HP-UX 11i v1, HP-UX 11i v2, HP-UX 11i v3
# Table of Contents

## About This Document
- Intended Audience ............................................................................................................. 11
- Document Organization ..................................................................................................... 11
- Typographical Conventions ................................................................................................ 11
- Related Information ............................................................................................................ 12
- HP Encourages Your Comments ........................................................................................ 12

## 1 Introduction
- MSP Overview .................................................................................................................... 13
  - Multimedia Streaming Architecture ............................................................................. 13
  - MSP Framework ............................................................................................................ 14
  - Real-Time Protocol ........................................................................................................ 15
  - Real-Time Streaming Protocol ....................................................................................... 16
  - Session Description Protocol ......................................................................................... 16
- Files Provided with the MSP SDK ...................................................................................... 16
  - Include Files Supplied with the SDK for C ................................................................... 16
  - Libraries Supplied with the SDK for C ......................................................................... 17

## 2 Using MSP APIs
- RTP Library ......................................................................................................................... 19
  - RTP Session APIs ........................................................................................................... 20
    - Return Values .............................................................................................................. 21
  - RTP Data Packet APIs ................................................................................................... 21
  - RTCP Control Packet APIs ............................................................................................ 23
  - RTP I/O Control Option APIs ........................................................................................ 23
  - Monitoring RTP and RTCP I/O Conditions .................................................................. 25
  - Other APIs ..................................................................................................................... 25
- RTP Call Order Sequence ................................................................................................... 26
- RTSP Library ....................................................................................................................... 28
  - RTSP Data Structures .................................................................................................... 28
  - RTSP APIs ...................................................................................................................... 29
    - RTSP Connection APIs ................................................................................................. 31
    - RTSP Session APIs ....................................................................................................... 36
    - RTSP Message APIs ..................................................................................................... 38
    - Request or Response Line Information APIs ............................................................. 41
    - Header Information APIs ............................................................................................ 45
    - Message Body APIs .................................................................................................... 49
    - RTSP Header APIs ....................................................................................................... 51
    - Cache Control Header ................................................................................................. 51
The librtp APIs................................................................................................................107
The librtp Data Structures...............................................................................................110
The librtp Enumerated Values........................................................................................111

Index........................................................................................................................................115
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-1</td>
<td>Multimedia Streaming Architecture</td>
<td>14</td>
</tr>
<tr>
<td>1-2</td>
<td>MSP Framework</td>
<td>15</td>
</tr>
<tr>
<td>2-1</td>
<td>RTP Call Order Sequence</td>
<td>27</td>
</tr>
<tr>
<td>2-2</td>
<td>RTSP Client-Server Call Order Sequence</td>
<td>67</td>
</tr>
<tr>
<td>2-3</td>
<td>Creating an SDP Packet</td>
<td>78</td>
</tr>
<tr>
<td>2-4</td>
<td>Parsing an SDP Packet</td>
<td>79</td>
</tr>
</tbody>
</table>
## List of Tables

1-1 Header Files Shipped with the MSP SDK.................................................................17
1-2 Libraries Shipped with the MSP SDK..................................................................17
2-1 The option Values.................................................................................................24
2-2 The state Argument Values................................................................................36
2-3 Method Values....................................................................................................39
2-4 SDP Session Description......................................................................................68
2-5 SDP Media Description.........................................................................................68
2-6 The `sdp_get` and `sdp_add` APIs.....................................................................71
2-7 The `sdp_get_media*` and `sdp_add_media*` APIs..............................................74
3-1 RTP Troubleshooting Tips....................................................................................81
3-2 RTSP Troubleshooting Tips................................................................................83
A-1 The `librtsp` APIs...........................................................................................107
A-2 The `librtsp` Data Structures..........................................................................110
A-3 The `librtsp` Enumerated Values...................................................................111
About This Document

This manual describes how to install and use the HP-UX Multimedia Streaming Protocols (MSP) Software Developer’s Kit (SDK) on the HP-UX 11i v1, HP-UX 11i v2, and HP-UX 11i v3 operating systems.

This manual assumes that the HP-UX 11i v1, HP–UX 11i v2, or HP-UX 11i v3 operating system software and the appropriate files, scripts, and subsets are installed.

The document printing date and part number indicate the document’s current edition. The printing date will change when a new edition is printed. Minor changes may be made at reprint without changing the printing date. The document part number will change when extensive changes are made.

The latest version of the document will be available at: http://www.docs.hp.com

Document updates can be issued between editions to correct errors or document product changes. To ensure that you receive the updated or new edition, subscribe to the appropriate support service.

Contact your HP sales representative for details.

Intended Audience

This manual is intended for application developers responsible for developing multimedia streaming applications using the Multimedia Streaming Protocol (MSP) APIs. Developers are expected to have knowledge of operating system concepts, library functions, and C coding. They should also have knowledge of Transmission Control Protocol/Internet Protocol (TCP/IP) networking concepts and network configuration. This manual is not a C, MSP or TCP/IP tutorial.

Document Organization

The HP-UX Multimedia Streaming Protocols (MSP) Programmer’s Guide is organized as follows:

Chapter 1  Chapter 1 (page 13) presents an overview of MSP and lists the components that the MSP software contains.

Chapter 2  Chapter 2 (page 19) describes the libraries in the MSP software.

Chapter 3  Chapter 3 (page 81) describes how to troubleshoot the MSP libraries.

Chapter 4  Chapter 4 (page 85) includes sample programs for RTP, RTSP, and SDP.

Typographical Conventions

This document uses the following typographical conventions:

audit(5)  An HP-UX manpage. The name of the manpage is audit and 5 is the section in the HP-UX Reference. On the web and on the Instant
Information CD, it may be a link to the manpage itself. From the HP-UX command line, you can enter “man audit” or “man 5 audit” to view the manpage. See man(1).

Book Title
The title of a book. On the web and on the Instant Information CD, it may be a link to the book itself.

KeyCap
The name of a keyboard key. Note that Return and Enter both refer to the same key.

Emphasis
Text that is emphasized.

Emphasis
Text that is strongly emphasized.

Term
The defined use of an important word or phrase.

ComputerOut
Text displayed by the computer.

UserInput
Commands and other text that you type.

Command
A command name or qualified command phrase.

Variable
The name of a variable that you may replace in a command or function or information in a display that represents several possible values.

[ ]
The contents are optional in formats and command descriptions.

{}  The contents are required in formats and command descriptions. If the contents are a list separated by @, you must choose one of the items.

...
The preceding element may be repeated an arbitrary number of times.

| Separates items in a list of choices.

Related Information

Many sections of this manual refer to RFCs (for example, RFC 2327 (SDP: Session Description Protocol)) for more information on certain networking topics. These documents publicize Internet standards, new research concepts, and status memos about the Internet. You can access the full range of RFC documents and information about the Internet Engineering Task Force (IETF) at the following URL:

http://www.ietf.org/rfc.html

HP Encourages Your Comments

HP encourages your comments concerning this document. We are committed to providing documentation that meets your needs. Send any errors found, suggestions for improvement, or compliments to:

feedback@fc.hp.com

Include the document title, manufacturing part number, and any comment, error found, or suggestion for improvement you have concerning this document.
1 Introduction

This chapter presents an overview of the Multimedia Streaming Protocols (MSP) Software Developer’s Kit (SDK) and its supported features on your HP-UX 11i v1, HP-UX 11i v2, and HP-UX 11i v3 operating systems.

This chapter discusses the following topics:
- “MSP Overview” (page 13)
- “Files Provided with the MSP SDK” (page 16)

MSP Overview

MSP enables you to transfer audio and video files to a remote location in real time. Streaming multimedia data is a transaction between the server and client. The client, such as RealPlayer or Winamp, is a user application that accesses the media. The server is an application that provides all the client applications with the multimedia content. Unlike the download-and-play mechanism, the multimedia streaming client starts playing the media packets as soon as they arrive, without holding back to receive the entire file. While this technology reduces the client’s storage requirements and startup time for the media to be played, it introduces a strict timing relationship between the server and the client.

MSP defines the transaction that is used to establish a connection and transmit the media from the server and client. The HP-UX MSP SDK uses the following suite of protocols:
- Real-Time Protocol (RTP)
- Real-Time Control Protocol (RTCP)
- Real-Time Streaming Protocol (RTSP)
- Session Description Protocol (SDP)

The MSP framework for HP-UX multimedia streaming servers comprises libraries required for implementing these protocols and transmitting real-time data. These libraries use underlying transport mechanisms, such as TCP and UDP, to deliver services. The MSP implementation on HP-UX offers high performance and enables you to take advantage of the scalability, reliability, and high availability of the HP-UX operating system.

The MSP SDK conforms to the following RFCs:
- RFC 2326 – Real Time Streaming Protocol (RTSP)
- RFC 2327 – SDP: Session Description Protocol

Multimedia Streaming Architecture

Figure 1-1 illustrates the architecture of a typical multimedia streaming setup.
Multimedia streaming setup, as shown in Figure 1-1, includes two types of interactions. In the server part, the multimedia streaming server accepts multimedia data or input from any of the following sources:

- Live broadcast, such as a digital camera connected to the computer port.
- Data stored in the form of media.
- Data stored on machines in a network.

The streaming server processes the real-time multimedia data and sends it to the client, such as a mobile phone via a network.

**MSP Framework**

Figure 1-2 illustrates the HP-UX MSP framework.
The HP-UX MSP framework comprises the following modules:
• Real-Time Protocol (RTP)
• Real-Time Control Protocol (RTCP)
• Real-Time Streaming Protocol (RTSP)
• Session Description Protocol (SDP)

The MSP framework contains the RTSP and SDP modules in the user space and the RTP and RTCP modules in the kernel space. Applications can use the APIs provided in the user space MSP library to access the kernel components, namely, RTP and RTCP.

Real-Time Protocol

RTP is a transport protocol that provides end-to-end network transport functions for applications transmitting data with real-time properties, such as interactive audio and video. Services that use RTP include payload type identification, sequence numbering, timestamping and delivery monitoring. Applications run RTP on top of the User Datagram Protocol (UDP). RTP includes RTCP, a closely linked protocol, to provide a mechanism for reporting feedback on the transmitted real-time data.

You can use RTP in the following scenarios:
• Multicast audio conference
• Audio and video conference

For information on RTP APIs, see “RTP Library” (page 19).

Real-Time Streaming Protocol

RTSP controls the transfer of real-time media data and serves as a network remote control for multimedia services. The client requests a selected media file from the server using RTSP. The server, in response, returns information required by the client to transfer the data. Applications run RTSP on top of TCP.

For information on RTSP APIs, see “RTSP Library” (page 28).

Session Description Protocol

The Session Description Protocol (SDP) describes the general real-time multimedia sessions. A multimedia client uses SDP to announce a conference session by periodically multicasting an announcement packet to a familiar multicast address and port using the Session Announcement Protocol (SAP).

SDP conveys information about media streams in multimedia sessions and allows the recipients of a session description to participate in the session. SDP is normally used in an Internetwork but can also be used for multimedia conferences in other network environments.

For more information on SDP APIs, see “SDP Library” (page 67).

Files Provided with the MSP SDK

MSP is a Software Developer’s Kit (SDK) that contains C header files, C libraries, and example programs. To install the SDK, you must download the SDK package from the HP software depot web site at:

http://www.software.hp.com

Unpack the files to the directory of your choice. When you unpack the SDK depot, it will create several directories and populate them with the various files provided with the MSP SDK. The directories and files supplied with the SDK are detailed in the following sections:

• “Include Files Supplied with the SDK for C” (page 16)
• “Libraries Supplied with the SDK for C” (page 17)

Include Files Supplied with the SDK for C

Table 1-1 lists the header files that the MSP SDK for C includes.
### Table 1-1 Header Files Shipped with the MSP SDK

<table>
<thead>
<tr>
<th>Header File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/usr/include/rtp.h</td>
<td>Specifies the RTP header file.</td>
</tr>
<tr>
<td>/usr/include/rtsp.h</td>
<td>Specifies the RTSP header file.</td>
</tr>
<tr>
<td>/usr/include/sdp.h</td>
<td>Specifies the SDP header file.</td>
</tr>
</tbody>
</table>

### Libraries Supplied with the SDK for C

The MSP SDK for C includes different library sets. The specific library files that you link with depend on the type of application or applications you are building.

In the HP-UX 11i v2 and HP-UX 11i v3 operating systems both the 32-bit and 64-bit versions of MSP libraries are delivered but only the 32-bit version of MSP libraries are delivered in the HP-UX 11i v1 operating system.

Table 1-2 outlines the library files provided with the MSP SDK for C.

### Table 1-2 Libraries Shipped with the MSP SDK

<table>
<thead>
<tr>
<th>Library File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>librtp</td>
<td>Specifies the RTP library for HP-UX.</td>
</tr>
<tr>
<td>librtsps</td>
<td>Specifies the RTSP library for HP-UX.</td>
</tr>
<tr>
<td>libsdp</td>
<td>Specifies the SDP library for HP-UX.</td>
</tr>
</tbody>
</table>
This chapter describes how to use the Application Programming Interfaces (APIs) in the MSP SDK.

It discusses the following topics:

- “RTP Library” (page 19)
- “RTP Call Order Sequence” (page 26)
- “RTSP Library” (page 28)
- “RTSP Call Order Sequence” (page 63)
- “SDP Library” (page 67)
- “SDP Call Order Sequence” (page 76)

RTP Library

This section describes the APIs in the RTP library.

The RTP library contains various categories of APIs, and the RTP header file includes the relevant data structures. The following lists the types of APIs that the RTP library contains:

- RTP session APIs
  
  The RTP descriptor, rd, identifies the RTP session. It facilitates all the RTP operations. It encapsulates information on data transaction and RTCP feedback. The library provides APIs for retrieving information about members of an RTP session, sending or receiving media data provided to the appropriate API, with the help of rd.
  
  The following lists the RTP session APIs:

  - rtp_open
  - rtp_close

- RTP data packet APIs

  The RTP packet data structure begins with the mandatory fixed header to be transmitted in an RTP packet. The optional data, such as the contributing source information and extension header, follow this fixed header. Finally, the media data follows this information. The APIs provided by this library require the RTP packet in such a format. The RTP library fills in the version sequence number and synchronization source identifier fields of the RTP header.

  The following lists the RTP data packet APIs:

  - rtp_send
  - rtp_recv
  - rtp_free_buf
• RTCP control packet APIs
  The RTP library provides APIs for retrieving control packets from the library. The library automatically generates the control packets and sends them to all the members of the RTP session.
  The following lists the RTCP control packet APIs:
  — rtcp_recv
  — rtcp_free_buf

• RTP I/O control option APIs
  The RTP library provides options for setting and retrieving the members of the active RTP session that can be configured by the application. The library also provides APIs for configuring these options and retrieving the current option settings.
  You can use the rtp_ioctl API to specify and retrieve options.

• RTP monitoring and RTCP I/O conditions APIs
  The RTP library provides APIs for ascertaining the possibility of sending RTP packets and for receiving RTP and RTCP packets in a particular RTP session, identified by the rd.
  The following lists the RTP and RTCP monitoring APIs:
  — rtp_poll
  — rtcp_poll

• Other APIs
  The following lists the other APIs that the RTP library includes:
  — rtcp_parse_pkt
  — rtcp_free_pkt

RTP Session APIs

RTP session APIs use rd, an integer value, to encapsulate the handling of lower level protocols (UDP over IP) by the RTP library.
  The following lists the APIs that the RTP session module contains:
• rtp_open
  You can use rtp_open to initiate a transport session using the UDP transport protocol.
  The declaration of rtp_open is as follows:
  
  # include <rtp.h>
  int rtp_open(char *rtp_addr, char *rtcp_addr);
  To execute rtp_open, pass the following socket structures as arguments:
— **rtp_addr**

This is a pointer to an IPv4 or IPv6 socket address structure.

— **rtcp_addr**

This is a pointer to an IPv4 or IPv6 socket address structure.

These arguments point to a structure that specifies the address and port on which RTP and RTCP packets are received.

- **rtp_close**

You can use `rtp_close` to terminate an RTP session, represented by `rd`, that is obtained by calling `rtp_open`. This API closes the RTP session and frees all the memory associated with the session and internal structures that map to lower level protocols.

The declaration of `rtp_close` is as follows:

```c
#include <rtp.h>
int rtp_close(int rd);
```

**Return Values**

Unless explicitly stated, the RTP APIs return zero or a positive integer upon success; upon failure, they return a specific error number, which is a negative value.

**RTP Data Packet APIs**

An RTP data packet consists of a fixed RTP header of length 12 octets, followed by the optional Contributing Source (CSRC) list and extension headers. The length of the header helps in assessing the end of the data block.

RTP data packet APIs use `rtp_pkt_t`, a data structure, as follows:

```c
typedef struct {
    unsigned short v:2;      /* packet type */
    unsigned short p:1;      /* padding flag */
    unsigned short x:1;      /* header extension flag */
    unsigned short cc:4;     /* CSRC count */
    unsigned short m:1;      /* marker bit */
    unsigned short pt:7;     /* payload type */
    uint16_t seq;            /* sequence number */
    uint32_t ts;             /* timestamp */
    uint32_t ssrc;           /* synchronization source */
    uint32_t *csrcp;         /* CSRC list */
    rtp_xtn_hdr_t *xtnhdrp;  /* extension headers */
    char *datap;             /* RTP data */
    int datalen; } rtp_pkt_t;
```

The RTP packet data structures contain all the optional fields of an RTP packet, such as the CSRC list and the extension headers, as pointers. If these pointers are NULL, APIs, for example `rtp_send`, treat them as non-existent.
Extension headers use `rtp_xtn_hdr_t`, a data structure, as follows:

```c
/* RTP Extension header */
typedef struct {
    uint16_t                 profile;
    uint16_t                 len;
    char                    *hdrp;
} rtp_xtn_hdr_t;
```

The following lists the RTP data packet APIs:

- **rtp_send**
  
  You can use this API to send an RTP packet to all the members that an application specifies, using the `rtp_ioctl` APIs.
  
  The declaration of `rtp_send` is as follows:
  ```c
  # include <rtp.h>
  int rtp_send(int rd, rtp_pkt_t *pkt);
  ```
  
  where `pkt` is a pointer to the `rtp_pkt_t` structure.

- **rtp_recv**
  
  You can use `rtp_recv` to receive and parse RTP packets.
  
  The declaration of `rtp_recv` is as follows:
  ```c
  # include <rtp.h>
  int rtp_recv(int rd, rtp_pkt_t *pkt, char **buf, int *buflen);
  ```
  
  where `rd` is an integer value that represents an open RTP session, `pkt` is a pointer to the `rtp_pkt_t` structure that contains the parsed RTP packet, `buf` is the buffer where the packet will be physically placed, and `buflen` is the length of `buf`.

  The `buf` buffer obtains memory, upon request. To file such a request for memory allocation, you must send a valid pointer that points to NULL; that is, `*buf` should be NULL for enabling memory allocation, but `buf` itself is not NULL. When the `pkt` argument is NULL, the RTP packet is not parsed. When `*buf` is not NULL, `buflen` points to the size of the buffer, `buf`.

  Upon success, `rtp_recv` returns the number of bytes written into the buffer `buf`. Upon failure, `rtp_recv` returns an error number, which is a negative value. If the return value is `RTP_EMOREDATA`, then `buflen` points to the total number of bytes required for holding the entire RTP DATA packet. Contents of `*buf` are not valid when an error is returned. To receive RTP packets, the buffer size must be at least 8 bytes more than the sum of the expected size of the RTP packet and the size of the `rtp_xtn_hdr_t` structure.

- **rtp_free_buf**

  You can use `rtp_free_buf` to free the memory allocated by `rtp_recv`.
  
  The declaration of `rtp_free_buf` is as follows:
  ```c
  # include <rtp.h>
  int rtp_free_buf(char *buf);
  ```
where \texttt{buf} is the size of the buffer. The return value of \texttt{rtp\_free\_buf} is not predictable if \texttt{buf} is not valid or not allocated by \texttt{rtp\_recv}.

**RTCP Control Packet APIs**

This section describes the RTCP control packet APIs that the RTP module includes. The following lists the RTCP control packet APIs:

- \texttt{rtcp\_recv}
  
  You can use \texttt{rtcp\_recv} to receive RTCP packets.
  
  The declaration of \texttt{rtcp\_recv} is as follows:
  
  ```c
  # include <rtp.h>
  int rtcp_recv(int rd, char **buf, int *buflen);
  ```
  
  where \texttt{rd} is an integer value, \texttt{buf} is the size of the buffer, and \texttt{buflen} is the length of the buffer.
  
  The \texttt{buf} buffer obtains memory, upon request. To trigger memory allocation, you must send a valid pointer that points to NULL; that is, \texttt{*buf} should be NULL for enabling memory allocation, but itself is not NULL. You must call \texttt{rtcp\_parse\_pkt} separately for parsing the RTCP packet.

- \texttt{rtcp\_free\_buf}
  
  You can use \texttt{rtcp\_free\_buf} to free the memory allocated by \texttt{rtcp\_recv}.
  
  The declaration of \texttt{rtcp\_free\_buf} is as follows:
  
  ```c
  # include <rtp.h>
  int rtcp_free_buf(char *buf);
  ```
  
  The return value of \texttt{rtcp\_free\_buf} is not predictable if \texttt{buf} is not valid or not allocated by \texttt{rtcp\_recv}.

**RTP I/O Control Option APIs**

To set or retrieve I/O control options for a particular RTP session, use \texttt{rtp\_ioctl}.

The declaration of \texttt{rtp\_ioctl} is as follows:

```c
# include <rtp.h>
int rtp_ioctl(int rd, int option, ...);
```

You must pass the following arguments to \texttt{rtp\_ioctl}:

- \texttt{rd}
  
  The \texttt{rd} argument is an RTP session descriptor that specifies the session for which the option needs to be obtained or modified. You must obtain \texttt{rd} by calling \texttt{rtp\_open}.
The `option` argument indicates the type of operation to be performed. Each of the options expects one or more arguments. The `option` argument can take one of the values listed in Table 2-1.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTP_GSSRC (uint32_t *ssrc)</td>
<td>Obtains the Synchronization Source (SSRC) identifier.</td>
</tr>
<tr>
<td>RTP_GMEMCOUNT (uint32_t *ssrc)</td>
<td>Obtains the current count of members.</td>
</tr>
<tr>
<td>RTP_GMEMLIST (char *buf, int buflen)</td>
<td>Obtains the current list of members.</td>
</tr>
<tr>
<td>RTP_SSENDTOADDR</td>
<td>Sets an address to which the RTP/RTCP packets need to be sent.</td>
</tr>
<tr>
<td>RTCP_SSENDTOADDR (struct sockaddr *)</td>
<td></td>
</tr>
<tr>
<td>RTP_GSENDTOADDRNUM</td>
<td>Obtains the current count of addresses to which RTP/RTCP packets are sent.</td>
</tr>
<tr>
<td>RTCP_GSENDTOADDRNUM (uint64_t *cnt)</td>
<td></td>
</tr>
<tr>
<td>RTP_GSENDTOADDRMLIST</td>
<td>Obtains the current list of addresses to which RTP/RTCP packets are sent.</td>
</tr>
<tr>
<td>RTCP_GSENDTOADDRMLIST (char *buf, int buflen)</td>
<td></td>
</tr>
<tr>
<td>RTCP_SSDES (rtcp_sdes_item_t *item)</td>
<td>Sets the values for Source Description (SDES) items for the RTP session descriptor.</td>
</tr>
<tr>
<td>RTCP_GSDES (rtcp_sdes_item_t *item)</td>
<td>Obtains the values for SDES items for the RTP session descriptor.</td>
</tr>
<tr>
<td>RTCP_STIMESTAMPRATE (struct rtp_ioctsrate *srate)</td>
<td>Set the sampling rate for the media carried in the RTP packets, which are received on the RTP session descriptor. The jitter calculations will be correct only if the RTCP_STIMESTAMPRATE option is set.</td>
</tr>
<tr>
<td>RTCP_GTIMESTAMPRATE (struct rtp_ioctsrate *srate)</td>
<td>Get the sampling rate for the media carried in the RTP packets, which are received on the RTP session descriptor. The value set using the RTCP_STIMESTAMPRATE ioctl is retrieved.</td>
</tr>
</tbody>
</table>
Table 2-1 The option Values (continued)

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTP_AMCAST (struct ip_mreq *mreq)</td>
<td>Request to join the multicast group to receive multicast RTP packets on the RTP session descriptor.</td>
</tr>
<tr>
<td>RTCP_AMCAST (struct ip_mreq *mreq)</td>
<td>Request to join the multicast group to receive multicast RTCP packets on the RTP session descriptor.</td>
</tr>
</tbody>
</table>

For more information, type `man 3n rtp_ioctl` at the HP-UX command prompt.

Monitoring RTP and RTCP I/O Conditions

The RTP library provides a general mechanism for reporting I/O conditions associated with a set of RTP descriptors and for waiting until one or more specified conditions become true. Specified I/O conditions include the ability to read or write data without blocking and error conditions.

These APIs use the `rtp_poll_t` or the `rtcp_poll_t` data structure, as follows, to poll the I/O conditions for a set of RTP session descriptors:

```c
struct rtp_pollrd {
    int     rd;      /* rtp descriptor */
    short   events;  /* Requested conditions */
    short   revents; /* Reported conditions */
};
typedef struct rtp_pollrd rtp_pollrd_t;
typedef struct rtp_pollrd rtcp_pollrd_t;
```

The APIs that monitor RTP or RTCP I/O conditions are `rtp_poll` and `rtcp_poll`. In effect, `rtp_poll` or `rtcp_poll` polls the RTP/RTCP port for the availability of an RTP/RTCP packet for reading.

The declarations of `rtp_poll` and `rtcp_poll` are as follows:

```c
#include <rtp.h>
int rtp_poll(rtp_pollrd_t [], int nfds, int timeout);
#include <rtp.h>
int rtcp_poll(rtp_pollrd_t [], int nfds, int timeout);
```

The `rtp_poll` API also examines the ability of the RTP port to take in another RTP packet for transmission. This is not applicable to RTCP, as the library initiates the transmission of non-BYE RTCP packets and the BYE RTCP packet in a few cases, such as collision of SSRC.

Other APIs

The following lists the other APIs that the RTP library includes:
• rtcp_parse_pkt

The rtcp_parse_pkt API uses rtcp_pkt_t, the RTCP packet data structure that begins with the common header, as follows:

```c
typedef struct rtcp_pkt {
    rtcp_common_t *common;
    union {
        struct rtcp_sr *sr;
        struct rtcp_rr *rr;
        struct rtcp_sdes *sdes;
        struct rtcp_app *app;
        struct rtcp_bye *bye;
    } rtcp_unit;
};
typedef struct rtcp_pkt rtcp_pkt_t;
```

The declaration of rtcp_parse_pkt is as follows:

```c
int rtcp_parse_pkt(rtcp_pkt_t **pkt, char *buf, int buflen);
```

where buf contains the RTCP packet that was received using rtcp_recv, buflen contains the length of buf in bytes, and pkt is an array of type rtcp_pkt_t dynamically allocated by the rtcp_parse_pkt.

Upon success, rtcp_parse_pkt returns the number of elements in pkt; upon failure, it returns a negative number to indicate errors. You can free the memory allocated to pkt by calling rtcp_free_pkt.

• rtcp_free_pkt

You can use rtcp_free_pkt to free the memory allocated by rtcp_parse_pkt.

The declaration of rtcp_free_pkt is as follows:

```c
#include <rtp.h>
int rtcp_free_pkt(rtcp_pkt_t *pkt);
```

where pkt is a pointer to the rtcp_pkt_t structure.

**RTP Call Order Sequence**

Figure 2-1 illustrates the RTP call order sequence.
An application initiates a session for data transport using `rtp_open`. It may optionally set control options for the session using the `rtp_ioctl` interface. The control options set the addresses of the participating members for the session using the `RTP_SSENDTOADDRLST` option through `rtp_ioctl`.

To send a packet containing media data to all the participating members of a session using `rtp_send`, the application creates a media packet in the format prescribed by the RTP packet structure, `rtp_pkt_t`.

The application can send an RTP packet, stored in the RTP packet structure, using `rtp_send`. 

---

**Figure 2-1 RTP Call Order Sequence**

- `rtp_open`
- One or more `rtp_ioctl` interfaces
- Repoll
- Poll for `rtp_send` (rtp poll)
- `rtp_send`
- `rtp_close`
The RTP library initiates RTCP packets and eliminates the need for an application to initiate the RTCP packet. The library collects and maintains statistics. The library also handles timers for sending RTCP packets.

You can receive RTP and RTCP packets using `rtp_recv` and `rtcp_recv`, respectively. You can parse the compound RTCP packets by using `rtcp_parse_pkt`.

You can free the resources for the buffers using the corresponding freeing APIs.

To close an RTP session, use `rtp_close`.

**RTSP Library**

The Real-Time Streaming Protocol, or RTSP (RFC 2326), is an application-level protocol for control over the delivery of data with real-time properties. This protocol controls multiple data delivery sessions; it also provides a means for choosing delivery channels and mechanisms, such as UDP, multicast UDP, and TCP, as described in RFC 1889-RTP.

The RTSP library, `librtsp`, includes 40 interfaces for creating, sending, and receiving RTSP messages.

**RTSP Data Structures**

The RTSP library includes the following data structures:

- **rtsp_methods**
  This is an array of pointers to locations that contain the RTSP method strings. You must index this array using the enumerated RTSP method code.
  For example, `rtsp_methods [RTSP_DESCRIBE]` is a pointer that points to the location containing the `DESCRIBE` string.

- **rtsp_hdrs**
  This is an array of pointers to locations that contain the RTSP header field strings. You must index this array using the enumerated RTSP header field code.
  For example, `rtsp_hdrs [RTSP_CONTENT_LOCATION_HDR]` is a pointer that points to the location containing the `CONTENT_LOCATION` string.

- **rtsp_status_code**
  This is an array of RTSP status code integers. You must index this array using the enumerated RTSP status code.
  For example, `rtsp_status_code [RTSP_100]` is a pointer that points to the location containing the `100` string.

- **rtsp_status_code_str**
  This is an array of pointers to locations that contain the RTSP status codes as strings. You must index this array using the RTSP status code.
For example, `rtsp_status_code_str [RTSP_100]` is a pointer that points to the location containing the `100` string.

- `rtsp_status_text`
  This is an array of pointers to locations that contain the RTSP reason phrase strings corresponding to the RTSP status codes. You must index this array using the enumerated RTSP status code.
  For example, `rtsp_status_text [RTSP_100]` is a pointer that points to the location containing the string `Continue`. The reason phrase corresponding to the status code is `100`.

**RTSP APIs**

The following lists the types of APIs that the RTSP library provides:

- **Connection APIs**
  The following lists the RTSP connection APIs:
  - `rtsp_open`
  - `rtsp_create_conn`
  - `rtsp_recv`
  - `rtsp_close`

- **Session APIs**
  The following lists the RTSP session APIs:
  - `rtsp_init_session`
  - `rtsp_free_session_flds`
  - `rtsp_free_session`

- **Message APIs**
  The following lists the RTSP message APIs:
  - `rtsp_init_request_msg`
  - `rtsp_init_response_msg`
  - `rtsp_is_request_msg`
  - `rtsp_is_response_msg`
  - `rtsp_send_msg`
  - `rtsp_free_msg`

The following lists the APIs that set or retrieve the request or response line information in RTSP messages:

- `rtsp_set_msg_request_line`
- `rtsp_get_msg_request_line`
- `rtsp_set_msg_response_line`
- `rtsp_get_msg_response_line`
The following lists the APIs that set or retrieve the header information in RTSP messages:

- `rtsp_set_msg_hdr`
- `rtsp_get_msg_hdr`
- `rtsp_copy_msg_hdr`
- `rtsp_append_msg_hdr`

The following lists the APIs that set or retrieve the body of RTSP messages:

- `rtsp_set_msg_body`
- `rtsp_get_msg_body`

### Header APIs

You must specify the header information for most of the header types in the form of strings. However, you may have to define data structures for certain headers. You can use the header APIs to manipulate these header data structures. To store a list of values, you must link these data structures together.

The following lists the types of RTSP headers:

- **Cache control header**
  
  The `rtsp_cache_t` data structure stores information about the cache control header. The RTSP header file, `<rtsp.h>`, contains the definitions of this data structure and the various enumerated values for its members.

  The following lists the cache control APIs:

  - `rtsp_init_cache`
  - `rtsp_alloc_cache`
  - `rtsp_free_cache`

- **Range header**
  
  The `rtsp_range_t` data structure stores information about the range header. This data structure is a union of the `rtsp_smpte_t`, `rtsp_npt_t` and `rtsp_utc_t` data structures, which hold information about the SMPTE, NPT, and UTC timestamps, respectively. The RTSP header file, `<rtsp.h>` contains the definitions of these data structures and the various enumerated values for their members.

  The following lists the range header APIs:

  - `rtsp_init_range`
  - `rtsp_alloc_range`
  - `rtsp_free_range`
— RTP-info header

The `rtsp_rtpinfo_t` data structure stores information about the RTP-info header. The RTSP header file, `<rtsp.h>`, contains the definition of this data structure.

The following lists the RTP-info header APIs:

- `rtsp_init_rtpinfo`
- `rtsp_alloc_rtpinfo`
- `rtsp_free_rtpinfo`

— Transport header

The `rtsp_xport_spec_t` data structure stores information about the transport header. The RTSP header file, `rtsp.h`, contains the definitions of this data structure and the various enumerated values for its members.

The following lists the transport header `xport_spec` APIs:

- `rtsp_init_rtpinfo`
- `rtsp_alloc_rtpinfo`
- `rtsp_free_rtpinfo`

• URL

The RTSP URL data structure, `rtsp_url_t`, stores information about the RTSP URL, such as the protocol being used, host, port, and absolute path. It includes the API, `rtsp_parse_url`, that parses the RTSP URL specified as a string in RFC 2326 format into the `rtsp_url_t` structure.

• Connection option

You can set or retrieve the connection options and attributes of an RTSP connection, such as socket identifier and mapped session. Certain options can only be set and other options can only be retrieved.

The following lists the APIs that set or retrieve connection options:

- `rtsp_set_conn_opt`
- `rtsp_get_conn_opt`

RTSP Connection APIs

In order to send or receive messages to or from the peer, the application must first create an RTSP connection using either the `rtsp_open` or the `rtsp_create_conn` API. The RTSP connection data structure, `rtsp_conn_t`, identifies the transport connection to the peer and stores information, such as the socket descriptor, transport protocol being used, and message buffers for the connection. The `rtsp.h` file does not contain a complete definition of `rtsp_conn_t`. Hence, an application can use only pointers to the `rtsp_conn_t` structure.
The following describes the RTSP connection APIs:

- **rtsp_open**

  To create a new RTSP connection, use *rtsp_open*.

  The declaration of *rtsp_open* is as follows:

  ```
  #include <rtsp.h>

  rtsp_error_t rtsp_open(
      rtsp_url_t *rtspurl,
      rtsp_flags_t flags,
      rtsp_conn_t **conn
  );
  ```

  The *rtsp_open* function dynamically allocates memory for a new *rtsp_conn_t* structure and opens a transport connection to the peer. The *rtspurl* argument must point to the *rtsp_urt_t* structure, which contains the peer information. The address of the new *rtsp_conn_t* structure is returned in the location pointed by the *conn* argument. You must initialize the location pointed to by the *conn* argument to NULL before calling the function.

  You must pass the following arguments to *rtsp_open*:
  - *rtspurl* - Pointer to the *rtsp_url_t* structure.
  - *flags* - RTSP connection flags set by means of a bitwise OR operation. Currently, the *rtsp.h* file contains the declaration of only one connection flag, *RTSP_CFLAG_CS*.
  - *conn* - Pointer where the location for the new RTSP connection is returned.

  Upon success, the *rtsp_open* API returns *RTSP_SUCCESS*, and the pointer to the new *rtsp_conn_t* structure is returned in the location pointed to by the *conn* argument. Upon failure, *rtsp_open* returns one of the following error values:
  - **RTSP_EINVAL**
    - The *rtspurl* argument is NULL.
    - The *conn* argument is NULL.
    - The location pointed to by the *conn* argument is not NULL.
    - The *flags* argument contains an invalid value.
  - **RTSP_ENOMEM**
    Dynamic memory allocation failure.
— RTSP_ESYS
  System call error. Global variable errno contains the specific error number.
— RTSP_ENAMRESOLV
  Name resolution error.

• rtsp_create_conn
To create an RTSP connection, use rtsp_create_conn.
The declaration of rtsp_create_conn is as follows:
#include <rtsp.h>

rtsp_error_t rtsp_create_conn(
  int sockd,
  rtsp_conn_t **conn
);
The rtsp_create_conn function dynamically allocates memory for a new rtsp_conn_t structure and stores the socket descriptor identified by the sockd argument in the new rtsp_conn_t structure. The sockd argument must contain a connected SOCK_STREAM socket. The address of the new rtsp_conn_t structure is returned in the location pointed to by the conn argument. You must initialize the location pointed to by the conn argument to NULL before calling the function.
You must pass the following arguments to rtsp_create_conn:
— sockd
  Socket descriptor identifying the transport connection to the peer.
— conn
  Pointer where the location for the new RTSP connection is returned.

Upon success, the rtsp_create_conn API returns RTSP_SUCCESS, and the pointer to the new rtsp_conn_t structure is returned in the location pointed to by the conn argument. Upon failure, rtsp_create_conn returns one of the following error values:
— RTSP EINVAL
  ◦ The sockd argument does not contain a SOCK_STREAM socket.
  ◦ The conn argument is NULL.
  ◦ The location pointed to by the conn argument is not NULL.
— RTSP_ENOMEM
  Dynamic memory allocation failure.
— RTSP_ESYS
  System call error. Global variable errno contains the specific error number.
You can use `rtsp_recv` to receive an RTSP message or an interleaved media packet from an RTSP connection.

The declaration of `rtsp_close` is as follows:

```c
#include <rtsp.h>

rtsp_pckt_type_t rtsp_recv(
    rtsp_conn_t *conn,
    void *pckt,
    int *size,
    int *channel
);
```

You must pass the following arguments to `rtsp_recv`:

- `conn`
  Pointer to a location containing the `rtsp_conn_t` structure.

- `pckt`
  Pointer where the location of the received RTSP message or the interleaved media packet is returned.

- `size`
  Pointer to a location where the length, in bytes, of the interleaved media data is returned.

- `channel`
  Pointer to a location where the channel of the interleaved media data is returned.

Upon success, `rtsp_close` returns one of the following values:

- `RTSP_PCKT_REQ_MSG`
  Successful completion. Received data is an RTSP request message.

- `RTSP_PCKT_RSP_MSG`
  Successful completion. Received data is an RTSP response message.

- `RTSP_PCKT_DATA`
  Successful completion. Received data is an interleaved media stream.

  System call error. Global variable `errno` contains the specific error number.

- `<0`
  Failure.

Upon failure, `rtsp_recv` returns one of the following error values:
- RTSP_EINVAL
  - The `pkt` argument does not contain a SOCK_STREAM socket.
  - The `conn` argument is NULL.
  - The `size` argument is required and is NULL.
  - The `channel` argument is required and is NULL.
- RTSP_ENOBUFFS
  No buffer space to accommodate the received data.
- RTSP_ENOMEM
  Dynamic memory allocation failure.
- RTSP_ECONNABORTED
  The RTSP connection was closed by the peer.
- RTSP_ESYS
  System call error. Global variable `errno` contains the specific error number.

For more information, type `man 3 rtsp_recv` at the HP-UX command prompt.

- rtsp_close

You can use `rtsp_close` to close an RTSP connection.

The declaration of `rtsp_close` is as follows:

```c
#include <rtsp.h>

rtsp_error_t rtsp_close(
    rtsp_conn_t *conn
);
```

The `rtsp_close` function closes the transport connection to the peer associated with the RTSP connection and frees the dynamically allocated memory for the `rtsp_conn_t` structure and its members. The `conn` argument points to a `rtsp_conn_t` structure obtained by calling either `rtsp_open` or `rtsp_create_conn`.

Upon success, `rtsp_close` returns `RTSP_SUCCESS`; upon failure, it returns one of the following error values:

- RTSP_EINVAL
  - The `conn` argument is NULL.
- RTSP_ESYS
  - System call error. Global variable `errno` contains the specific error number.

For more information, type `man 3 rtsp_close` at the HP-UX command prompt.
The RTSP session data structure encapsulates all the necessary information about an RTSP session, such as the session URL, session ID and the state of the session. The udata member of this structure stores the session data that is specific to the application. The rtsp.h file contains the definition of the rtsp_session_t data structure.

The following is the definition of rtsp_session_t data structure:

typedef struct rtsp_session
{
    char    *url;
    char    *id;
    uint16_t idlen;
    rtsp_state_t state;
    rtsp_conn_t *conn;
    void     *udata;
} rtsp_session_t;

The url structure member contains the session URL and is initialized with the string present in the location pointed to by the url argument. If the url argument is NULL, the url structure member is set to NULL.

The id structure member contains the session ID and is initialized with the string present in the location pointed by the session_id argument. If the session_id argument is NULL, the id structure member is set to NULL.

The idlen structure member contains the length of the string present in the location pointed to by the id structure member.

The state structure member contains the state of the RTSP session. This member is initialized to RTSP_STATE_INIT. It takes one of the values described in Table 2-2.

Table 2-2 The state Argument Values

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTSP_STATE_INIT</td>
<td>Session is initialized.</td>
</tr>
<tr>
<td>RTSP_STATE_READY</td>
<td>SETUP or PAUSE message transaction was successfully completed.</td>
</tr>
<tr>
<td>RTSP_STATE_PLAY</td>
<td>PLAY message transaction was successfully completed.</td>
</tr>
<tr>
<td>RTSP_STATE_RECORD</td>
<td>RECORD message transaction was successfully completed.</td>
</tr>
</tbody>
</table>

The conn structure member is a pointer to the associated RTSP connection for this RTSP session. If there is no associated RTSP connection, the conn structure member is NULL.

The udata structure member is a void pointer and is for storing any session-specific application data. It is initialized to NULL.

There is no association between an RTSP session and an RTSP connection. Hence, different messages pertaining to a particular RTSP session may transmit multiple RTSP connections. However, an application may choose to create such an association. For
more information on creating an association between session and connection, see “RTSP Connection Option APIs” (page 61).

The following lists the RTSP session APIs:

- **rtsp_init_session**

  You can use `rtsp_init_session` to initialize the members of the `rtsp_session_t` structure with the arguments supplied by the application.

  The declaration of `rtsp_init_session` is as follows:

  ```c
  #include <rtsp.h>

  rtsp_error_t rtsp_init_session(
      char *url,
      char *session_id,
      rtsp_session_t *session
  );
  ```

  You must use the following arguments to `rtsp_init_session`:

  - **url**
    
    This is a pointer to a NULL terminated string that contains the session URL.

  - **session_id**
    
    This is a pointer to a NULL terminated string that contains the session ID.

  - **session**
    
    This is a pointer to a `rtsp_session_t` structure to be initialized.

  For more information on `rtsp_session_t` and its definition, type `man 3 rtsp_init_session` at the HP-UX command prompt.

  Upon success, `rtsp_init_session` returns `RTSP_SUCCESS`; upon failure, it returns one of the following error values:

  - **RTSP_EINVAL**
    
    The `session` argument is NULL.

  - **RTSP_ENOMEM**
    
    Dynamic memory allocation failure.

- **rtsp_free_session_flds**

  You can use `rtsp_free_session_flds` to free the resources associated with the `rtsp_session_t` structure members.

  The declaration of `rtsp_free_session_flds` is as follows:

  ```c
  #include <rtsp.h>

  void rtsp_free_session_flds(
      rtsp_session_t *session
  );
  ```
You must pass `session`, a pointer to an `rtsp_session_t` structure as an argument to `rtsp_free_session_flds`.

The `rtsp_free_session_flds` API does not return a value.

For more information, type `man 3 rtsp_free_session_flds` at the HP-UX command prompt.

- **rtsp_free_session**
  
  You can use `rtsp_free_session` to free the dynamically allocated memory for the `rtsp_session_t` structure and its members.
  
  The declaration of `rtsp_free_session` is as follows:

  ```c
  #include <rtsp.h>

  void rtsp_free_session(
     rtsp_session_t *session
  );
  ```

  You must pass `session`, a pointer to an `rtsp_session_t` structure to be freed, as an argument to `rtsp_free_session`.

  The `rtsp_free_session` API does not return a value.

  For more information, type `man 3 rtsp_free_session` at the HP-UX command prompt.

**RTSP Message APIs**

The RTSP message structure, `rtsp_msg_t`, stores complete information about an RTSP message: the request or response line, various RTSP header fields, and the message body.

The following lists the APIs that create and initialize the request and response messages:

- **rtsp_init_request_msg**
  
  You can use `rtsp_init_request_msg` to create and initialize an RTSP request message, and set the method and URL.
  
  The declaration of `rtsp_init_request_msg` is as follows:

  ```c
  #include <rtsp.h>

  rtsp_error_t rtsp_init_request_msg(
     rtsp_method_t method,
     char *url,
     rtsp_msg_t **msg
  );
  ```

  You must use the following arguments to `rtsp_init_request_msg`:
  
  - **method**
    
    The `method` argument specifies an enumerated RTSP method type and can take one of the values described in Table 2-3.
## Table 2-3 Method Values

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTSP_DESCRIBE</td>
<td>DESCRIBE request message.</td>
</tr>
<tr>
<td>RTSP_ANNOUNCE</td>
<td>ANNOUNCE request message.</td>
</tr>
<tr>
<td>RTSP_GET_PARAMETER</td>
<td>GET_PARAMETER request message.</td>
</tr>
<tr>
<td>RTSP_OPTIONS</td>
<td>OPTIONS request message.</td>
</tr>
<tr>
<td>RTSP_PAUSE</td>
<td>PAUSE request message.</td>
</tr>
<tr>
<td>RTSP_PLAY</td>
<td>PLAY request message.</td>
</tr>
<tr>
<td>RTSP_RECORD</td>
<td>RECORD request message.</td>
</tr>
<tr>
<td>RTSP_REDIRECT</td>
<td>REDIRECT request message.</td>
</tr>
<tr>
<td>RTSP_SETUP</td>
<td>SETUP request message.</td>
</tr>
<tr>
<td>RTSP_SET_PARAMETER</td>
<td>SET_PARAMETER request message.</td>
</tr>
<tr>
<td>RTSP_TEARDOWN</td>
<td>TEARDOWN request message.</td>
</tr>
</tbody>
</table>

- **url**
  The `url` argument points to a location containing the NULL terminated URL string.

- **msg**
  This points to a location where the address of the new RTSP request message is returned.

Upon success, `rtsp_init_request_msg` returns `RTSP_SUCCESS` and places the address of the allocated `rtsp_msg_t` structure in the location pointed to by the `msg` argument. Upon failure, it returns one of the following error values:

- **RTSP EINVAL**
  - The value in the `method` argument is invalid.
  - The `url` argument is NULL.
  - The `msg` argument is NULL.
  - The value in location pointed by the `msg` argument is not NULL.

- **RTSPENOMEM**
  Dynamic memory allocation failure.

- **rtsp_init_response_msg**
  You can use `rtsp_init_response_msg` to create and initialize an RTSP response message.
The `rtsp_init_response_msg` function dynamically allocates memory for the `rtsp_msg_t` structure and sets the status code and the reason phrase in the message to the specified values.

The declaration of `rtsp_init_response_msg` is as follows:

```c
#include <rtsp.h>

rtsp_error_t rtsp_init_response_msg(
    int status_code,
    uint8_t *reason_phrase,
    rtsp_msg_t *req_msg,
    rtsp_msg_t **msg
);
```

You must pass the following arguments to `rtsp_init_response_msg`:

- **status_code**
  Integer status code to be set in the message.

- **reason_phrase**
  Pointer to a location containing the NULL terminated `reason_phrase` to be set in the message. If the `reason_phrase` argument is NULL, then the API uses the reason phrase corresponding to the status code.

- **req_msg**
  Pointer to a location containing an RTSP request message from which the Cseq, Session, and Timestamp headers are copied.

- **msg**
  Pointer to a location where the address of the new RTSP response message is returned.

Upon success, `rtsp_init_response_msg` returns `RTSP_SUCCESS` and places the address of the allocated `rtsp_msg_t` structure in the location pointed to by the `msg` argument. Upon failure, `rtsp_init_response_msg` returns one of the following error values:

- **RTSP_EINVAL**
  - The `status_code` argument contains an invalid value.
  - The `msg` argument is NULL.
  - The value in the location pointed to by the `msg` argument is not NULL.

- **RTSP_ENOMEM**
  Dynamic memory allocation failure.
For more information, type `man 3 rtsp_init_response_msg` at the HP-UX command prompt.

- `rtsp_is_request_msg` and `rtsp_is_response_msg`
  You can use `rtsp_is_request_msg` and `rtsp_is_response_msg` functions to examine if an RTSP message is a request message or a response message, respectively.

  The declarations of these APIs are as follows:
  ```c
  #include <rtsp.h>

  int rtsp_is_request_msg(
      rtsp_msg_t *msg
  );

  int rtsp_is_response_msg(
      rtsp_msg_t *msg
  );
  ```

  You must pass the `msg` argument, which points to a location that contains the `rtsp_msg_t` structure.

  If the RTSP message pointed to by the `msg` argument is an RTSP request message, `rtsp_is_request_msg` returns 1. Otherwise, it returns 0. If the RTSP message pointed to by the `msg` argument is an RTSP response message, `rtsp_is_response_msg` returns 1. Otherwise, it returns 0.

  For more information on `rtsp_is_request_msg`, type `man 3 rtsp_is_request_msg` at the HP-UX command prompt. For more information on `rtsp_is_response_msg`, type `man 3 rtsp_is_response_msg` at the HP-UX command prompt.

Request or Response Line Information APIs

You can use the following APIs to set or retrieve the request and response lines of an RTSP message:

- `rtsp_set_msg_request_line`

  You can use `rtsp_set_msg_request_line` to set the RTSP method and the URL information in an RTSP message.

  The declaration of `rtsp_set_msg_request_line` is as follows:
  ```c
  #include <rtsp.h>

  rtsp_error_t rtsp_set_msg_request_line(
      rtsp_msg_t *msg,
      rtsp_method_t method,
  );
  ```
You must pass the following arguments to \texttt{rtsp_set_msg_request_line}:

- \textit{msg}
  
  Pointer to a location containing an RTSP request message.

- \textit{method}
  
  Enumerated RTSP method type. See Table 2-3 (page 39) for information on method types.

- \textit{url}
  
  Pointer to a location containing the NULL URL string.

Upon success, \texttt{rtsp_set_msg_request_line} returns \texttt{RTSP_SUCCESS}; upon failure, it returns one of the following error values:

- \texttt{RTSP_EINVAL}
  
  - The \textit{msg} argument is NULL.
  - The \textit{url} argument is NULL.
  - The \textit{method} argument contains an invalid method type.
  - The location pointed to by the \textit{msg} argument does not contain a request message.

- \texttt{RTSP_ENOMEM}
  
  Dynamic memory allocation failure.

\textbullet{} \texttt{rtsp_get_msg_request_line}

You can use \texttt{rtsp_get_msg_request_line} to retrieve the RTSP method and the URL information from an RTSP message.

The declaration of \texttt{rtsp_get_msg_request_line} is as follows:

\begin{verbatim}
#include <rtsp.h>

rtsp_error_t rtsp_get_msg_request_line(
    rtsp_msg_t *msg,
    char *url,
    uint16_t len
);  
\end{verbatim}

You must pass the following arguments to \texttt{rtsp_get_msg_request_line}:
- **msg**
  Pointer to a location containing an RTSP request message.

- **url**
  Pointer to a location where the URL in the message is returned.

- **len**
  Length, in bytes, of the location pointed by the **url** argument.

Upon success, **rtsp_get_msg_request_line** returns the method of the RTSP message in the form of an enumerated method code; upon failure, it returns **RTSP_EINVAL** in any of the following cases:

- The **msg** argument is NULL.
- The **url** argument is not NULL and the **len** argument contains an invalid value.
- The location pointed to by the **msg** argument does not contain an RTSP request message.

See Table 2-3 (page 39) for information on method values.

For more information, type `man 3 rtsp_get_msg_request_line` at the HP-UX command prompt.

**• rtsp_set_msg_response_line**

You can use **rtsp_set_msg_response_line** to set the status code and reason phrase information in an RTSP response message.

The declaration of **rtsp_set_msg_response_line** is as follows:

```c
#include <rtsp.h>

rtsp_error_t rtsp_set_msg_response_line(
    rtsp_msg_t *msg,
    int status,
    uint8_t *reason_phrase
);
```

You must pass the following arguments to **rtsp_set_msg_response_line**:

- **msg**
  Pointer to a location containing an RTSP response message.

- **status**
  Integer status code to be set in the message.

- **reason_phrase**
  Pointer to a location containing the NULL terminated reason phrase to be set in the message. If the **reason_phrase** argument is NULL, then the API uses the reason phrase corresponding to the status code.
Upon success, \texttt{rtsp\_set\_msg\_response\_line} returns \texttt{RTSP\_SUCCESS}; upon failure, \texttt{rtsp\_set\_msg\_response\_line} returns one of the following error values:

- \texttt{RTSP\_EINVAL}
  - The \texttt{msg} argument is NULL.
  - The \texttt{status} argument contains an invalid status code.
  - The location pointed to by the \texttt{msg} argument does not contain a response message.

- \texttt{RTSP\_ENOMEM}
  Dynamic memory allocation failure.

For more information, type \texttt{man 3 rtsp\_set\_msg\_response\_line} at the HP-UX command prompt.

- \texttt{rtsp\_get\_msg\_response\_line}

  You can use \texttt{rtsp\_get\_msg\_response\_line} to retrieve the status code and reason phrase information from an RTSP response message.

  The declaration of \texttt{rtsp\_get\_msg\_response\_line} is as follows:

  ```c
  #include <rtsp.h>

  rtsp_error_t rtsp_get_msg_response_line(
    rtsp_msg_t *msg,
    uint8_t *reason_phrase,
    uint16_t len
  );
  ```

  You must pass the following arguments to \texttt{rtsp\_get\_msg\_response\_line}:

  - \texttt{msg}
    Pointer to a location containing an RTSP response message.

  - \texttt{reason\_phrase}
    Pointer to a location where the reason phrase in the message is returned.

  - \texttt{len}
    Length, in bytes, of the location pointed to by the \texttt{reason\_phrase} argument.

  Upon success, \texttt{rtsp\_get\_msg\_response\_line} returns the integer status code of the RTSP response message. Upon failure, \texttt{rtsp\_get\_msg\_response\_line} returns \texttt{RTSP\_EINVAL} in any of the following cases:

  - The \texttt{msg} argument is NULL.
  - The location pointed to by the \texttt{msg} argument does not contain a response message.
  - The \texttt{reason\_phrase} argument is not NULL and \texttt{len} argument contains an invalid value.
For more information, type `man 3 rtsp_get_msg_response_line` at the HP-UX command prompt.

Header Information APIs

You can use the following APIs to set, modify, and retrieve header information in RTSP messages:

- `rtsp_set_msg_hdr`

  You can use `rtsp_set_msg_hdr` to set the header field in an RTSP message. The declaration of `rtsp_set_msg_hdr` is as follows:

  ```c
  #include <rtsp.h>
  
  rtsp_error_t rtsp_set_msg_hdr(
    rtsp_msg_t *msg,
    rtsp_hdr_type_t hdr_type,
    void *value,
    int size
  );
  ```

  You must pass the following arguments to `rtsp_set_msg_hdr`:
  - `msg`  
    Pointer to a location containing the `rtsp_msg_t` structure.
  - `hdr_type`  
    Enumerated RTSP header field type.
  - `value`  
    Pointer to a location containing the value for the header field.
  - `size`  
    Size (in bytes) of the location pointed to by the `value` argument.
Upon success, the \texttt{rtsp\_set\_msg\_hdr} function returns \texttt{RTSP\_SUCCESS}; upon failure, \texttt{rtsp\_set\_msg\_hdr} returns one of the following error values:

- \texttt{RTSP\_EINVAL}
  - The \texttt{msg} argument is NULL.
  - The value contained in the location pointed to by the \texttt{value} argument is invalid.
  - The value contained in the \texttt{size} argument is invalid.

- \texttt{RTSP\_EINVHDR}
  - Value contained in the \texttt{hdr\_type} argument is not a valid enumerated RTSP header type.
  - The header field specified by the \texttt{hdr\_type} argument cannot be set in the RTSP message pointed to by the \texttt{msg} argument.

- \texttt{RTSP\_ENOMEM}
  Dynamic memory allocation failure.

For more information, type \texttt{man 3 rtsp\_set\_msg\_hdr} at the HP-UX command prompt.

- \texttt{rtsp\_get\_msg\_hdr}

You can use \texttt{rtsp\_get\_msg\_hdr} to retrieve the header field from an RTSP message.

The declaration of \texttt{rtsp\_get\_msg\_hdr} is as follows:

```c
#include <rtsp.h>

tfsp_error_t rtsp_set_msg_hdr(
    rtsp_msg_t *msg,
    rtsp_hdr_type_t hdr_type,
    void *value,
    int *size
);
```

You can use the same function to retrieve headers from both request and response messages.

You must pass the following arguments to \texttt{rtsp\_get\_msg\_hdr}:

- \texttt{msg}
  Pointer to a location containing the \texttt{rtsp\_msg\_t} structure.

- \texttt{hdr\_type}
  Enumerated RTSP header field type.
— *value*
   Pointer to a location where the value for the header field is returned.

— *size*
   Pointer to a location containing the size (in bytes) of the location pointed to by the *value* argument.

Upon success, `rtsp_get_msg_hdr` returns one of the following values:

— **RTSP_SUCCESS**
  Successful completion. The location pointed to by the *value* argument stores header information.

— **RTSP_HDR_NOT_IN_MSG**
  The RTSP message does not contain the header indicated by the *hdr_type* argument.

Upon failure, `rtsp_get_msg_hdr` returns one of the following values:

— **RTSP_EINVAL**
  ◦ The *msg* argument is NULL.
  ◦ The *value* argument is NULL.
  ◦ The *size* argument is required and is NULL.
  ◦ The value contained in the location pointed to by the *value* argument is not NULL and the value in the location pointed to by the *size* argument is invalid.

— **RTSP_EINVHDR**
  ◦ The value contained in the *hdr_type* argument is not a valid enumerated RTSP header type.
  ◦ The header field specified by the *hdr_type* argument cannot be retrieved from the RTSP message.

— **RTSP_ENOMEM**
  Dynamic memory allocation failure.

*rtsp_copy_msg_hdr*

You can use `rtsp_copy_msg_hdr` to copy a header from one RTSP message to another.

The declaration of `rtsp_copy_msg_hdr` is as follows:

```c
#include <rtsp.h>

rtsp_error_t rtsp_copy_msg_hdr(
    rtsp_msg_t *dst_msg,
    rtsp_msg_t *src_msg,
    rtsp_hdr_type_t hdr_type
);
```
You must pass the following arguments to \texttt{rtsp\_copy\_msg\_hdr}:

- \texttt{dst\_msg}
  Pointer to a location containing the \texttt{rtsp\_msg\_t} structure.

- \texttt{src\_msg}
  Pointer to a location containing the \texttt{rtsp\_msg\_t} structure.

- \texttt{hdr\_type}
  Enumerated RTSP header field type.

Upon success, the existing information for the header in \texttt{dst\_msg} is overwritten with the header information from \texttt{src\_msg}.

Upon success, \texttt{rtsp\_copy\_msg\_hdr} returns \texttt{RTSP\_SUCCESS}; upon failure, it returns one of the following error values:

- \texttt{RTSP\_HDR\_NOT\_IN\_MSG}
  The \texttt{src\_msg} argument does not contain the header indicated by the \texttt{hdr\_type} argument.

- \texttt{<0}
  Failure.

For more information on enumerated header field types, type \texttt{man 3 rtsp\_copy\_msg} at the HP-UX command prompt.

- \texttt{rtsp\_append\_msg\_hdr}

You can use \texttt{rtsp\_append\_msg\_hdr} to append the header information in RTSP request and response messages.

The declaration of \texttt{rtsp\_append\_msg\_hdr} is as follows:

\begin{verbatim}
#include <rtsp.h>

rtsp_error_t rtsp_append_msg_hdr(
    rtsp_msg_t *msg,
    rtsp_hdr_type_t hdr_type,
    void *value,
    int size
);
\end{verbatim}

You must pass the following arguments to \texttt{rtsp\_append\_msg\_hdr}:

- \texttt{msg}
  Pointer to a location containing the \texttt{rtsp\_msg\_t} structure.

- \texttt{hdr\_type}
  Enumerated RTSP header field type.
— `value`
  Pointer to a location containing the value for the header field.

— `size`
  Size (in bytes) of the location pointed to by the `value` argument.

Upon success, `rtsp_append_msg_hdr` returns `RTSP_SUCCESS`; upon failure, it returns one of the following error values:

- **RTSP_EINVAL**
  - The `msg` argument is NULL.
  - The value contained in the location pointed to by the `value` argument is invalid.
  - The value contained in the `size` argument is invalid.

- **RTSP_EINVHDR**
  - The value contained in the `hdr_type` argument is not a valid enumerated RTSP header type.
  - The value contained in the `hdr_type` argument is not an appendable header field.
  - The header field specified by the `hdr_type` argument cannot be set in the RTSP message.

- **RTSP_ENOMEM**
  Dynamic memory allocation failure.

**Message Body APIs**

You can use the following APIs to set or retrieve the body of an RTSP message:

- **rtsp_set_msg_body**
  To set the body of an RTSP message, use `rtsp_set_msg_body`. The declaration of `rtsp_set_msg_body` is as follows:
  ```c
  #include <rtsp.h>

  rtsp_error_t rtsp_set_msg_body(
      rtsp_msg_t *msg,
      uint8_t *body,
      uint16_t content_len
  );
  ```

  You must pass the following arguments to `rtsp_set_msg_body`:
— **msg**
  Pointer to a location containing the `rtsp_msg_t` structure.

— **body**
  Pointer to a location containing the body of the RTSP message.

— **content_len**
  Length, in bytes, of the message body.

The `rtsp_set_msg_body` function overwrites the body of an RTSP message with the specified value and sets the content length header in the RTSP message to the value specified by the `content_len` argument. If the `body` argument is NULL, the message body, if any, is removed and the content length header information in the message is also removed.

Upon success, `rtsp_set_msg_body` returns `RTSP_SUCCESS`; upon failure, it returns one of the following error values:

— **RTSP_EINVAL**
  - The `msg` argument is NULL.
  - The `body` argument is not NULL, but the `content_len` argument contains an invalid value.

— **RTSP_ENOMEM**
  Dynamic memory allocation failure.

For more information, type `man 3 rtsp_set_msg_body` at the HP-UX command prompt.

• **rtsp_get_msg_body**
To retrieve the body of an RTSP message, use `rtsp_get_msg_body`. The declaration of `rtsp_get_msg_body` is as follows:

```c
#include <rtsp.h>

rtsp_error_t rtsp_get_msg_body(
    rtsp_msg_t *msg,
    uint8_t *body,
    uint16_t *content_len
);
```

You must pass the following arguments to `rtsp_get_msg_body`: 

---
— `msg`
  Pointer to a location containing the `rtsp_msg_t` structure.

— `body`
  Pointer to a location where the body of the RTSP message is returned.

— `content_len`
  Pointer to a location containing the size of the location pointed to by the `body` argument.

Ensure that the location pointed to by the `content_len` argument contains the size of the buffer in the location pointed to by the `body` argument before calling `rtsp_get_msg_body`. The `rtsp_get_msg_body` function updates the location pointed to by the `content_len` argument with the actual size of the data returned in the location pointed to by the `body` argument.

Upon success, `rtsp_get_msg_body` returns `RTSP_SUCCESS`; upon failure, it returns one of the following error values:

— `RTSP_EINVAL`
  ◦ The `msg` argument is NULL.
  ◦ The `content_len` argument is NULL.
  ◦ The `body` argument is NULL.
  ◦ The `msg` argument is NULL.
  ◦ The location pointed to by the `content_len` argument contains an invalid value.

For more information, type `man 3 rtsp_get_msg_body` at the HP-UX command prompt.

RTSP Header APIs

You must specify the header information for most of the header types in the form of strings. However, you can define data structures for certain headers. You can also link these data structures together to store a list of values.

The following lists the types of RTSP headers:

- Cache control header
- Range header
- RTP-info header
- Transport header

Cache Control Header

The `rtsp_cache_t` data structure stores information about the cache control header. The `rtsp.h` file contains the definition of this data structure and the various enumerated values as follows:
typedef struct rtsp_cache {
    int32_t     directive:8;
    int32_t     seconds:24;
    uint8_t     *cache_ext;
    uint16_t    cache_ext_len;
    struct rtsp_cache *next;
} rtsp_cache_t;

The directive structure member in the rtsp_cache_t structure contains the cache directive and takes one of the following values:

- RTSP_CACHE_NONE
- RTSP_CACHE_MAX_STALE
- RTSP_CACHE_NO_CACHE
- RTSP_CACHE_MIN_FRESH
- RTSP_CACHE_ONLYIF_CACHED
- RTSP_CACHE_PUBLIC
- RTSP_CACHE_PRIVATE
- RTSP_CACHE_NO_TRANSFORM
- RTSP_CACHE_MUST_REVAL
- RTSP_CACHE_PROXY
- RTSP_CACHE_MAX_AGE
- RTSP_CACHE_EXT

The seconds structure member contains the time information, in seconds, for the RTSP_CACHE_MAX_STALE, RTSP_CACHE_MIN_FRESH, and RTSP_CACHE_MAX_AGE cache directives.

The following lists the cache control header APIs:

- rtsp_init_cache
  You can use rtsp_init_cache to initialize the structure members of the rtsp_cache_t structure with their default values.
  The declaration of rtsp_init_cache is as follows:

      #include <rtsp.h>

      void rtsp_init_cache(
          rtsp_cache_t *cache
      );

  You must pass cache, a pointer to the rtsp_cache_t structure, as an argument to rtsp_init_cache.
  The rtsp_init_cache function does not return a value.
For more information, type `man 3 rtsp_init_cache` at the HP-UX command prompt.

- **rtsp_alloc_cache**
  You can use `rtsp_alloc_cache` to dynamically allocate memory for the `rtsp_cache_t` structure and to initialize the structure members with application-specific values.
  The declaration of `rtsp_alloc_cache` is as follows:
  ```c
  #include <rtsp.h>

  rtsp_cache_t *rtsp_alloc_cache(
       rtsp_cache_t *cache2dup
  );
  ```
  You must pass `cache2dup`, a pointer to the `rtsp_cache_t` structure, as an argument to `rtsp_alloc_cache`.
  Upon success, `rtsp_init_cache` returns a pointer to the `rtsp_cache_t` structure. If memory cannot be dynamically allocated, the `rtsp_alloc_cache` function returns NULL.
  For more information, type `man 3 rtsp_alloc_cache` at the HP-UX command prompt.

- **rtsp_free_cache**
  You can use `rtsp_free_cache` to free the dynamically allocated memory for the `rtsp_cache_t` structure and its members.
  The declaration of `rtsp_free_cache` is as follows:
  ```c
  #include <rtsp.h>

  void rtsp_free_cache(
       rtsp_cache_t *cache
  );
  ```
  You must pass `cache`, a pointer to the `rtsp_cache_t` structure to be freed, as an argument to `rtsp_free_cache`.
  The `rtsp_free_cache` function does not return a value.
  For more information, type `man 3 rtsp_free_cache` at the HP-UX command prompt.

**Range Header**

The `rtsp_range_t` data structure stores information about the range header. This data structure is a union of the `rtsp_smpte_t`, `rtsp_npt_t`, and `rtsp_utc_t` data structures, which hold information about the SMPTE, NPT, and UTC timestamps, respectively. The `rtsp.h` file contains definitions of these data structures and the various enumerated values for their members.
The definition of the `rtsp_range_t` data structure is as follows:

```c
typedef struct rtsp_range
{
    uint8_t             range_type;
    uint8_t             range_flag;
    union
    {
        struct
        {
            rtsp_npt_t  start;
            rtsp_npt_t  end;
        } npt;
        struct
        {
            rtsp_utc_t  start;
            rtsp_utc_t  end;
        } utc;
        struct
        {
            rtsp_smpte_t start;
            rtsp_smpte_t end;
        } smpte;
    } range;
    struct rtsp_range   *next;
} rtsp_range_t;
```

The `range_type` structure member identifies the type of timestamp. For more information on the `rtsp_cache_t` data structure and enumerated values for its members, type `man 3 rtsp_init_range` at the HP-UX command prompt.

The following lists the RTSP range APIs:

- **rtsp_init_range**

  You can use `rtsp_init_range` to initialize the members of the `rtsp_range_t` structure with the default values.

  The declaration of `rtsp_init_range` is as follows:

  ```c
  #include <rtsp.h>

  void rtsp_init_range(
      rtsp_range_t *range
  );
  ```

  You must pass `range`, a pointer to the `rtsp_range_t` structure, as an argument to `rtsp_init_range`.  

The `rtsp_init_range` function does not return a value. For more information, type `man 3 rtsp_init_range` at the HP-UX command prompt.

- `rtsp_alloc_range`
  You can use `rtsp_alloc_range` to dynamically allocate memory for the `rtsp_range_t` structure and to initialize its members with application-specific values.
  
  The declaration of `rtsp_alloc_range` is as follows:
  ```c
  #include <rtsp.h>
  
  rtsp_range_t *rtsp_alloc_range(
      rtsp_range_t *range2dup
  );
  ```
  
  You must pass `range2dup`, a pointer to the `rtsp_range_t` structure, as an argument to `rtsp_alloc_range`.
  
  Upon success, `rtsp_alloc_range` returns a pointer to the `rtsp_range_t` structure. If memory cannot be dynamically allocated, the `rtsp_alloc_range` function returns NULL. For more information, type `man 3 rtsp_alloc_range` at the HP-UX command prompt.

- `rtsp_free_range`
  You can use `rtsp_free_range` to free the dynamically allocated memory for the `rtsp_range_t` structure and its members.
  
  The declaration of `rtsp_free_range` is as follows:
  ```c
  #include <rtsp.h>
  
  void rtsp_free_range(
      rtsp_range_t *range
  );
  ```
  
  You must pass `range`, a pointer to the `rtsp_range_t` structure to be freed, as an argument to `rtsp_free_range`.
  
  The `rtsp_free_range` function does not return a value.
  
  For more information, type `man 3 rtsp_free_range` at the HP-UX command prompt.

RTP-Info Header

The `rtsp_rtpinfo_t` data structure stores information about the RTP-info header. The `rtsp.h` file contains the definition of `rtsp_rtpinfo_t` data structure as follows:

```c
typedef struct rtsp_rtpinfo
{
    char       *url;
    int32_t     seq;
};
```
int64_t rtptime;
struct rtsp_rtpinfo *next;
} rtsp_rtpinfo_t;

The url member of the rtsp_rtpinfo_t structure is a pointer to the location containing the stream URL specified in the RTP-info header. The seq member of the structure contains the sequence number and the rtptime member contains the RTP timestamp specified in the RTP-info header.

The following lists the RTP-info header APIs:

- **rtsp_init_rtpinfo**
  
  You can use rtsp_init_rtpinfo to initialize the members of rtsp_rtpinfo_t data structure with default values.

  The declaration of rtsp_init_rtpinfo is as follows:

  ```c
  #include <rtsp.h>
  
  void rtsp_init_rtpinfo(
      rtsp_rtpinfo_t *rtpinfo
  );
  ```

  You must pass rtpinfo, a pointer to the rtsp_rtpinfo_t structure, as an argument to rtsp_init_rtpinfo.

  The rtsp_init_rtpinfo function does not return a value.

  For more information, type `man 3 rtsp_init_rtpinfo` at the HP-UX command prompt.

- **rtsp_alloc_rtpinfo**
  
  You can use rtsp_alloc_rtpinfo to dynamically allocate memory for the rtsp_rtpinfo_t structure and to initialize its members with application-specific values.

  The declaration of rtsp_alloc_rtpinfo is as follows:

  ```c
  #include <rtsp.h>
  
  rtpinfo_t *rtsp_alloc_rtpinfo(
      rtsp_rtpinfo_t *rtpinfo2dup
  );
  ```

  You must pass rtpinfo2dup, a pointer to the rtsp_rtpinfo_t structure, as an argument to rtsp_alloc_rtpinfo.

  Upon success, rtsp_alloc_rtpinfo returns a pointer to the rtsp_rtpinfo_t structure. If memory cannot be dynamically allocated, the rtsp_alloc_rtpinfo function returns NULL.
For more information, type `man 3 rtsp_alloc_rtpinfo` at the HP-UX command prompt.

- **rtsp_free_rtpinfo**

  You can use `rtsp_free_rtpinfo` to free the dynamically allocated memory for the `rtsp_rtpinfo_t` structure and its members.

  The declaration of `rtsp_free_rtpinfo` is as follows:

  ```c
  #include <rtsp.h>

  void rtsp_free_rtpinfo(
      rtsp_rtpinfo_t *rtpinfo
  );
  ```

  You must pass `rtpinfo2dup`, a pointer to the `rtsp_rtpinfo_t` structure to be freed, as an argument to `rtsp_free_rtpinfo`.

  Upon success, `rtsp_free_rtpinfo` returns `RTSP_SUCCESS`. Upon failure, it returns an error number indicating an error.

  For more information, type `man 3 rtsp_free_rtpinfo` at the HP-UX command prompt.

**Transport Header**

The `rtsp_xport_spec_t` data structure stores information about the transport header. The `rtsp.h` file contains the definitions of this data structure and the various enumerated values for its members.

The definition of `rtsp_xport_spec_t` data structure is as follows:

```c
typedef struct rtsp_xport_spec
{
    uint8_t  xport_id;
    uint8_t  src_addr;
    uint8_t  dest_addr;
    int16_t  channel_start;
    int16_t  channel_end;
    uint16_t layers;
    uint16_t ttl;
    rtsp_port_range_t port;
    rtsp_port_range_t cport;
    rtsp_port_range_t sport;
    char      ssrc[9];
    uint8_t ncast:2;
    uint8_t mode:3;
    uint8_t append:1;
    uint8_t reserved:2;
} rtsp_xport_spec_t;
```
struct rtsp_xport_spec  *next;

} rtsp_xport_spec_t;

The xport_id structure member points to a location that contains the transport specifier in the form transport/profile/lower-transport.

The src_addr and dest_addr structure members are pointers to the locations that contain the source and destination address parameters in the transport specifier.

The channel_start and channel_end structure members contain the range of the interleaved transport specifier parameter.

The layers and ttl structure members contain the multicast time-to-live and the multicast layers information.

The port, cport and sport structure members contain the multicast session RTP/RTCP port pair, unicast RTP/RTCP port pair on client, and unicast RTP/RTCP port pair on server, respectively. The RTP/RTCP port pair information is stored in the rtsp_port_range_t structure defined in the rtsp.h file.

The rtp member in rtsp_port_range_t structure contains the RTP port and the rtcp structure member contains the RTCP port.

The ssrc structure member contains the RTP synchronization source identifier for the unicast session.

The mode parameter can either be PLAY or RECORD. This information is stored in the mode structure member, which takes one of the following values:
- RTSP_MODE_PLAY
- RTSP_MODE_RECORD

The append structure member must be set to 1, if the append parameter is specified in the transport specifier.

The reserved structure member is currently ignored.

The following lists the RTSP transport header APIs:
- rtsp_init_xport_spec
  You can use rtsp_init_xport_spec to initialize the structure members of the rtsp_xport_spec_t structure with the default values.
  The declaration of rtsp_init_xport_spec is as follows:
  ```c
  #include <rtsp.h>

  void rtsp_init_xport_spec(
    rtsp_xport_spec_t *xport_spec
  );
  ```
  You must pass xport_spec, a pointer to the rtsp_xport_spec_t structure, as an argument to rtsp_init_xport_spec.
  The rtsp_init_xport_spec function does not return a value.
For more information, type `man 3 rtsp_init_xport_spec` at the HP-UX command prompt.

- **rtsp_alloc_xport_spec**
  You can use `rtsp_alloc_xport_spec` to dynamically allocate memory for `rtsp_xport_spec_t` structure and to initialize its members with application-specific values.
  
  The declaration of `rtsp_alloc_xport_spec` is as follows:
  
  ```c
  #include <rtsp.h>

  rtsp_xport_spec_t *rtsp_alloc_xport_spec(
      rtsp_xport_spec_t *xspec2dup
  );
  ```

  You must pass `xspec2dup`, a pointer to a `rtsp_xport_spec_t` structure, as an argument to `rtsp_alloc_xport_spec`.

  Upon success, `rtsp_alloc_xport_spec` returns a pointer to the `rtsp_xport_spec_t` structure. If memory cannot be dynamically allocated, `rtsp_alloc_xport_spec` returns `NULL`.

  For more information, type `man 3 rtsp_alloc_xport_spec` at the HP-UX command prompt.

- **rtsp_free_xport_spec**
  You can use `rtsp_free_xport_spec` to free the dynamically allocated memory for the `rtsp_xport_spec_t` structure and its members.

  The declaration of `rtsp_free_xport_spec` is as follows:
  
  ```c
  #include <rtsp.h>

  void rtsp_free_xport_spec(
      rtsp_xport_spec_t *xport_spec
  );
  ```

  You must pass `xport_spec`, a pointer to the `rtsp_xport_spec_t` structure to be freed, as an argument to `rtsp_free_xport_spec`.

  The `rtsp_free_xport_spec` function does not return a value.

  For more information, type `man 3 rtsp_free_xport_spec` at the HP-UX command prompt.

- **rtsp_free_xport_spec_flds**
  You can use `rtsp_free_xport_spec_flds` to free the dynamically allocated memory for the `rtsp_xport_spec_t` structure members.

  The declaration of `rtsp_free_xport_spec_flds` is as follows:
  
  ```c
  #include <rtsp.h>
  ```
void rtsp_free_xport_spec_flds(
    rtsp_xport_spec_t *xport_spec
);

You must pass xport_spec, a pointer to the rtsp_xport_spec_t structure, as an argument to rtsp_free_xport_spec_flds.

The rtsp_free_xport_spec_flds function does not return a value.

For more information, type man 3 rtsp_free_xport_spec_flds at the HP-UX command prompt.

RTSP URL APIs

The RTSP URL data structure, rtsp_url_t, stores information about the RTSP URL, such as the protocol being used, host, port, and absolute path.

The definition of the rtsp_url_t data structure is as follows:

typedef struct rtsp_url
{
    uint8_t     family;
    uint8_t     protocol;
    uint16_t    port;
    char        host[RTSP_MAX_HOST_LEN];
    char        abspath[RTSP_MAX_PATH_LEN];
} rtsp_url_t;

The family structure member can take one of the following values:
• RTSP_AF_NONE
• RTSP_AF_INET
• RTSP_AF_INET6

RTSP_AF_INET denotes an IPv4 address in the RTSP URL data structure and RTSP_AF_INET6 denotes an IPv6 address in the RTSP URL data structure. Note that IPv6 addresses in the RTSP URL string must be enclosed within [ and ] brackets.

The protocol structure member can take one of the following values:
• RTSP_PROTO_TCP
• RTSP_PROTO_UDP

RTSP_PROTO_TCP indicates TCP transport and RTSP_PROTO_UDP denotes UDP transport.

The port, host, and abspath structure members store information about the port, host, and absolute path specified in the RTSP URL. The host and the abspath structure members can contain only a maximum of RTSP_MAX_HOST_LEN and RTSP_MAX_PATH_LEN characters, respectively, and are NULL terminated.

RTSP_MAX_HOST_LEN and RTSP_MAX_PATH_LEN are defined in the rtsp.h file.

You can use rtsp_parse_url to parse a string containing the RTSP URL in RFC 2326 format into the rtsp_url_t structure.
You must pass the following arguments to `rtsp_parse_url`:

- **url**
  Pointer to a location containing a string in RFC 2326 RTSP URL format.

- **rtspurl**
  Pointer to a location containing the `rtsp_url_t` structure.

Upon success, `rtsp_parse_url` returns `RTSP_SUCCESS` and the parsed URL is stored in the location pointed to by the `rtspurl` argument. Upon failure, it returns one of the following error values:

- **RTSP_EINVAL**
  - The `url` argument is NULL.
  - The `rtspurl` argument is NULL.
  - The location pointed to by the `url` argument does not contain the string in RFC 2326 format.

- **RTSP_EPROTONOSUPPORT**
  UDP transport was specified and is currently not supported.

For more information, type `man 3 rtsp_parse_url` at the HP-UX command prompt.

**RTSP Connection Option APIs**

Options are attributes of an RTSP connection, such as socket identifier and mapped session, that can be set or retrieved. Certain options can only be set and other options can only be retrieved.

The following lists the RTSP connection APIs:

- **rtsp_set_conn_opt**
  You can use `rtsp_set_conn_opt` to set options associated with an RTSP connection.

  The declaration of `rtsp_set_conn_opt` is as follows:

  ```c
  #include <rtsp.h>

  rtsp_error_t rtsp_set_conn_opt(
      rtsp_conn_t *conn,
      rtsp_option_type_t opt_type,
      void *opt_value,
      int opt_size
  );
  ```
You must pass the following arguments to `rtsp_set_conn_opt`:

- **conn**
  RTSP connection for which the option is being set.

- **opt_type**
  Parameter that identifies the option being set.

- **opt_value**
  Pointer to a location containing the value of the option.

- **opt_size**
  Size of the data structure in the location pointed to by `opt_value`.

Upon success, `rtsp_set_conn_opt` returns `RTSP_SUCCESS`. Upon failure, it returns `RTSP_EINVAL` in any of the following cases:

- **RTSP_EINVAL**
  - The `conn` argument is NULL.
  - The option identified by `opt_type` is invalid.
  - The `opt_value` argument is required, but is specified as NULL.
  - The `opt_size` argument is required, and the value specified in the `opt_size` argument is invalid.

For more information, type `man 3 rtsp_set_conn_opt` at the HP-UX command prompt.

• **rtsp_get_conn_opt**

You can use `rtsp_get_conn_opt` to retrieve the options associated with an RTSP connection.

The declaration of `rtsp_get_conn_opt` is as follows:

```c
#include <rtsp.h>

rtsp_error_t rtsp_get_conn_opt(
    rtsp_conn_t *conn,
    rtsp_option_type_t opt_type,
    void *opt_value,
    int *opt_size
);
```

You must pass the following arguments to `rtsp_get_conn_opt`:

- **conn**
  RTSP connection for which the option is being retrieved.

- **opt_type**
  Parameter that identifies the option being retrieved.
— *opt_value*
  Pointer to a location where the value of the option is returned.

— *opt_size*
  Pointer to a location that contains the size, in bytes, of the data structure present
  in the location pointed to by *opt_value*.

You can retrieve the following options associated with an RTSP connection:

— **RTSP_OPT_SOCKET**
  *(int *)* retrieves the socket descriptor for the RTSP connection. The *opt_size*
  argument is ignored and is not updated on return.

— **RTSP_AF_INET**
  *(rtsp_session_t *)* retrieves the associated session for the RTSP connection.
  *opt_size* is ignored and is not updated on return.

Upon success, `rtsp_get_conn_opt` places the option value in the location
pointed to by *opt_value*, and places the size of the option value in the location
pointed to by *opt_size* (unless this argument is ignored). Upon failure,
`rtsp_get_conn_opt` returns RTSP_EINVAL one of the following error values:

— **RTSP_EINVAL**
  The *conn* argument is NULL.
  ◦ The *conn* argument is NULL.
  ◦ The value specified by *opt_type* is invalid.
  ◦ The *opt_value* argument is NULL.
  ◦ The *opt_size* argument is required and the value specified in the *opt_size*
    argument is invalid.

For more information, type `man 3 rtsp_get_conn_opt` at the HP-UX command
prompt.

**RTSP Call Order Sequence**

The RTSP call order sequence is as follows:

1. An application initially creates an RTSP connection structure, `rtsp_conn_t`,
   using either the `rtsp_open` or the `rtsp_create_conn` API. It requires the RTSP
   connection structure to send and receive RTSP messages. The application may
   optionally create and associate sessions using the RTSP session interfaces. For
   more information, type `man 5 rtsp` at the HP-UX command prompt.

2. The application creates a new RTSP message, represented by the `rtsp_msg_t`
   structure, using either the `rtsp_init_request_msg` or the
   `rtsp_init_response_msg` API.
3. Use the `rtsp_set_msg_hdr` API to set the RTSP header information in the message.
4. The application sends an RTSP message, identified by the `rtsp_msg_t` structure, using the `rtsp_send_msg` API.
5. Receive the RTSP messages using `rtsp_recv`. To identify the message type (request or response message), use the `rtsp_is_request_msg` and the `rtsp_is_response_msg` APIs.
6. Retrieve the request or response line information of a message using `rtsp_get_msg_request_line` and `rtsp_get_msg_response_line`.
7. Retrieve the header information in the RTSP message using `rtsp_get_msg_hdr`.
8. Free the resources allocated for the RTSP message using `rtsp_free_msg`.

The following lists the basic rules for using `rtsp_*_t` data structures in an application:

- For statically declared structures, call initialization routines. For pointer variables, call functions that allocate memory and initialize the structures. You may have to use a single function that performs both of these operations.
- Use the data structures.
- Deallocate resources for structure members, and deallocate resources for the structure. You may have to use a single function that performs both of these operations.

The following illustrates this sequence when the `rtsp_msg_t`, `rtsp_conn_t` and the `rtsp_xport_spec_t` data structures are used:

- For `rtsp_msg_t`, call `rtsp_init_request_msg`, `rtsp_init_response_msg`, or `rtcp_recv` before calling APIs that require arguments of type `rtsp_msg_t*`.

In other words, you must obtain the `rtsp_msg_t*` data structure from `rtsp_init_request_msg`, `rtsp_init_response_msg`, or `rtsp_recv`. After calling `rtsp_free_msg`, you must not reference the memory location passed to `rtsp_free_msg`.

The call sequence with respect to the `rtsp_msg_t` structure is as follows:

1. `rtsp_init_request_msg(..., &msg)` or `rtsp_init_response_msg(..., &msg)` or `rtsp_recv(..., &msg)`
2. APIs that require `rtsp_msg_t*` arguments.
3. `rtsp_free_msg(msg);`
NOTE: After calling `rtsp_free_msg`, do not use `msg` again without a call to (1) in the previous list.

- For `rtsp_conn_t`, call either `rtsp_open` or `rtsp_create_conn` before calling APIs that require arguments of type `rtsp_conn_t *`. In other words, you must obtain the `rtsp_conn_t*` data structure only from `rtsp_open` or `rtsp_create_conn`. After calling `rtsp_close`, you must not reference the memory location passed to the `rtsp_close`.

The call sequence with respect to `rtsp_conn_t` structure is as follows:
1. `rtsp_open (..., &conn)` or `rtsp_create_conn (..., &conn)`.
2. APIs that require `rtsp_conn_t*` arguments
3. `rtsp_close_conn(conn);`

NOTE: After calling `rtsp_close_conn`, do not use `conn` again without a call to (1) in the previous list.

- You must initialize `rtsp_*_t` data structures before using them. You may use certain APIs to initialize statically declared structures and some others to dynamically allocate memory and initialize the structures. You must call either of these APIs before passing these structures to other APIs.

The call sequence with respect to the `rtsp_xport_spec_t*` structure is as follows:
1. `xspec = rtsp_alloc_xport_spec(NULL)` or `rtsp_init_xport_spec(&xspec);`
2. Set information in the `xspec` structure, for example, as follows:
   `xspec->cport.rtp = 7000; xspec->cport.rtcp = 7001;`
3. Use `xspec` in RTSP APIs:
   `rtsp_set_msg_hdr(msg, RTSP_TRANSPORT_HDR, xspec, 0);`
4. Free the resources:
   `rtsp_free_xport_spec(xspec)`
NOTE: If you use `rtsp_*_t` structures as OUT arguments to APIs, or the API allocates memory and initializes the structures, you do not need to call the initialization functions for the structures. You may not have to do the first step in the previous sequence.
The following lists the types of descriptions included in a real-time multimedia Session Description Protocol (SDP):
• Session description
The session description includes the elements listed in Table 2-4.

Table 2-4 SDP Session Description

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td>Protocol version.</td>
</tr>
<tr>
<td>o</td>
<td>Session owner.</td>
</tr>
<tr>
<td>s</td>
<td>Session name.</td>
</tr>
<tr>
<td>i</td>
<td>Session information.</td>
</tr>
<tr>
<td>e</td>
<td>E-mail address.</td>
</tr>
<tr>
<td>c</td>
<td>Connection information.</td>
</tr>
<tr>
<td>b</td>
<td>Bandwidth information.</td>
</tr>
<tr>
<td>u</td>
<td>URI of description.</td>
</tr>
<tr>
<td>p</td>
<td>Phone number.</td>
</tr>
<tr>
<td>z</td>
<td>Time zone adjustments.</td>
</tr>
<tr>
<td>a</td>
<td>Session attributes.</td>
</tr>
<tr>
<td>k</td>
<td>Encryption key.</td>
</tr>
</tbody>
</table>

• Time description
The time description includes \( t \), the time period during which the session is active, and \( r \) that signifies zero or more repeat times.

• Media description
The media description includes the elements listed in Table 2-5.

Table 2-5 SDP Media Description

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>m</td>
<td>Media name and transport address.</td>
</tr>
<tr>
<td>i</td>
<td>Media title.</td>
</tr>
<tr>
<td>c</td>
<td>Connection information.</td>
</tr>
<tr>
<td>k</td>
<td>Encryption key.</td>
</tr>
<tr>
<td>b</td>
<td>Bandwidth information.</td>
</tr>
<tr>
<td>a</td>
<td>Media attributes.</td>
</tr>
</tbody>
</table>
**SDP APIs**

The APIs included in the SDP library have the following functions:

- “Parsing the Buffer and Filling Up the SDP Structure” (page 69)
- “Generating an SDP Packet from the SDP Structure” (page 69)
- “Obtaining or Setting SDP Structure Members” (page 70)
- “Obtaining or Adding the Structure Members for Media Descriptions” (page 74)
- “Initializing the SDP Structure and Allocating Space” (page 75)
- “Freeing the Space Allocated for the SDP Structure” (page 76)
- “Creating an SDP Packet” (page 76)
- “Parsing an SDP Packet” (page 78)

**Parsing the Buffer and Filling Up the SDP Structure**

The SDP header file, `sdp.h`, contains the declaration of `sdp_parse_buf` as follows:

```
#include <sdp.h>
int sdp_parse_buf(sdp_t *sdp, char *sdp_buf);
```

The `sdp_parse_buf` API parses the input buffer, `sdp_buf`, which contains SDP descriptions and fills up the SDP structure.

For more information, type `man 3 sdp_parse_buf` at the HP-UX command prompt.

**Return Values**

Upon success, `sdp_parse_buf` returns `SDP_SUCCESS`; upon failure, it returns one of the following error values:

- **SDP_EPROT**
  Non-RFC compliant description string encountered.

- **SDP_EEOF**
  EOF/end of buffer reached while parsing.

- **SDP_ENOMEM**
  Memory allocation failure.

- **SDP_EBUFOVFLW**
  Buffer overflow encountered.

- **SDP_ETYPE**
  Type is not expected in `<type>=` in the SDP description.

**Generating an SDP Packet from the SDP Structure**

The SDP header file, `sdp.h`, contains the declaration of `sdp_create_buf` as follows:

```
#include <sdp.h>
int sdp_create_buf(sdp_t *sdp, char **sdp_buf, size_t *bufsize);
```

```
The `sdp_create_buf` API generates an SDP packet from the SDP structure that is created using `sdp_parse_buf`.

For more information, type `man 3 sdp_create_buf` at the HP-UX command prompt.

**Return Values**

Upon success, `sdp_create_buf` returns `SDP_SUCCESS`; upon failure, it returns one of the following error values:

- **SDP_EPROT**
  Non-RFC compliant description string encountered.

- **SDP_EEOF**
  EOF/end of buffer reached while parsing.

- **SDP_ENOMEM**
  Memory allocation failure.

- **SDP_EBUFOVFLW**
  Buffer overflow encountered.

- **SDP_ETYPE**
  Type is not expected in `<type>=` in the SDP description.

**Obtaining or Setting SDP Structure Members**

The SDP header file, `sdp.h`, contains the declarations of `sdp_get*` and `sdp_add*` APIs as follows:

```c
#include <sdp.h>
void *sdp_get(sdp_t *sdp, sdp_media_list_t *media, int field_type);
char *sdp_get_version(sdp_t *sdp);
sdp_owner_t *sdp_get_owner(sdp_t *sdp);
char *sdp_get_name(sdp_t *sdp);
char *sdp_get_info(sdp_t *sdp);
char *sdp_get_uri(sdp_t *sdp);
sdp_email_list_t *sdp_get_email(sdp_t *sdp);
sdp_phone_list_t *sdp_get_phone(sdp_t *sdp);
int sdp_add_version(sdp_t *sdp, char *vers);
int sdp_add_owner(sdp_t *sdp, char *username, char *sesid,
                 char *ver, char *nettype, char *addrtype, char *addr);
int sdp_add_name(sdp_t *sdp, char *sname);
int sdp_add_info(sdp_t *sdp, char *sinfo);
int sdp_add_uri(sdp_t *sdp, char *uri);
int sdp_add_email(sdp_t *sdp, char *email);
int sdp_add_phone(sdp_t *sdp, char *phone);
sdp_connection_list_t *sdp_get_connection(sdp_t *sdp);
sdp_bandwidth_t *sdp_get_bandwidth(sdp_t *sdp);
sdp_time_list_t *sdp_get_time(sdp_t *sdp);
sdp_repeat_time_t *sdp_get_repeat(sdp_t *sdp);
```
sdp_adj_time_list_t *sdp_get_tzoneadjust(sdp_t *sdp);
sdp_encr_key_t *sdp_get_encrkey(sdp_t *sdp);
sdp_attribute_list_t *sdp_get_attribute(sdp_t *sdp);
int sdp_add_connection(sdp_t *sdp, char *nettype, char *addrtype,
char *addr, char *ttl, char *no_of_addr);
int sdp_add_bandwidth(sdp_t *sdp, char *bwtype, char *bandwidth);
int sdp_add_attribute(sdp_t *sdp, char *att_name, char *att_value);
int sdp_add_time(sdp_t *sdp, char *start, char *stop, char *interval,
char *act_dur, char *offset);
int sdp_add_tzoneadjust(sdp_t *sdp, char *adjtime, char *offset);
int sdp_add_encrkey(sdp_t *sdp, char *key_method, char *key_data);

“The sdp_get and sdp_add APIs” (page 71) lists the sdp_get* and sdp_add* APIs.

Table 2-6 The sdp_get and sdp_add APIs

<table>
<thead>
<tr>
<th>API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sdp_get_version()</td>
<td>Gets the SDP protocol version.</td>
</tr>
<tr>
<td>sdp_get_owner()</td>
<td>Gets information about the SDP session owner or the creator.</td>
</tr>
<tr>
<td>sdp_get_name()</td>
<td>Gets the SDP session name.</td>
</tr>
<tr>
<td>sdp_get_info()</td>
<td>Gets the SDP session information.</td>
</tr>
<tr>
<td>sdp_get_uri()</td>
<td>Gets Universal Resource Identifier (URI) information, if any.</td>
</tr>
<tr>
<td>sdp_get_email()</td>
<td>Gets the e-mail address or addresses of the person responsible for this conference. It returns the starting address of a list of e-mail addresses.</td>
</tr>
<tr>
<td>sdp_get_phone()</td>
<td>Gets the telephone number or numbers of the person responsible for this conference. It returns the starting address of a list of telephone numbers.</td>
</tr>
<tr>
<td>sdp_add_version()</td>
<td>Adds SDP protocol version to the SDP structure. vers points to the protocol version to be set.</td>
</tr>
<tr>
<td>sdp_add_owner()</td>
<td>Adds the owner or creator information in the SDP structure. The arguments username, sesid, ver, nettype, addrtype, and addr point to the user’s login name on the originating host, session ID, version number of the announcement, network type (IN), address type (IP4 or IP6), and address of the machine from which the session was created, respectively.</td>
</tr>
<tr>
<td>sdp_add_name()</td>
<td>Adds the session name. The sname argument points to the session name to be set.</td>
</tr>
<tr>
<td>sdp_add_info()</td>
<td>Adds the SDP session information. info points to the session information to be set.</td>
</tr>
<tr>
<td>API</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>sdp_add_uri()</code></td>
<td>Adds Universal Resource Identifier (URI) information to the SDP structure. Usually, a URI is a pointer to additional information about the conference. The <code>uri</code> argument points to this information.</td>
</tr>
<tr>
<td><code>sdp_add_email()</code></td>
<td>Adds the e-mail address of the contact person responsible for the conference. The <code>email</code> argument points to the e-mail address to be set.</td>
</tr>
<tr>
<td><code>sdp_add_phone()</code></td>
<td>Adds the phone number of the person responsible for the conference. The <code>phone</code> argument points to the telephone number.</td>
</tr>
<tr>
<td><code>sdp_get_connection()</code></td>
<td>Gets the connection information for the session.</td>
</tr>
<tr>
<td><code>sdp_get_bandwidth()</code></td>
<td>Gets the proposed bandwidth information.</td>
</tr>
<tr>
<td><code>sdp_get_time()</code></td>
<td>Gets the start and stop time information.</td>
</tr>
<tr>
<td><code>sdp_get_repeat()</code></td>
<td>Gets the repeat times for a session.</td>
</tr>
<tr>
<td><code>sdp_get_tzoneadjust()</code></td>
<td>Gets the time zone adjustment information.</td>
</tr>
<tr>
<td><code>sdp_get_encrkey()</code></td>
<td>Gets the encryption key information.</td>
</tr>
<tr>
<td><code>sdp_get_attribute()</code></td>
<td>Gets any attribute-related information. This usually specifies any additional information about the session.</td>
</tr>
<tr>
<td><code>sdp_add_connection()</code></td>
<td>Adds connection information. The arguments <code>nettype</code>, <code>addrtype</code>, <code>addr</code>, <code>ttl</code>, and <code>no_of_addr</code> point to network type (IN), address type (IP4 or IP6), base multicast address, time to live (TTL), and the number of addresses, respectively.</td>
</tr>
<tr>
<td><code>sdp_add_bandwidth()</code></td>
<td>Adds bandwidth information that specifies the proposed bandwidth to be used by the session. The <code>bwtype</code> and <code>bandwidth</code> arguments point to the modifier (bandwidth type) and bandwidth value, respectively.</td>
</tr>
<tr>
<td><code>sdp_add_attribute()</code></td>
<td>Adds general attributes, if any. The <code>att_name</code> and <code>att_value</code> arguments point to the attribute name and the attribute value, respectively.</td>
</tr>
<tr>
<td><code>sdp_add_time</code></td>
<td>Adds the start and stop times for the conference. The <code>start</code>, <code>stop</code>, <code>interval</code>, <code>act_dur</code>, and <code>offset</code> arguments point to the start time, stop time, repeat interval, active duration, and list of offsets from start time, respectively.</td>
</tr>
<tr>
<td>API</td>
<td>Description</td>
</tr>
<tr>
<td>-----</td>
<td>-------------</td>
</tr>
<tr>
<td>sdp_add_tzoneadjust()</td>
<td>Adds time zone adjustment information. The <em>adjtime</em> and <em>offset</em> arguments point to the adjust time and the offset, respectively.</td>
</tr>
<tr>
<td>sdp_add_encrkey()</td>
<td>Adds encryption key information. The <em>key_method</em> and <em>key_data</em> arguments point to the encryption method (mechanism used for obtaining a usable key) and the encryption key, respectively.</td>
</tr>
</tbody>
</table>

The *sdp_get* API gets an SDP structure member based on the *field_type* argument that it accepts. You can use *sdp_get* to obtain field information either from the session-level description or from the media-level description, based on the pointer values passed to it (*sdp* and *media*). If *sdp* is NULL, the API returns the field information from the media-level description. Some SDP structure members, such as bandwidth information, can be part of both session-level and media-level descriptions.

For information on *field_type*, type `man 3 sdp_get` at the HP-UX command prompt.

For APIs that return a pointer to the first item of the list of information, you can obtain the other items by following the link till the end of the list, indicated by a NULL, is reached. For example, you can retrieve the e-mail addresses in a session description using *sdp_get_email* as follows:

```c
sdp_email_list_t *smail;

    smail = (sdp_email_list_t *) sdp_get_email(sdp);
    while (smail->next != NULL) {
        ... /* process the email field smail->e_email */
        smail = smail->next;
    }
```

For more information on *sdp_get* routines, type `man 3 sdp_get_connection` at the HP-UX command prompt.

**Return Values**

Upon success, *sdp_get* routines return a pointer to the particular SDP structure member requested; otherwise, they return NULL.

Upon success, *sdp_add* routines return *SDP_SUCCESS*, indicating the addition of a particular SDP structure member. Otherwise, they return one of the following values:

- **SDP_ENOMEM**
  Memory allocation failure.

- **SDP_EPROT**
  Non-RFC compliant description string encountered.
Obtaining or Adding the Structure Members for Media Descriptions

The SDP header file, sdp.h, contains the declarations of sdp_get_media* and sdp_add_media* APIs as follows:

```c
#include <sdp.h>
sdp_media_info_t *sdp_get_media_info(sdp_media_list_t *med_des);
char *sdp_get_media_title(sdp_media_list_t *med_des);
sdp_connection_list_t *sdp_get_media_connection(sdp_media_list_t *med_des);
sdp_bandwidth_t *sdp_get_media_bandwidth(sdp_media_list_t *med_des);
sdp_encr_key_t *sdp_get_media_encrkey(sdp_media_list_t *med_des);
sdp_attribute_list_t *sdp_get_media_attribute(sdp_media_list_t *med_des);
int sdp_add_media_info(sdp_t *sdp, char *media, char *port,
char *number_of_port, char *transport, char *fmtlist);
int sdp_add_media_title(sdp_t *sdp, char *i_info);
int sdp_add_media_connection(sdp_t *sdp, char *nettype,
char *addrtype, char *addr, char *ttl, char *no_of_addr);
int sdp_add_media_bandwidth(sdp_t *sdp, char *bwtype, char *bandwidth);
int sdp_add_media_encrkey(sdp_t *sdp, char *key_method, char *key_data);
int sdp_add_media_attribute(sdp_t *sdp, char *att_name, char *att_value);
```

Table 2-7 lists the sdp_get_media* and sdp_add_media* APIs.

### Table 2-7 The sdp_get_media* and sdp_add_media* APIs

<table>
<thead>
<tr>
<th>API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sdp_get_media_info()</td>
<td>Gets complete media description information for a session. A session may have more than one media description. This returns a pointer to the list of media descriptions.</td>
</tr>
<tr>
<td>sdp_get_media_title()</td>
<td>Gets the media title.</td>
</tr>
<tr>
<td>sdp_get_media_connection()</td>
<td>Gets the media connection information in the media description.</td>
</tr>
<tr>
<td>sdp_get_media_bandwidth()</td>
<td>Gets the proposed bandwidth information in the media description.</td>
</tr>
<tr>
<td>sdp_get_media_encrkey()</td>
<td>Gets encryption key information in the media description.</td>
</tr>
<tr>
<td>sdp_get_media_attribute()</td>
<td>Gets attribute information in the media description, if any.</td>
</tr>
<tr>
<td>sdp_add_media_info()</td>
<td>Adds media information. The media, port, number_of_port, transport, and fmtlist arguments point to the media type (such as audio, video, application, data, and control), the transport port to which media stream will be sent, the number of ports, the transport protocol, and the format, respectively.</td>
</tr>
</tbody>
</table>

74 Using MSP APIs
### Table 2-7 The sdplib_get_media* and sdplib_add_media* APIs (continued)

<table>
<thead>
<tr>
<th>API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sdplib_add_media_title()</td>
<td>Adds media title information. The i_info argument points to the media title.</td>
</tr>
<tr>
<td>sdplib_add_media_connection()</td>
<td>Adds connection information for media. The nettype, addrtype, addr, ttl, and no_of_addr arguments point to network type (IN), address type (IP4 or IP6), base multicast address, time to live (TTL), and the number of addresses, respectively.</td>
</tr>
<tr>
<td>sdplib_add_media_bandwidth()</td>
<td>Adds the proposed bandwidth information for a media. The bwtype and bandwidth arguments point to the modifier (bandwidth type) and bandwidth value, respectively.</td>
</tr>
<tr>
<td>sdplib_add_media_encrkey()</td>
<td>Adds encryption key information for a media. The key_method and key_data arguments point to the encryption method (mechanism used for obtaining a usable key) and the encryption key, respectively.</td>
</tr>
<tr>
<td>sdplib_add_media_attribute()</td>
<td>Adds any additional attribute information for a particular media. The att_name and att_value arguments point to the attribute name and the attribute value, respectively.</td>
</tr>
</tbody>
</table>

### Return Values

Upon success, sdplib_get_media* routines return a pointer to the particular SDP structure member requested from a media description; otherwise, they return NULL.

Upon success, sdplib_add_media* routines return SDP_SUCCESS, indicating the addition of a particular SDP structure member. Otherwise, they return one of the following values:

- **SDP_ENOMEM**
  Memory allocation failure.
- **SDP_EPROT**
  Non-RFC compliant description string encountered.

### Initializing the SDP Structure and Allocating Space

The SDP header file, *sdplib.h*, contains the declaration of *sdplib_init* as follows:

```c
#include <sdplib.h>
sdplib_t *sdplib_init(void);
```

You can use *sdplib_init* to initialize the session description protocol (SDP) structure. The *sdplib.h* file contains the definition of an SDP structure as follows:
typedef struct {
    char *v_version;       /* v= protocol version */
    sdp_owner_t *owner_info; /* o= owner/creator info */
    char *s_name;          /* s= session name */
    char *i_info;          /* i= session information */
    char *u_uri;           /* u= uri */
    sdp_email_list_t *email; /* e= email address */
    sdp_phone_list_t *phone; /* p= phone number */
    sdp_connection_list_t *conn_info; /* c= connection info */
    sdp_bandwidth_t *bandw_info; /* b= bandwidth */
    sdp_time_list_t *ses_time; /* t= time description */
    sdp_adj_time_list_t *adj_time; /* z= time zone info */
    sdp_encr_key_t *key;     /* k= encryption key */
    sdp_attribute_list_t *attributes; /* a= attributes */
    sdp_media_list_t *media_des; /* m= media description */
} sdp_t

The `sdp_init` API allocates memory for the SDP structure and initializes the structure members. The `<sdp.h>` file contains the definitions of the following member types:

`sdp_owner_t, sdp_email_list_t, sdp_phone_list_t,
sdp_connection_list_t, sdp_bandwidth_t, sdp_time_list_t,
sdp_adj_time_list_t, sdp_encr_key_t, sdp_attribute_list_t, and
sdp_media_list_t.```

Return Values

Upon success, `sdp_init` returns a pointer to an appropriate SDP structure; upon failure, it returns NULL.

Freeing the Space Allocated for the SDP Structure

The SDP header file, `sdp.h`, contains the declaration of `sdp_free` as follows:

```c
#include <sdp.h>
void sdp_free(sdp_t *sdp)
```

The `sdp_free` API deallocates the space pointed to by `sdp`, which is a pointer to a block previously allocated by `sdp_init`. If `sdp` is a NULL pointer, no action occurs. The `sdp_free` API does not return a specific value.

SDP Call Order Sequence

This section outlines the SDP call order sequence.

It includes the following topics:

- “Creating an SDP Packet” (page 76)
- “Parsing an SDP Packet” (page 78)

Creating an SDP Packet

To create an SDP packet, use the SDP APIs in the following sequence:
1. Initiate sdp_t, the SDP structure.
   sdpt init

2. Fill up sdp_t using sdp_add* APIs.
   or
   Fill up sdp_t by parsing an input buffer containing SDP descriptions.
   sdpt parse_buf

3. Create buffer.
   sdpt create_buf

4. Free the buffer.
   sdpt_free

Figure 2-3 illustrates the call order sequence of APIs that create an SDP packet.
Figure 2-3 Creating an SDP Packet

Figure 2-4 illustrates the call order sequence of APIs that parse an SDP packet.

To parse an SDP packet, use the SDP APIs in the following sequence:

1. `sdp_init`
2. `sdp_parse_buf`
3. `sdp_get*`
4. `sdp_free`
Figure 2-4 Parsing an SDP Packet

- `sdp_init()`: Initialize `sdp_t`, the sdp structure
- `sdp_parse_buf()`: Parse an input buffer containing SDP descriptions and Fill up `sdp_t`
- `sdp_get*()`: Read the required elements of `sdp_t`
- Process/use the SDP packet
- `sdp_free()`: Free the sdp structure
This chapter describes how to troubleshoot the APIs that the Multimedia Streaming Protocols (MSP) Software Developer’s Kit (SDK) includes.

It discusses the following topics:
- "Troubleshooting RTP" (page 81)
- "Troubleshooting RTSP" (page 82)
- "Troubleshooting SDP" (page 83)

**Troubleshooting RTP**

Table 3-1 describes how to troubleshoot Real-Time Protocol (RTP) suite of APIs.

**Table 3-1 RTP Troubleshooting Tips**

<table>
<thead>
<tr>
<th>API</th>
<th>Scenario</th>
<th>Tips</th>
</tr>
</thead>
<tbody>
<tr>
<td>rtp_open</td>
<td>The library dumps core when rtp_open is called, or rtp_open returns negative error number, such as RTP_EINVAL.</td>
<td>The arguments passed to the call must be pointers to sockaddr_in or sockaddr_in6 socket address structures, with either AF_INET or AF_INET6 as family. The rtp_open API fails, returning a negative error value, when the socket address structures passed belong to different families.</td>
</tr>
<tr>
<td>rtp_close</td>
<td>The rtp_close API fails with RTP_EINVAL.</td>
<td>When this API receives an incorrect argument, or an already closed RTP descriptor, rtp_close returns RTP_EINVAL.</td>
</tr>
<tr>
<td>API</td>
<td>Scenario</td>
<td>Tips</td>
</tr>
<tr>
<td>-----------------</td>
<td>--------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>rtp_send</code></td>
<td>The <code>rtp_send</code> API does not send packets to the other side. The descriptor is valid and correct. It also does not return errors.</td>
<td>At least one successful call to <code>rtp_ioctl</code> for <code>RTP_SSENDTOADDR</code> should be made on a valid RTP descriptor before <code>rtp_send</code> can send packets to the other side.</td>
</tr>
<tr>
<td><code>rtp_recv</code> and</td>
<td>Second call to <code>rtp_recv</code> always fails or the user is not able to receive the second RTP packet, but the first is obtained successfully.</td>
<td>Check if the arguments are re-initialized. You may be passing NULL to <code>rtp_recv</code> for the buffer argument so that <code>rtp_recv</code> allocates its own buffer and return you the RTP packet. In such a case, the <code>buflen</code> field is set to the size of the first packet. If the <code>*buf</code> argument is not set to null in the second call, the same buffer is used and the maximum length of the buffer is assumed to be the size specified in <code>buflen</code>: the size of the first RTP packet. In such a case, if a longer RTP packet is received, <code>rtp_recv</code> returns an error.</td>
</tr>
<tr>
<td><code>rtcp_recv</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Troubleshooting RTSP**

Table 3-2 describes how to troubleshoot Real-Time Streaming Protocol (RTSP) suite of APIs.
**Table 3-2 RTSP Troubleshooting Tips**

<table>
<thead>
<tr>
<th>API</th>
<th>Scenario</th>
<th>Tips</th>
</tr>
</thead>
<tbody>
<tr>
<td>rtsp_get_msg_hdr</td>
<td>Using the address of an array when a char ** expected as an argument.</td>
<td>The following is invalid:</td>
</tr>
<tr>
<td></td>
<td>Example:</td>
<td>rtsp_get_msg_hdr(msg, RTSP_USER_AGENT_HDR, &amp;arr, &amp;size);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Instead, use the following:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rtsp_get_msg_hdr(msg, RTSP_USER_AGENT_HDR, ptr, &amp;size);</td>
</tr>
<tr>
<td>All rtsp_free APIs</td>
<td>Calling rtsp_free_* APIs with an argument whose memory is not</td>
<td>Use:</td>
</tr>
<tr>
<td></td>
<td>dynamically allocated.</td>
<td>rtsp_free_xport_spec(pxport)</td>
</tr>
<tr>
<td></td>
<td>Example:</td>
<td>Do not use:</td>
</tr>
<tr>
<td></td>
<td>rtsp_xport_spec_t xportspec;</td>
<td>rtsp_free_xport_specx(&amp;xportspec)</td>
</tr>
<tr>
<td></td>
<td>rtsp_xport_spec_t *pxport;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>....</td>
<td></td>
</tr>
<tr>
<td></td>
<td>pxport = rtsp_alloc_xport_spec(NULL);</td>
<td></td>
</tr>
<tr>
<td>All rtsp_free APIs</td>
<td>Referring pointers to rtsp_* structures after they are freed using the</td>
<td>References to pxport after calling the following is incorrect:</td>
</tr>
<tr>
<td></td>
<td>rtsp_free_* APIs.</td>
<td>rtsp_free_xport_spec(pxport)</td>
</tr>
<tr>
<td>All rtsp_* data</td>
<td>Using statically declared rtsp_* structures without initializing them.</td>
<td>The following is an incorrect use of</td>
</tr>
<tr>
<td>structures</td>
<td></td>
<td>rtsp_xport_spec_t:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rtsp_xport_spec_t xport; rtsp_msg_t *msg; // Initialize msg xport.sport.rtp = 7000; xport.sport.rtcp = 7001;rtsp_set_msg_hdr(msg, RTSP_TRANS_PORT_HDR, &amp;xport, 0);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Use xport; as follows:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rtsp_xport_spec_t xport; rtsp_msg_t *msg; // Initialize msg xport.sport.rtp = 7000; xport.sport.rtcp = 7001;rtsp_set_msg_hdr(msg, RTSP_TRANS_PORT_HDR, &amp;xport, 0);</td>
</tr>
</tbody>
</table>

**Troubleshooting SDP**

This section describes how to troubleshoot Session Description Protocol (SDP) suite of APIs.

**The sdp_create_buf API**

The definition of sdp_create_buf is as follows:

```c
sdp_create_buf(sdp_t *sdp, char **sdp_buf, size_t *bufsize);
```
The `sdp_create_buf` API may dump core the application if a wrong pointer to the API is passed, as in the following cases:

- The `sdp_create_buf` is called before allocating the first argument `sdp` using `sdp_init`, and filling up using `sdp_add*` routines.
- `*sdp_buf == NULL`, `sdp_create_buf` allocates the buffer using `malloc`, which needs to be freed by `free`.
- If `*sdp_buf != NULL`, the caller needs to allocate and free the buffer. In this case, `*bufsize` should contain the maximum size of the buffer.

**The sdp_get API**

The definition of `sdp_get` is as follows:

```
sdp_get*(sdp_t *sdp,..);
```

The `sdp_get` API may dump core the application if `sdp_parse_buf` and `sdp_add*` APIs do not initialize and fill up `sdp`. 
4 Sample Programs

This chapter contains sample programs for RTP, RTSP, and SDP. It includes the following sections:

- "RTP Sample Program" (page 85)
- "RTSP Sample Program" (page 92)
- "SDP Sample Program" (page 104)

RTP Sample Program

This section includes the following sample programs for RTP:

- **sender.c**
  
  When the user requests, the `sender.c` program sends an RTP packet.

- **recvver.c**
  
  When the user requests, the `recvver.c` program polls for an RTP packet and receives the packet.

The usage of these programs is as follows:

```
./sender [-a local-address] [-p local-rtp-port] [-d dest-addr] [-r remote-rtp-port]
./recvver [-a local-address] [-p local-rtp-port] [-d dest-addr] [-r remote-rtp-port]
```

Example:

```
./sender
or
./sender -p 8200 -r 7100
./recvver
or
./recvver -p 7100 -r 8200
```

The `sender.c` Program

The `sender.c` program is as follows:

```
/*
 * HP MSP-RTP Sample application
 * (c) Copyright 2003 Hewlett-Packard Company All Rights Reserved.
 */

#include <netinet/in.h>
#include <rtp.h>

/*
```
* This Application uses the interfaces provided by the
* RTP library to transmit data
*
*/
/* Macros for handling time as double */
#define tv2dbl(tv) ((tv).tv_sec + (tv).tv_usec / 1000000.0)
#define dbl2prnt(x)  ((int)((long)(x))
,((int)(((int)(1000000.0*((double)(x))))%((int)1000000)))

/* function to print Usage for this program */
Usage(char *prg)
{
    fprintf(stderr, "Usage: %s [-a local-address] [-p local-rtp-port]"
        " [-d dest-addr] [-r remote-rtp-port] n",prg);
}

/* The main routine */
int main(int argc, char **argv)
{
    char c;
    /*
    * by default:
    * IPv4 is to be handled unless specified
    * local ports 28000 for RTP and 2801 for RTCP are used
    * loop back addresses are used
    */
    int RTPport=28000,RTCPport;
    int remRTPport=17000,remRTCPport;
    char *caddr="127.0.0.1",*destaddr="127.0.0.1";
    struct sockaddr_in addr[2];
    int rd;
    int ret;
    struct sockaddr_in sin;
    int i=0,x;
    char ec="y";

    while ((c = getopt(argc, argv, "6a:p:d:r:"))!=EOF)
    {
        switch(c)
        {
            case 'a':
                caddr=optarg;
                break;
            case 'p':
                RTPport=atoi(optarg);
                break;
            case 'r':
                remRTPport=atoi(optarg);
                break;
            case 'd':
                destaddr=optarg;
                break;
            default:
            {
                Usage(argv[1]);
                exit(1);
            }
        }
    }

86 Sample Programs
RTCPport = RTPport + 1;
addr[0].sin_family=AF_INET;
inet_pton(AF_INET,caddr,&addr[0].sin_addr.s_addr);
addr[0].sin_port=RTPport;
addr[1].sin_family=AF_INET;
inet_pton(AF_INET,caddr,&addr[1].sin_addr.s_addr);
addr[1].sin_port=RTCPport;
rd = rtp_open((char*)addr, (char*)(addr+1));
if(rd>0)
{
    printf("opened RTP descriptor %d n"
"Receiving RTP packets on port %d n"
"tand RTCP packets on port %d n"
,rd
,addr[0].sin_port
,addr[1].sin_port
);
}
else
{
    fprintf(stderr,"rtp_open() failed n");
exit(1);
}
remRTCPport=remRTPport+1;
sin.sin_family=AF_INET;
sin.sin_port=remRTPport;
sin.sin_addr.s_addr/inet_addr(destaddr);
if(sin.sin_addr.s_addr < 0)
{
    fprintf(stderr,"The destination address %s is not valid n",destaddr);
exit(1);
}
ret = rtp_ioctl(rd,RTP_SSENDTOADDR,&sin);
if(ret<0)
{
    fprintf(stderr,"could not add %s to the RTP SENDTO list n",destaddr);
}
else
{
    printf("Destination %s:%d addded to RTP SENDTO list n",destaddr
,remRTPport
);
}
sin.sin_port=remRTCPport;
ret = rtp_ioctl(rd,RTCP_SSENDTOADDR,&sin);
if(ret<0)
{
    fprintf(stderr,"could not add %s to the RTCP SENDTO list n",destaddr);
}
else
{
    printf("Destination %s:%d addded to RTCP SENDTO list n",destaddr
,remRTCPport
);
/* constructing an RTP packet */ The RTP packet which we construct does not contain actual media data, but contains alphabets. Media streaming applications would replace this with actual media packets. */

rtp_pkt_t pkt;
pkt.v=2;
pkt.p=0;
pkt.x=0;
pkt.cc=0;
pkt.m=1;
pkt.pt=7;
pkt.seq=1;
struct timeval tv;
struct timezone tzp;

gettimeofday(&tv,&tzp);
pkt.ts=tv2dbl(tv);
pkt.csrcp=0;
pkt.xtnhdrp=0;
pkt.datap=(char*)malloc(9);
strcpy(pkt.datap,"abcdefgh",8);
pkt.datalen=8;

i=0;

printf("send an RTP packet? ");
fflush(stdin);
scanf("%c",&ec);

/* The library automatically generates RTCP packets. */

while((ec=='y') (ec=='Y'))
{
    i++;
    if((ret= rtp_send(rd,&pkt))<0) break;
    printf("send another packet? ");
    fflush(stdin);
scanf("%c",&ec);
}

if(ret<0)
    printf("rtp_send failed ");
else
    printf("%d RTP packet(s) sent ",i);

The recver.c Program

The recver.c program is as follows:

/* HP MSP-RTP Sample application */
(c) Copyright 2003 Hewlett-Packard Company All Rights Reserved.
*/
#include <netinet/in.h>
#include <rtp.h>

/*
 * This Application uses the interfaces provided by the
 * RTP library to transmit data
 *
 */

/* Macros for handling time as double */
#define tv2dbl(tv) ((tv).tv_sec + (tv).tv_usec / 1000000.0)
#define dbl2prnt(x)  (((int)(((int)(1000000.0*((double)(x))))%((int)1000000))))

/* function to print Usage for this program */
Usage(char *prg)
{
    fprintf(stderr, "Usage: %s 
" [a local-address] [-p local-rtp-port] 
[ -d dest-addr] [-r remote-rtp-port] n", prg);
}

/* The main routine */
int main(int argc, char **argv)
{
    char c;
    /*
     * by default:
     * IPv4 is to be handled unless specified
     * local ports 17000 for RTP and 17001 for RTCP are used
     * loop back addresses are used
     */
    int RTPport=17000, RTCPPport;
    int remRTPport=28000, remRTCPPport;
    char *caddr="127.0.0.1", *destaddr="127.0.0.1";
    struct sockaddr_in addr[2];
    int rd;
    int ret;
    struct sockaddr_in sin;
    int i=0, x;
    char ec='y';
    struct rtp_pollrd rfds[1];
    char buf[1024], *ptr = buf;

    while ((c = getopt(argc, argv, "6a:p:d:r:")) != EOF)
    {
        switch(c)
        {
            case 'a':
                caddr = optarg;
                break;
            case 'p':
                RTPport = atoi(optarg);
                break;
            case 'r':
                remRTPport = atoi(optarg);
                break;
            case 'd':
                destaddr = optarg;
                break;
            }
default:
{
    Usage(argv[1]);
    exit(1);
}

RTCPport = RTPport + 1;

addr[0].sin_family=AF_INET;
inet_pton(AF_INET,caddr,&addr[0].sin_addr.s_addr);
    addr[0].sin_port=RTPport;

addr[1].sin_family=AF_INET;
inet_pton(AF_INET,caddr,&addr[1].sin_addr.s_addr);
    addr[1].sin_port=RTCPport;

rd = rtp_open((char*)addr, (char*)(addr+1));
    if(rd>0)
    {
        printf("tOpened RTP descriptor %d n"
               " tReceiving RTP packets on port %d n"
               " tand RTCP packets on port %d n"
               ,rd
               ,addr[0].sin_port
               ,addr[1].sin_port
               );
    }
    else
    {
        fprintf(stderr,"rtp_open() failed n");
        exit(1);
    }

remRTCPport=remRTPport+1;

sin.sin_family=AF_INET;
    sin.sin_port=remRTPport;
    sin.sin_addr.s_addr=inet_addr(destaddr);

    if(sin.sin_addr.s_addr < 0)
    {
        fprintf(stderr,"The destination address %s is not valid n",destaddr);
        exit(1);
    }

ret = rtp_ioctl(rd,RTP_SSENDTOADDR,&sin);
    if(ret<0)
    {
        fprintf(stderr,"could not add %s to the RTP SENDTO list n",destaddr);
    }
    else
    {
        printf("tDestination %s:%d added to RTP SENDTO list n",destaddr
               ,remRTCPport
               );
    }

    sin.sin_port=remRTCPport;
    ret = rtp_ioctl(rd,RTCP_SSENDTOADDR,&sin);
    if(ret<0)
    {
        fprintf(stderr,"could not add %s to the RTCP SENDTO list n",destaddr);
else

    printf(" tDestination %s:%d added to RTCP SENDTO list n", destaddr, remRTCPport);

/*
 * The library automatically generates RTCP packets.
 */

/* poll for an RTP packet */

printf("polling for an RTP packet... ");

rfds[0].rd=rd;
rfds[0].events=RTP_PIN;

if((ret=rtp_poll(rfds,1,-1))<0)
{
    fprintf(stderr," rtp_poll() failed ");
    exit(1);
}

printf("An RTP packet is ready to be received  
    "Receive it now?");
fflush(stdin);
scanf(“%c”,&ec);
i=0;

int buflen;
while((ec!=’n’)&&(ec!=’N’))
{
    i++;
    buflen=sizeof buf;

    /*
     * receiving an RTP packet
     * if ptr points to NULL instead of buf the
     * library would automatically allocate the
     * required memory for the RTP packet.
     */

    if((ret= rtp_recv(rd,(rtp_pkt_t *)NULL,&ptr,&buflen))<0) break;
    printf("receive another packet? ");
    fflush(stdin);
    scanf(“%c”,&ec);
}

if(ret<0)
{
    /*
     * if memory were requested from the library
     * a check for RTP_EMOREDATA would have been
     * redundant
     */

    if(ret==RTP_EMOREDATA)
        fprintf(stderr,"The buffer passed to rtp_recv() was small n");
    else
        fprintf(stderr,"rtp_recv() failed n");
}
else
    printf(“%d RTP packet(s) received n”,i);
}
Make File

The make file for RTP is as follows:

```make
all : sender recver
sender: sender.c
    cc -O sender.c -o sender -l rtp
recver: recver.c
    cc -O recver.c -o recver -l rtp
```

Compiling the Sample RTP Program

This section describes how to compile the sample RTP program. Before compiling the program, ensure that the `rtp.h` header file and the `librtp` library that come with HP-UX MSP are available on the machine. The `rtp.h` header file is assumed to be in the `/usr/include` directory and the `librtp` library in the `/usr/lib` directory. To compile, issue the following commands:

```
$ cc recver.c -o recver -lrtp
$ cc sender.c -o sender -lrtp
```
Alternatively, you can use the make file to compile the RTP programs as follows:

```
$ make all
or
$ make sender
$ make recver
```

RTSP Sample Program

You can use the `sampleclient.c` program, a minimal RTSP client, to send RTSP messages to a server. The program is interactive and allows you to choose the type of RTSP message to be sent to a server. Based on the RTSP message type, you may be prompted to enter additional information.

The following is a sample program for the RTSP module of the MSP suite of libraries:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <rtsp.h>

/*
connect2server :
* Parse the url string and create a RTSP
* Connection to the server identified by the url.
*/

rtsp_conn_t *
connect2server(char *mediaurl)
{ 
```
rtsp_error_t err;
rtsp_conn_t *rtspconn = NULL;
rtsp_url_t rtspurl;

/* Parse the url string and fill up rtspurl */
err = rtsp_parse_url(mediaurl, &rtspurl);
if( err!=RTSP_SUCCESS )
{
    printf("Error parsing media url %s 
", mediaurl);
    return rtspconn;
}
printf("Connecting to server ... ");

/* Create a RTSP Connection */
err = rtsp_open(&rtspurl, RTSP_CFLAG_CS, &rtspconn);
if( err!=RTSP_SUCCESS )
{
    printf("Error 
");
    return rtspconn;
}
printf("Success 
");
return rtspconn;

void
display_msg_menu()
{
    printf("Available RTSP Messages : 
"
        "1. DESCRIBE 
2. SETUP 
3. PLAY 
4. PAUSE 
5. TEARDOWN 
6. EXIT 
"Choose one [1-6]: ");
}

/*
* The basic flow of operations for the process_* routines
* is as follows :
* Create and Initialize the request message.
* Set the header information in the request message.
* Send the request message to the server
* Free the request message
* Receive the response message from the server.
* Retrieve header information from the response message.
* Retrieve message body from the response message.
* Free the response message
*
* In all the process_* routines, the following variables
* are used.
* req_msg => The RTSP Request Message
* rsp_msg => The RTSP Response Message
* err => error number returned by RTSP APIs
rtspconn => The RTSP Connection over which messages are sent and received.

process_describe:
Creates and sends a DESCRIBE Request message to the server. Receives and processes the response message.

rtsp_error_t process_describe(rtsp_conn_t *rtspconn, char *mediaurl)
{
    rtsp_msg_t      *reqmsg = NULL;
    rtsp_msg_t      *rspmsg = NULL;
    rtsp_error_t    err;
    char            accepttype[] = "application/sdp";
    uint8_t         *sdpbuf;
    uint16_t        sdpbuflen;

    /* Create a DESCRIBE RTSP Request Message */
    err = rtsp_init_request_msg(RTSP_DESCRIBE, mediaurl, &reqmsg);
    if( err!=RTSP_SUCCESS )
        return err;

    /* Specify the Accept RTSP Header in the Request Message */
    err = rtsp_set_msg_hdr(reqmsg, RTSP_ACCEPT_HDR, accepttype,
                            strlen(accepttype));
    if( err!=RTSP_SUCCESS )
    {
        rtsp_free_msg(reqmsg);
        return err;
    }

    /* Send the Request Message */
    err = rtsp_send_msg(rtspconn, NULL, reqmsg, 0);
    rtsp_free_msg(reqmsg);
    if( err<0 )
        return err;

    /* Receive the response for the request Message. We pass NULL for channel and size argument since we are sure we would be receiving only a RTSP Message and not an interleaved media stream. */
    err = rtsp_recv(rtspconn, &rspmsg, NULL, NULL);
    if( err<0 )
        return err;

    /* Retrieve the Content-Length Header information */

err = rtsp_get_msg_hdr(rspmsg, RTSP_CONTENT_LENGTH_HDR, &sdpbuflen, NULL);

if ( err!=RTSP_SUCCESS ) {
    rtsp_free_msg(rspmsg);
    return err;
}

/* Allocate a buffer to hold the message body
(i.e) SDP Description */
sdpbuf = malloc(sdpbuflen*sizeof(uint8_t));
if( !sdpbuf ) {
    rtsp_free_msg(rspmsg);
    return err;
}

/* Get the body of the Response Message */
err = rtsp_get_msg_body(rspmsg, (uint8_t *)sdpbuf, &sdpbuflen);
if( err!=RTSP_SUCCESS ) {
    free(sdpbuf);
    rtsp_free_msg(rspmsg);
    return err;
}

printf("SDP Description for %s is : n", mediaurl, (int)sdpbuflen, sdpbuf);
free(sdpbuf);

/* Free the resources allocated for the RTSP Message */
rtsp_free_msg(rspmsg);
return RTSP_SUCCESS;
}

rtsp_error_t
process_setup(rtsp_conn_t *rtspconn, char *setupurl, int crtpport, int crtcpport) {
    rtsp_msg_t      *reqmsg = NULL;
    rtsp_msg_t      *rspmsg = NULL;
    rtsp_error_t    err;
    rtsp_xport_spec_t xspec, *serverxspec=NULL;

RTSP Sample Program  95
char *sessionid;
int sessionidlen;

/* Create a SETUP RTSP Request Message */
err = rtsp_init_request_msg(RTSP_SETUP, setupurl, &reqmsg);
if( err!=RTSP_SUCCESS )
    return err;

/* Initialize and fill up the Transport Header */
rtsp_init_xport_spec(&xspec);
xspec.xport_id = (uint8_t *)"RTP/AVP";
xspec.ncast = RTSP_TRANSMIT_UNICAST;
xspec.cport.rtp = crtpport;
xspec.cport.rtcp = crtcppport;

/* Set the Transport RTSP Header in the Request Message */
err = rtsp_set_msg_hdr(reqmsg, RTSP_TRANSPORT_HDR,
                        &xspec, 0);
if( err!=RTSP_SUCCESS )
    {rtsp_free_msg(reqmsg);
     return err;
    }

/* Send the Request Message */
err = rtsp_send_msg(rtspconn, NULL, reqmsg, 0);

/* Free the resources allocated for the request message */
rtsp_free_msg(reqmsg);

/* Check for rtsp_send_msg errors */
if( err<0 )
    return err;

/* Receive the response for the request Message. We pass
 * NULL for channel and size argument since we are sure we
 * would be receiving only a RTSP Message and not an
 * interleaved media stream. */
err = rtsp_recv(rtspconn, &rspmsg, NULL, NULL);
if( err<0 )
    return err;

/* Get Transport Header information from the Response */
err = rtsp_get_msg_hdr(rspmsg, RTSP_TRANSPORT_HDR,
                        &serverxspec, NULL);
if( err!=RTSP_SUCCESS )
    {rtsp_free_msg(rspmsg);
     return err;
    }
printf(" nServer RTP and RTCP ports are %d %d n", (int)serverxspec->sport.rtp, (int)serverxspec->sport.rtcp);

/* Free the resources allocated to the entire linked * list of rtsp_xport_spec_t structures */
rtsp_free_xport_spec(serverxspec);

/* Retrieve Session Header information from message */
err = rtsp_get_msg_hdr(rspmsg, RTSP_SESSION_HDR, &sessionid, &sessionidlen);
if( err!=RTSP_SUCCESS )
{
   rtsp_free_msg(rspmsg);
   return err;
}

printf("Session ID is %.s n", sessionidlen, sessionid);
free(sessionid);

/* Free the resources allocated for the response Message */
rtsp_free_msg(rspmsg);
return RTSP_SUCCESS;
}

/*
 * process_play :
 * Creates and sends a PLAY Request message to the server.
 * Receives and process the response message for PLAY.
 */

rtsp_error_t
process_play(rtsp_conn_t *rtspconn, char *mediaurl)
{

   rtsp_msg_t *reqmsg = NULL;
   rtsp_msg_t *rspmsg = NULL;
   rtsp_error_t err;
   rtsp_rtpinfo_t *rtpinfo = NULL, *prtpinfo;

   /* Create a PLAY RTSP Request Message */
   err = rtsp_init_request_msg(RTSP_PLAY, mediaurl, &reqmsg);
   if( err!=RTSP_SUCCESS )
      return err;

   /* Send the Request Message */
   err = rtsp_send_msg(rtspconn, NULL, reqmsg, 0);

   /* Free the resources allocated for the request message */
   rtsp_free_msg(reqmsg);

   return RTSP_SUCCESS;
}
/* Check for rtsp_send_msg errors */
if( err<0 )
  return err;

/* Receive the response for the request Message. We pass
 * NULL for channel and size argument since we are sure we
 * would be receiving only a RTSP Message and not an
 * interleaved media stream. */
err = rtsp_recv(rtspconn, &rspmsg, NULL, NULL);
if( err<0 )
  return err;

/* Retrieve RTP-Info Header information from the message */
err = rtsp_get_msg_hdr(rspmsg, RTSP_RTP_INFO_HDR,
                       &rtpinfo, NULL);
if( err!=RTSP_SUCCESS )
{
  rtsp_free_msg(rspmsg);
  return err;
}

printf(" nServer RTP Info : n");

/*
 * The RTP-Info header can have multiple rtp-info-spec
 * entries. The multiple rtp-info-spec entries are linked
 * together through the next structure member
 */
for(prtpinfo=rtpinfo; prtpinfo!=NULL; prtpinfo=prtpinfo->next)
{
  printf(" t");

  /* If url member is NULL, then URL was not
   * specified in the rtp-info-spec entry in the header
   */
  if(prtpinfo->url)
    printf("URL=%s, ", prtpinfo->url);

  /* If seq member is -1, then sequence number was not
   * specified in the rtp-info-spec entry in the header
   */
  if(prtpinfo->seq!=-1)
    printf("seq=%d, ", prtpinfo->seq);

  /* If rtptime member is -1, then RTP Timestamp was not
   * specified in the rtp-info-spec entry in the header
   */
  if(prtpinfo->rtptime!=-1)
    printf("rtptime=%lld", prtpinfo->rtptime);
printf(" n");
}
/* Free the resources allocated to the entire linked
 * list of rtsp_rtpinfo_t structures
 */
rtsp_free_rtpinfo(rtpinfo);
/* Free the resources allocated for the Response Message */
rtsp_free_msg(rspmsg);
return RTSP_SUCCESS;
}
/*
* process_pause :
* Creates and sends a PAUSE Request message to the
* server. Receives and process the response message.
*/
rtsp_error_t process_pause(rtsp_conn_t *rtspconn, char *mediaurl)
{
    rtsp_msg_t *reqmsg = NULL;
    rtsp_msg_t *rspmsg = NULL;
    rtsp_error_t err;
    int statuscode;

    /* Create a PAUSE RTSP Request Message */
    err = rtsp_init_request_msg(RTSP_PAUSE, mediaurl, &reqmsg);
    if( err!=RTSP_SUCCESS )
        return err;

    /* Send the Request Message */
    err = rtsp_send_msg(rtspconn, NULL, reqmsg, 0);

    /* Free the resources allocated for the request message */
    rtsp_free_msg(reqmsg);

    /* Check for rtsp_send_msg errors */
    if( err<0 )
        return err;

    /* Receive the response for the request Message. We pass
     * NULL for channel and size argument since we are sure we
     * would be receiving only a RTSP Message and not an
     * interleaved media stream.
     */
    err = rtsp_recv(rtspconn, &rspmsg, NULL, NULL);
    if( err<0 )
        return err;
/* Get the status code of the Response Message */
statuscode = rtsp_get_msg_response_line(rspmsg, NULL,
                                      NULL);
if (statuscode!=200 )
{
    rtsp_free_msg(rspmsg);
    return err;
}

/* Free the resources allocated for the Response Message */
rtsp_free_msg(rspmsg);
printf("nPAUSE Succeeded n");
return RTSP_SUCCESS;
}

rtsp_error_t
process_teardown(rtsp_conn_t *rtspconn, char *mediaurl)
{
    rtsp_msg_t      *reqmsg = NULL;
    rtsp_msg_t      *rspmsg = NULL;
    rtsp_error_t    err;
    int             statuscode;
    rtsp_session_t  *session;

    /* Create a TEARDOWN RTSP Request Message */
    err = rtsp_init_request_msg(RTSP_TEARDOWN, mediaurl,
                                      &reqmsg);
    if( err!=RTSP_SUCCESS )
        return err;

    /* Send the Request Message */
    err = rtsp_send_msg(rtspconn, NULL, reqmsg, 0);

    /* Free the resources allocated for the request message */
    rtsp_free_msg(reqmsg);

    /* Check for rtsp_send_msg errors */
    if( err<0 )
        return err;

    /* Receive the response for the request message. We pass
     * NULL for channel and size argument since we are sure we
     * would be receiving only a RTSP Message and not an
     * interleaved media stream.*/
err = rtsp_recv(rtspconn, &rspmsg, NULL, NULL);
if (err<0)
    return err;

/* Get the status code of the Response Message */
statuscode = rtsp_get_msg_response_line(rspmsg, NULL, NULL);
if (statuscode!=200)
{
    rtsp_free_msg(rspmsg);
    return err;
}

/* Free the resources allocated for the Response Message */
rtsp_free_msg(rspmsg);

printf("nTEARDOWN Succeeded n");
return RTSP_SUCCESS;

main(int argc, char *argv[])
{
    /*
    * mediaurl points to the url passed as a
    * command-line argument to the program
    */
    char *mediaurl;

    /* msgcode identifies the RTSP Method chosen by the user */
    int msgcode;

    /* rtspconn represents the RTSP Connection to the peer */
    rtsp_conn_t *rtspconn=NULL;

    /* mapsession represents the mapped session for the
    * RTSP connection
    */
    rtsp_session_t *mapsession=NULL;

    /* err will hold the error numbers returned by RTSP APIs */
    rtsp_error_t err;

    /*
    * crtpport and crtcpport will hold the client
    * RTP and RTCP ports for SETUP Request
    */
    int crtpport, crtcpport;

    /* setupurl holds the URL to be used in SETUP Request */
    char setupurl[256];
int setupurllen;

/*
 * The mthd variable will hold the enumerated RTSP
 * Method type for the Method chosen by the user
 */
rtsp_method_t mthd;

int close=0;

if( argc!=2 )
{
    printf("Usage : %s <media url> n", argv[0]);
    exit(-1);
}
mediaurl = argv[1];

/* Connect to the server specified by the mediaurl */
rtspconn = connect2server(mediaurl);
if( rtspconn==NULL )
    exit(-1);

do
{
    display_msg_menu();
    scanf("%d", &msgcode);

    switch(msgcode)
    {
    case 1:
        mthd = RTSP_DESCRIBE;
        err = process_describe(rtspconn, mediaurl);
        break;

    case 2:
        mthd = RTSP_SETUP;
        printf("Enter SETUP URL (max 255 char) : ");
        fflush(stdin);
        fgets(setupurl, 256, stdin);
        setupurllen = strlen(setupurl);
        if( setupurl[setupurllen-1]==’n’ )
            setupurl[setupurllen-1]=’0’;
        printf("Enter Client RTP RTCP Ports : ");
        scanf("%d %d", &crtpport, &crtcpport);
        err = process_setup(rtspconn, setupurl,
                           crtpport, crtcpport);
        break;

    case 3:
        mthd = RTSP_PLAY;
        err = process_play(rtspconn, mediaurl);
break;

case 4:
    mthd = RTSP_PAUSE;
    err = process_pause(rtspconn, mediaurl);
    break;

case 5:
    mthd = RTSP_TEARDOWN;
    err = process_teardown(rtspconn, mediaurl);
    close = 1;
    break;

case 6:
    close = 1;
    err = RTSP_SUCCESS;
    break;

default:
    err = RTSP_SUCCESS;
    printf("Invalid value \n");
    break;
}

/*
* The rtsp_methods array contains the strings for RTSP
* Methods. The array index should be an enumerated RTSP
* Method type
*/
if( err!=RTSP_SUCCESS )
    printf("Error occurred during %s message "
            "processing \n", rtsp_methods[mthd]);

} while(!close);

/* Close the RTSP Connection and free the resources */
rtsp_get_conn_opt(rtspconn, RTSP_OPT_SESSION,
                  &mapsession, NULL);
if( mapsession )
    rtsp_free_session(mapsession);
rtsp_close(rtspconn);

}

The usage of this sample program is as follows:
sampleclient <mediaurl>

Example:
sampleclient rtsp://serveraddress/mediafile
Compiling the Sample RTSP Program

This section describes how to compile the sample RTSP program.

Before compiling the program, ensure that the rtsp.h header file and the librtsp library provided by the HP-UX MSP suite are available on the machine. The rtsp.h header file is assumed to be in the /usr/include directory and the librtsp library in the /usr/lib directory. To compile, issue the following command:

```
$ cc sampleclient.c -o sampleclient -lrtsp
```

SDP Sample Program

The following is a sample program for SDP module of the MSP suite of libraries:

```c
#include <stdio.h>
#include <fcntl.h>
#include "sdp.h"

#define SDP_BUFSIZE 1024

sdp_t *init_parse_sdp(char *);
void print_session_level_fields(sdp_t *);
void print_media_descriptions(sdp_t *);
void print_media_attributes(sdp_t *);
int parse_media(sdp_t *, char *, char **);

main(int argc, char **argv)
{
    sdp_t *sdp;
    char sbuf[SDP_BUFSIZE];
    int fd;
    int n;
    if( argc <= 1)
        fd = open("./sample.in", O_RDONLY);
    else
        fd = open(argv[1], O_RDONLY);
        n=read(fd, sbuf, sizeof(sbuf));
        sbuf[n]= '0';
    sdp = init_parse_sdp(sbuf);
    if (sdp != NULL) {
        print_session_level_fields(sdp);
        print_media_descriptions(sdp);
        print_media_attributes(sdp);
    }
    sdp_free(sdp);
    return;
}

sdp_t *
init_parse_sdp(char *sbuf)
{
    sdp_t *psdp = NULL;
    int sdp_ret;
    if ((psdp = sdp_init()) == NULL) /* initialize sdp structure */
        return NULL;
    /* Parse the input buffer and fill up the sdp structure */
    if (sdp_ret = sdp_parse_buf(psd, sbuf) != SDP_SUCCESS)
        return NULL;
    return psdp;
}

/* Prints some of the session level descriptions */
```
void print_session_level_fields(sdp_t *sdp) {
    sdp_email_list_t *semail = sdp->email;
    printf("Session Name: %s
", sdp->s_name);
    printf("Session Information: %s
", sdp->i_info);
    if (semail != NULL) {
        do {
            printf("%s
", semail->e_email);
            semail = semail->next;
        } while (semail != NULL);
    }
    printf("URI: %s
", sdp->u_uri);
    return;
}

void print_media_descriptions(sdp_t *sdp) {
    sdp_media_list_t *mdes = sdp->media_des;
    if (mdes != NULL) {
        do {
            printf("Media Title: %s
", mdes->i_info);
            mdes = mdes->next;
        } while (mdes != NULL);
    }
    return;
}

void print_media_attributes(sdp_t *sdp) {
    int i, count = 0;
    char sbuf[SDP_BUFSIZE];
    char *stream[SDP_BUFSIZE];
    stream[0] = NULL;
    stream[1] = NULL;
    stream[2] = NULL;
    count = parse_media(sdp, sbuf, stream);
    printf("Media Attributes: 
");
    for(i=0; i<count; i++) {
        printf("%s
", stream[i]);
    }
    return;
}

int parse_media(sdp_t *sdp, char *sdpbuf, char **stream) {
    sdp_attribute_list_t *media_attr = NULL;
    sdp_media_list_t *mdes = NULL;
    int streamct = 0;
    char buf[SDP_BUFSIZE];

    mdes = (sdp_media_list_t *) sdp->media_des;
    while (mdes != NULL) {
        /* get sdp media attribute */
        media_attr = (sdp_attribute_list_t *) sdp_get_media_attribute(mdes);
        if(media_attr != NULL) {
            do {
                snprintf(buf, SDP_BUFSIZE, "a=%s:%s 0", media_attr->a_attr_name, media_attr->a_attr_val);
                *(stream + streamct) = (char *) malloc(sizeof(buf));
                strcpy(stream[streamct], buf);
                streamct++;
            } while (media_attr != NULL);
        }
    }
    return;
}
media_attr = media_attr->next;
} while(media_attr != NULL);
}
mdes = mdes->next;
}
return streamct;
}

Sample Output

A sample output of the SDP program is as follows:

v=0
o=mhandley 2890844526 2890842807 IN IP4 126.16.64.4
s=SDP Seminar
i=A Seminar on the session description protocol
u=http://www.cs.ucl.ac.uk/staff/M.Handley/sdp.03.ps
e=mjh@isi.edu (Mark Handley)
c=IN IP4 224.2.17.12/127
t=2873397496 2873404696
a=recvonly
m=audio 49170 RTP/AVP 0
i=MP3
m=video 51372 RTP/AVP 31
i=MPeg
a=orient:landscape
m=application 32416 udp wb
i=Netscape
a=orient:portrait
The librtsp Summary

This appendix presents a summary of the Real Time Streaming Protocol (RTSP) library that the Multimedia Streaming Protocol (MSP) Software Developer’s Kit (SDK) includes. The appendix contains the following sections:

- “The librtsp APIs” (page 107)
- “The librtsp Data Structures” (page 110)
- “The librtsp Enumerated Values” (page 111)

The librtsp APIs

Table A-1 lists and describes the librtsp APIs that the RTSP library provides.

<table>
<thead>
<tr>
<th>API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Connection</strong></td>
<td></td>
</tr>
<tr>
<td>rtsp_open</td>
<td>Creates a new rtsp_conn_t structure and opens a transport connection to the peer.</td>
</tr>
<tr>
<td>rtsp_create_conn</td>
<td>Creates a new rtsp_conn_t structure using an existing transport connection.</td>
</tr>
<tr>
<td>rtsp_set_conn_opt</td>
<td>Sets connection options in the rtsp_conn_t structure.</td>
</tr>
<tr>
<td>rtsp_get_conn_opt</td>
<td>Gets connection options from the rtsp_conn_t structure.</td>
</tr>
<tr>
<td>rtsp_close</td>
<td>Closes an RTSP connection.</td>
</tr>
<tr>
<td><strong>Session</strong></td>
<td></td>
</tr>
<tr>
<td>rtsp_init_session</td>
<td>Initializes the members of rtsp_session_t structure.</td>
</tr>
<tr>
<td>rtsp_free_session_flds</td>
<td>Frees the memory allocated for the rtsp_session_t structure members.</td>
</tr>
<tr>
<td>rtsp_free_session</td>
<td>Frees the memory allocated for the rtsp_session_t structure and its members.</td>
</tr>
<tr>
<td><strong>Message Creation/Destroy</strong></td>
<td></td>
</tr>
<tr>
<td>rtsp_init_request_msg</td>
<td>Creates a new rtsp_msg_t structure and sets the RTSP method and URL information.</td>
</tr>
<tr>
<td>API</td>
<td>Description</td>
</tr>
<tr>
<td>-----</td>
<td>-------------</td>
</tr>
<tr>
<td>rtsp_init_response_msg</td>
<td>Creates a new <code>rtsp_msg_t</code> structure and sets the RTSP status code and reason phrase information.</td>
</tr>
<tr>
<td>rtsp_free_msg</td>
<td>Frees the memory allocated for the <code>rtsp_msg_t</code> structure and its members.</td>
</tr>
</tbody>
</table>

**Message Type**

| rtsp_is_request_msg | Checks if the `rtsp_msg_t` structure represents an RTSP request message. |
| rtsp_is_response_msg | Checks if the `rtsp_msg_t` structure represents an RTSP response message. |

**Message Request/Response Line**

| rtsp_set_msg_request_line | Sets the RTSP method and URL information in the `rtsp_msg_t` structure. |
| rtsp_get_msg_request_line | Gets the RTSP method and URL information from the `rtsp_msg_t` structure. |
| rtsp_set_msg_response_line | Sets the RTSP status code and reason phrase information in the `rtsp_msg_t` structure. |
| rtsp_get_msg_response_line | Gets the RTSP status code and reason phrase information from the `rtsp_msg_t` structure. |

**Message Header**

| rtsp_set_msg_hdr | Sets header information in the `rtsp_msg_t` structure. |
| rtsp_get_msg_hdr | Gets header information from the `rtsp_msg_t` structure. |
| rtsp_copy_msg_hdr | Copies header from one `rtsp_msg_t` structure to another. |
| rtsp_append_msg_hdr | Appends header information in the `rtsp_msg_t` structure. |

**Message Body**

<p>| rtsp_set_msg_body | Sets message body in the <code>rtsp_msg_t</code> structure. |</p>
<table>
<thead>
<tr>
<th>API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rtsp_get_msg_body</td>
<td>Gets message body from the rtsp_msg_t structure.</td>
</tr>
<tr>
<td>Message Send/Receive</td>
<td></td>
</tr>
<tr>
<td>rtsp_send_msg</td>
<td>Sends an RTSP message over an RTSP connection.</td>
</tr>
<tr>
<td>rtsp_send_stream</td>
<td>Sends interleaved binary data over an RTSP connection.</td>
</tr>
<tr>
<td>rtsp_recv</td>
<td>Receives either an RTSP message or interleaved data from an RTSP connection.</td>
</tr>
<tr>
<td>Cache Control Header</td>
<td></td>
</tr>
<tr>
<td>rtsp_init_cache</td>
<td>Initializes the members of the rtsp_cache_t structure.</td>
</tr>
<tr>
<td>rtsp_alloc_cache</td>
<td>Creates a new rtsp_cache_t structure and initializes the structure members.</td>
</tr>
<tr>
<td>rtsp_free_cache</td>
<td>Frees the memory allocated for the entire linked list of rtsp_cache_t structures and their members.</td>
</tr>
<tr>
<td>Range Header</td>
<td></td>
</tr>
<tr>
<td>rtsp_init_range</td>
<td>Initializes the members of the rtsp_range_t structure.</td>
</tr>
<tr>
<td>rtsp_alloc_range</td>
<td>Creates a new rtsp_range_t structure and initializes the structure members.</td>
</tr>
<tr>
<td>rtsp_free_range</td>
<td>Frees the memory allocated for the entire linked list of rtsp_range_t structures and their members.</td>
</tr>
<tr>
<td>RTP-Info Header</td>
<td></td>
</tr>
<tr>
<td>rtsp_init_rtpinfo</td>
<td>Initializes the members of the rtsp_rtpinfo_t structure.</td>
</tr>
<tr>
<td>rtsp_alloc_rtpinfo</td>
<td>Creates a new rtsp_rtpinfo_t structure and initializes the structure members.</td>
</tr>
<tr>
<td>rtsp_free_rtpinfo</td>
<td>Frees the memory allocated for the entire linked list of rtsp_rtpinfo_t structures and their members.</td>
</tr>
<tr>
<td>Transport Header</td>
<td></td>
</tr>
</tbody>
</table>
Table A-1 The librtsp APIs (continued)

<table>
<thead>
<tr>
<th>API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rtsp_init_xport_spec</td>
<td>Initializes the members of the rtsp_xport_spec_t structure.</td>
</tr>
<tr>
<td>rtsp_alloc_xport_spec</td>
<td>Creates a new rtsp_xport_spec_t structure and initializes the structure</td>
</tr>
<tr>
<td></td>
<td>members.</td>
</tr>
<tr>
<td>rtsp_free_xport_spec_flds</td>
<td>Frees the memory allocated for the rtsp_xport_spec_t structure members.</td>
</tr>
<tr>
<td>rtsp_free_xport_spec</td>
<td>Frees the memory allocated for the entire linked list of rtsp_xport_spec_t structures and their members.</td>
</tr>
</tbody>
</table>

RTSP URL

| rtsp_parse_url | Parses the RTSP URL string into the rtsp_url_t structure. |

The librtsp Data Structures

Table A-2 lists and describes the librtsp data structures that the RTSP library provides.

Table A-2 The librtsp Data Structures

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rtsp_conn_t</td>
<td>Represents an RTSP connection.</td>
</tr>
<tr>
<td>rtsp_session_t</td>
<td>Represents an RTSP session.</td>
</tr>
<tr>
<td>rtsp_msg_t</td>
<td>Represents an RTSP message.</td>
</tr>
<tr>
<td>rtsp_cache_t</td>
<td>Represents the RTSP cache control header.</td>
</tr>
<tr>
<td>rtsp_range_t</td>
<td>Represents the RTSP range header.</td>
</tr>
<tr>
<td>rtsp_npt_t</td>
<td>Represents the NPT timestamp.</td>
</tr>
<tr>
<td>rtsp_utc_t</td>
<td>Represents the UTC timestamp.</td>
</tr>
<tr>
<td>rtsp_smpte_t</td>
<td>Represents the SMPTE timestamp.</td>
</tr>
<tr>
<td>rtsp_timestamp_t</td>
<td>Represents the RTSP timestamp header.</td>
</tr>
<tr>
<td>rtsp_rtpinfo_t</td>
<td>Represents the RTSP RTP-info header.</td>
</tr>
<tr>
<td>rtsp_xport_spec_t</td>
<td>Represents the RTSP transport header.</td>
</tr>
<tr>
<td>rtsp_url_t</td>
<td>Represents an RTSP URL.</td>
</tr>
</tbody>
</table>

The librtsp Arrays
Table A-2 The librtsp Data Structures (continued)

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rtsp_methods</td>
<td>Array of RTSP method strings.</td>
</tr>
<tr>
<td>rtsp_hdrs</td>
<td>Array of RTSP header field strings.</td>
</tr>
<tr>
<td>rtsp_status_code</td>
<td>Array of RTSP status code integers.</td>
</tr>
<tr>
<td>rtsp_status_code_str</td>
<td>Array of RTSP status code strings.</td>
</tr>
<tr>
<td>rtsp_status_text</td>
<td>Array of RTSP reason phrase strings.</td>
</tr>
</tbody>
</table>

The librtsp Enumerated Values

Table A-3 lists and describes the enumerated values that the RTSP library uses.

Table A-3 The librtsp Enumerated Values

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods</td>
<td></td>
</tr>
<tr>
<td>RTSP_DESCRIBE</td>
<td>DESCRIBE method.</td>
</tr>
<tr>
<td>RTSP_ANNOUNCE</td>
<td>ANNOUNCE method.</td>
</tr>
<tr>
<td>RTSP_GET_PARAMETER</td>
<td>GET_PARAMETER method.</td>
</tr>
<tr>
<td>RTSP_OPTIONS</td>
<td>OPTIONS method.</td>
</tr>
<tr>
<td>RTSP_PAUSE</td>
<td>PAUSE method.</td>
</tr>
<tr>
<td>RTSP_PLAY</td>
<td>PLAY method.</td>
</tr>
<tr>
<td>RTSP_RECORD</td>
<td>RECORD method.</td>
</tr>
<tr>
<td>RTSP_REDIRECT</td>
<td>REDIRECT method.</td>
</tr>
<tr>
<td>RTSP_SETUP</td>
<td>SETUP method.</td>
</tr>
<tr>
<td>RTSP_SET_PARAMETER</td>
<td>SET_PARAMETER method.</td>
</tr>
<tr>
<td>RTSP_TEARDOWN</td>
<td>TEARDOWN method.</td>
</tr>
<tr>
<td>Timestamps</td>
<td></td>
</tr>
<tr>
<td>RTSP_RANGE_NPT</td>
<td>Normal Play Time (NPT).</td>
</tr>
<tr>
<td>RTSP_RANGE.UTC</td>
<td>Absolute time as ISO 8601 timestamps using UTC.</td>
</tr>
<tr>
<td>RTSP_RANGE_SMPTE</td>
<td>SMPTE relative timestamp.</td>
</tr>
<tr>
<td>RTSP_RANGE_SMPTE_25</td>
<td>SMPTE-25 format.</td>
</tr>
<tr>
<td>RTSP_RANGE_SMPTE_30_DROP</td>
<td>SMPTE 30 drop format.</td>
</tr>
</tbody>
</table>
Table A-3 The librtsp Enumerated Values (continued)

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTSP_RANGE_TIME</td>
<td>Range ‘time’ specifier in UTC format.</td>
</tr>
<tr>
<td>Cache Directives</td>
<td></td>
</tr>
<tr>
<td>RTSP_CACHE_NO_CACHE</td>
<td>no-cache directive.</td>
</tr>
<tr>
<td>RTSP_CACHE_MAX_STALE</td>
<td>max-stale directive.</td>
</tr>
<tr>
<td>RTSP_CACHE_MIN_FRESH</td>
<td>min-fresh directive.</td>
</tr>
<tr>
<td>RTSP_CACHE_ONLYIF_CACHED</td>
<td>only-if-cached directive.</td>
</tr>
<tr>
<td>RTSP_CACHE_PUBLIC</td>
<td>public directive.</td>
</tr>
<tr>
<td>RTSP_CACHE_PRIVATE</td>
<td>private directive.</td>
</tr>
<tr>
<td>RTSP_CACHE_NO_TRANSFORM</td>
<td>no-transform directive.</td>
</tr>
<tr>
<td>RTSP_CACHE_MUST_REVAL</td>
<td>must-revalidate directive.</td>
</tr>
<tr>
<td>RTSP_CACHE_PROXY_REVAL</td>
<td>proxy-revalidate directive.</td>
</tr>
<tr>
<td>RTSP_CACHE_MAX_AGE</td>
<td>max-age directive.</td>
</tr>
<tr>
<td>RTSP_CACHE_EXT</td>
<td>Application-defined directive.</td>
</tr>
<tr>
<td>Connection Options</td>
<td></td>
</tr>
<tr>
<td>RTSP_OPT_SOCKET</td>
<td>Socket for the RTSP connection.</td>
</tr>
<tr>
<td>RTSP_OPT_SESSION</td>
<td>Mapped RTSP session for the RTSP connection.</td>
</tr>
<tr>
<td>Transport Mode Parameter</td>
<td></td>
</tr>
<tr>
<td>RTSP_MODE_PLAY</td>
<td>PLAY mode.</td>
</tr>
<tr>
<td>RTSP_MODE_RECORD</td>
<td>RECORD mode.</td>
</tr>
<tr>
<td>Transport Delivery Parameter</td>
<td></td>
</tr>
<tr>
<td>RTSP_TRANSMIT_UNICAST</td>
<td>Unicast delivery.</td>
</tr>
<tr>
<td>RTSP_TRANSMIT_MULTICAST</td>
<td>Multicast delivery.</td>
</tr>
<tr>
<td>Header Fields</td>
<td></td>
</tr>
<tr>
<td>RTSP_ACCEPT_HDR</td>
<td>Accept header.</td>
</tr>
<tr>
<td>RTSP_ACCEPT_ENCODING_HDR</td>
<td>Accept-encoding header.</td>
</tr>
<tr>
<td>RTSP_ACCEPT_LANGUAGE_HDR</td>
<td>Accept-language header.</td>
</tr>
<tr>
<td>RTSP_ACCEPT_RANGE_HDR</td>
<td>Accept-ranges header.</td>
</tr>
<tr>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>RTSP_ALLOW_HDR</td>
<td>Allow header.</td>
</tr>
<tr>
<td>RTSP_AUTHORIZATION_HDR</td>
<td>Authorization header.</td>
</tr>
<tr>
<td>RTSP_BANDWIDTH_HDR</td>
<td>Bandwidth header.</td>
</tr>
<tr>
<td>RTSP_BLOCKSIZE_HDR</td>
<td>Blocksize header.</td>
</tr>
<tr>
<td>RTSP_CACHE_CONTROL_HDR</td>
<td>Cache control header.</td>
</tr>
<tr>
<td>RTSP_CONNECTION_HDR</td>
<td>Connection header.</td>
</tr>
<tr>
<td>RTSP_CONTENT_BASE_HDR</td>
<td>Content-base header.</td>
</tr>
<tr>
<td>RTSP_CONTENT_ENCODING_HDR</td>
<td>Content-encoding header.</td>
</tr>
<tr>
<td>RTSP_CONTENT_LANGUAGE_HDR</td>
<td>Content-language header.</td>
</tr>
<tr>
<td>RTSP_CONTENT_LENGTH_HDR</td>
<td>Content-length header.</td>
</tr>
<tr>
<td>RTSP_CONTENT_LOCATION_HDR</td>
<td>Content-location header.</td>
</tr>
<tr>
<td>RTSP_CONTENT_TYPE_HDR</td>
<td>Content-type header.</td>
</tr>
<tr>
<td>RTSP_CSEQ_HDR</td>
<td>Cseq header.</td>
</tr>
<tr>
<td>RTSP_DATE_HDR</td>
<td>Date header.</td>
</tr>
<tr>
<td>RTSP_EXPIRES_HDR</td>
<td>Expires header.</td>
</tr>
<tr>
<td>RTSP_FROM_HDR</td>
<td>From header.</td>
</tr>
<tr>
<td>RTSP_IF_MATCH_HDR</td>
<td>If-match header.</td>
</tr>
<tr>
<td>RTSP_IF_MODIFIED_SINCE_HDR</td>
<td>If-modified-since header.</td>
</tr>
<tr>
<td>RTSP_LAST_MODIFIED_HDR</td>
<td>Last-modified header.</td>
</tr>
<tr>
<td>RTSP_LOCATION_HDR</td>
<td>Location header.</td>
</tr>
<tr>
<td>RTSP_PROXY_AUTHENTICATE_HDR</td>
<td>Proxy-authenticate header.</td>
</tr>
<tr>
<td>RTSP_PROXY_REQUIRE_HDR</td>
<td>Proxy-require header.</td>
</tr>
<tr>
<td>RTSP_PUBLIC_HDR</td>
<td>Public header.</td>
</tr>
<tr>
<td>RTSP_RANGE_HDR</td>
<td>Range header.</td>
</tr>
<tr>
<td>RTSPREFERER_HDR</td>
<td>Referer header.</td>
</tr>
<tr>
<td>RTSP_REQUIRE_HDR</td>
<td>Require header.</td>
</tr>
<tr>
<td>RTSP_RETRY_AFTER_HDR</td>
<td>Retry-after header.</td>
</tr>
<tr>
<td>RTSP_RTP_INFO_HDR</td>
<td>RTP-Info header.</td>
</tr>
<tr>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>RTSP_SCALE_HDR</td>
<td>Scale header.</td>
</tr>
<tr>
<td>RTSP_SESSION_HDR</td>
<td>Session header.</td>
</tr>
<tr>
<td>RTSP_SERVER_HDR</td>
<td>Server header.</td>
</tr>
<tr>
<td>RTSP_SPEED_HDR</td>
<td>Speed header.</td>
</tr>
<tr>
<td>RTSP_SUPPORTED_HDR</td>
<td>Supported header.</td>
</tr>
<tr>
<td>RTSP_TIMESTAMP_HDR</td>
<td>Timestamp header.</td>
</tr>
<tr>
<td>RTSP_TRANSPORT_HDR</td>
<td>Transport header.</td>
</tr>
<tr>
<td>RTSP_UNSUPPORTED_HDR</td>
<td>Unsupported header.</td>
</tr>
<tr>
<td>RTSP_USER_AGENT_HDR</td>
<td>User-agent header.</td>
</tr>
<tr>
<td>RTSP_VARY_HDR</td>
<td>Vary header.</td>
</tr>
<tr>
<td>RTSP_VIA_HDR</td>
<td>Via header.</td>
</tr>
<tr>
<td>RTSP_WWW_AUTHENTICATE_HDR</td>
<td>WWW-authenticate header.</td>
</tr>
</tbody>
</table>
Index