an error.	Read	char array
qtssCliSesCurrentBitRate The movie bit rate.	Read	UInt32

*continued*

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

Attribute Name and Content	Read/Write	Data Type
<code>qtssCliSesPacketLossPercent</code> Percentage of packets lost; 5 = 50%	Read	Float32
<code>qtssCliSesTimeConnectedinMsec</code> Time in milliseconds that the client session has been connected.	Read	SInt64
<code>qtssCliSesCounterID</code> A counter-based unique ID for the session.	Read	UInt32

## QTSS\_FileObject

A `QTSS_FileObject` is the collection of attributes that describe a file that has been opened. Table 2-5 lists the attributes for the object `QTSS_FileObject`. These attributes are valid for all roles that receive a value of type `QTSS_FileObject` in the structure the server passes to them.

### Note

All of the attributes for the object `QTSS_FileObject` are preemptive safe, so they can be read by calling `QTSS_GetValue` (page 106) or `QTSS_GetValuePtr` (page 108). ♦

**Table 2-5** Attributes of the object `QTSS_FileObject`

Attribute Name and Content	Read/Write	Data Type
<code>qtssFlObjStream</code> The stream reference for this file object.	Read	<code>QTSS_StreamRef</code>
<code>qtssFlObjFileSysModuleName</code> The name of the file system module that handles this file object	Read	char array

*continued*

**Table 2-5** Attributes of the object QTSS\_FileObject (continued)

Attribute Name and Content	Read/Write	Data Type
qtssF1ObjLength The length of the file in bytes.	Read/write	UInt64
qtssF1ObjPosition The current position in bytes of the file’s file pointer from the beginning of the file (byte zero).	Read	UInt64
qtssF1ObjModDate The date and time of the last time the file was modified.	Read/write	QTSS_TimeVal (page 170)

**QTSS\_ModuleObject**

A QTSS\_ModuleObject is the collection of attributes that describe a module, including its name, version number, a description of what the module does, its preferences, and what roles the module is registered for.

Table 2-6 lists the attributes for the QTSS\_ModuleObject object. These attributes are valid for all roles that receive a value of type QTSS\_ModuleObject in the structure the server passes to them.

**Note**  
With the exception of `qtssModDesc` and `qtssModVersion`, `QTSS_ModuleObject` attributes are preemptive safe and can be read by calling `QTSS_GetValue` (page 106) or `QTSS_GetValuePtr` (page 108). ♦

**Table 2-6**      Attributes of the `QTSS_ModuleObject` object

Attribute Name and Content	Read/Write	Data Type
<code>qtssModName</code> The module's name.	Read	char array
<code>qtssModDesc</code> Description of what the module does.	Read/write	char array
<code>qtssModVersion</code> The module's version number in the format 0xMM.m.v.bbbb, where M = major version, m = minor version, v = very minor version, and b = build number.	Read/write	UInt32
<code>qtssModRoles</code> List of all the roles for which this module is registered.	Read	QTSS_Role (page 167)
<code>qtssModPrefs</code> An object whose attributes store the preferences for this module.	Read	QTSS_ModulePrefsObject (page 140)

### QTSS\_ModulePrefsObject

The module preferences object type `QTSS_ModulePrefsObject` is the collection of attributes that contain a module's preferences.

Each module is responsible for adding attributes to its module preferences object type and setting their values. The values of the attributes in the module preferences object are persistent between invocations of the server because the

server writes the module preferences object for each module to a configuration file that the server reads when it is started.

QTSS\_PrefsObject

A QTSS\_PrefsObject is the collection of attributes that contain server preferences. Table 2-7 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 106). ♦

Table 2-7 Attributes of the object QTSS\_PrefsObject

Attribute Name and Content	Read/Write	Data Type
qtssPrefsRTSPTimeout Amount of time in seconds the server tells clients it will wait before disconnecting idle RTSP clients.	Read/write	UInt32
qtssPrefsRealRTSPTimeout The amount of time in seconds the server actually waits before disconnecting idle RTSP clients. This timer is reset each time the server receives a new RTSP request from the client. A value of zero means that there is no timeout.	Read/write	UInt32
qtssPrefsRTPTIMEOUT The amount of time in seconds the server will wait before disconnecting idle RTP clients. This timer is reset each time the server receives an RTCP status packet from a client. A value of zero means there is no timeout.	Read/write	UInt32

continued

**Table 2-7** Attributes of the object QTSS\_PrefsObject (continued)

Attribute Name and Content	Read/Write	Data Type
<p>qtssPrefsMaximumConnections</p> <p>The maximum number of concurrent RTP connections the server allows. A value of -1 means that an unlimited number of connections are allowed.</p>	Read/write	SInt32
<p>qtssPrefsMaximumBandwidth</p> <p>The maximum amount of bandwidth the server is allowed to serve in K bits. If the server exceeds this value, it responds to new client requests for additional streams with RTSP error 453, "Not Enough Bandwidth". A value of -1 means the amount bandwidth the server is allowed to serve is unlimited.</p>	Read/write	SInt32
<p>qtssPrefsMovieFolder</p> <p>The path to the root movie folder.</p>	Read/write	char array
<p>qtssPrefsRTSPIPAddr</p> <p>Specifies the IP address in dotted-decimal format the server should accept RTSP client connections on. A value of 0 means the server should accept connections on all IP addresses that are currently enabled on the system.</p>	Read/write	char array
<p>qtssPrefsBreakOnAssert</p> <p>If true, the server will stop and enter the debugger when an assert fails</p>	Read/write	Bool16
<p>qtssPrefsAutoRestart</p> <p>If true, the server automatically restarts itself if it crashes.</p>	Read/write	Bool16
<p>qtssPrefsTotalBytesUpdate</p> <p>The interval in seconds between updates of the server's total bytes and current bandwidth statistics.</p>	Read/write	UInt32

*continued*

**Table 2-7** Attributes of the object QTSS\_PrefsObject (continued)

Attribute Name and Content	Read/Write	Data Type
qtssPrefsAvgBandwidthUpdate The interval in seconds between computations of the server's average bandwidth.	Read/write	UInt32
qtssPrefsSafePlayDuration If the server finds it is serving more than its allowed maximum bandwidth (using the average bandwidth computation), it will attempt to disconnect the most recently connected clients until the average bandwidth drops to acceptable levels. However, it will not disconnect clients if they've been connected for longer than the time in seconds specified by this attribute. If this value is set to zero, the server does not disconnect clients.	Read/write	UInt32
qtssPrefsModuleFolder The path to the folder containing dynamic loadable server modules. The configuration file sets this attribute to "/usr/local/sbin/StreamingServerModules". The built-in error log module that loads before all other modules uses the following seven attributes:	Read/write	char array
qtssPrefsErrorLogName Sets the name of the error log file. The configuration file sets this value to "Error".	Read/write	char array
qtssPrefsErrorLogDir Sets the path to the directory containing the error log file. The configuration file sets this value to "/Library/QuickTimeStreaming/Logs".	Read/write	char array
qtssPrefsErrorRollInterval The interval in days between rolling the error log file. A value of zero means that there is no interval.	Read/write	UInt32

*continued*

**Table 2-7** Attributes of the object QTSS\_PrefsObject (continued)

Attribute Name and Content	Read/Write	Data Type
<p>qtssPrefsMaxErrorLogSize</p> <p>The maximum size in bytes of the error log. A value of zero means that the server does not impose a limit.</p>	Read/write	UInt32
<p>qtssPrefsErrorLogVerbosity</p> <p>Sets the verbosity level of messages the error logger logs. The following values are meaningful:</p> <p>0 = log fatal errors  1 = log fatal errors and warnings  2 = log fatal errors, warnings, asserts  3 = log fatal errors, warnings, asserts, and debug messages</p>	Read/write	UInt32
<p>qtssPrefsScreenLogging</p> <p>Set to true to write error log messages to the terminal window. Note that in order to see the messages, the server must be launched from the command line in foreground mode (triggered by the use of the -d flag).</p>	Read/write	Bool16
<p>qtssPrefsErrorLogEnabled</p> <p>Set to true to enable error logging.</p>	Read/write	Bool16
<p>qtssPrefsTCPMinThinDelayToleranceInMSec</p> <p>If a packet is late by less than this number of milliseconds, the server stops thinning. (Thinning is reducing the bitrate of the stream when there is not enough bandwidth between the client and the server to transmit the full stream.) This attribute applies to all stream transports, not just TCP. This is a default time that can be overridden by the late-tolerance field of x-RTP-Options.</p>	Read/write	SInt32

*continued*



**Table 2-7** Attributes of the object QTSS\_PrefsObject (continued)

Attribute Name and Content	Read/Write	Data Type
qtssPrefsTCPMaxThinDelayToleranceInMSec If a packet is late by this number of milliseconds, the server reduces the bit rate of the stream (called “thinning”). This attribute applies to all stream transports, not just TCP. This is a default time that can be overridden by the late-tolerance field of x-RTP-Options.	Read/write	SInt32
qtssPrefsTCPVideoDelayToleranceInMSec If a video packet is late by this number of milliseconds, the server drops it. This applies to all stream transports, not just TCP. This is a default time that can be overridden by the late-tolerance field of x-RTP-Options.	Read/write	SInt32
qtssPrefsTCPAudioDelayToleranceInMSec If an audio packet is late by this number of milliseconds, the server drops it. This applies to all stream transports, not just TCP. This is a default time that can be overridden by the late-tolerance field of x-RTP-Options.	Read/write	SInt32
qtssPrefsMinTCPBufferSizeInBytes When streaming over TCP, sets the minimum size in bytes of the TCP send buffer.	Read/write	UInt32
qtssPrefsMaxTCPBufferSizeInBytes When streaming over TCP, sets the maximum size in bytes of the TCP send buffer.	Read/write	UInt32
qtssPrefsTCPSecondsToBuffer When streaming over TCP, uses the movie’s bit rate to scale the size of the TCP send buffer to fit the specified number of seconds.	Read/write	Float32

*continued*

**Table 2-7** Attributes of the object QTSS\_PrefsObject (continued)

Attribute Name and Content	Read/Write	Data Type
<p>qtssPrefsDoReportHTTPConnectionAddress</p> <p>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.</p>	Read/write	Bool16
<p>qtssPrefsRunUserName</p> <p>Run under the specified user name.</p>	Read/write	char array
<p>qtssPrefsRunGroupName</p> <p>Run under the specified group name.</p>	Read/write	char array
<p>qtssPrefsSrcAddrInTransport</p> <p>If set to true, the server will add its source address to its transport. headers. This is necessary on certain networks where the source address is not necessarily known.</p>	Read/write	Bool16
<p>qtssPrefsRTSPPorts</p> <p>Ports for accepting RTSP client connections. BY default, ports 554 and 7070 are set. Add port 80 to the list if you are streaming across the Internet and want clients behind firewalls to be able to connect to the server.</p>	Read/write	UInt16
<p>qtssPrefsMaxRetransDelayInMsec</p> <p>For reliable UPD, the maximum interval between when a retransmit is supposed to be sent and when it actually is sent. Lower values means smoother flow but slower server performance.</p>	Read/write	UInt32
<p>qtssPrefsDefaultWindowSizeInK</p> <p>The default window size in K bytes, used when the client doesn't specify its initial window size.</p>	Read/write	UInt32
<p>qtssPrefsAckLoggingEnabled</p> <p>Enables detailed logging of UDP acknowledgments and retransmits, used for debugging only.</p>	Read/write	Bool16

*continued*

**Table 2-7** Attributes of the object QTSS\_PrefsObject (continued)

Attribute Name and Content	Read/Write	Data Type
qtssPrefsRTCPollIntervalInMsec For reliable UDP, the time in milliseconds between server checks for incoming RTCP packets. A longer interval means better server performance but grosser mis-estimates of packet round-trip times.	Read/write	UInt32
qtssPrefsRTCPSockRcvBufSizeInK For reliable UDP, the size in K bytes for the RTCP UDP socket receive buffers. In general, this buffer needs to be big enough to absorb bursts of RTCP acknowledgements. A low value may cause acknowledgements to be dumped by the kernel.	Read/write	UInt32
qtssPrefsOverbufferBucketInterval The duration in milliseconds for buckets in the overbuffer. This is also the minimum time the server will wait between sending packet data to a client.	Read/write	UInt32
qtssPrefsTCPThickIntervalInSec Time in seconds server must wait before attempting to increase the bit rate of the client even if conditions clear up. (Increasing the bit rate is known as “thickening”.)	Read/write	UInt32
qtssPrefsAltTransportIPAddr The server appends its own IP address to the Transport header. If you want an alternate address placed there, use this attribute to specify the address.	Read/write	char
qtssPrefsMaxAdvanceSendTimeInSec The farthest in advance the server will send a packet to a client that supports overbuffering.	Read/write	UInt32

*continued*

**Table 2-7** Attributes of the object QTSS\_PrefsObject (continued)

Attribute Name and Content	Read/Write	Data Type
<code>qtssPrefsReliableUDPSlowStart</code> Set to <code>true</code> if reliable UDP slow start is enabled. Disabling UDP slow start may lead to an initial burst of packet loss due to mis-estimate of the client's available bandwidth. Enabling UDP slow start may lead to premature reduction of the bit rate (known as "thinning").	Read/write	Bool16
<code>qtssPrefsAutoDeleteSDPFiles</code> Set to <code>true</code> if automatic delete of SDP files is enabled. SDP files in the Movies directory tree are deleted after the SDP end time or when a broadcaster's RTSP-controlled SDP session ends. SDPs controlled by an RTSP session contain "a=x-broadcastcontrol:RTSP". Changes take affect at the end of the current interval.	Read/write	Bool16
<code>qtssPrefsAuthenticationScheme</code> Set this to be the authentication scheme you want the server to use. The currently supported values are "basic", "digest", and "none".	Read/write	char
<code>qtssPrefsDeleteSDPFilesInterval</code> The interval in seconds at which the server checks for and deletes SDP files whose "t= value" has timed out. The internal default value is ten seconds. The minimum internal value is one second. Changes take affect at the end of the current interval.	Read/write	UInt32

### QTSS\_RTPStreamObject

A `QTSS_RTPStreamObject` is the collection of attributes that describe a particular RTP stream. Table 2-8 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 106) or `QTSS_GetValuePtr` (page 108). ♦

**Table 2-8** Attributes of the object `QTSS_RTPStreamObject`

Attribute Name and Content	Read/Write	Data Type
<code>qtssRTPStrTrackID</code> Unique ID that identifies each RTP stream.	Read/write	UInt32
<code>qtssRTPStrSSRC</code> 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
<code>qtssRTPStrPayloadName</code> Name of the media for this stream. This attribute is empty unless a module explicitly sets it.	Read/write	char array
<code>qtssRTPStrPayloadType</code> Payload type of the media for this stream. The value of this attribute is <code>qtssUnknownPayloadType</code> unless a module sets it <code>qtssVideoPayloadType</code> or <code>qtssAudioPayloadType</code> .	Read/write	QTSS_RTPPayloadType (page 167)
<code>qtssRTPStrFirstSeqNumber</code> 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 <code>QTSS_Play</code> (page 130). The server uses this attribute to generate a proper RTSP PLAY response.	Read/write	SInt16

*continued*

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

Attribute Name and Content	Read/Write	Data Type
<code>qtssRTPStrFirstTimestamp</code> RTP timestamp of the first RTP packet generated for this stream after the last <code>PLAY</code> request was issued. If known, this attribute must be set by a module before calling <code>QTSS_Play</code> (page 130). The server uses this attribute to generate a proper RTSP <code>PLAY</code> response.	Read/write	<code>SInt32</code>
<code>qtssRTPStrTimescale</code> Timescale for the track. If known, this must be set before calling <code>QTSS_Play</code> (page 130).	Read/write	<code>SInt32</code>
<code>qtssRTPStrBufferDelayInSecs</code> 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.  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.	Read	<code>Float32</code>
<code>qtssRTPStrFractionLostPackets</code> The fraction of packets that have been lost for this stream.	Read	<code>UInt32</code>
<code>qtssRTPStrTotalLostPackets</code> The total number of packets that have been lost for this stream.	Read	<code>UInt32</code>
<code>qtssRTPStrJitter</code> Cumulative jitter for this stream.	Read	<code>UInt32</code>
<code>qtssRTPStrRecvBitRate</code> Average bit rate received by the client in bits per second.	Read	<code>UInt32</code>
<code>qtssRTPStrAvgLateMilliseconds</code> Average in milliseconds of packets that the client received late.	Read	<code>UInt16</code>

*continued*

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

Attribute Name and Content	Read/Write	Data Type
<code>qtssRTPStrAudioDryCount</code> Number of times the audio has run dry.	Read	UInt16
<code>qtssRTPStrIsTCP</code> If this RTP stream is being sent over TCP, this attribute is <code>true</code> . If this RTP stream is being sent over UDP, this attribute is <code>false</code> .	Read	Bool16
<code>qtssRTPStrStreamRef</code> A <code>QTSS_StreamRef</code> used for sending RTP or RTCP packets to the client. Use <code>QTSS_WriteFlags</code> to specify whether each packet is an RTP or RTCP packet.	Read	<code>QTSS_StreamRef</code>
<code>qtssRTPStrTransportType</code> The transport type.	Read	<code>QTSS_RTPTransportType</code> (page 168)

## 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 106) or `QTSS_GetValuePtr` (page 108).



## QTSS\_RTSPRequestObject

A `QTSS_RTSPRequestObject` is the collection of attributes that describe a particular RTSP request. Table 2-9 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 106) or `QTSS_GetValuePtr` (page 108). ♦

**Table 2-9** Attributes of the object `QTSS_RTSPRequestObject`

Attribute Name and Content	Read/Write	Data Type
<code>qtssRTSPReqFullRequest</code> The complete RTSP request as sent by the client. This attribute is available in every role that receives an object of type <code>QTSS_RTSPRequestObject</code> .	Read	char array
<code>qtssRTSPReqMethodStr</code> The RTSP method of this request.	Read	char array
<code>qtssRTSPReqFilePath</code> URI for this request, converted to a local file system path.	Read	char array
<code>qtssRTSPReqURI</code> URI for this request.	Read	char array
<code>qtssRTSPReqFilePathTrunc</code> Same as <code>qtssRTSPReqFilePath</code> , but without the last element of the path.	Read	char array

*continued*

**Table 2-9** Attributes of the object `QTSS_RTSPRequestObject` (continued)

Attribute Name and Content	Read/Write	Data Type
<code>qtssRTSPReqFileName</code> All characters after the last path separator in the file system path.	Read	char array
<code>qtssRTSPReqFileDigit</code> If the URI ends with one or more digits, this attribute points to those digits.	Read	char array
<code>qtssRTSPReqAbsoluteURL</code> The full URL starting with “rtsp://”.	Read	char array
<code>qtssRTSPReqTruncAbsoluteURL</code> The URL without last element of the path.	Read	char array
<code>qtssRTSPReqMethod</code> The RTSP method as a value of type <code>QTSS_RTSPMethod</code> .	Read	<code>QTSS_RTSPMethod</code>
<code>qtssRTSPReqStatusCode</code> The current status code for the request as <code>QTSS_RTSPStatusCode</code> . By default, the value is <code>qtssSuccessOK</code> . If a module sets this attribute and calls <code>QTSS_SendRTSPHeaders</code> , the status code in the header that the server generates contains the value of this attribute.	Read/write	<code>QTSS_RTSPStatusCode</code>
<code>qtssRTSPReqStartTime</code> The start time specified in the Range header of the PLAY request.	Read	<code>Float64</code>
<code>qtssRTSPReqStopTime</code> The stop time specified in the Range header of the PLAY request.	Read	<code>Float64</code>
<code>qtssRTSPReqRespKeepAlive</code> Set this attribute to <code>true</code> if you want the server to keep the connection open after completion of the request. Otherwise, set this attribute to <code>false</code> if you want the server to terminate the connection upon completion of the request.	Read/write	<code>Bool16</code>

**Table 2-9** Attributes of the object `QTSS_RTSPRequestObject` (continued)

Attribute Name and Content	Read/Write	Data Type	<i>continued</i>
<code>qtssRTSPReqRootDir</code> 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 array	
<code>qtssRTSPReqRealStatusCode</code> Same as the <code>qtssRTSPReqStatusCode</code> attribute but translated from a <code>QTSS_RTSPStatusCode</code> to an actual RTSP status code.	Read	UInt32	
<code>qtssRTSPReqStreamRef</code> A value of type <code>QTSS_StreamRef</code> for sending data to the RTSP client. This stream reference, unlike the one provided as an attribute in the <code>QTSS_RTSPSessionObject</code> , never returns <code>QTSS_WouldBlock</code> in response to a <code>QTSS_Write</code> or a <code>QTSS_WriteV</code> call.	Read	<code>QTSS_StreamRef</code>	
<code>qtssRTSPReqUserName</code> The decoded user name, if provided by the RTSP request.	Read	char array	
<code>qtssRTSPReqURLRealm</code> The authorization entity for the client to display in the following string: "Please enter password for <i>realm</i> at <i>server-name</i> . The default value of this attribute is "Streaming Server".	Read/write	char array	
<code>qtssRTSPReqIfModSinceDate</code> If the RTSP request contains an If-Modified-Since header, this attribute is the if-modified date converted to a value of type <code>QTSS_TimeVal</code> .	Read	<code>QTSS_TimeVal</code> (page 170)	

*continued*

**Table 2-9** Attributes of the object `QTSS_RTSPRequestObject` (continued)

Attribute Name and Content	Read/Write	Data Type
<code>qtssRTSPReqRespMsg</code> The error message that is sent back to the client if the response was an error. A module sending an RTSP error to the client should set this attribute to be a text message that describes why the error occurred. It is also useful to write this message to a log file. Once the RTSP response has been sent, this attribute contains the response message.	Read/write	char array
<code>qtssRTSPReqContentLen</code> Content length of incoming RTSP request body.	Read	UInt32
<code>qtssRTSPReqSpeed</code> Value of the speed header.	Read	Float32
<code>qtssRTSPReqLateTolerance</code> Value of the late-tolerance field in the x-RTP-Options header, or -1 if not present..	Read	Float32

## QTSS\_RTSPSessionObject

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

Table 2-10 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 2-10** Attributes of the object `QTSS_RTSPSessionObject`

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

*continued*

**Table 2-10** Attributes of the object `QTSS_RTSPSessionObject` (continued)

Attribute Name and Content	Read/Write	Data Type
<code>qtssRTSPSesEventCntxt</code> An event context for the RTCP connection to the client. This attribute should primarily be used to wait for flow-controlled <code>EV_WR</code> event when responding to a client.	Read	<code>QTSS_EventContextRef</code>
<code>qtssRTSPSesType</code> The RTSP session type. Possible values are <code>qtssRTSPSession</code> , <code>qtssRTSPHTTPSession</code> (an HTTP tunnelled RTSP session), and <code>qtssRTSPHTTPInputSession</code> . Sessions of type <code>qtssRTSPHTTPInputSession</code> are usually very short lived.	Read	<code>QTSS_RTSPSessionType</code> (page 169)
<code>qtssRTSPSesStreamRef</code> A <code>QTSS_StreamRef</code> used for sending data to the RTSP client.	Read	<code>QTSS_RTSPSessionStream</code> (page 169)

## QTSS\_ServerObject

A `QTSS_ServerObject` is the collection of attributes that describe a particular QuickTime Streaming Server. Table 2-11 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 2-11.

**Table 2-11** 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 The API version supported by this server. The format of this value is 0xMMMMmmmm, where <i>M</i> is the major version number and <i>m</i> is the minor version number.	Read	UInt32
qtssSvrDefaultDNSName The “default” DNS name of the server.	Read	char array
qtssSvrDefaultIPAddr The “default” IP address of the server.	Read	UInt32
qtssSvrServerName The name of the server.	Read	char array
qtssSvrServerVersion The version of the server.	Read	char array
qtssSvrServerBuildDate Date that the server was built.	Read	char array
qtssSvrRTSPServerHeader The Server header that the server uses when responding to RTSP clients.	Read	char array

*continued*

**Table 2-11** 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	Read/write	QTSS_ServerState (page 170)
The current state of the server. Possible values are		
qtssStartingUpState		
qtssRunningState		
qtssRefusingConnectionsState		
qtssFatalErrorState		
qtssShuttingDownState		
qtssIdleState		
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.		
qtssSvrRTSPPorts	Read	char array
An indexed attribute containing all the ports the server is listening on.		
qtssSvrIsOutOfDescriptors	Read	Bool16
If the server has run out of file descriptors, this attribute is true; otherwise, this attribute is false.		
qtssRTSPCurrentSessionCount	Read	UInt32
The number of clients that are currently connected over standard RTSP.		
qtssRTSPHTTPCurrentSessionCount	Read	UInt32
The number of clients that are currently connected over RTSP/HTTP.		

*continued*



**Table 2-11** 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 <code>QTSS_GetValuePtr</code> :		
<code>qtssRTPSvrNumUDPSockets</code> Number of UDP sockets currently being used by the server.	Read	UInt32
<code>qtssRTPSvrCurConn</code> The number of clients currently connected to the server.	Read	UInt32
<code>qtssRTPSvrTotalConn</code> Total number of clients that have connected to the server since the server started up.	Read	UInt32
<code>qtssRTPSvrCurBandwidth</code> Current bandwidth being output by the server in bits per second.	Read	UInt32
<code>qtssRTPSvrTotalBytes</code> Total number of bytes output since the server started up.	Read	UInt64
<code>qtssRTPSvrAvgBandwidth</code> Average bandwidth output by the server in bits per second.	Read	UInt32
<code>qtssRTPSvrCurPackets</code> Current packets per second being output by the server.	Read	UInt32
<code>qtssRTPSvrTotalPackets</code> Total number of bytes output since the server started up.	Read	UInt64
<code>qtssSvrHandledMethods</code> The methods that the server supports. Modules should append the methods they support to this attribute in their <code>QTSS_Initialize_Role</code> .	Read/write	QTSS_RTSPMethod (page 169)

*continued*

**Table 2-11** 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 <code>QTSS_GetValuePtr</code> :		
<code>qtssSvrCurrentTimeMilliseconds</code> The server's current time in milliseconds. Getting the value of this attribute is equivalent to calling <code>QTSS_Milliseconds</code> (page 97).	Read	<code>QTSS_TimeVal</code> (page 170)
<code>qtssSvrCPULoadPercent</code> The percentage of CPU time the server is currently using.	Read	<code>Float32</code>

**Table 2-11** Attributes of the object QTSS\_ServerObject (continued)

Attribute Name and Content	Read/Write	Data Type
The following attributes are preemptive safe and can be read by QTSS_GetValuePtr:		
qtssSvrModuleObjects A module object representing each module.	Read	QTSS_ModuleObject (page 139)
qtssSvrStartupTime The time at which the server started up.	Read	QTSS_TimeVal (page 170)
qtssSvrGMTOffsetInHrs The time zone in which the server is running (offset from GMT in hours).	Read	SInt32
qtssSvrDefaultIPAddrStr The default IP address of the server as a string.	Read	char array
qtssSvrPreferences An object representing each the server's preferences.	Read	QTSS_PrefsObject (page 141)
qtssSvrMessages An object containing the server's error messages.	Read	QTSS_Object (page 167)

## Other QTSS Data Types

This section describes other QTSS data types. The data types are

- QTSS\_AttrDataType (page 164)
- QTSS\_AttributeID (page 165)
- QTSS\_AttrPermission (page 166)
- QTSS\_CliSesTeardownReason (page 166)
- QTSS\_Object (page 167)
- QTSS\_Role (page 167)
- QTSS\_RTTPPayloadType (page 167)
- QTSS\_RTPSessionState (page 168)
- QTSS\_RTPTransportType (page 168)
- QTSS\_RTSPMethod (page 169)

- QTSS\_RTSPSessionStream (page 169)
- QTSS\_RTSPSessionType (page 169)
- QTSS\_RTSPStatusCode (page 169)
- QTSS\_StreamRef (page 170)
- QTSS\_TimeVal (page 170)
- QTSS\_ServerState (page 170)

## QTSS\_AttrDataType

---

Each QTSS attribute has an associated data type. The `QTSS_AttrDataType` enumeration defines values that describe attribute data types. Having an attribute's data type helps the server and modules handle an attribute value without having specific knowledge about the attribute.

The `QTSS_AttrDataType` is defined as:

```
enum
{
    qtssAttrDataTypeUnknown      = 0,
    qtssAttrDataTypeCharArray    = 1,
    qtssAttrDataTypeBool16      = 2,
    qtssAttrDataTypeSInt16      = 3,
    qtssAttrDataTypeUInt16      = 4,
    qtssAttrDataTypeSInt32      = 5,
    qtssAttrDataTypeUInt32      = 6,
    qtssAttrDataTypeSInt64      = 7,
    qtssAttrDataTypeUInt64      = 8,
    qtssAttrDataTypeQTSS_Object = 9,
    qtssAttrDataTypeQTSS_StreamRef = 10,
    qtssAttrDataTypeFloat32     = 11,
    qtssAttrDataTypeFloat64     = 12,
    qtssAttrDataTypeVoidPointer = 13,
    qtssAttrDataTypeTimeVal     = 14,
    qtssAttrDataTypeNumTypes    = 15
};
typedef UInt32 QTSS_AttrDataType;
```

**Constant descriptions**`qtssAttrDataTypeUnknown`

The data type is unknown.

`qtssAttrDataTypeCharArray`

The data type is a character array.

`qtssAttrDataTypeBool16`

The data type is a 16-bit Boolean value.

`qtssAttrDataTypeSInt16`

The data type is a signed 16-bit integer.

`qtssAttrDataTypeUInt16`

The data type is an unsigned 16-bit integer.

`qtssAttrDataTypeSInt32`

The data type is a signed 32-bit integer.

`qtssAttrDataTypeUInt32`

The data type is an unsigned 32-bit integer.

`qtssAttrDataTypeSInt64`

The data type is a signed 64-bit integer.

`qtssAttrDataTypeQTSS_Object`The data type is a `QTSS_Object` (page 167).`qtssAttrDataTypeQTSS_StreamRef`The data type is a `QTSS_StreamRef` (page 170).`qtssAttrDataTypeFloat32`The data type is a `Float32`.`qtssAttrDataTypeFloat64`The data type is a `Float64`.`qtssAttrDataTypeVoidPointer`

The data type is a pointer to a void.

`qtssAttrDataTypeTimeVal`The data type is a `QTSS_TimeVal` (page 170).`qtssAttrDataTypeNumTypes`

The data type is a value that describes the number of types.

**QTSS\_AttributeID**

---

A `QTSS_AttributeID` is a signed 32-bit integer that uniquely identifies an attribute. It is defined as

```
typedef SInt32 QTSS_AttributeID;
```

## QTSS\_AttrPermission

---

The `QTSS_AttrPermission` data type is an enumeration that describes whether an attribute is readable, writable, and preemptive safe. The data type of the `qtssAttrPermissions` attribute of the `QTSS_AttrInfoObject` object type is of type `QTSS_AttrPermission`.

The `QTSS_AttrPermission` enumeration is defined as:

```
enum
{
    qtssAttrModeRead      = 1,
    qtssAttrModeWrite     = 2,
    qtssAttrModePreempSafe= 4
};
typedef UInt32 QTSS_AttrPermission;
```

Once set, attribute permissions cannot be changed.

## QTSS\_CliSesTeardownReason

---

The `QTSS_CliSesTeardownReason` enumeration defines values that identify why a session is closing. The `QTSS_RTPSessionState` enumeration is defined as

```
enum
{
    qtssCliSesTearDownClientRequest= 0,
    qtssCliSesTearDownUnsupportedMedia = 1,
    qtssCliSesTearDownServerShutdown = 2,
    qtssCliSesTearDownServerInternalErr = 3
};
typedef UInt32 QTSS_CliSesTeardownReason;
```

## QTSS\_Object

---

A `QTSS_Object` is a pointer to a value that identifies a particular object. The `QTSS_Object` is defined as

```
typedef void* QTSS_Object;
```

The `QTSS_Object` is used to define other objects:

```
typedef QTSS_Object QTSS_RTPStreamObject;  
typedef QTSS_Object QTSS_RTSPSessionObject;  
typedef QTSS_Object QTSS_RTSPRequestObject;  
typedef QTSS_Object QTSS_RTSPHeaderObject;  
typedef QTSS_Object QTSS_ClientSessionObject;  
typedef QTSS_Object QTSS_ServerObject;  
typedef QTSS_Object QTSS_PrefsObject;  
typedef QTSS_Object QTSS_TextMessagesObject;  
typedef QTSS_Object QTSS_FileObject;  
typedef QTSS_Object QTSS_ModuleObject;  
typedef QTSS_Object QTSS_ModulePrefsObject;  
typedef QTSS_Object QTSS_AttrInfoObject;
```

## QTSS\_Role

---

A `QTSS_Role` is an unsigned 32-bit integer that defines values that correspond to module roles. It is defined as

```
typedef UInt32 QTSS_Role;
```

## QTSS\_RTPPayloadType

---

The `QTSS_RTPPayloadType` enumeration defines values that a module uses to specify the stream's payload type when it adds an RTP stream to a client session. The enumeration is defined as

```
enum
{
    qtssUnknownPayloadType = 0,
    qtssVideoPayloadType = 1,
    qtssAudioPayloadType = 2
};
typedef UInt32 QTSS_RTPPayloadType;
```

## QTSS\_RTPSessionState

---

The `QTSS_RTPSessionState` enumeration defines values that identify the state of an RTP session. The `QTSS_RTPSessionState` enumeration is defined as

```
enum
{
    qtssPausedState = 0,
    qtssPlayingState = 1
};
typedef UInt32 QTSS_RTPSessionState;
```

## QTSS\_RTPTransportType

---

The `QTSS_RTPTransportType` enumeration defines values for RTP transports. The enumeration is defined as

```
enum
{
    qtssRTPTransportTypeUDP = 0,
    qtssRTPTransportTypeReliableUDP = 1,
    qtssRTPTransportTypeTCP = 2
};
typedef UInt32 QTSS_RTPTransportType;
```



## QTSS\_RTSPMethod

---

The `QTSS_RTSPMethod` enumeration defines values for each method in the RTSP protocol. This enumeration is fully defined in `QTSS_RTSPProtocol.h`.

## QTSS\_RTSPSessionStream

---

A `QTSS_RTSPSessionStream` is a value of type `QTSS_StreamRef` used for sending data to the RTSP client.

## QTSS\_RTSPSessionType

---

The `QTSS_RTSPSessionType` enumeration defines values that describe various RTSP session types. The enumeration is defined as

```
enum
{
    qtssRTSPSession          = 0,
    qtssRTSPHTTPSession      = 1,
    qtssRTSPHTTPInputSession = 2
};
typedef UInt32 QTSS_RTSPSessionType;
```

A `qtssRTSPHTTPInputSession` is the input half of an RTSPHTTP session. These session types are usually very short lived.

## QTSS\_RTSPStatusCode

---

The `QTSS_RTSPStatusCode` enumeration defines values for each RTSP response status code that the RTSP protocol defines. This enumeration is fully defined in `QTSSRTSPProtocol.h`.

## QTSS\_StreamRef

---

A `QTSS_StreamRef` is a pointer to a value that identifies a particular stream. The `QTSS_StreamRef` is defined as

```
typedef void* QTSS_StreamRef;
```

The `QTSS_StreamRef` is used to define other stream references:

```
typedef QTSS_StreamRef QTSS_ErrorLogStream;  
typedef QTSS_StreamRef QTSS_FileStream;  
typedef QTSS_StreamRef QTSS_RTSPSessionStream;  
typedef QTSS_StreamRef QTSS_RTSPRequestStream;  
typedef QTSS_StreamRef QTSS_RTPStreamStream;  
typedef QTSS_StreamRef QTSS_SocketStream;
```

## QTSS\_TimeVal

---

A `QTSS_TimeVal` is a signed 64-bit integer used to store time values. It is defined as

```
typedef SInt64 QTSS_TimeVal;
```

## QTSS\_ServerState

---

The `QTSS_ServerState` enumeration defines values that describe the server's state. Modules can set the server's state by setting the value of the `qtssSvrState` attribute in the `QTSS_ServerObject` object. The enumeration is defined as

```
enum  
{  
    qtssStartingUpState      = 0,  
    qtssRunningState         = 1,  
    qtssRefusingConnectionsState = 2,  
    qtssFatalErrorState      = 3,  
    qtssShuttingDownState    = 4,  
};
```

## QuickTime Streaming Server Module Reference

```

        qtssIdleState          = 5
    };
typedef UInt32 QTSS_ServerState;

```

**Constant descriptions**

qtssStartingUpState The server is starting up.

qtssRunningState The server is running.

qtssRefusingConnectionsState

Setting the server to this state causes the server to refuse new connections.

qtssFatalErrorState Setting the server to this state causes the server to quit.

qtssShuttingDownState Setting the server to this state causes the server to quit.

qtssIdleState Setting the server to this state causes the server to refuse new connections and disconnect existing connections.



# Index

---

## A

---

ACTIONDATA special tag 72  
ADD command option 82  
adding  
    attributes 30, 48–49, 98, 99  
    roles 94  
    services 53, 123  
Admin Protocol  
    access types 83  
    ADD command option 82  
    authorization 79  
    changing server settings 90  
    command options 82  
    data  
        access 79  
        references 81  
        types 84  
    DEL command option 82  
    GET command option 82  
    methods 79  
    parameter options 83  
    query  
        functionality 80  
        options 81  
    request  
        header 79  
        method 79  
        syntax 80  
    response method 79  
    responses  
        array values 86  
        errors 87  
        examples of 87  
        ok response 85  
        response data 85  
        root value 87  
        unauthorized response 84  
    session state 79

    special paths 90  
adminprotocol-lib.pl 68  
Advise File role 62  
allocating memory 96  
ANNOUNCE response 128  
attribute data types  
    converting  
        QTSS\_StringToValue 112  
        QTSS\_TypeStringToType 110  
        QTSS\_TypeToTypeString 111  
        QTSS\_ValueToString 113  
attributes  
    adding 30, 48–49, 98, 99  
    callback routines 97–113  
    IDs 105  
    information, getting 102, 103, 104  
    qtssRTPStrBufferDelayInSecs 26  
    qtssRTSPReqAbsoluteURL 23  
    qtssRTSPReqFullRequest 23  
    qtssRTSPReqRootDir 23  
    qtssRTSPReqStreamRef 50  
    removing 101  
    values of  
        getting 45–47, 106, 107, 108  
        removing 113  
        setting 47–48, 109–110  
authentication 69

---

## B

---

blocking I/O 26  
building modules  
    code fragment 17  
    compiled in server 16  
built-in services 54

## C

---

- CGIs 68
- changing root folder 95
- Client Session Closing role 28, 40–41
- client session objects 44
- Close File role 64
- compiling modules 16
- conventions, naming 27
- converting time 97
- CONVERTMSECTIMETOSTR special tag 74
- CONVERTTOLOCALTIME special tag 72
- customizing Web Admin 78

## D

---

- data types
  - Admin Protocol 84
  - naming conventions 27
  - QTSS\_AttrDataType 164
  - QTSS\_AttributeID 165
  - QTSS\_AttrInfoObject 134
  - QTSS\_AttrPermission 166
  - QTSS\_ClientSessionObject 135–137
  - QTSS\_CliSesTeardownReason 166
  - QTSS\_FileObject 138–139
  - QTSS\_ModuleObject 139–140
  - QTSS\_ModulePrefsObject 140–141
  - QTSS\_Object 167
  - QTSS\_PrefsObject 141–148
  - QTSS\_Role 167
  - QTSS\_RTTPPayloadType 167
  - QTSS\_RTTPSessionState 168
  - QTSS\_RTTPStreamObject 148–152
  - QTSS\_RTTPTransportType 168
  - QTSS\_RTSPHeaderObject 152
  - QTSS\_RTSPMethod 169
  - QTSS\_RTSPRequestObject 153–156
  - QTSS\_RTSPSessionObject 156–158
  - QTSS\_RTSPSessionStream 169
  - QTSS\_RTSPSessionType 169
  - QTSS\_RTSPStatusCode 169
  - QTSS\_ServerObject 158–163

- QTSS\_ServerState 170
- QTSS\_StreamRef 170
- QTSS\_TimeVal 170
- DELcommand option 82
- deleting memory 96
- DESCRIBE response 128
- dispatch routine 18
- dynamic modules 20

## E

---

- ECHODATA special tag 69
- Error Log
  - role 28, 32–33
- error messages 32

## F

---

- file system callback routines 121–123
- file systems
  - Advise File role 62
  - Close File role 64
  - Open File Preprocess role 59–60
  - Open File role 61–62
  - Read File role 62–63
  - Request Event File role 64–65
- Filter role 22, 28, 33–34
- flushing data 121
- FORMATFLOAT special tag 73
- functions
  - LoadCompiledInModules 16
  - QTSS\_AddAttribute 30
  - QTSS\_AddInstanceAttribute 99
  - QTSS\_AddRole 29, 94
  - QTSS\_AddService 53, 123
  - QTSS\_AddStaticAttribute 98
  - QTSS\_AppendRTSPHeader 24, 126
  - QTSS\_Delete 96
  - QTSS\_DoService 125
  - QTSS\_Flush 121
  - QTSS\_GetAttrInfoByID 102

QTSS\_GetAttrInfoByName 102, 103, 104  
 QTSS\_GetValue 106  
 QTSS\_GetValueAsString 107  
 QTSS\_GetValuePtr 108  
 QTSS\_IDForAttr 105  
 QTSS\_IDForService 124  
 QTSS\_Milliseconds 97  
 QTSS\_MilliSecsTo1970Secs 26, 97  
 QTSS\_New 34, 96  
 QTSS\_Play 24  
 QTSS\_RemoveInstanceAttribute 101  
 QTSS\_RemoveValue 113  
 QTSS\_SendRTSPHeaders 24, 126  
 QTSS\_SendStandardRTSPResponse 24, 127  
 QTSS\_SetValue 109–110  
 QTSS\_SStringToValue 112  
 QTSS\_TypeStringToType 110  
 QTSS\_TypeToTypeString 111  
 QTSS\_ValueToString 113  
 QTSS\_Write 24, 119  
 QTSS\_WriteV 24, 120

## G

---

GETcommand option 82  
 GETDATA special tag 70  
 getting  
   attribute IDs 105  
   attribute values 45–47  
   server time 97  
 GETVALUE special tag 70

## H

---

HASVALUE special tag 71  
 headers  
   appending to 126  
   sending 126  
 HTMLIZE special tag 76

## I

---

IDs, attribute 105  
 Initialize role 20, 28, 30  
 instance attributes  
   adding 99  
   removing 101  
 I/O, blocking 26

## L

---

language types 23  
   loadable bundle project type 17  
 LoadCompiledInModules function 16  
 log files 32

## M

---

main routine 17–18  
 MAKEARRAY special tag 71  
 memory  
   allocating 96  
   deleting 96  
 MODIFYDATA special tag 74  
 modules  
   call order 29  
   compiling 16  
   roles 27–42  
   static 17  
 monitoring server status 77–78  
 mutexes 25

## N

---

name conflicts, preventing 17  
 naming conventions 27

## O

---

### objects

- client session 44
- RTSP request 21, 23

### object types

- qtssClientSessionObjectType 44
- qtssPrefsObjectType 45
- qtssRTPStreamObjectType 44
- qtssRTSPHeaderObjectType 44
- qtssRTSPRequestObjectType 44
- qtssRTSPServerObjectType 45
- qtssRTSPSessionObjectType 44
- qtssTextMessageObjectType 45

Open File Preprocess role 59–60

Open File role 61–62

## P

---

parse.cgi CGI script 68

parsewithinput.cgi CGI script 68

parsing RTSP requests 23

PLAY response 128

Postprocessor role 24, 28, 38–39

preference file 91

Preprocessor role 23, 28, 36–37

preventing name conflicts 17

PRINTF special tag 75

PRINTHTMLFORMATFILE special tag 75

PROCESSFILE special tag 76

Process role 28

project type, loadable bundle 17

## Q

---

### QTSS

- services 52–54

- streams 50–52

QTSS\_AddAttribute function 30

QTSS\_AddInstanceAttribute function 99

QTSS\_AddRole function 29, 94

QTSS\_AddService function 53, 123

QTSS\_AddStaticAttribute function 98

QTSS\_AdviseFile\_Params structure 62

QTSS\_AppendRTSPHeader function 24, 126

QTSS\_AttrDataType data type 164

QTSS\_AttributeID data type 165

QTSS\_AttrInfoObject data type 134

QTSS\_AttrPermission data type 166

QTSS\_ClientSessionClosing\_Params  
structure 41

QTSS\_ClientSessionObject data type 135–137

QTSS\_CliSesTeardownReason data type 166

QTSS\_Delete function 96

QTSS\_DoService function 125

QTSS\_ErrorLog\_Params structure 32

QTSS\_FileObject data type 138–139

QTSS\_Flush function 121

QTSS\_GetAttrInfoByID function 102

QTSS\_GetAttrInfoByName function 102, 103, 104

QTSS\_GetValueAsString function 107

QTSS\_GetValue function 106

QTSS\_GetValuePtr function 108

QTSS\_IDForAttr function 105

QTSS\_IDForService function 124

QTSS\_Initialize\_Params structure 30

QTSS\_Milliseconds function 97

QTSS\_MilliSecsTo1970Secs function 26, 97

QTSS\_ModuleObject data type 139–140

QTSS\_ModulePrefsObject data type 140–141

QTSS\_New function 34, 96

QTSS\_Object data type 167

QTSS\_OpenFile\_Params structure 60, 61, 64

QTSS\_Play function 24

QTSS\_PrefsObject data type 141–148

QTSS\_ReadFile\_Params structure 63

QTSS\_RemoveInstanceAttribute function 101

QTSS\_RemoveValue function 113

QTSS\_RequestEventFile\_Params structure 64

QTSS\_Role data type 167

QTSS\_RTCPProcess\_Params structure 42

QTSS\_RTPPayloadType data type 167

QTSS\_RTPSendPackets\_Params structure 40

QTSS\_RTPSessionState data type 168

QTSS\_RTPStreamObject data type 148–152

QTSS\_RTPTransportType data type 168



QTSS\_RTSPHeaderObject data type 152  
 QTSS\_RTSPMethod data type 169  
 QTSS\_RTSPRequestObject data type 153–156  
 QTSS\_RTSPSessionObject data type 156–158  
 QTSS\_RTSPSessionStream data type 169  
 QTSS\_RTSPSessionType data type 169  
 QTSS\_RTSPStatusCode data type 169  
 QTSS\_SendRTSPHeaders function 24, 126  
 QTSS\_SendStandardRTSPResponse function 24, 127  
 QTSS\_ServerObject data type 158–163  
 QTSS\_ServerState data type 170  
 QTSS\_SetValue function 109–110  
 QTSS\_StandardRTSP\_Params structure 33, 34  
 QTSS\_StreamRef data type 170  
 QTSS\_StringToValue function 112  
 QTSS\_TimeVal data type 170  
 QTSS\_TypeStringToType function 110  
 QTSS\_TypeToTypeString function 111  
 QTSS\_ValueToStribg function 113  
 QTSS\_Write function 24, 119  
 QTSS\_WriteV function 24, 120  
 qtssClientSessionObjectType object type 44  
 qtssPrefsObjectType object type 45  
 qtssRTPStrBufferDelayInSecs attribute 26  
 qtssRTPStreamObjectType object type 44  
 qtssRTSPHeaderObjectType object type 44  
 qtssRTSPReqAbsoluteURL attribute 23  
 qtssRTSPReqFullRequestattribute 23  
 qtssRTSPReqRootDir attribute 23  
 qtssRTSPReqStreamRef attribute 50  
 qtssRTSPRequestObjectType object type 44  
 qtssRTSPSessionObjectType object type 44  
 qtssServerObjectObjectType object type 45  
 qtssTextMessageObjectType object type 45

## R

Read File role 62–63  
 Real Time Streaming Protocol  
   See RTSP  
 Real Time Transport Protocol  
   See RTP

Register role 20, 28, 29  
 removing instance attributes 101  
 Request Event File role 64–65  
 Request for Comments  
   See RFCs  
 request object, RTSP 21, 23  
 Request role 28, 37–38  
 Reread Preferences  
   role 28, 31  
   service 54  
 responding to RTSP requests 24  
 RFCs  
   1945 79  
   2396 79  
 roles  
   Advise File 62  
   Client Session Closing 28, 40–41  
   Close File 64  
   Error Log 28, 32–33  
   Initialize 20, 28, 30  
   Open File 61–62  
   Open File Preprocess 59–60  
   Read File 62–63  
   Register 20, 28, 29  
   Request Event File 64–65  
   Reread Preferences 28, 31  
   RTCP Process 28, 41–42  
   RTP Send Packets 24, 28, 40  
   RTSP Filter 22, 28, 33–34  
   RTSP Postprocessor 24, 28, 38–39  
   RTSP Preprocessor 23, 28, 36–37  
   RTSP Request 28, 37–38  
   RTSP Route 23, 28, 34–36  
   Shutdown 20, 28, 31  
 root folder, changing 95  
 Route role 23, 28, 34–36  
 routines  
   attribute callback 97–113  
   dispatch 18  
   file system callback 121–123  
   RTP callback 129–133  
   RTSP header callback 125–128  
   service callback 123–125  
   stream callback 114–121  
   utility callback 93–97

- RTCP Process role 28, 41–42
- RTP
  - callback routines 129–133
  - Send Packets role 24, 28, 40
  - sessions 25
- RTSP
  - Filter role 22, 28, 33–34
  - header callback routines 125–128
  - Postprocessor role 24, 28, 38–39
  - Preprocessor role 23, 28, 36–37
  - request objects 21, 23, 44
  - Request role 28, 37–38
  - requests
    - parsing 23
    - responding to 24
  - responses, sending 29, 127
  - Route role 23, 28, 34–36
  - sessions 23
- runtime environment 25

## S

---

- sending
  - RTSP headers 126
  - RTSP responses 29, 127
- Send Packets role 24, 28, 40
- server
  - settings, modifying 77–78, 90
  - status, monitoring 77–78
  - time 26
- service
  - callback routines 123–125
  - IDs, getting 124
- services
  - adding 53, 123
  - built-in 54
  - QTSS 52–54
  - Reread Preferences 54
  - using 53, 125
- sessions
  - client 23, 24, 40
  - header 44
  - RTP 25

- RTSP 23
- setting attribute values 47–48
- settings, modifying 77–78
- SETUP response 128
- shutdown, server 19
- Shutdown role 20, 28, 31
- source code, server 16
- special tags
  - ACTIONDATA 72
  - CONVERTMSECTIMETOSTR 74
  - CONVERTTOLOCALTIME 72
  - ECHODATA 69
  - FORMATFLOAT 73
  - GETDATA 70
  - GETVALUE 70
  - HASVALUE 71
  - HTMLIZE 76
  - MAKEARRAY 71
  - MODIFYDATA 74
  - PRINTFILE 75
  - PRINTHTMLFORMATFILE 75
  - PROCESSFILE 76
  - VALUEEQUALS 72
- startup, server 19
- static attributes
  - adding 98
- static modules 17, 20
- status, monitoring 77–78
- stream callback routines 114–121
- streams, QTSS 50–52
- structures
  - QTSS\_AdviseFile\_Params 62
  - QTSS\_ClientSessionClosing\_Params 41
  - QTSS\_ErrorLog\_Params 32
  - QTSS\_Initialize\_Params 30
  - QTSS\_OpenFile\_Params 60, 61, 64
  - QTSS\_ReadFile\_Params 63
  - QTSS\_RequestEventFile\_Params 64
  - QTSS\_RTCPPProcess\_Params 42
  - QTSS\_RTSPSendPackets\_Params 40
  - QTSS\_StandardRTSP\_Params 33, 34
- symbols, preventing name conflicts 17
- synchronous I/O 26

## T

---

TEARDOWN response 128  
threads 25  
time  
    converting 97  
    getting 97  
    server 26

## U

---

using  
    services 125  
using services 53  
utility callback routines 93–97

## V

---

VALUEEQUALS special tag 72  
verbosity level 32

## W

---

Web Admin  
    customizing 78  
    using 68–78  
writing  
    data to client 119, 120  
    log files 32