JRTPLIB 3.4.0

Jori Liesenborgs
jori@lumumba.uhasselt.be

January 19, 2006

Developed at the The Expertise Centre for Digital Media (EDM), a research institute of the Hasselt University

http://www.edm.uhasselt.be/
http://www.uhasselt.be/

Contents

Acknowledgment

I would like thank the people at the Expertise Centre for Digital Media for giving me the opportunity to create this rewrite of the library.

1 Introduction

This document describes JRTPLIB version 3.4.0, an object-oriented library written in C++ which aims to help developers in using the Real-time Transport Protocol (RTP) as described in RFC 3550.

The library makes it possible for the user to send and receive data using RTP, without worrying about SSRC collisions, scheduling and transmitting RTCP data etc. The user only needs to provide the library with the payload data to be sent and the library gives the user access to incoming RTP and RTCP data.

1.1 Design idea

The library provides several classes which can be helpful in creating RTP applications. Most users will probably need just the `RTPSession` class for building an application. This class provides the necessary functions for sending RTP data and handles the RTCP part internally.

For applications such as a mixer or translator using the `RTPSession` class will not be a good solution. Other components can be used for this purpose: a transmission component, an SSRC table, an RTCP scheduler etc. Using these, it should be much easier to build all kinds of applications.

1.2 Changes from version 2.x

One of the most important changes is probably the fact that this version is based on RFC 3550 and the 2.x versions were based upon RFC 1889 which is now obsolete.

Also, the 2.x series was created with the idea that the user would only need to use the `RTPSession` class which meant that the other classes were not very useful by themselves. This version on the other hand, aims to provide many useful components to aid the user in building RTP capable applications.

In this version, the code which is specific for the underlying protocol by which RTP packets are transported, is bundled in a class which inherits its interface from a class called `RTPTransmitter`. This makes it easy for different underlying protocols to be supported. Currently there is support for UDP over IPv4 and UDP over IPv6.

2 Copyright license

The library code uses the following copyright license:

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

There are two reasons for using this license. First, since this is the license of the 2.x series, it only seemed natural that this rewrite would contain the same license. Second, since the RTP protocol is deliberately incomplete RTP profiles can, for example, define additional header fields. The best way to deal with this is to adapt the library code itself and that's why I like to keep the license as free as possible.

3 Using JRTPLIB 3.4.0

This section gives information about how to use the library. First, some simple examples will be given which illustrate how you can work with the `RTPSession` class. Afterwards, a complete description of the API will be given.

3.1 Getting started with the `RTPSession` class

To use RTP, you'll have to create an `RTPSession` object. The constructor of the `RTPSession` class takes a parameter of type `RTPTransmitter::TransmissionProtocol` and defaults to `RTPTransmitter::IPv4UDPProto`. This means that unless specified otherwise, the UDP over IPv4 transmission component will be used. Let's suppose that this is the kind of session you want to create, then this is our code so far:

```
RTPSession session;
```

To actually create the session, you'll have to call the `Create` member function which takes two arguments: the first one is of type `RTPSessionParams` and specifies the general options for the session. One parameter of this class must be set explicitly, otherwise the session will not be created successfully. This parameter is the timestamp unit of the data you intend to send and can be calculated by dividing a certain time interval (in seconds) by the number of samples in that interval. So, assuming that we'll send $8000Hz$ voice data, we can use this code:

```
RTPSessionParams sessionparams;

sessionparams.SetOwnTimestampUnit(1.0/8000.0);
```

The other parameters will probably depend on the actual RTP profile you intend to work with. For a complete description of the `RTPSessionParams` class, see section 3.2.20.

The second argument of the `Create` function is a pointer to an `RTPTransmissionParams` instance and describes the parameters for the transmission component. Since there can be several transmission components, you'll have to use a class which inherits `RTPTransmissionParams` and which is appropriate for the component you've chosen. For our UDP over IPv4 component, the class to be used is `RTPUDPv4TransmissionParams`. Assuming that we want our RTP portbase to be 8000, we can do the following:

```
RTPUDPv4TransmissionParams transparams;

transparams.SetPortbase(8000);
```

Now, we're ready to call the `Create` member function of `RTPSession`. The return value is stored in the integer `status` so we can check if something went wrong. If this value is negative, it indicates that some error occurred. A description of what this error code means can be retrieved by calling `RTPGetErrorString`:

```
int status = session.Create(sessionparams,&transparams);
if (status < 0)
{
    std::cerr << RTPGetErrorString(status) << std::endl;
    exit(-1);
}
```

If the session was created with success, this is probably a good point to specify to which destinations RTP and RTCP data should be sent. This is done by a call to the `RTPSession` member function `AddDestination`. This function takes an argument of type `RTPAddress`. This is an abstract class and for the UDP over IPv4 transmitter the actual class to be used is `RTPIPv4Address`. Suppose that we want to send our data to a process running on the same host at port 9000, we can do the following:

```
u_int8_t localip[]={127,0,0,1};
RTPIPv4Address addr(localip,9000);

status = session.AddDestination(addr);
if (status < 0)
{
    std::cerr << RTPGetErrorString(status) << std::endl;
    exit(-1);
}
```

If the library was compiled with JThread support, incoming data is processed in the background. If JThread support was not enabled at compile time or if you specified in the session parameters that no poll thread should be used, you'll have to call the `RTPSession` member function `Poll` regularly to process incoming data and to send RTCP data when necessary. For now, let's assume that we're working with the poll thread enabled.

Lets suppose that for a duration of one minute, we want to send packets containing $20ms$ (or 160 samples) of silence and we want to indicate when a packet from someone else has been received. Also suppose we have $L8$ data as defined in RFC 3551 and want to use payload type 96. First, we'll set some default values:

```
session.SetDefaultPayloadType(96);
session.SetDefaultMark(false);
session.SetDefaultTimestampIncrement(160);
```

Next, we'll create the buffer which contains 160 silence samples and create an `RTPTime` instance which indicates $20ms$ or 0.020 seconds. We'll also store the current time so we'll know when one minute has passed.

```
u_int8_t silencebuffer[160];

for (int i = 0 ; i < 160 ; i++)
    silencebuffer[i] = 128;

RTPTime delay(0.020);
RTPTime starttime = RTPTime::CurrentTime();
```

Next, the main loop will be shown. In this loop, a packet containing 160 bytes of payload data will be sent. Then, data handling can take place but this part is described later in the text. Finally, we'll wait $20ms$ and check if sixty seconds have passed:

```
bool done = false;
while (!done)
{
    status = session.SendPacket(silencebuffer,160);
```

```
    if (status < 0)
    {
        std::cerr << RTPGetErrorString(status) << std::endl;
        exit(-1);
    }

    //
    // Inspect incoming data here
    //

    RTPTime::Wait(delay);

    RTPTime t = RTPTime::CurrentTime();
    t -= starttime;
    if (t > RTPTime(60.0))
        done = true;
}
```

Information about participants in the session, packet retrieval etc, has to be done between calls to the `RTPSession` member functions `BeginDataAccess` and `EndDataAccess`. This ensures that the background thread doesn't try to change the same data you're trying to access. We'll iterate over the participants using the `GotoFirstSource` and `GotoNextSource` member functions. Packets from the currently selected participant can be retrieved using the `GetNextPacket` member function which returns a pointer to an instance of the `RTPPacket` class. When you don't need the packet anymore, it has to be deleted. The processing of incoming data will then be as follows:

```
session.BeginDataAccess();
if (session.GotoFirstSource())
{
    do
    {
        RTPPacket *packet = session.GetNextPacket();
        if (packet)
        {
            std::cout << "Got packet with extended sequence number "
                      << packet->GetExtendedSequenceNumber()
                      << " from SSRC " << packet->GetSSRC()
                      << std::endl;
            delete packet;
        }
    } while (session.GotoNextSource());
}
session.EndDataAccess();
```

Information about the currently selected source can be obtained by using the `GetCurrentSourceInfo` member function of the `RTPSession` class. This function returns a pointer to an instance of `RTPSourceData` which contains all information about that source: sender reports from that source, receiver reports, SDES info etc. The `RTPSourceData` class is described in detail in section 3.2.15.

When the main loop is finished, we'll send a BYE packet to inform other participants of our departure and clean up the `RTPSession` class. Also, we want to wait at most 10 seconds for the BYE packet to be sent, otherwise we'll just leave the session without sending a BYE packet.

```
delay = RTPTime(10.0);
session.BYEDestroy(delay,"Time's up",9);
```

The complete code of the program is given in `example2.cpp` and is shown on the following pages. More detailed information about the `RTPSession` class can be found in section 3.2.21.

```
#include "rtpsession.h"
#include "rtpsessionparams.h"
#include "rtpudpv4transmitter.h"
#include "rtpipv4address.h"
#include "rtptimeutilities.h"
#include "rtppacket.h"
#include <stdlib.h>
#include <iostream>

int main(void)
{
#ifdef WIN32
    WSADATA dat;
    WSAStartup(MAKEWORD(2,2),&dat);
#endif // WIN32

    RTPSession session;

    RTPSessionParams sessionparams;
    sessionparams.SetOwnTimestampUnit(1.0/8000.0);

    RTPUDPv4TransmissionParams transparams;
    transparams.SetPortbase(8000);

    int status = session.Create(sessionparams,&transparams);
    if (status < 0)
    {
        std::cerr << RTPGetErrorString(status) << std::endl;
        exit(-1);
    }

    uint8_t localip[]={127,0,0,1};
    RTPIPv4Address addr(localip,9000);

    status = session.AddDestination(addr);
    if (status < 0)
    {
        std::cerr << RTPGetErrorString(status) << std::endl;
        exit(-1);
    }

    session.SetDefaultPayloadType(96);
    session.SetDefaultMark(false);
    session.SetDefaultTimestampIncrement(160);

    uint8_t silencebuffer[160];
```

```
for (int i = 0 ; i < 160 ; i++)
    silencebuffer[i] = 128;

RTPTime delay(0.020);
RTPTime starttime = RTPTime::CurrentTime();

bool done = false;
while (!done)
{
    status = session.SendPacket(silencebuffer,160);
    if (status < 0)
    {
        std::cerr << RTPGetErrorString(status) << std::endl;
        exit(-1);
    }

    session.BeginDataAccess();
    if (session.GotoFirstSource())
    {
        do
        {
            RTPPacket *packet = session.GetNextPacket();
            if (packet)
            {
                std::cout << "Got packet with "
                          << "extended sequence number "
                          << packet->GetExtendedSequenceNumber()
                          << " from SSRC " << packet->GetSSRC()
                          << std::endl;
                delete packet;
            }
        } while (session.GotoNextSource());
    }
    session.EndDataAccess();

    RTPTime::Wait(delay);

    RTPTime t = RTPTime::CurrentTime();
    t -= starttime;
    if (t > RTPTime(60.0))
        done = true;
}

delay = RTPTime(10.0);
session.BYEDestroy(delay,"Time's up",9);
```

```
#ifdef WIN32
    WSACleanup();
#endif // WIN32
    return 0;
}
```

3.2 The complete API

Here, the complete API of the library will be explained. This will be done in a more or less bottom-to-top fashion.

3.2.1 Library version

Header: `rtplibraryversion.h`

The `RTPLibraryVersion` class has a static member which creates an instance of this class:

```
static RTPLibraryVersion GetVersion();
```

The user can access the version data using the following member functions:

- `int GetMajorNumber() const`
  Returns the major version number.
- `int GetMinorNumber() const`
  Returns the minor version number.
- `int GetDebugNumber() const`
  Returns the debug version number.
- `std::string GetVersionString() const`
  Returns a string describing the library version.

3.2.2 Error codes

Header: `rtperrors.h`

Unless specified otherwise, functions with a return type `int` will return a negative value when an error occurred and zero or a positive value upon success. A description of the error code can be obtained by using the following function:

```
std::string RTPGetErrorString(int errcode)
```

3.2.3 Time utilities

Header: `rtptimeutilities.h`

`RTPNTPTime`

This is a simple wrapper for the most significant word (MSW) and least significant word (LSW) of an NTP timestamp. The class has the following members:

- `RTPNTPTime(u_int32_t m, u_int32_t l)`
  This constructor creates and instance with MSW `m` and LSW `l`.

- `u_int32_t GetMSW() const`
  Returns the most significant word.
- `u_int32_t GetLSW() const`
  Returns the least significant word.

`RTPTime`

This class is used to specify wallclock time, delay intervals etc. It stores a number of seconds and a number of microseconds and it has the following interface:

- `RTPTime(u_int32_t seconds, u_int32_t microseconds)`
  Creates an instance corresponding to `seconds` and `microseconds`.
- `RTPTime(double t)`
  Creates an `RTPTime` instance representing `t` which is expressed in units of seconds.
- `RTPTime(RTPNTPTime ntptime)`
  Creates an instance that corresponds to `ntptime`. If the conversion cannot be made, both the seconds and the microseconds are set to zero.
- `u_int32_t GetSeconds() const`
  Returns the number of seconds stored in this instance.
- `u_int32_t GetMicroSeconds() const`
  Returns the number of microseconds stored in this instance.
- `double GetDouble() const`
  Returns the time stored in this instance, expressed in units of seconds.
- `RTPNTPTime GetNTPTime() const`
  Returns the NTP time corresponding to the time stored in this instance.
- `static RTPTime CurrentTime()`
  Returns an RTPTime instance representing the current wallclock time. This is expressed as a number of seconds since 00:00:00 UTC, January 1, 1970.
- `static void Wait(const RTPTime &delay)`
  This function waits the amount of time specified in `delay`.

The following operators are defined in the `RTPTime` class:

- `operator-=`
- `operator+=`
- `operator<`

- operator>
- operator<=
- operator>=

3.2.4 RTPRandom

Header: rtprandom.h

The RTPRandom class can be used to generate random numbers. It has the following member functions:

- u_int8_t GetRandom8()
  Returns a random eight bit value.
- u_int16_t GetRandom16()
  Returns a random sixteen bit value.
- u_int32_t GetRandom32()
  Returns a random thirty-two bit value.
- double GetRandomDouble()
  Returns a random number between 0.0 and 1.0.

3.2.5 RTCPSDESInfo

Header: rtcpsdesinfo.h

The class RTCPSDESInfo is a container for RTCP SDES information. The interface is the following:

- void Clear()
  Clears all SDES information.
- int SetCNAME(const u_int8_t *s, size_t l)
  Sets the SDES CNAME item to s with length l.
- int SetName(const u_int8_t *s, size_t l)
  Sets the SDES name item to s with length l.
- int SetEMail(const u_int8_t *s, size_t l)
  Sets the SDES e-mail item to s with length l.
- int SetPhone(const u_int8_t *s, size_t l)
  Sets the SDES phone item to s with length l.
- int SetLocation(const u_int8_t *s, size_t l)
  Sets the SDES location item to s with length l.
- int SetTool(const u_int8_t *s, size_t l)
  Sets the SDES tool item to s with length l.

- `int SetNote(const u_int8_t *s, size_t l)`
  Sets the SDES note item to `s` with length `l`.

- `u_int8_t *GetCNAME(size_t *len) const`
  Returns the SDES CNAME item and stores its length in `len`.

- `u_int8_t *GetName(size_t *len) const`
  Returns the SDES name item and stores its length in `len`.

- `u_int8_t *GetEMail(size_t *len) const`
  Returns the SDES e-mail item and stores its length in `len`.

- `u_int8_t *GetPhone(size_t *len) const`
  Returns the SDES phone item and stores its length in `len`.

- `u_int8_t *GetLocation(size_t *len) const`
  Returns the SDES location item and stores its length in `len`.

- `u_int8_t *GetTool(size_t *len) const`
  Returns the SDES tool item and stores its length in `len`.

- `u_int8_t *GetNote(size_t *len) const`
  Returns the SDES note item and stores its length in `len`.

If SDES private item support was enabled at compile time, the following member functions are also available:

- `int SetPrivateValue(const u_int8_t *prefix, size_t prefixlen, const u_int8_t *value, size_t valuelen)`
  Sets the entry for the prefix string specified by `prefix` with length `prefixlen` to contain the value string specified by `value` with length `valuelen`. If the maximum allowed number of prefixes was reached, the error code `ERR_RTP_SDES_MAXPRIVITEMS` is returned.

- `int DeletePrivatePrefix(const u_int8_t *s, size_t len)`
  Deletes the entry for the prefix specified by `s` with length `len`.

- `void GotoFirstPrivateValue()`
  Starts the iteration over the stored SDES private item prefixes and their associated values.

- `bool GetNextPrivateValue(u_int8_t **prefix, size_t *prefixlen, u_int8_t **value, size_t *valuelen)`
  If available, returns `true` and stores the next SDES private item prefix in `prefix` and its length in `prefixlen`. The associated value and its length are then stored in `value` and `valuelen`. Otherwise, it returns `false`.

- `bool GetPrivateValue(const u_int8_t *prefix, size_t prefixlen, u_int8_t **value, size_t *valuelen) const`
  Looks for the entry which corresponds to the SDES private item prefix `prefix` with length `prefixlen`. If found, the function returns `true` and stores the associated value and its length in `value` and `valuelen` respectively.

3.2.6 `RTPTransmitter`

Header: `rtptransmitter.h`

The abstract class `RTPTransmitter` specifies the interface for actual transmission components. Currently, three implementations exist: an UDP over IPv4 transmitter and an UDP over IPv6 transmitter. The `TransmissionProtocol` type is used to specify a specific kind of transmitter:

```
enum TransmissionProtocol { IPv4UDPProto, IPv6UDPProto,
                            UserDefinedProto };
```

The `UserDefinedProto` can be used to select your own transmission component when using the `RTPSession` class. In this case, you'll have to implement the `RTPSession` member function `NewUserDefinedTransmitter()` which should return a pointer to your own `RTPTransmitter` implementation.

Three kind of receive modes can be specified using the `ReceiveMode` type:

```
enum ReceiveMode { AcceptAll, AcceptSome, IgnoreSome };
```

Depending on the mode set, incoming data is processed differently:

- `AcceptAll`
  All incoming data is accepted, no matter where it originated from.
- `AcceptSome`
  Only data coming from specific sources will be accepted.
- `IgnoreSome`
  All incoming data is accepted, except for data coming from a specific set of sources.

The interface defined by the `RTPTransmission` class is the following:

- `int Init(bool threadsafe)` This function must be called before the transmission component can be used. Depending on the value of `threadsafe`, the component will be created for thread-safe usage or not.

- `int Create(size_t maxpacksize, const RTPTransmissionParams *transparams)`
  Prepares the component to be used. The parameter `maxpacksize` specifies the maximum size a packet can have: if the packet is larger it will not be transmitted. The `transparams` parameter specifies a pointer to an `RTPTransmissionParams` instance. This is also an abstract class and each actual component will define its own parameters by inheriting a a class from `RTPTransmissionParams`. If `transparams` is NULL, the default transmission parameters for the component will be used.

- `int Destroy()`
  By calling this function, buffers are cleared and the component cannot be used anymore. Only when the `Create` function is called again can the component be used again.

- `RTPTransmissionInfo *GetTransmissionInfo()`
  This function returns an instance of a subclass of `RTPTransmissionInfo` which will give some additional information about the transmitter (a list of local IP addresses for example). Currently, either an instance of `RTPUDPv4TransmissionInfo` or `RTPUDPv6TransmissionInfo` is returned, depending on the type of the transmitter. The user has to delete the returned instance when it is no longer needed.

- `int GetLocalHostName(u_int8_t *buffer, size_t *bufferlength)`
  Looks up the local host name based upon internal information about the local host's addresses. This function might take some time since a DNS query might be done. `bufferlength` should initially contain the number of bytes that may be stored in `buffer`. If the function succeeds, `bufferlength` is set to the number of bytes stored in `buffer`. Note that the data in `buffer` is not NULL-terminated. If the function fails because the buffer isn't large enough, it returns `ERR_RTP_TRANS_BUFFERLENGTHTOOSMALL` and stores the number of bytes needed in `bufferlength`.

- `bool ComesFromThisTransmitter(const RTPAddress *addr)`
  Returns `true` if the address specified by `addr` is one of the addresses of the transmitter.

- `size_t GetHeaderOverhead()`
  Returns the amount of bytes that will be added to the RTP packet by the underlying layers (excluding the link layer).

- `int Poll()`
  Checks for incoming data and stores it.

- `bool NewDataAvailable()` Returns `true` if packets can be obtained using the `GetNextPacket` member function.

- `RTPRawPacket *GetNextPacket()`
  Returns the raw data of a received RTP packet (received during the `Poll` function) in an `RTPRawPacket` instance.

- `int WaitForIncomingData(const RTPTime &delay,bool *dataavailable = 0)`
  Waits at most a time `delay` until incoming data has been detected. If `dataavailable` is not `NULL`, it should be set to `true` if data was actually read and to `false` otherwise.

- `int AbortWait()`
  If the previous function has been called, this one aborts the waiting.

- `int SendRTPData(const void *data, size_t len)`
  Send a packet with length `len` containing `data` to all RTP addresses of the current destination list.

- `int SendRTCPData(const void *data, size_t len)`
  Send a packet with length `len` containing `data` to all RTCP addresses of the current destination list.

- `void ResetPacketCount()`
  The transmitter keeps track of the amount of RTP and RTCP packets that were sent. This functions resets those counts.

- `u_int32_t GetNumRTPPacketsSent()`
  Returns the number of RTP packets sent.

- `u_int32_t GetNumRTCPPacketsSent()`
  Returns the number of RTCP packets sent.

- `int AddDestination(const RTPAddress &addr)`
  Adds the address specified by `addr` to the list of destinations.

- `int DeleteDestination(const RTPAddress &addr)`
  Deletes the address specified by `addr` from the list of destinations.

- `void ClearDestinations()`
  Clears the list of destinations.

- `bool SupportsMulticasting()`
  Returns `true` if the transmission component supports multicasting.

- `int JoinMulticastGroup(const RTPAddress &addr)`
  Joins the multicast group specified by `addr`.

- `int LeaveMulticastGroup(const RTPAddress &addr)`
  Leaves the multicast group specified by `addr`.

- `void LeaveAllMulticastGroups()`
  Leaves all the multicast groups that have been joined.

- `int SetReceiveMode(RTPTransmitter::ReceiveMode m)`
  Sets the receive mode to m, which is one of the following: `RTPTransmitter::AcceptAll`, `RTPTransmitter::AcceptSome`, `RTPTransmitter::IgnoreSome`. Note that if the receive mode is changed, all information about the addresses to ignore or to accept is lost.
- `int AddToIgnoreList(const RTPAddress &addr)`
  Adds `addr` to the list of addresses to ignore.
- `int DeleteFromIgnoreList(const RTPAddress &addr)`
  Deletes `addr` from the list of addresses to ignore.
- `void ClearIgnoreList()`
  Clears the list of addresses to ignore.
- `int AddToAcceptList(const RTPAddress &addr)`
  Adds `addr` to the list of addresses to accept.
- `int DeleteFromAcceptList(const RTPAddress &addr)`
  Deletes `addr` from the list of addresses to accept.
- `void ClearAcceptList()`
  Clears the list of addresses to accept.
- `int SetMaximumPacketSize(size_t s)`
  Sets the maximum packet size which the transmitter should allow to `s`.

UDP over IPv4 transmitter

Header:
`rtpudpv4transmitter.h`

Inherits:
`RTPTransmitter`

The class `RTPUDPv4Transmitter` inherits the `RTPTransmitter` interface and implements a transmission component which user UDP over IPv4 to send and receive RTP and RTCP data.

The component's parameters are described by the class `RTPUDPv4TransmissionParams` and is described in section 3.2.7. The functions which have an `RTPAddress` argument require an argument of type `RTPIPv4Address` which is described in section 3.2.9. The `GetTransmissionInfo` member function of the `RTPUDPv4Transmitter` class returns an instance of type `RTPUDPv4TransmissionInfo` which is described in section 3.2.8.

UDP over IPv6 transmitter

Header:
`rtpudpv6transmitter.h`

Inherits:
`RTPTransmitter`

The class `RTPUDPv6Transmitter` inherits the `RTPTransmitter` interface and implements a transmission component which user UDP over IPv6 to send and receive RTP and RTCP data.

The component's parameters are described by the class `RTPUDPv6TransmissionParams` and is described in section 3.2.7. The functions which have an `RTPAddress` argument require an argument of type `RTPIPv6Address` which is described in section 3.2.9. The `GetTransmissionInfo` member function of the `RTPUDPv6Transmitter` class returns an instance of type `RTPUDPv6TransmissionInfo` which is described in section 3.2.8.

3.2.7 `RTPTransmissionParams`

Header:
`rtptransmitter.h`

The `RTPTransmissionParams` class is an abstract class which will have a specific implementation for a specific kind of transmission component. All actual implementations inherit the following function which identify the component type for which these parameters are valid:

```
RTPTransmitter::TransmissionProtocol GetTransmissionProtocol()
```

Parameters for the UDP over IPv4 transmitter

Header:
`rtpudpv4transmitter.h`
Inherits:
`RTPTransmissionParams`

The `RTPUDPv4TransmissionParams` class represents the parameters used by the UDP over IPv4 transmission component. By default, the multicast TTL is set to 1 and the portbase is set to 5000. The interface of this class is the following:

- `void SetBindIP(u_int32_t ip)`
  Sets the IP address which is used to bind the sockets to `ip`.
- `void SetPortbase(u_int16_t pbase)`
  Sets the RTP portbase to `pbase`. This has to be an even number.
- `void SetMulticastTTL(u_int8_t mcastTTL)`
  Sets the multicast TTL to be used to `mcastTTL`.
- `void SetLocalIPList(std::list<u_int32_t> &iplist)`
  Passes a list of IP addresses which will be used as the local IP addresses.
- `void ClearLocalIPList()`
  Clears the list of local IP addresses. An empty list will make the transmission component itself determine the local IP addresses.
- `u_int32_t GetBindIP() const`
  Returns the IP address which will be used to bind the sockets.
- `u_int16_t GetPortbase() const`
  Returns the RTP portbase which will be used.
- `u_int8_t GetMulticastTTL() const`
  Returns the multicast TTL which will be used.

- `const std::list<u_int32_t> &GetLocalIPList() const`
  Returns the list of local IP addresses.

Parameters for the UDP over IPv6 transmitter

Header: `rtpudpv6transmitter.h`
Inherits: `RTPTransmissionParams`

The `RTPUDPv6TransmissionParams` class represents the parameters used by the UDP over IPv6 transmission component. By default, the multicast TTL is set to 1 and the portbase is set to 5000. The interface of this class is the following:

- `void SetBindIP(in6_addr ip)`
  Sets the IP address which is used to bind the sockets to `ip`.
- `void SetPortbase(u_int16_t pbase)`
  Sets the RTP portbase to `pbase`. This has to be an even number.
- `void SetMulticastTTL(u_int8_t mcastTTL)`
  Sets the multicast TTL to be used to `mcastTTL`.
- `void SetLocalIPList(std::list<in6_addr> &iplist)`
  Passes a list of IP addresses which will be used as the local IP addresses.
- `void ClearLocalIPList()`
  Clears the list of local IP addresses. An empty list will make the transmission component itself determine the local IP addresses.
- `in6_addr GetBindIP() const`
  Returns the IP address which will be used to bind the sockets.
- `u_int16_t GetPortbase() const`
  Returns the RTP portbase which will be used.
- `u_int8_t GetMulticastTTL() const`
  Returns the multicast TTL which will be used.
- `const std::list<in6_addr> &GetLocalIPList() const`
  Returns the list of local IP addresses.

3.2.8 `RTPTransmissionInfo`

Header: `rtptransmitter.h`

The `RTPTransmissionInfo` class is an abstract class which will have a specific implementation for a specific kind of transmission component. All actual implementations inherit the following function which identify the component type for which these parameters are valid:

```
RTPTransmitter::TransmissionProtocol GetTransmissionProtocol()
```

Info about the UDP over IPv4 transmitter

Header:
`rtpudpv4transmitter.h`

The class `RTPUDPv4TransmissionInfo` gives some additional information about the UDP over IPv4 transmission component. The following member functions are available:

- `std::list<u_int32_t> GetLocalIPList() const`
  Returns the list of IPv4 addresses the transmitter considers to be the local IP addresses.
- `int GetRTPSocket() const`
  Returns the socket descriptor used for receiving and transmitting RTP packets.
- `int GetRTCPSocket() const`
  Returns the socket descriptor used for receiving and transmitting RTCP packets.

Info about the UDP over IPv6 transmitter

Header:
`rtpudpv6transmitter.h`

The class `RTPUDPv6TransmissionInfo` gives some additional information about the UDP over IPv6 transmission component. The following member functions are available:

- `std::list<in6_addr> GetLocalIPList() const`
  Returns the list of IPv4 addresses the transmitter considers to be the local IP addresses.
- `int GetRTPSocket() const`
  Returns the socket descriptor used for receiving and transmitting RTP packets.
- `int GetRTCPSocket() const`
  Returns the socket descriptor used for receiving and transmitting RTCP packets.

3.2.9 `RTPAddress`

Header:
`rtpaddress.h`

The class `RTPAddress` is an abstract class which is used to specify destinations, multicast groups etc. To check which actual implementation is used, the class defines the following type:

```
enum AddressType { IPv4Address, IPv6Address, UserDefinedAddress };
```

The type `RTPAddress::IPv4Address` is used by the UDP over IPv4 transmitter; `RTPAddress::IPv6Address` is used by the UDP over IPv6 transmitter. The `RTPAddress::UserDefinedAddress` type can be useful when using a user-defined transmission component.

The class defines the following interface:

- `AddressType GetAddressType() const`
  Returns the type of address the actual implementation represents.
- `RTPAddress *CreateCopy() const`
  Creates a copy of the `RTPAddress` instance.
- `bool IsSameAddress(const RTPAddress *addr) const`
  Checks if the address `addr` is the same address as the one this instance represents. Implementations must be able ti handle a NULL argument.
- `bool IsFromSameHost(const RTPAddress *addr) const`
  Checks if the address `addr` represents the same host as this instance. Implementations must be able to handle a NULL argument.

`RTPIPv4Address`

Header: `rtpipv4address.h`
Inherits: `RTPAddress`

This class is used by the UDP over IPv4 transmission component. The following member functions are defined in this class:

- `RTPIPv4Address(u_int32_t ip = 0, u_int16_t port = 0)`
  Creates an instance with IP address `ip` and port number `port`. Both are interpreted in host byte order.
- `RTPIPv4Address(const u_int8_t ip[4], u_int16_t port = 0)`
  Creates an instance with IP address `ip` and port `port`. The port number is interpreted in host byte order.
- `void SetIP(u_int32_t ip)`
  Sets the IP address for this instance to `ip` which is assumed to be in host byte order.
- `void SetIP(const u_int8_t ip[4])`
  Sets the IP address of this instance to `ip`.
- `void SetPort(u_int16_t port)`
  Sets the port number for this instance to `port` which is interpreted in host byte order.
- `u_int32_t GetIP() const`
  Returns the IP address contained in this instance in host byte order.

- `u_int16_t GetPort() const`
  Returns the port number of this instance in host byte order.

When an `RTPIPv4Address` is used in one of the multicast functions of the transmitter, the port number is ignored. When an instance is used in one of the accept or ignore functions of the transmitter, a zero port number represents all ports for the specified IP address.

`RTPIPv6Address`

Header: `rtpipv6address.h`
Inherits: `RTPAddress`

This class is used by the UDP over IPv4 transmission component. The following member functions are defined:

- `RTPIPv6Address()`
  Creates an instance with IP address and port number set to zero.
- `RTPIPv6Address(const u_int8_t ip[16], u_int16_t port = 0)`
  Creates an instance with IP address `ip` and port number `port`. The port number is assumed to be in host byte order.
- `RTPIPv6Address(in6_addr ip, u_int16_t port = 0)`
  Creates an instance with IP address `ip` and port number `port`. The port number is assumed to be in host byte order.
- `void SetIP(in6_addr ip)`
  Sets the IP address for this instance to `ip`.
- `void SetIP(const u_int8_t ip[16])`
  Sets the IP address for this instance to `ip`.
- `void SetPort(u_int16_t port)`
  Sets the port number for this instance to `port`, which is interpreted in host byte order.
- `void GetIP(u_int8_t ip[16]) const`
  Copies the IP address of this instance in `ip`.
- `in6_addr GetIP() const`
  Returns the IP address of this instance.
- `u_int16_t GetPort() const`
  Returns the port number contained in this instance in host byte order.

When an `RTPIPv6Address` is used in one of the multicast functions of the transmitter, the port number is ignored. When an instance is used in one of the accept or ignore functions of the transmitter, a zero port number represents all ports for the specified IP address.

3.2.10 RTPRawPacket

Header: rtprawpacket.h

The RTPRawPacket class is used by the transmission component to store the incoming RTP and RTCP data in. It has the following interface:

- RTPRawPacket(u_int8_t *data, size_t datalen, RTPAddress *address, RTPTime &recvtime, bool rtp)
  Creates an instance which stores data from data with length datalen. Only the pointer to the data is stored, no actual copy is made! The address from which this packet originated is set to address and the time at which the packet was received is set to recvtime. The flag which indicates whether this data is RTP or RTCP data is set to rtp.
- u_int8_t *GetData()
  Returns the pointer to the data which is contained in this packet.
- size_t GetDataLength() const
  Returns the length of the packet described by this instance.
- RTPTime GetReceiveTime() const
  Returns the time at which this packet was received.
- const RTPAddress *GetSenderAddress() const
  Returns the address stored in this packet.
- bool IsRTP() const
  Returns true if this data is RTP data, false if it is RTCP data.
- void ZeroData()
  Sets the pointer to the data stored in this packet to zero. This will prevent a delete call for the actual data when the destructor of RTPRawPacket is called. This function is used by the RTPPacket and RTCPCompoundPacket classes to obtain the packet data (without having to copy it) and to make sure the data isn't deleted when the destructor of RTPRawPacket is called.

3.2.11 RTPPacket

Header: rtppacket.h

The RTPPacket class can be used to parse a RTPRawPacket instance if it represents RTP data. The class can also be used to create a new RTP packet according to the parameters specified by the user. The interface of this class is the following:

- RTPPacket(RTPRawPacket &rawpack)
  Creates an RTPPacket instance based upon the data in rawpack.
- RTPPacket(u_int8_t payloadtype, const void *payloaddata, size_t payloadlen, u_int16_t seqnr, u_int32_t timestamp, u_int32_t ssrc, bool gotmarker, u_int8_t numcsrcs, const u_int32_t *csrcs, bool gotextension,

`u_int16_t extensionid, u_int16_t extensionlen_numwords, const void *extensiondata, size_t maxpacksize = 0)`
Creates an new buffer for an RTP packet and fills in the fields according to the specified parameters. If `maxpacksize` is not equal to zero, an error is generated if the total packet size would exceed `maxpacksize`. The arguments of the constructor are self-explanatory. Note that the size of a header extension is specified in a number of 32-bit words.

- `RTPPacket(u_int8_t payloadtype, const void *payloaddata, size_t payloadlen, u_int16_t seqnr, u_int32_t timestamp, u_int32_t ssrc, bool gotmarker, u_int8_t numcsrcs, const u_int32_t *csrcs, bool gotextension, u_int16_t extensionid, u_int16_t extensionlen_numwords, const void *extensiondata, void *buffer, size_t buffersize)`
  Pretty much the same function as the previous one, except that data is stored in an external buffer `buffer` with buffer size `buffersize`.

- `int GetCreationError() const`
  If an error occurred in one of the constructors, this function returns the error code.

- `bool HasExtension() const`
  Returns `true` if the RTP packet has a header extension and `false` otherwise.

- `bool HasMarker() const`
  Returns `true` is the marker bit was set and `false` otherwise.

- `int GetCSRCCount() const`
  Returns the number of CSRCs contained in this packet.

- `u_int32_t GetCSRC(int num) const`
  Returns a specific CSRC identifier. The parameter `num` can go from 0 to `GetCSRCCount()`-1.

- `u_int8_t GetPayloadType() const`
  Returns the payload type of the packet.

- `u_int32_t GetExtendedSequenceNumber() const`
  Returns the extended sequence number of the packet. When the packet is just received, only the low 16 bits will be set. The high 16 bits can be filled in later.

- `u_int16_t GetSequenceNumber() const`
  Returns the sequence number of this packet.

- `void SetExtendedSequenceNumber(u_int32_t seq)`
  Sets the extended sequence number of this packet to `seq`.

- `u_int32_t GetTimestamp() const`
  Returns the timestamp of this packet.

- `u_int32_t GetSSRC() const`
  Returns the SSRC identifier stored in this packet.

- `u_int8_t *GetPacketData() const`
  Returns a pointer to the data of the entire packet.

- `u_int8_t *GetPayloadData() const`
  Returns a pointer to the actual payload data.

- `size_t GetPacketLength() const`
  Returns the length of the entire packet.

- `size_t GetPayloadLength() const`
  Returns the payload length.

- `u_int16_t GetExtensionID() const`
  If a header extension is present, this function returns the extension identifier.

- `u_int8_t *GetExtensionData() const`
  Returns a pointer to the header extension data.

- `size_t GetExtensionLength() const`
  Returns the length of the header extension data.

- `RTPTime GetReceiveTime() const`
  When an `RTPPacket` instance is created from an `RTPRawPacket` instance, the raw packet's reception time is stored in the `RTPPacket` instance. This function then retrieves that time.

3.2.12 `RTCPCompoundPacket`

Header: `rtpcompoundpacket.h`

This class describes an RTCP compound packet and has the following interface:

- `RTCPCompoundPacket(RTPRawPacket &rawpack)`
  Creates an `RTCPCompoundPacket` instance from the data in `rawpack`.

- `int GetCreationError()`
  If the raw packet data in the constructor could not be parsed, this function returns the error code of what went wrong. If the packet had an invalid format, the return value is `ERR_RTP_RTCPCOMPOUND_INVALIDPACKET`.

- `u_int8_t *GetCompoundPacketData()`
  Returns a pointer to the data of the entire RTCP compound packet.

- `size_t GetCompoundPacketLength()`
  Returns the size of the entire RTCP compound packet

- `void GotoFirstPacket()`
  Starts the iteration over the individual RTCP packets in the RTCP compound packet.
- `RTCPPacket *GetNextPacket()`
  Returns a pointer to the next individual RTCP packet. Note that no `delete` call may be done on the `RTCPPacket` instance which is returned. The `RTCPPacket` class is described below.

`RTCPPacket`

Header: `rtcppacket.h`

The class `RTCPPacket` is a base class for specific types of RTCP packets. The following type in the `RTCPPacket` class identifies the different packet types:

```
enum PacketType { SR, RR, SDES, BYE, APP, Unknown };
```

The class contains the following member functions:

- `bool IsKnownFormat() const`
  Returns `true` if the subclass was able to interpret the data and `false` otherwise.
- `PacketType GetPacketType() const`
  Returns the actual packet type which the subclass implements:
  - `RTCPPacket::SR`: indicates an `RTCPSRPacket` instance
  - `RTCPPacket::RR`: indicates an `RTCPRRPacket` instance
  - `RTCPPacket::SDES`: indicates an `RTCPSDESPacket` instance
  - `RTCPPacket::BYE`: indicates an `RTCPBYEPacket` instance
  - `RTCPPacket::APP`: indicates an `RTCPAPPPacket` instance
  - `RTCPPacket::Unknown`: indicates an `RTCPUnknownPacket` instance
- `u_int8_t *GetPacketData()`
  Returns a pointer to the data of this RTCP packet.
- `size_t GetPacketLength() const`
  Returns the length of this RTCP packet.

`RTCPSRPacket`

Header: `rtcpsrpacket.h`

Inherits: `RTCPPacket`

This class describes an RTCP sender report packet. The interface is the following:

- `RTCPSRPacket(u_int8_t *data, size_t datalen)`
  Creates an instance based on the data in `data` with length `datalen`. Since the `data` pointer is referenced inside the class (no copy of the data is made) one must make sure that the memory it points to is valid as long as the class instance exists.

- `u_int32_t GetSenderSSRC() const`
  Returns the SSRC of the participant who sent this packet.

- `RTPNTPTime GetNTPTimestamp() const`
  Returns the NTP timestamp contained in the sender report.

- `u_int32_t GetRTPTimestamp() const`
  Returns the RTP timestamp contained in the sender report.

- `u_int32_t GetSenderPacketCount() const`
  Returns the sender's packet count contained in the sender report.

- `u_int32_t GetSenderOctetCount() const`
  Returns the sender's octet count contained in the sender report.

- `int GetReceptionReportCount() const`
  Returns the number of reception report blocks present in this packet.

- `u_int32_t GetSSRC(int index) const`
  Returns the SSRC of the reception report block described by `index` which may have a value from 0 to `GetReceptionReportCount()`-1. Note that no check is performed to see if `index` is valid.

- `u_int8_t GetFractionLost(int index) const`
  Returns the 'fraction lost' field of the reception report described by `index` which may have a value from 0 to `GetReceptionReportCount()`-1. Note that no check is performed to see if `index` is valid.

- `int32_t GetLostPacketCount(int index) const`
  Returns the number of lost packets in the reception report block described by `index` which may have a value from 0 to `GetReceptionReportCount()`-1. Note that no check is performed to see if `index` is valid.

- `u_int32_t GetExtendedHighestSequenceNumber(int index) const`
  Returns the extended highest sequence number of the reception report block described by `index` which may have a value from 0 to `GetReceptionReportCount()`-1. Note that no check is performed to see if `index` is valid.

- `u_int32_t GetJitter(int index) const`
  Returns the jitter field of the reception report block described by `index` which may have a value from 0 to `GetReceptionReportCount()`-1. Note that no check is performed to see if `index` is valid.

- `u_int32_t GetLSR(int index) const`
  Returns the LSR field of the reception report block described by `index` which may have a value from 0 to `GetReceptionReportCount()`-1. Note that no check is performed to see if `index` is valid.

- `u_int32_t GetDLSR(int index) const`
  Returns the DLSR field of the reception report block described by `index` which may have a value from 0 to `GetReceptionReportCount()`-1. Note that no check is performed to see if `index` is valid.

`RTCPRRPacket`

Header: `rtcprrpacket.h`
Inherits: `RTCPPacket`

This class describes an RTCP receiver report packet. The interface is the following:

- `RTCPRRPacket(u_int8_t *data, size_t datalen)`
  Creates an instance based on the data in `data` with length `datalen`. Since the `data` pointer is referenced inside the class (no copy of the data is made) one must make sure that the memory it points to is valid as long as the class instance exists.

- `u_int32_t GetSenderSSRC() const`
  Returns the SSRC of the participant who sent this packet.

- `int GetReceptionReportCount() const`
  Returns the number of reception report blocks present in this packet.

- `u_int32_t GetSSRC(int index) const`
  Returns the SSRC of the reception report block described by `index` which may have a value from 0 to `GetReceptionReportCount()`-1. Note that no check is performed to see if `index` is valid.

- `u_int8_t GetFractionLost(int index) const`
  Returns the 'fraction lost' field of the reception report described by `index` which may have a value from 0 to `GetReceptionReportCount()`-1. Note that no check is performed to see if `index` is valid.

- `int32_t GetLostPacketCount(int index) const`
  Returns the number of lost packets in the reception report block described by `index` which may have a value from 0 to `GetReceptionReportCount()`-1. Note that no check is performed to see if `index` is valid.

- `u_int32_t GetExtendedHighestSequenceNumber(int index) const`
  Returns the extended highest sequence number of the reception report block described by `index` which may have a value from 0 to `GetReception-ReportCount()`-1. Note that no check is performed to see if `index` is valid.

- `u_int32_t GetJitter(int index) const`
  Returns the jitter field of the reception report block described by `index` which may have a value from 0 to `GetReceptionReportCount()`-1. Note that no check is performed to see if `index` is valid.

- `u_int32_t GetLSR(int index) const`
  Returns the LSR field of the reception report block described by `index` which may have a value from 0 to `GetReceptionReportCount()`-1. Note that no check is performed to see if `index` is valid.

- `u_int32_t GetDLSR(int index) const`
  Returns the DLSR field of the reception report block described by `index` which may have a value from 0 to `GetReceptionReportCount()`-1. Note that no check is performed to see if `index` is valid.

`RTCPSDESPacket`

Header: `rtcpsdespacket.h`
Inherits: `RTCPPacket`

This class describes an RTCP SDES packet. In the `RTCPSDESPacket` class, the following type is defined:

```
enum ItemType { None, CNAME, NAME, EMAIL, PHONE,
                LOC, TOOL, NOTE, PRIV, Unknown };
```

This type is used to identify the type of an SDES item. The type `None` is used when the iteration over the items has finished; `Unknown` is used when there really is an item present, but the type is not a standard type.

The class interface is the following:

- `RTCPSDESPacket(u_int8_t *data, size_t datalen)`
  Creates an instance based on the data in `data` with length `datalen`. Since the `data` pointer is referenced inside the class (no copy of the data is made) one must make sure that the memory it points to is valid as long as the class instance exists.

- `int GetChunkCount() const`
  Returns the number of SDES chunks in the SDES packet. Each chunk has its own SSRC identifier.

- `bool GotoFirstChunk()`
  Starts the iteration. If no SDES chunks are present, the function returns `false`. Otherwise, it returns `true` and sets the current chunk to be the first chunk.

- `bool GotoNextChunk()`
  Sets the current chunk to the next available chunk. If no next chunk is present, this function returns `false`, otherwise it returns `true`.

- `u_int32_t GetChunkSSRC() const`
  Returns the SSRC identifier of the current chunk.

- `bool GotoFirstItem()`
  Starts the iteration over the SDES items in the current chunk. If no SDES items are present, the function returns `false`. Otherwise, the function sets the current item to be the first one and returns `true`.

- `bool GotoNextItem()`
  If there's another item in the chunk, the current item is set to be the next one and the function returns `true`. Otherwise, the function returns `false`.

- `ItemType GetItemType() const`
  Returns the SDES item type of the current item in the current chunk.

- `size_t GetItemLength() const`
  Returns the item length of the current item in the current chunk.

- `u_int8_t *GetItemData()`
  Returns the item data of the current item in the current chunk.

If SDES private item support was enabled at compile-time, the following members can be used if the current item is an SDES private item.

- `size_t GetPRIVPrefixLength()`
  Returns the length of the prefix string of the private item.

- `u_int8_t *GetPRIVPrefixData()`
  Returns the actual data of the prefix string.

- `size_t GetPRIVValueLength()`
  Returns the length of the value string of the private item.

- `u_int8_t *GetPRIVValueData()`
  Returns the actual value data of the private item.

`RTCPAPPPacket`

Header: `rtcpapppacket.h`
Inherits: `RTCPPacket`

The class `RTCPAPPPacket` describes an RTCP APP packet. The following member functions are available:

- `RTCPAPPPacket(u_int8_t *data, size_t datalen)`
  Creates an instance based on the data in `data` with length `datalen`. Since the `data` pointer is referenced inside the class (no copy of the data is made) one must make sure that the memory it points to is valid as long as the class instance exists.

- `u_int8_t GetSubType() const`
  Returns the subtype contained in the APP packet.
- `u_int32_t GetSSRC() const`
  Returns the SSRC of the source which sent this packet.
- `u_int8_t *GetName()`
  Returns the name contained in the APP packet. This always consists of four bytes and is not NULL-terminated.
- `u_int8_t *GetAPPData()`
  Returns a pointer to the actual data.
- `size_t GetAPPDataLength() const`
  Returns the length of the actual data.

`RTCPBYEPacket`

Header: `rtcpbyepacket.h`
Inherits: `RTCPPacket`

An RTCP BYE packet is represented by the class `RTCPBYEPacket` which has the following member functions:

- `RTCPBYEPacket(u_int8_t *data, size_t datalen)`
  Creates an instance based on the data in `data` with length `datalen`. Since the `data` pointer is referenced inside the class (no copy of the data is made) one must make sure that the memory it points to is valid as long as the class instance exists.
- `int GetSSRCCount() const`
  Returns the number of SSRC identifiers present in this BYE packet.
- `u_int32_t GetSSRC(int index) const`
  Returns the SSRC described by `index` which may have a value from 0 to `GetSSRCCount()`-1. Note that no check is performed to see if `index` is valid.
- `bool HasReasonForLeaving() const`
  Returns true if the BYE packet contains a reason for leaving.
- `size_t GetReasonLength() const`
  Returns the length of the string which describes why the source(s) left.
- `u_int8_t *GetReasonData()`
  Returns the actual reason data.

`RTCPUnknownPacket`

Header: `rtcpunknownpacket.h`
Inherits: `RTCPPacket`

This class doesn't have any extra member functions besides the ones it inherited. Note that since an unknown packet type doesn't have any format to check against, the `IsKnownFormat` function will trivially return `true`. Only the following constructor is available:

- `RTCPUnknownPacket(u_int8_t *data, size_t datalen)`
  Creates an instance based on the data in `data` with length `datalen`. Since the `data` pointer is referenced inside the class (no copy of the data is made) one must make sure that the memory it points to is valid as long as the class instance exists.

3.2.13 `RTCPCompoundPacketBuilder`

Header: `rtpcompoundpacketbuilder.h`
Inherits: `RTCPCompoundPacket`

The `RTCPCompoundPacketBuilder` class can be used to construct an RTCP compound packet. It inherits the member functions of `RTCPCompoundPacket` which can be used to access the information in the compound packet once it has been built successfully. The member functions described below return `ERR_RTP_RTCPCOMPPACKBUILDER_NOTENOUGHBYTESLEFT` if the action would cause the maximum allowed size to be exceeded.

- `int InitBuild(size_t maxpacketsize)`
  Starts building an RTCP compound packet with maximum size `maxpacketsize`. New memory will be allocated to store the packet.
- `int InitBuild(void *externalbuffer, size_t buffersize)`
  Starts building a packet. Data will be stored in `externalbuffer` which can contain `buffersize` bytes.
- `int StartSenderReport(u_int32_t senderssrc, const RTPNTPTime &ntptimestamp, u_int32_t rtptimestamp, u_int32_t packetcount, u_int32_t octetcount)`
  Tells the packet builder that the packet should start with a sender report which will contain the sender information specified by this function's arguments. Once the sender report is started, report blocks can be added using the `AddReportBlock` function.
- `int StartReceiverReport(u_int32_t senderssrc)`
  Tells the packet builder that the packet should start with a receiver report which will contain the sender SSRC `senderssrc`. Once the sender report is started, report blocks can be added using the `AddReportBlock` function.
- `int AddReportBlock(u_int32_t ssrc, u_int8_t fractionlost, int32_t packetslost, u_int32_t exthighestseq, u_int32_t jitter, u_int32_t lsr, u_int32_t dlsr)`
  Adds the report block information specified by the function's arguments. If more than 31 report blocks are added, the builder will automatically use a new RTCP receiver report packet.

- `int AddSDESSource(u_int32_t ssrc)`
  Starts an SDES chunk for participant `ssrc`.

- `int AddSDESNormalItem(RTCPSDESPacket::ItemType t, const void *itemdata, u_int8_t itemlength)`
  Adds a normal (non-private) SDES item of type `t` to the current SDES chunk. The item's value will have length `itemlength` and will contain the data `itemdata`.

- `int AddBYEPacket(u_int32_t *ssrcs, u_int8_t numssrcs, const void *reasondata, u_int8_t reasonlength)`
  Adds a BYE packet to the compound packet. It will contain `numssrcs` source identifiers specified in `ssrcs` and will indicate as reason for leaving the string of length `reasonlength` containing data `reasondata`.

- `int AddAPPPacket(u_int8_t subtype, u_int32_t ssrc, const u_int8_t name[4], const void *appdata, size_t appdatalen)`
  Adds the APP packet specified by the arguments to the compound packet. Note that `appdatalen` has to be a multiple of four.

- `int EndBuild()`
  Finished building the compound packet. If successful, the `RTCPCompoundPacket` member functions can be used to access the RTCP packet data.

If the library was compiled with SDES private item support, the following member function is also available:

- `int AddSDESPrivateItem(const void *prefixdata, u_int8_t prefixlength, const void *valuedata, u_int8_t valuelength)`
  Adds an SDES PRIV item described by the function's arguments to the current SDES chunk.

3.2.14 `RTPSources`

Header: `rtpsources.h`

The `RTPSources` class represents a table in which information about the participating sources is kept. The class has member functions to process RTP and RTCP data and to iterate over the participants. Note that a NULL address is used to identify packets from our own session. The class also provides some overridable functions which can be used to catch certain events (new SSRC, SSRC collision, ...).

When probation support is enabled, you can select one of three probation types:

```
enum ProbationType { NoProbation, ProbationDiscard, ProbationStore };
```

When `NoProbation` is selected, the probation algorithm will not be used to validate new sources. When `ProbationDiscard` is used, the probation algorithm is activated but received packets will be discarded until the source is validated. To activate the probation routine and store the packets which are received before the source is validated, the mode `ProbationStore` can be selected.

The RTPSources class interface is the following:

- `RTPSources(ProbationType probationtype = ProbationStore)`
  In the constructor you can select the probation type you'd like to use. This only makes a difference when probation support was enabled at compilation time.

- `void Clear()`
  Clears the source table.

- `int CreateOwnSSRC(u_int32_t ssrc)`
  Creates an entry for our own SSRC identifier.

- `int DeleteOwnSSRC()`
  Deletes the entry for our own SSRC identifier.

- `void SentRTPPacket()`
  For our own SSRC entry, the sender flag is updated based upon outgoing packets instead of incoming packets. This function should be called if our own session has sent an RTP packet.

- `int ProcessRawPacket(RTPRawPacket *rawpack, RTPTransmitter *trans, bool acceptownpackets)`
  Processes a raw packet `rawpack`. The instance `trans` will be used to check if this packet is one of our own packets. The flag `acceptownpackets` indicates whether own packets should be accepted or ignored.

- `int ProcessRawPacket(RTPRawPacket *rawpack, RTPTransmitter *trans[], int numtrans, bool acceptownpackets)`
  Same function as the previous one, except that every transmitter in the array `trans` of length `numtrans` is used to check if the packet is from our own session.

- `int ProcessRTPPacket(RTPPacket *rtppack, const RTPTime &receivetime, const RTPAddress *senderaddress, bool *stored)`
  Processes an `RTPPacket` instance `rtppack` which was received at time `receivetime`. And which originated from `senderaddress`. The `senderaddress` parameter must be NULL if the packet was sent by the local participant. The flag `stored` indicates whether the packet was stored in the table or not. If so, the `rtppack` instance may not be deleted.

- `int ProcessRTCPCompoundPacket(RTCPCompoundPacket *rtcpcomppack, const RTPTime &receivetime, const RTPAddress *senderaddress)`

Processes the RTCP compound packet `rtcpcomppack` which was received at time `receivetime` from `senderaddress`. The `senderaddress` parameter must be NULL if the packet was sent by the local participant.

- `int ProcessRTCPSenderInfo(u_int32_t ssrc, const RTPNTPTime &ntptime, u_int32_t rtptime, u_int32_t packetcount, u_int32_t octetcount, const RTPTime &receivetime, const RTPAddress *senderaddress)`
  Process the sender information of SSRC `ssrc` into the source table. The information was received at time `receivetime` from address `senderaddress`. The `senderaddress` parameter must be NULL if the packet was sent by the local participant.

- `int ProcessRTCPReportBlock(u_int32_t ssrc, u_int8_t fractionlost, int32_t lostpackets, u_int32_t exthighseqnr, u_int32_t jitter, u_int32_t lsr, u_int32_t dlsr, const RTPTime &receivetime, const RTPAddress *senderaddress)`
  Processes the report block information which was sent by participant `ssrc` into the source table. The information was received at time `receivetime` from address `senderaddress`. The `senderaddress` parameter must be NULL if the packet was sent by the local participant.

- `int ProcessSDESNormalItem(u_int32_t ssrc, RTCPSDESPacket::ItemType t, size_t itemlength, const void *itemdata, const RTPTime &receivetime, const RTPAddress *senderaddress)`
  Processes the non-private SDES item from source `ssrc` into the source table. The information was received at time `receivetime` from address `senderaddress`. The `senderaddress` parameter must be NULL if the packet was sent by the local participant.

- `int ProcessBYE(u_int32_t ssrc, size_t reasonlength, const void *reasondata, const RTPTime &receivetime, const RTPAddress *senderaddress)`
  Processes the BYE message for SSRC `ssrc`. The information was received at time `receivetime` from address `senderaddress`. The `senderaddress` parameter must be NULL if the packet was sent by the local participant.

- `int UpdateReceiveTime(u_int32_t ssrc, const RTPTime &receivetime, const RTPAddress *senderaddress)`
  If we heard from source `ssrc`, but no actual data was added to the source table (for example, if no report block was meant for us), this function can be used to indicate that something was received from this source. This will prevent a premature timeout for this participant. The message was received at time `receivetime` from address `senderaddress`. The `senderaddress` parameter must be NULL if the packet was sent by the local participant.

- `bool GotoFirstSource()`
  Starts the iteration over the participants by going to the first member in the table. If a member was found, the function returns `true`, otherwise it returns `false`.

- `bool GotoNextSource()`
  Sets the current source to be the next source in the table. If we're already at the last source, the function returns `false`, otherwise it returns `true`.

- `bool GotoPreviousSource()`
  Sets the current source to be the previous source in the table. If we're at the first source, the function returns `false`, otherwise it returns `true`.

- `bool GotoFirstSourceWithData()`
  Sets the current source to be the first source in the table which has `RTPPacket` instances that we haven't extracted yet. If no such member was found, the function returns `false`, otherwise it returns `true`.

- `bool GotoNextSourceWithData()`
  Sets the current source to be the next source in the table which has `RTPPacket` instances that we haven't extracted yet. If no such member was found, the function returns `false`, otherwise it returns `true`.

- `bool GotoPreviousSourceWithData()`
  Sets the current source to be the previous source in the table which has `RTPPacket` instances that we haven't extracted yet. If no such member was found, the function returns `false`, otherwise it returns `true`.

- `RTPSourceData *GetCurrentSourceInfo()`
  Returns the `RTPSourceData` instance for the currently selected participant.

- `RTPSourceData *GetSourceInfo(u_int32_t ssrc)`
  Returns the `RTPSourceData` instance for the participant identified by `ssrc`, or NULL if no such entry exists.

- `RTPPacket *GetNextPacket()`
  Extracts the next packet from the received packets queue of the current participant.

- `bool GotEntry(u_int32_t ssrc)`
  Returns `true` if an entry for participant `ssrc` exists and `false` otherwise.

- `RTPSourceData *GetOwnSourceInfo()`
  If present, it returns the `RTPSourceData` instance of the entry which was created by `CreateOwnSSRC`.

- `void Timeout(const RTPTime &curtime, const RTPTime &timeoutdelay)`
  Assuming that the current time is `curtime`, time out the members from whom we haven't heard during the previous time interval `timeoutdelay`.

- `void SenderTimeout(const RTPTime &curtime, const RTPTime &timeoutdelay)`
  Assuming that the current time is `curtime`, remove the sender flag for senders from whom we haven't received any RTP packets during the previous time interval `timeoutdelay`.

- `void BYETimeout(const RTPTime &curtime, const RTPTime &timeoutdelay)`
  Assuming that the current time is `curtime`, remove the members who sent a BYE packet more than the time interval `timeoutdelay` ago.

- `void NoteTimeout(const RTPTime &curtime, const RTPTime &timeoutdelay)`
  Assuming that the current time is `curtime`, clear the SDES NOTE items which haven't been updated in during the previous time interval `timeoutdelay`.

- `void MultipleTimeouts(const RTPTime &curtime, const RTPTime &sendertimeout, const RTPTime &byetimeout, const RTPTime &generaltimeout, const RTPTime &notetimeout)`
  Combines the previous four functions. This is more efficient than calling all four functions since only one iteration is needed in this function.

- `int GetSenderCount() const`
  Returns the number of participants which are marked as a sender.

- `int GetTotalCount() const`
  Returns the total number of entries in the source table.

- `int GetActiveMemberCount() const`
  Returns the number of members which have been validated and which haven't sent a BYE packet yet.

If SDES private item support was enabled at compile-time, the following member function is also available:

- `int ProcessSDESPrivateItem(u_int32_t ssrc, size_t prefixlen, const void *prefixdata, size_t valuelen, const void *valuedata, const RTPTime &receivetime, const RTPAddress *senderaddress)`
  Processes the SDES private item from source `ssrc` into the source table. The information was received at time `receivetime` from address `senderaddress`. The `senderaddress` parameter must be NULL if the packet was sent by the local participant.

By inheriting your own class from `RTPSources` and overriding one or more of the functions below, your application can be informed of certain events:

- `void OnRTPPacket(RTPPacket *pack, const RTPTime &receivetime, const RTPAddress *senderaddress)`
  Is called when an RTP packet is about to be processed.

- `void OnRTCPCompoundPacket(RTCPCompoundPacket *pack, const RTPTime &receivetime, const RTPAddress *senderaddress)`
  Is called when an RTCP packet is about to be processed.

- `void OnSSRCCollision(RTPSourceData *srcdat, const RTPAddress *senderaddress, bool isrtp)`
  Is called when an SSRC collision was detected. The instance `srcdat` is the one present in the table, the address `senderaddress` is the one that collided with one of the addresses and `isrtp` indicates against which address of `srcdat` the check failed.

- `void OnCNAMECollision(RTPSourceData *srcdat, const RTPAddress *senderaddress, const u_int8_t *cname, size_t cnamelength)`
  Is called when another CNAME was received than the one already present for source `srcdat`.

- `void OnNewSource(RTPSourceData *srcdat)`
  Is called when a new entry `srcdat` is added to the source table.

- `void OnRemoveSource(RTPSourceData *srcdat)`
  Is called when the entry `srcdat` is about to be deleted from the source table.

- `void OnTimeout(RTPSourceData *srcdat)`
  Is called when participant `srcdat` is timed out.

- `void OnBYETimeout(RTPSourceData *srcdat)`
  Is called when participant `srcdat` is timed after having sent a BYE packet.

- `void OnBYEPacket(RTPSourceData *srcdat)`
  Is called when a BYE packet has been processed for source `srcdat`.

- `void OnAPPPacket(RTCPAPPPacket *apppacket, const RTPTime &receivetime, const RTPAddress *senderaddress)`
  In called when an RTCP APP packet `apppacket` has been received at time `receivetime` from address `senderaddress`.

- `void OnUnknownPacketType(RTCPPacket *rtcppack, const RTPTime &receivetime, const RTPAddress *senderaddress)`
  Is called when an unknown RTCP packet type was detected.

- `void OnUnknownPacketFormat(RTCPPacket *rtcppack, const RTPTime &receivetime, const RTPAddress *senderaddress)`
  Is called when an unknown packet format for a known packet type was detected.

- `void OnNoteTimeout(RTPSourceData *srcdat)`
  Is called when the SDES NOTE item for source `srcdat` has been timed out.

3.2.15 `RTPSourceData`

Header: `rtpsourcedata.h`

The `RTPSourceData` class contains all information about a member of the session. The interface is the following:

- `u_int32_t GetSSRC() const`
  Returns the SSRC identifier for this member.

- `bool HasData() const`
  Returns `true` if there are RTP packets which can be extracted.

- `RTPPacket *GetNextPacket()`
  Extracts the first packet of this participants RTP packet queue.

- `void FlushPackets()`
  Clears the participant's RTP packet list.

- `bool IsOwnSSRC() const`
  Returns `true` if the participant was added using the `RTPSources` member function `CreateOwnSSRC` and returns `false` otherwise.

- `bool IsCSRC() const`
  Returns true if the source identifier is actually a CSRC from an RTP packet.

- `bool IsSender() const`
  Returns `true` if this member is marked as a sender and `false` if not.

- `bool IsValidated() const`
  Returns `true` if the participant is validated: this is so if a number of consecutive RTP packets have been received or if a CNAME item has been received for this participant.

- `bool IsActive() const`
  Returns `true` if the source was validated and had not yet sent a BYE packet.

- `void SetProcessedInRTCP(bool v)`
  This function is used by the `RTCPPacketBuilder` class to mark whether this participant's information has been processed in a report block or not.

- `bool IsProcessedInRTCP() const`
  This function is used by the `RTCPPacketBuilder` class and returns whether this participant has been processed in a report block or not.

- `bool IsRTPAddressSet() const`
  Returns `true` if the address from which this participant's RTP packets originate has already been set.

- `bool IsRTCPAddressSet() const`
  Returns `true` if the address from which this participant's RTCP packets originate has already been set.

- `const RTPAddress *GetRTPDataAddress() const`
  Returns the address from which this participant's RTP packet originate. If the address has been set and the returned value is NULL, this indicates that it originated from the local participant.

- `const RTPAddress *GetRTCPDataAddress() const`
  Returns the address from which this participant's RTCP packet originate. If the address has been set and the returned value is NULL, this indicates that it originated from the local participant.

- `bool ReceivedBYE() const`
  Returns `true` if we received a BYE message for this participant and `false` otherwise.

- `u_int8_t *GetBYEReason(size_t *len) const`
  Returns the reason for leaving contained in the BYE packet of this participant. The length of the reason is stored in `len`.

- `RTPTime GetBYETime() const`
  Returns the time at which the BYE packet was received.

- `void SetTimestampUnit(double tsu)`
  Sets the value for the timestamp unit to be used in jitter calculations for data received from this participant. If not set, the library uses an approximation for the timestamp unit which is calculated from two consecutive RTCP sender reports. The timestamp unit is defined as a time interval divided by the number of samples in that interval: for $8000Hz$ audio this would be 1.0/8000.0.

- `double GetTimestampUnit() const`
  Returns the timestamp unit used for this participant.

- `bool SR_HasInfo() const`
  Returns `true` if an RTCP sender report has been received from this participant.

- `RTPNTPTime SR_GetNTPTimestamp() const`
  Returns the NTP timestamp contained in the last sender report.

- `u_int32_t SR_GetRTPTimestamp() const`
  Returns the RTP timestamp contained in the last sender report.

- `u_int32_t SR_GetPacketCount() const`
  Returns the packet count contained in the last sender report.

- `u_int32_t SR_GetByteCount() const`
  Returns the octet count contained in the last sender report.

- `RTPTime SR_GetReceiveTime() const`
  Returns the time at which the last sender report was received.

- `bool SR_Prev_HasInfo() const`
  Returns `true` if more than one RTCP sender report has been received.

- `RTPNTPTime SR_Prev_GetNTPTimestamp() const`
  Returns the NTP timestamp contained in the second to last sender report.

- `u_int32_t SR_Prev_GetRTPTimestamp() const`
  Returns the RTP timestamp contained in the second to last sender report.
- `u_int32_t SR_Prev_GetPacketCount() const`
  Returns the packet count contained in the second to last sender report.
- `u_int32_t SR_Prev_GetByteCount() const`
  Returns the octet count contained in the second to last sender report.
- `RTPTime SR_Prev_GetReceiveTime() const`
  Returns the time at which the second to last sender report was received.
- `bool RR_HasInfo() const`
  Returns `true` if this participant sent a receiver report with information about the reception of our data.
- `double RR_GetFractionLost() const`
  Returns the fraction lost value from the last report.
- `int32_t RR_GetPacketsLost() const`
  Returns the number of lost packets contained in the last report.
- `u_int32_t RR_GetExtendedHighestSequenceNumber() const`
  Returns the extended highest sequence number contained in the last report.
- `u_int32_t RR_GetJitter() const`
  Returns the jitter value from the last report.
- `u_int32_t RR_GetLastSRTimestamp() const`
  Returns the LSR value from the last report.
- `u_int32_t RR_GetDelaySinceLastSR() const`
  Returns the DLSR value from the last report.
- `RTPTime RR_GetReceiveTime() const`
  Returns the time at which the last report was received.
- `bool RR_Prev_HasInfo() const`
  Returns `true` if this participant sent more than one receiver report with information about the reception of our data.
- `double RR_Prev_GetFractionLost() const`
  Returns the fraction lost value from the second to last report.
- `int32_t RR_Prev_GetPacketsLost() const`
  Returns the number of lost packets contained in the second to last report.
- `u_int32_t RR_Prev_GetExtendedHighestSequenceNumber() const`
  Returns the extended highest sequence number contained in the second to last report.

- `u_int32_t RR_Prev_GetJitter() const`
  Returns the jitter value from the second to last report.
- `u_int32_t RR_Prev_GetLastSRTimestamp() const`
  Returns the LSR value from the second to last report.
- `u_int32_t RR_Prev_GetDelaySinceLastSR() const`
  Returns the DLSR value from the second to last report.
- `RTPTime RR_Prev_GetReceiveTime() const`
  Returns the time at which the second to last report was received.
- `bool INF_HasSentData() const`
  Returns `true` if validated RTP packets have been received from this participant.
- `int32_t INF_GetNumPacketsReceived() const`
  Returns the total number of received packets from this participant.
- `u_int32_t INF_GetBaseSequenceNumber() const`
  Returns the base sequence number of this participant.
- `u_int32_t INF_GetExtendedHighestSequenceNumber() const`
  Returns the extended highest sequence number received from this participant.
- `u_int32_t INF_GetJitter() const`
  Returns the current jitter value for this participant.
- `RTPTime INF_GetLastMessageTime() const`
  Returns the time at which something was last heard from this member.
- `RTPTime INF_GetLastRTPPacketTime() const`
  Returns the time at which the last RTP packet was received.
- `double INF_GetEstimatedTimestampUnit() const`
  Returns the estimated timestamp unit. The estimate is made from two consecutive sender reports.
- `u_int32_t INF_GetNumPacketsReceivedInInterval() const`
  Returns the number of packets received since a new interval was started with `INF_StartNewInterval`.
- `u_int32_t INF_GetSavedExtendedSequenceNumber() const`
  Returns the extended sequence number which was stored by the `INF_StartNewInterval` call.
- `void INF_StartNewInterval()`
  Starts a new interval to count received packets in. This also stores the current extended highest sequence number to be able to calculate the packet loss during the interval.

- `RTPTime INF_GetRoundtripTime() const`
  Estimates the round trip time by using the LSR and DLSR info from the last receiver report.

- `RTPTime INF_GetLastSDESNoteTime() const`
  Returns the time at which the last SDES NOTE item was received.

- `u_int8_t *SDES_GetCNAME(size_t *len) const`
  Returns a pointer to the SDES CNAME item of this participant and stores its length in `len`.

- `u_int8_t *SDES_GetName(size_t *len) const`
  Returns a pointer to the SDES name item of this participant and stores its length in `len`.

- `u_int8_t *SDES_GetEMail(size_t *len) const`
  Returns a pointer to the SDES e-mail item of this participant and stores its length in `len`.

- `u_int8_t *SDES_GetPhone(size_t *len) const`
  Returns a pointer to the SDES phone item of this participant and stores its length in `len`.

- `u_int8_t *SDES_GetLocation(size_t *len) const`
  Returns a pointer to the SDES location item of this participant and stores its length in `len`.

- `u_int8_t *SDES_GetTool(size_t *len) const`
  Returns a pointer to the SDES tool item of this participant and stores its length in `len`.

- `u_int8_t *SDES_GetNote(size_t *len) const`
  Returns a pointer to the SDES note item of this participant and stores its length in `len`.

If SDES private item support was enabled at compile-time, the following member functions are also available:

- `void SDES_GotoFirstPrivateValue()`
  Starts the iteration over the stored SDES private item prefixes and their associated values.

- `bool SDES_GetNextPrivateValue(u_int8_t **prefix, size_t *prefixlen, u_int8_t **value, size_t *valuelen)`
  If available, returns `true` and stores the next SDES private item prefix in `prefix` and its length in `prefixlen`. The associated value and its length are then stored in `value` and `valuelen`. Otherwise, it returns `false`.

- `bool SDES_GetPrivateValue(u_int8_t *prefix, size_t prefixlen, u_int8_t **value, size_t *valuelen) const`
  Looks for the entry which corresponds to the SDES private item prefix `prefix` with length `prefixlen`. If found, the function returns `true` and stores the associated value and its length in `value` and `valuelen` respectively.

3.2.16 `RTPPacketBuilder`

Header: `rtppacketbuilder.h`

This class can be used to build RTP packets and is a bit more high-level than the `RTPPacket` class: it generates an SSRC identifier, keeps track of timestamp and sequence number etc.

The interface is the following:

- `int Init(size_t maxpacksize)`
  Initializes the builder to only allow packets with a size below `maxpacksize`
- `void Destroy()`
  Cleans up the builder.
- `u_int32_t GetPacketCount()`
  Returns the number of packets which have been created with the current SSRC identifier.
- `u_int32_t GetPayloadOctetCount()`
  Returns the number of payload octets which have been generated with this SSRC identifier.
- `int SetMaximumPacketSize(size_t maxpacksize)`
  Sets the maximum allowed packet size to `maxpacksize`.
- `int AddCSRC(u_int32_t csrc)`
  Adds a CSRC to the CSRC list which will be stored in the RTP packets.
- `int DeleteCSRC(u_int32_t csrc)`
  Deletes a CSRC from the list which will be stored in the RTP packets.
- `void ClearCSRCList()`
  Clears the CSRC list.
- `int BuildPacket(const void *data, size_t len)`
  Builds a packet with payload `data` and payload length `len`. The payload type, marker and timestamp increment used will be those that have been set using the `SetDefault` functions below.
- `int BuildPacket(const void *data, size_t len, u_int8_t pt, bool mark, u_int32_t timestampinc)`

Builds a packet with payload `data` and payload length `len`. The payload type will be set to `pt`, the marker bit to `mark` and after building this packet, the timestamp will be incremented with `timestamp`.

- `int BuildPacketEx(const void *data, size_t len, u_int16_t hdrextID, const void *hdrextdata, size_t numhdrextwords)`
  Builds a packet with payload `data` and payload length `len`. The payload type, marker and timestamp increment used will be those that have been set using the `SetDefault` functions below. This packet will also contain an RTP header extension with identifier `hdrextID` and data `hdrextdata`. The length of the header extension data is given by `numhdrextwords` which expresses the length in a number of 32-bit words.

- `int BuildPacketEx(const void *data, size_t len, u_int8_t pt, bool mark, u_int32_t timestampinc, u_int16_t hdrextID, const void *hdrextdata, size_t numhdrextwords)`
  Builds a packet with payload `data` and payload length `len`. The payload type will be set to `pt`, the marker bit to `mark` and after building this packet, the timestamp will be incremented with `timestamp`. This packet will also contain an RTP header extension with identifier `hdrextID` and data `hdrextdata`. The length of the header extension data is given by `numhdrextwords` which expresses the length in a number of 32-bit words.

- `u_int8_t *GetPacket()`
  Returns a pointer to the last built RTP packet data.

- `size_t GetPacketLength()`
  Returns the size of the last built RTP packet.

- `int SetDefaultPayloadType(u_int8_t pt)`
  Sets the default payload type to `pt`.

- `int SetDefaultMark(bool m)`
  Sets the default marker bit to `m`.

- `int SetDefaultTimestampIncrement(u_int32_t timestampinc)`
  Sets the default timestamp increment to `timestampinc`.

- `int IncrementTimestamp(u_int32_t inc)`
  This function increments the timestamp with the amount given by `inc`. This can be useful if, for example, a packet was not sent because it contained only silence. Then, this function should be called to increment the timestamp with the appropriate amount so that the next packets will still be played at the correct time at other hosts.

- `int IncrementTimestampDefault()`
  This function increments the timestamp with the amount given set by the `SetDefaultTimestampIncrement` member function. This can be useful if, for example, a packet was not sent because it contained only silence. Then,

this function should be called to increment the timestamp with the appropriate amount so that the next packets will still be played at the correct time at other hosts.

- `u_int32_t CreateNewSSRC()`
  Creates a new SSRC to be used in generated packets. This will also generate new timestamp and sequence number offsets.

- `u_int32_t CreateNewSSRC(RTPSources &sources)`
  Creates a new SSRC to be used in generated packets. This will also generate new timestamp and sequence number offsets. The source table `sources` is used to make sure that the chosen SSRC isn't used by another participant yet.

- `u_int32_t GetSSRC()`
  Returns the current SSRC identifier.

- `u_int32_t GetTimestamp()`
  Returns the current RTP timestamp.

- `u_int16_t GetSequenceNumber()`
  Returns the current sequence number.

- `RTPTime GetPacketTime()`
  Returns the time at which a packet was generated. This is not necessarily the time at which the last RTP packet was generated: if the timestamp increment was zero, the time is not updated.

- `u_int32_t GetPacketTimestamp()`
  Returns the RTP timestamp which corresponds to the time returned by the previous function.

3.2.17 `RTCPPacketBuilder`

Header: `rtcppacketbuilder.h`

The class `RTCPPacketBuilder` can be used to build RTCP compound packets. This class is more high-level than the `RTCPCompoundPacketBuilder` class: it uses the information of an `RTPPacketBuilder` instance and of a `RTPSources` instance to automatically generate the next compound packet which should be sent. It also provides functions to determine when SDES items other than the CNAME item should be sent.

Its class interface is the following:

- `RTCPPacketBuilder(RTPSources &sources,RTPPacketBuilder &rtppackbuilder)`
  Creates an instance which will use the source table `sources` and the RTP packet builder `rtppackbuilder` to determine the information for the next RTCP compound packet.

- `int Init(size_t maxpacksize, double timestampunit, const void *cname, size_t cnamelen)`
  Initializes the builder to use the maximum allowed packet size `maxpacksize`, timestamp unit `timestampunit` and the SDES CNAME item specified by `cname` with length `cnamelen`. The timestamp unit is defined as a time interval divided by the number of samples in that interval: for $8000Hz$ audio this would be 1.0/8000.0.

- `void Destroy()`
  Cleans up the builder.

- `int SetTimestampUnit(double tsunit)`
  Sets the timestamp unit to be used to `tsunit`. The timestamp unit is defined as a time interval divided by the number of samples in that interval: for $8000Hz$ audio this would be 1.0/8000.0.

- `int SetMaximumPacketSize(size_t maxpacksize)`
  Sets the maximum size allowed size of an RTCP compound packet to `maxpacksize`.

- `int SetPreTransmissionDelay(const RTPTime &delay)`
  This function allows you to inform RTCP packet builder about the delay between sampling the first sample of a packet and sending the packet. This delay is taken into account when calculating the relation between RTP timestamp and wallclock time, used for inter-media synchronization.

- `int BuildNextPacket(RTCPCompoundPacket **pack)`
  Builds the next RTCP compound packet which should be sent and stores it in `pack`.

- `int BuildBYEPacket(RTCPCompoundPacket **pack, const void *reason, size_t reasonlength,bool useSRifpossible = true)`
  Builds a BYE packet with reason for leaving specified by `reason` and length `reasonlength`. If `useSRifpossible` is set to `true`, the RTCP compound packet will start with a sender report if allowed. Otherwise, a receiver report is used.

- `void SetNameInterval(int count)`
  After all possible sources in the source table have been processed, the class will check if other SDES items need to be sent. If `count` is zero or negative, nothing will happen. If `count` is positive, an SDES name item will be added after the sources in the source table have been processed `count` times.

- `void SetEMailInterval(int count)`
  After all possible sources in the source table have been processed, the class will check if other SDES items need to be sent. If `count` is zero or negative, nothing will happen. If `count` is positive, an SDES e-mail item will be added after the sources in the source table have been processed `count` times.

- `void SetLocationInterval(int count)`
  After all possible sources in the source table have been processed, the class will check if other SDES items need to be sent. If `count` is zero or negative, nothing will happen. If `count` is positive, an SDES location item will be added after the sources in the source table have been processed `count` times.

- `void SetPhoneInterval(int count)`
  After all possible sources in the source table have been processed, the class will check if other SDES items need to be sent. If `count` is zero or negative, nothing will happen. If `count` is positive, an SDES phone item will be added after the sources in the source table have been processed `count` times.

- `void SetToolInterval(int count)`
  After all possible sources in the source table have been processed, the class will check if other SDES items need to be sent. If `count` is zero or negative, nothing will happen. If `count` is positive, an SDES tool item will be added after the sources in the source table have been processed `count` times.

- `void SetNoteInterval(int count)`
  After all possible sources in the source table have been processed, the class will check if other SDES items need to be sent. If `count` is zero or negative, nothing will happen. If `count` is positive, an SDES note item will be added after the sources in the source table have been processed `count` times.

- `int SetLocalName(const void *s, size_t len)`
  Sets the SDES name item for the local participant to the value `s` with length `len`.

- `int SetLocalEMail(const void *s, size_t len)`
  Sets the SDES e-mail item for the local participant to the value `s` with length `len`.

- `int SetLocalLocation(const void *s, size_t len)`
  Sets the SDES location item for the local participant to the value `s` with length `len`.

- `int SetLocalPhone(const void *s, size_t len)`
  Sets the SDES phone item for the local participant to the value `s` with length `len`.

- `int SetLocalTool(const void *s, size_t len)`
  Sets the SDES tool item for the local participant to the value `s` with length `len`.

- `int SetLocalNote(const void *s, size_t len)`
  Sets the SDES note item for the local participant to the value `s` with length `len`.

3.2.18 RTPCollisionList

Header: rtpcollisionlist.h

This class represents a list of addresses from which SSRC collisions were detected. Its interface is this:

- `void Clear()`
  Clears the list of addresses.
- `int UpdateAddress(const RTPAddress *addr, const RTPTime &receivetime, bool *created)`
  Updates the entry for address `addr` to indicate that a collision was detected at time `receivetime`. If the entry did not exist yet, the flag `created` is set to `true`, otherwise it is set to `false`.
- `bool HasAddress(const RTPAddress *addr) const`
  Returns `true` if the address `addr` appears in the list.
- `void Timeout(const RTPTime &currenttime, const RTPTime &timeoutdelay)`
  Assuming that the current time is given by `currenttime`, this function times out entries which haven't been updated in the previous time interval specified by `timeoutdelay`.

3.2.19 RTCPScheduler

Header: rtcpscheduler.h

This class determines when RTCP compound packets should be sent. It has the following interface:

- `RTCPScheduler(RTPSources &sources)`
  Creates an instance which will use the source table `RTPSources` to determine when RTCP compound packets should be scheduled. Note that for correct operation the `sources` instance should have information about the own SSRC (added by `CreateOwnSSRC`).
- `void Reset()`
  Resets the scheduler.
- `void SetParameters(const RTCPSchedulerParams &params)`
  Sets the scheduler parameters to be used to `params`. The `RTCPSchedulerParams` class is described below.
- `RTCPSchedulerParams GetParameters() const`
  Returns the currently used scheduler parameters.
- `void SetHeaderOverhead(size_t numbytes)`
  Sets the header overhead from underlying protocols (for example UDP and IP) to `numbytes`.

- `size_t GetHeaderOverhead() const`
  Returns the currently used header overhead.

- `void AnalyseIncoming(RTCPCompoundPacket &rtcpcomppack)`
  For each incoming RTCP compound packet, this function has to be called for the scheduler to work correctly.

- `void AnalyseOutgoing(RTCPCompoundPacket &rtcpcomppack)`
  For each outgoing RTCP compound packet, this function has to be called for the scheduler to work correctly.

- `void ActiveMemberDecrease()`
  This function has to be called each time a member times out or sends a BYE packet.

- `void ScheduleBYEPacket(size_t packetsize)`
  Asks the scheduler to schedule an RTCP compound packet containing a BYE packet. The compound packet has size `packetsize`.

- `RTPTime GetTransmissionDelay()`
  Returns the delay after which an RTCP compound will possibly have to be sent. The `IsTime` member function should be called afterwards to make sure that it actually is time to send an RTCP compound packet.

- `bool IsTime()`
  This function returns `true` if it's time to send an RTCP compound packet and `false` otherwise. If the function returns `true` it will also have calculated the next time at which a packet should be sent, so if it is called again right away, it will return `false`.

- `RTPTime CalculateDeterministicInterval(bool sender = false)`
  Calculates the deterministic interval at this time. This is used – together with a certain multiplier – to time out members, senders etc.

Scheduler parameters

Header: `rtcpscheduler.h`

The `RTCPSchedulerParams` class describes the parameters to be used by the scheduler. Its interface is the following:

- `int SetRTCPBandwidth(double bw)`
  Sets the RTCP bandwidth to be used to `bw` (in bytes per second).

- `double GetRTCPBandwidth() const`
  Returns the used RTCP bandwidth in bytes per second.

- `int SetSenderBandwidthFraction(double fraction)`
  Sets the fraction of the RTCP bandwidth reserved for senders to `fraction`.

- `double GetSenderBandwidthFraction() const`
  Returns the fraction of the RTCP bandwidth reserved for senders.
- `int SetMinimumTransmissionInterval(const RTPTime &t)`
  Sets the minimum (deterministic) interval between RTCP compound packets to `t`.
- `RTPTime GetMinimumTransmissionInterval() const`
  Returns the minimum RTCP transmission interval.
- `void SetUseHalfAtStartup(bool usehalf)`
  If `usehalf` is `true`, only use half the minimum interval before sending the first RTCP compound packet.
- `bool GetUseHalfAtStartup() const`
  Returns if only half the minimum interval should be used before sending the first RTCP compound packet.
- `void SetRequestImmediateBYE(bool v)`
  If `v` is `true`, the scheduler will schedule a BYE packet to be sent immediately if allowed.
- `bool GetRequestImmediateBYE()`
  Returns if the scheduler will schedule a BYE packet to be sent immediately if allowed.

The parameters default to the following values:

- RTCP bandwidth: 1000 bytes per second
- Sender bandwidth fraction: 25%
- Minimum interval: 5 seconds
- Use half minimum interval at startup: yes
- Send BYE packet immediately if possible: yes

3.2.20 `RTPSessionParams`

Header: `rtpsessionparams.h`

Describes the parameters for to be used by an `RTPSession` instance. Note that the own timestamp unit must be set to a valid number, otherwise the session can't be created. The interface of this class is the following:

- `int SetUsePollThread(bool usethread)`
  If `usethread` is `true`, the session will use a poll thread to automatically process incoming data and to send RTCP packets when necessary.

- `bool IsUsingPollThread() const`
  Returns whether the session should use a poll thread or not.

- `void SetMaximumPacketSize(size_t max)`
  Sets the maximum allowed packet size for the session.

- `size_t GetMaximumPacketSize() const`
  Returns the maximum allowed packet size.

- `void SetAcceptOwnPackets(bool accept)`
  If the argument is `true`, the session should accept its own packets and store them accordingly in the source table.

- `bool AcceptOwnPackets() const`
  Returns `true` if the session should accept its own packets.

- `void SetReceiveMode(RTPTransmitter::ReceiveMode recvmode)`
  Sets the receive mode to be used by the session.

- `RTPTransmitter::ReceiveMode GetReceiveMode() const`
  Returns the currently set receive mode.

- `void SetOwnTimestampUnit(double tsunit)`
  Sets the timestamp unit for our own data. The timestamp unit is defined as a time interval in seconds divided by the number of samples in that interval. For example, for $8000 Hz audio$, the timestamp unit would be 1.0/8000.0. Since this value is initially set to an illegal value, the user must set this to an allowed value to be able to create a session.

- `double GetOwnTimestampUnit() const`
  Returns the currently set timestamp unit.

- `void SetResolveLocalHostname(bool v)`
  If `v` is set to `true`, the session will ask the transmitter to find a host name based upon the IP addresses in its list of local IP addresses. If set to `false`, a call to `gethostname` or something similar will be used to find the local hostname. Note that the first method might take some time.

- `bool GetResolveLocalHostname() const`
  Returns whether the local hostname should be determined from the transmitter's list of local IP addresses or not.

- `void SetProbationType(RTPSources::ProbationType probtype)`
  If probation support is enabled, this function sets the probation type to be used.

- `RTPSources::ProbationType GetProbationType() const`
  Returns the probation type which will be used.

- `void SetSessionBandwidth(double sessbw)`
  Sets the session bandwidth in bytes per second.

- `double GetSessionBandwidth() const`
  Returns the session bandwidth in bytes per second.

- `void SetControlTrafficFraction(double frac)`
  Sets the fraction of the session bandwidth to be used for control traffic.

- `double GetControlTrafficFraction() const`
  Returns the fraction of the session bandwidth that will be used for control traffic.

- `void SetSenderControlBandwidthFraction(double frac)`
  Sets the minimum fraction of the control traffic that will be used by senders.

- `double GetSenderControlBandwidthFraction() const`
  Returns the minimum fraction of the control traffic that will be used by senders.

- `void SetMinimumRTCPTransmissionInterval(const RTPTime &t)`
  Set the minimal time interval between sending RTCP packets.

- `RTPTime GetMinimumRTCPTransmissionInterval() const`
  Returns the minimal time interval between sending RTCP packets.

- `void SetUseHalfRTCPIntervalAtStartup(bool usehalf)`
  If `usehalf` is set to `true`, the session will only wait half of the calculated RTCP interval before sending its first RTCP packet.

- `bool GetUseHalfRTCPIntervalAtStartup() const`
  Returns whether the session will only wait half of the calculated RTCP interval before sending its first RTCP packet or not.

- `void SetRequestImmediateBYE(bool v)`
  If `v` is `true`, the session will send a BYE packet immediately if this is allowed.

- `bool GetRequestImmediateBYE() const`
  Returns whether the session should send a BYE packet immediately (if allowed) or not.

- `void SetSenderReportForBYE(bool v)`
  When sending a BYE packet, this indicates whether it will be part of an RTCP compound packet that begins with a sender report or a receiver report. Of course, a sender report will only be used if allowed.

- `bool GetSenderReportForBYE() const`
  Returns `true` if a BYE packet will be sent in an RTCP compound packet which starts with a sender report. If a receiver report will be used, the function returns `false`.

- `void SetSenderTimeoutMultiplier(double m)`
  Sets the multiplier to be used when timing out senders.

- `double GetSenderTimeoutMultiplier() const`
  Returns the multiplier to be used when timing out senders.
- `void SetSourceTimeoutMultiplier(double m)`
  Sets the multiplier to be used when timing out members.
- `double GetSourceTimeoutMultiplier() const`
  Returns the multiplier to be used when timing out members.
- `void SetBYETimeoutMultiplier(double m)`
  Sets the multiplier to be used when timing out a member after it has sent a BYE packet.
- `double GetBYETimeoutMultiplier() const`
  Returns the multiplier to be used when timing out a member after it has sent a BYE packet.
- `void SetCollisionTimeoutMultiplier(double m)`
  Sets the multiplier to be used when timing out entries in the collision table.
- `double GetCollisionTimeoutMultiplier() const`
  Returns the multiplier to be used when timing out entries in the collision table.
- `void SetNoteTimeoutMultiplier(double m)`
  Sets the multiplier to be used when timing out SDES NOTE information.
- `double GetNoteTimeoutMultiplier() const`
  Returns the multiplier to be used when timing out SDES NOTE information.

The default values for the parameters are the following:

- Use poll thread: yes
- Maximum packet size: 1400 bytes
- Accept own packets: no
- Receive mode: accept all packets
- Resolve local hostname: no
- Probation type: `ProbationStore`
- Session bandwidth: 10000 bytes per second
- Control traffic fraction: 5%
- Sender's fraction of control traffic: 25%

- Minimum RTCP interval: 5 seconds
- Use half minimum interval at startup: yes
- Send BYE packet immediately if allowed: yes
- Use sender report for BYE packet: yes
- Sender timeout multiplier: 2
- Member timeout multiplier: 5
- Timeout multiplier after BYE packet: 1
- Timeout multiplier for entries in collision table: 10
- Timeout multiplier for SDES NOTE items: 25

3.2.21 `RTPSession`

Header: `rtpsession.h`

For most RTP based applications, the `RTPSession` class will probably be the one to use. It handles the RTCP part completely internally, so the user can focus on sending and receiving the actual data.

Note that the `RTPSession` class is not meant to be thread safe. The user should use some kind of locking mechanism to prevent different threads from using the same `RTPSession` instance.

The `RTPSession` class interface is the following:

- `RTPSession(RTPTransmitter::TransmissionProtocol proto = RTPTransmitter::IPv4UDPProto)`
  Creates an instance which will use the transmission component given by `proto`. When `proto` indicates the a user-defined transmitter, the `NewUserDefinedTransmitter()` member function should be implemented.
- `int Create(const RTPSessionParams &sessparams, const RTPTransmissionParams *transparams = 0)`
  Creates the session with parameters `sessparams` and transmission parameters `transparams`. If `transparams` is NULL, the defaults for the requested transmitter will be used.
- `void Destroy()`
  Leaves the session without sending a BYE packet.
- `void BYEDestroy(const RTPTime &maxwaittime, const void *reason, size_t reasonlength)`
  Sends a BYE packet and leaves the session. At most a time `maxwaittime` will be waited to send the BYE packet. If this time expires, the session

will be left without sending a BYE packet. The BYE packet will contain as reason for leaving `reason` with length `reasonlength`.

- `bool IsActive()`
  Returns whether the session has been created or not.

- `u_int32_t GetLocalSSRC()`
  Returns our own SSRC.

- `int AddDestination(const RTPAddress &addr)`
  Adds `addr` to the list of destinations.

- `int DeleteDestination(const RTPAddress &addr)`
  Deletes `addr` from the list of destinations.

- `void ClearDestinations()`
  Clears the list of destination.

- `bool SupportsMulticasting()`
  Returns `true` if multicasting is supported.

- `int JoinMulticastGroup(const RTPAddress &addr)`
  Joins the multicast group specified by `addr`.

- `int LeaveMulticastGroup(const RTPAddress &addr)`
  Leaves the multicast group specified by `addr`.

- `void LeaveAllMulticastGroups()`
  Leaves all multicast groups.

- `int SendPacket(const void *data, size_t len)`
  Sends the RTP packet with payload `data` which has length `len`. The used payload type, marker and timestamp increment will be those that have been set using the `SetDefault` member functions.

- `int SendPacket(const void *data, size_t len, u_int8_t pt, bool mark, u_int32_t timestampinc)`
  Sends the RTP packet with payload `data` which has length `len`. It will use payload type `pt`, marker `mark` and after the packet has been built, the timestamp will be incremented by `timestampinc`.

- `int SendPacketEx(const void *data, size_t len, u_int16_t hdrextID, const void *hdrextdata, size_t numhdrextwords)`
  Sends the RTP packet with payload `data` which has length `len`. The packet will contain a header extension with identifier `hdrextID` and containing data `hdrextdata`. The length of this data is given by `numhdrextwords` and is specified in a number of 32-bit words. The used payload type, marker and timestamp increment will be those that have been set using the `SetDefault` member functions.

- `int SendPacketEx(const void *data, size_t len, u_int8_t pt, bool mark, u_int32_t timestampinc, u_int16_t hdrextID, const void *hdrextdata, size_t numhdrextwords)`
  Sends the RTP packet with payload `data` which has length `len`. It will use payload type `pt`, marker `mark` and after the packet has been built, the timestamp will be incremented by `timestampinc`. The packet will contain a header extension with identifier `hdrextID` and containing data `hdrextdata`. The length of this data is given by `numhdrextwords` and is specified in a number of 32-bit words.

- `int SetDefaultPayloadType(u_int8_t pt)`
  Sets the default payload type for RTP packets to `pt`.

- `int SetDefaultMark(bool m)`
  Sets the default marker for RTP packets to `m`.

- `int SetDefaultTimestampIncrement(u_int32_t timestampinc)`
  Sets the default value to increment the timestamp with to `timestampinc`.

- `int IncrementTimestamp(u_int32_t inc)`
  This function increments the timestamp with the amount given by `inc`. This can be useful if, for example, a packet was not sent because it contained only silence. Then, this function should be called to increment the timestamp with the appropriate amount so that the next packets will still be played at the correct time at other hosts.

- `int IncrementTimestampDefault()`
  This function increments the timestamp with the amount given set by the `SetDefaultTimestampIncrement` member function. This can be useful if, for example, a packet was not sent because it contained only silence. Then, this function should be called to increment the timestamp with the appropriate amount so that the next packets will still be played at the correct time at other hosts.

- `int SetPreTransmissionDelay(const RTPTime &delay)`
  This function allows you to inform the library about the delay between sampling the first sample of a packet and sending the packet. This delay is taken into account when calculating the relation between RTP timestamp and wallclock time, used for inter-media synchronization.

- `RTPTransmissionInfo *GetTransmissionInfo()`
  This function returns an instance of a subclass of `RTPTransmissionInfo` which will give some additional information about the transmitter (a list of local IP addresses for example). The user has to delete the returned instance when it is no longer needed.

- `int Poll()`
  If you're not using the poll thread, this function must be called regularly to process incoming data and to send RTCP data when necessary.

- `int WaitForIncomingData(const RTPTime &delay,bool *dataavailable = 0)`
  Waits at most a time `delay` until incoming data has been detected. Only works when you're not using the poll thread. If `dataavailable` is not `NULL`, it should be set to `true` if data was actually read and to `false` otherwise.

- `int AbortWait()`
  If the previous function has been called, this one aborts the waiting. Only works when you're not using the poll thread.

- `RTPTime GetRTCPDelay()`
  Returns the time interval after which an RTCP compound packet may have to be sent. Only works when you're not using the poll thread.

- `int BeginDataAccess()`
  The following member functions (till `EndDataAccess`) need to be accessed between a call to `BeginDataAccess` and `EndDataAccess`. The `BeginDataAccess` makes sure that the poll thread won't access the source table at the same time. When the `EndDataAccess` is called, the lock on the source table is freed again.

- `bool GotoFirstSource()`
  Starts the iteration over the participants by going to the first member in the table. If a member was found, the function returns `true`, otherwise it returns `false`.

- `bool GotoNextSource()`
  Sets the current source to be the next source in the table. If we're already at the last source, the function returns `false`, otherwise it returns `true`.

- `bool GotoPreviousSource()`
  Sets the current source to be the previous source in the table. If we're at the first source, the function returns `false`, otherwise it returns `true`.

- `bool GotoFirstSourceWithData()`
  Sets the current source to be the first source in the table which has `RTPPacket` instances that we haven't extracted yet. If no such member was found, the function returns `false`, otherwise it returns `true`.

- `bool GotoNextSourceWithData()`
  Sets the current source to be the next source in the table which has `RTPPacket` instances that we haven't extracted yet. If no such member was found, the function returns `false`, otherwise it returns `true`.

- `bool GotoPreviousSourceWithData()`
  Sets the current source to be the previous source in the table which has `RTPPacket` instances that we haven't extracted yet. If no such member was found, the function returns `false`, otherwise it returns `true`.

- `RTPSourceData *GetCurrentSourceInfo()`
  Returns the `RTPSourceData` instance for the currently selected participant.

- `RTPSourceData *GetSourceInfo(u_int32_t ssrc)`
  Returns the `RTPSourceData` instance for the participant identified by `ssrc`, or NULL if no such entry exists.

- `RTPPacket *GetNextPacket()`
  Extracts the next packet from the received packets queue of the current participant.

- `int EndDataAccess()`
  See `BeginDataAccess`.

- `int SetReceiveMode(RTPTransmitter::ReceiveMode m)`
  Sets the receive mode to `m`, which can be one of the following:

  - `RTPTransmitter::AcceptAll`
    All incoming data is accepted, no matter where it originated from.
  - `RTPTransmitter::AcceptSome`
    Only data coming from specific sources will be accepted.
  - `RTPTransmitter::IgnoreSome`
    All incoming data is accepted, except for data coming from a specific set of sources.

  Note that when the receive mode is changed, the list of addressed to be ignored or accepted will be cleared.

- `int AddToIgnoreList(const RTPAddress &addr)`
  Adds `addr` to the list of addresses to ignore.

- `int DeleteFromIgnoreList(const RTPAddress &addr)`
  Deletes `addr` from the list of addresses to ignore.

- `void ClearIgnoreList()`
  Clears the list of addresses to ignore.

- `int AddToAcceptList(const RTPAddress &addr)`
  Adds `addr` to the list of addresses to accept.

- `int DeleteFromAcceptList(const RTPAddress &addr)`
  Deletes `addr` from the list of addresses to accept.

- `void ClearAcceptList()`
  Clears the list of addresses to accept.

- `int SetMaximumPacketSize(size_t s)`
  Sets the maximum allowed packet size to `s`.

- `int SetSessionBandwidth(double bw)`
  Sets the session bandwidth to `bw`, which is specified in bytes per second.

- `int SetTimestampUnit(double u)`
  Sets our own timestamp unit to `u`. The timestamp unit is defined as a time interval divided by the number of samples in that interval: for $8000Hz$ audio this would be 1.0/8000.0.

- `void SetNameInterval(int count)`
  After all possible sources in the source table have been processed, the RTCP packet builder will check if other (non-CNAME) SDES items need to be sent. If `count` is zero or negative, nothing will happen. If `count` is positive, an SDES name item will be added after the sources in the source table have been processed `count` times.

- `void SetEMailInterval(int count)`
  After all possible sources in the source table have been processed, the RTCP packet builder will check if other (non-CNAME) SDES items need to be sent. If `count` is zero or negative, nothing will happen. If `count` is positive, an SDES e-mail item will be added after the sources in the source table have been processed `count` times.

- `void SetLocationInterval(int count)`
  After all possible sources in the source table have been processed, the RTCP packet builder will check if other (non-CNAME) SDES items need to be sent. If `count` is zero or negative, nothing will happen. If `count` is positive, an SDES location item will be added after the sources in the source table have been processed `count` times.

- `void SetPhoneInterval(int count)`
  After all possible sources in the source table have been processed, the RTCP packet builder will check if other (non-CNAME) SDES items need to be sent. If `count` is zero or negative, nothing will happen. If `count` is positive, an SDES phone item will be added after the sources in the source table have been processed `count` times.

- `void SetToolInterval(int count)`
  After all possible sources in the source table have been processed, the RTCP packet builder will check if other (non-CNAME) SDES items need to be sent. If `count` is zero or negative, nothing will happen. If `count` is positive, an SDES tool item will be added after the sources in the source table have been processed `count` times.

- `void SetNoteInterval(int count)`
  After all possible sources in the source table have been processed, the RTCP packet builder will check if other (non-CNAME) SDES items need to be sent. If `count` is zero or negative, nothing will happen. If `count` is positive, an SDES note item will be added after the sources in the source table have been processed `count` times.

- `int SetLocalName(const void *s, size_t len)`
  Sets the SDES name item for the local participant to the value `s` with length `len`.

- `int SetLocalEMail(const void *s, size_t len)`
  Sets the SDES e-mail item for the local participant to the value `s` with length `len`.

- `int SetLocalLocation(const void *s, size_t len)`
  Sets the SDES location item for the local participant to the value `s` with length `len`.

- `int SetLocalPhone(const void *s, size_t len)`
  Sets the SDES phone item for the local participant to the value `s` with length `len`.

- `int SetLocalTool(const void *s, size_t len)`
  Sets the SDES tool item for the local participant to the value `s` with length `len`.

- `int SetLocalNote(const void *s, size_t len)`
  Sets the SDES note item for the local participant to the value `s` with length `len`.

In case you specified in the constructor that you want to use your own transmission component, you should override the following function:

- `RTPTransmitter *NewUserDefinedTransmitter()`

The `RTPTransmitter` instance returned by this function will then be used to send and receive RTP and RTCP packets. Note that when the session is destroyed, this `RTPTransmitter` instance will be destroyed with a `delete` call.

By inheriting your own class from `RTPSession` and overriding one or more of the functions below, certain events can be detected:

- `void OnRTPPacket(RTPPacket *pack, const RTPTime &receivetime, const RTPAddress *senderaddress)`
  Is called when an RTP packet is about to be processed.

- `void OnRTCPCompoundPacket(RTCPCompoundPacket *pack, const RTPTime &receivetime, const RTPAddress *senderaddress)`
  Is called when an RTCP packet is about to be processed.

- `void OnSSRCCollision(RTPSourceData *srcdat, const RTPAddress *senderaddress, bool isrtp)`
  Is called when an SSRC collision was detected. The instance `srcdat` is the

one present in the table, the address `senderaddress` is the one that collided with one of the addresses and `isrtp` indicates against which address of `srcdat` the check failed.

- `void OnCNAMECollision(RTPSourceData *srcdat, const RTPAddress *senderaddress, const u_int8_t *cname, size_t cnamelength)`
  Is called when another CNAME was received than the one already present for source `srcdat`.

- `void OnNewSource(RTPSourceData *srcdat)`
  Is called when a new entry `srcdat` is added to the source table.

- `void OnRemoveSource(RTPSourceData *srcdat)`
  Is called when the entry `srcdat` is about to be deleted from the source table.

- `void OnTimeout(RTPSourceData *srcdat)`
  Is called when participant `srcdat` is timed out.

- `void OnBYETimeout(RTPSourceData *srcdat)`
  Is called when participant `srcdat` is timed after having sent a BYE packet.

- `void OnBYEPacket(RTPSourceData *srcdat)`
  Is called when a BYE packet has been processed for source `srcdat`.

- `void OnAPPPacket(RTCPAPPPacket *apppacket, const RTPTime &receivetime, const RTPAddress *senderaddress)`
  In called when an RTCP APP packet `apppacket` has been received at time `receivetime` from address `senderaddress`.

- `void OnUnknownPacketType(RTCPPacket *rtcppack, const RTPTime &receivetime, const RTPAddress *senderaddress)`
  Is called when an unknown RTCP packet type was detected.

- `void OnUnknownPacketFormat(RTCPPacket *rtcppack, const RTPTime &receivetime, const RTPAddress *senderaddress)`
  Is called when an unknown packet format for a known packet type was detected.

- `void OnNoteTimeout(RTPSourceData *srcdat)`
  Is called when the SDES NOTE item for source `srcdat` has been timed out.

- `void OnPollThreadError(int errcode)`
  Is called when error `errcode` was detected in the poll thread.

- `void OnPollThreadStep()`
  Is called each time the poll thread loops. This happens when incoming data was detected or when its time to send an RTCP compound packet.

4 Contact

If you have any questions, remarks or requests about the library or if you think you've discovered a bug, you can contact me at:

`jori@lumumba.uhasselt.be`

The home page of the library is:

`http://research.edm.uhasselt.be/jori/jrtplib/jrtplib.html`

There is also a mailing list for the library. To subscribe to the list, send an e-mail with the text `subscribe jrtplib` as the message body (not the subject) to `majordomo@edm.uhasselt.be` and you'll receive further instructions.