ORTP 库入门

(Version 1.04)

【摘要】：RTP（Real-time Transport Protocol）是用于 Internet 上针对多媒体数据流的一种传输协议，做流媒体传输方面的应用离不 开 RTP 协议的实现及使用，为了更加快速地在项目中应用 RTP 协议实现流媒体的传输，我们一般会选择使用一些 RTP 库，例如使用 c++语言编写的 JRTPLIB 库，网上关于 RTP 协议以及 JRTPLIB 库的介绍已经很多了，文本主要介绍实现了 RTP 协议的另一种开源库—— ORTP 库，这个库是纯使用 c 语言编写，由于我们的项目是基于 Linux 下的 c 语言编程，故我们选择了 ortp 作为我们的第三方库，在此我也对该库进行一 个简单地介绍，希望对其他 ortp 的初学者有所帮助。

【关键词】：ORTP Lib,RTP,H.264 ,NALU

目录

1. 简介

ORTP 是一个支持 RTP 以及 RFC3550 协议的库，有如下的特性：

（1）使用 C 语言编写，可以工作于 windows， Linux， 以及 Unix 平台

（2）实现了 RFC3550 协议，提供简单易用的 API。支持多种配置，RFC3551 为默认的配置。

（3）支持单线程下的多个 RTP 会话，支持自适应抖动处理。

（4）基于 GPL 版权声明。

ORTP 可以在其官方网站上（http://www。linphone。org/index。php/eng/code_review/ortp）下载， 下载解压后得到 ORTP 的源码包和示例程序（tests）。其帮助文档在 docs 目录下，也可以在 http://download。savannah。gnu。org/releases/linphone/ortp/docs/在线查看。

2. 编译安装

2.1 PC 下安装

进入主目录：cd ~/ortp-0。20。0，解压，执行如下命令：

- **。/configure**
- **make clean**
- **make all**
- **make install**

装好以后系统环境如下，静态动态库安装到了/usr/local/lib 目录下，包括 libortp。la、libortp。so libortp。so。8 libortp。so。8。0。0。头文件在/usr/local/include 目录 ortp 目录下。库文件复制到/usr/lib 文件夹下，头文件复制到/usr/include 下。

2.2 交叉编译

进入主目录：cd ~/ortp-0。20。0 解压执行如下命令：

只生成动态链接库（默认只生成动态链接库，不生成静态库）

- **。/configure --prefix=/root/dm8168_ortp --host= i686-pc-linux-gnu --target=arm-none-linux-gnueabi --disable-static CC=arm-none-linux-gnueabi-gcc**

只生成静态链接库

- **。/configure --prefix=/root/dm8168_ortp --host= i686-pc-linux-gnu --target=arm-none-linux-gnueabi --disable-static CC=arm-none-linux-gnueabi-gcc**

- **make clean**
- **make all**
- **make install**

/root/dm8168_ortp 是 ortp 的安装目录，为了能在 DM8168 平台上运行程序，需要把库文件(lib 文件夹里边的除文件夹 pkgconfig 以外的文件)和头文件拷贝到嵌入式文件系统中。分别拷贝到\usr\lib 和\usr\include 中。到此 ortp 的交叉编译环境移植成功。注意将 include 文件夹下的 ortp 文件夹整个拷贝到\usr\include 中，如果只拷贝里面的。h 文件，在程序编译时会报错。移植到开发板，配置参数时可以考虑加 perf，这样可以去掉一些功能，从而在程序运行时降低 CPU 的损耗。

2.3 demo 测试

1. Pc 端程序编译
   - **cd ~/ortp-0。20。0/src/tests**
   - **gcc rtprecv。c –o rtprecv –lortp**
2. ARM 端程序编译
   - **arm-none-linux-gnueabi-gcc rtpsend 。c -o rtpsend_arm -L/root/dm8168_ortp/lib -I/root/dm8168_ortp/include -lortp**
3. 测试

在 PC 端运行**。/rtprecv recvFile 5000** 在 ARM 端运行 **。/rtpsend_arm test 10。108。20。26 5000**，可完成从 ARM 端发送文件 test 到 PC 端。

3. 主要函数介绍

3.1 rtp_session_init

函数原型：void rtp_session_init (RtpSession * session， int mode)

函数功能：执行 rtp 会话的一些必要的初始化工作

参数含义：

session： rtp 会话结构体，含有一些 rtp 会话的基本信息

mode： 传输模式，有以下几种，决定本会话的一些特性。

RTP_SESSION_RECVONLY：只进行 rtp 数据的接收

RTP_SESSION_SENDONLY：只进行 rtp 数据的发送

RTP_SESSION_SENDRECV：可以进行 rtp 数据的接收和发送

执行的操作：

1。设置 rtp 包缓冲队列的最大长度

2。根据传输模式设置标志变量的值

3。随机产生 SSRC 和同步源描述信息

4。传入全局的 av_profile，即使用默认的 profile 配置

5。初始化 rtp 包缓冲区队列

6。发送负载类型默认设置为 0（pcmu 音频），接收负载类型默认设置为-1（未定义）

7。将 session 的其他成员的值均设置一个默认值。

代码实现：

```
void ortp_init()
{
	static bool_t initialized=FALSE;
	if (initialized) return;
	initialized=TRUE;

#ifdef WIN32
	win32_init_sockets();
#endif

	av_profile_init(&av_profile);
	ortp_global_stats_reset();//重置 rtp_stats 结构
	init_random_number_generator();
	ortp_message("oRTP-" ORTP_VERSION " initialized.");
}
```

主要涉及结构体：RtpProfile av_profile 在 payloadtype.h 中定义

```
struct _RtpProfile
{
	char *name;
	PayloadType *payload[RTP_PROFILE_MAX_PAYLOADS];
};
/**
 * The RTP profile is a table RTP_PROFILE_MAX_PAYLOADS entries to make the matching
```

* between RTP payload type number and the PayloadType that defines the type of

* media.

**/

/*这个结构是用于将 RTP 负载类型数字与具体的负载类型定义关联，例如，

rtp_profile_set_payload(&av_profile, 96, &payload_type_h264);

rtp_session_set_payload_type(session, 96);

就是将 96 与 h264 关联起来

*/

3.2 rtp_session_set_scheduling_mode

函数原型：void rtp_session_set_scheduling_mode (RtpSession * session， int yesno)

函数功能： RtpScheduler 管理多个 session 的调度和收发的控制，本函数设置是否使用该 session 调度管理功能。

参数含义：

session： rtp 会话结构体

yesno: 是否使用 rtp session 的系统调度功能

说明：

如果 yesno 为 1，则表明使用系统的 session 调度管理功能，意味着可以使用以下功能：

1． 可以使用 session_set_select 在多个 rtp 会话之间进行选择，根据时间戳判定某个会话是否到达了收发的时间。

2． 可以使用 rtp_session_set_blocking_mode()设置是否使用阻塞模式来进行 rtp 包的发送和接收。

如果 yesno 为 0，则表明该会话不受系统管理和调度。

关于 rtp session 的管理和调度，由全局的变量 RtpScheduler *__ortp_scheduler 来负责，该变量必须通过 ortp_scheduler_init() 来进行初始化操作。

3.3 rtp_session_set_blocking_mode

函数原型：void rtp_session_set_blocking_mode (RtpSession * session， int yesno)

函数功能：设置是否使用阻塞模式，

参数含义：

session： rtp 会话结构体

yesno: 是否使用阻塞模式

说明：

阻塞模式只有在 scheduling mode 被开启的情况下才能使用，本函数决定了 rtp_session_recv_with_ts() 和 rtp_session_send_with_ts()两个函数的行为，如果启用了阻塞模式，则 rtp_session_recv_with_ts()会 一直阻塞直到接收 RTP 包的时间点到达（这个时间点由该函数参数中所定义的时间戳来决定），当接收完 RTP 数据包后，该函数才会返回。同 样，rtp_session_send_with_ts()也会一直阻塞直到需要被发送的 RTP 包的时间点到达，发送结束后，函数才返回。

3.4 rtp_session_signal_connect

函数原型：int rtp_session_signal_connect (RtpSession * session， const char *signal， RtpCallback cb，unsigned long user_data)

函数功能：本函数提供一种方式，用于通知应用程序各种可能发生的 RTP 事件（信号）。可能通过注册回调函数的形式来实现本功能。

参数含义：

session： rtp 会话结构体

signal： 信号的名称

cb：回调函数

user_data：传递给回调函数的数据

返回值：0 表示成功，-EOPNOTSUPP 表示信号名称不存在，-1 表示回调函数绑定错误

说明：

信号的名称必须是以下字符串中的一种：

"ssrc_changed" : 数据流的同步源标识改变

"payload_type_changed" : 数据流的负载类型改变

"telephone-event_packet" : telephone-event RTP 包(RFC2833)被接收

"telephone-event" : telephone event 发生

"network_error" : 网络错误产生，传递给回调函数的是描述错误的字符串（const char *型）或者错误码（int 型）

"timestamp_jump" : 接收到的数据包发生了时间戳的跳跃。

要取消事件（信号）的监听，可以使用下面这个函数

int rtp_session_signal_disconnect_by_callback (RtpSession * session， const char * signal_name，RtpCallback cb)

3.5 rtp_session_set_local_addr

函数原型：int rtp_session_set_local_addr(RtpSession * session， const char * addr，int port)

函数功能：设置本地 rtp 数据监听地址

参数含义：

session： rtp 会话结构体

addr： 本地 IP 地址，例如 127。0。0。1，如果为 NULL，则系统分配 0。0。0。0

port： 监听端口，如果设置为-1，则系统为其自动分配端口

返回值： 0 表示成功

说明：

如果是 RTP_SESSION_SENDONLY（只发送）型会话，则不需要进行本设置，而必须设置 rtp_session_set_remote_addr() 来设置远程目的地址。

如果采用了系统自动分配监听端口，则可以通过 int rtp_session_get_local_port(const RtpSession *session) 来获取系统分配的监听端口号。

3.6 rtp_session_set_remote_addr

函数原型：int rtp_session_set_remote_addr (RtpSession * session， const char * addr， int port)

函数功能：设置 RTP 发送的目的地址

参数含义：

session： rtp 会话结构体

addr：目的 IP 地址

port：目的地址的监听端口号

返回值： 0 表示成功

3.7 rtp_session_set_send_payload_type

函数原型：int rtp_session_set_send_payload_type (RtpSession * session， int paytype)

函数功能：设置 RTP 发送数据的负载类型

参数含义：

session： rtp 会话结构体

paytype：负载类型

返回值： 0 表示成功，-1 表示负载未定义

说明：

负载类型在 payloadtype。h 文件中有详细的定义，RTP 接收端有着类似的负载类型设置函数，int

rtp_session_set_recv_payload_type (RtpSession * session， int paytype) ，注意，发送的负载类型必须与接收的负载类型一致才能正常完成收发。

3.8 rtp_session_send_with_ts

函数原型：int rtp_session_send_with_ts (RtpSession * session， const char * buffer， int len，uint32_t userts)

函数功能：发送 RTP 数据包

参数含义：

session： rtp 会话结构体

buffer： 需要发送的 RTP 数据的缓冲区

len： 需要发送的 RTP 数据的长度

userts： 本 RTP 数据包的时间戳

返回值： 成功发送到网络中的字节数

说明：

发送 RTP 数据需要自己管理时间戳的递增，每调用一次本函数，请根据实际情况对 userts 进行递增，具体递增的规则见 RTP 协议中的说明。

例如：如果发送的是采样率为 90000Hz 的视频数据包，每秒 25 帧，则时间戳的增量为：90000/25 = 3600

时间戳的起始值为随机值，建议设置为 0。

3.9 rtp_session_recv_with_ts

函数原型：int rtp_session_recv_with_ts (RtpSession * session， char * buffer，int len， uint32_t time， int * have_more)

函数功能：接收 RTP 数据包

参数含义：

session： rtp 会话结构体

buffer： 存放接收的 RTP 数据的缓冲区

len： 期望接收的 RTP 数据的长度

time：期望接收的 RTP 数据的时间戳

have_more：标识接收缓冲区是否还有数据没有传递完。当用户给出的缓冲区不够大时，为了标识缓冲区数据未取完，则 have_more 指向的数据为 1，期望用户以同样的时间戳再次调用本函数；否则为 0，标识取完。

3.10 rtp_session_destroy

函数原型： void rtp_session_destroy(RtpSession *session)

函数功能：摧毁 rtp 会话对象，释放资源

函数参数：session 已经创建的 RTP 会话对象。

4. RTP 结构简介

RTP 协议是 IETF 在 1996 年提出的适合实时数据传输的新型协议。RTP 协议实际上是由实时传输协议 RTP（Real-time Transport Protocol）和实时传输控制协议 RTCP（Real-time Transport Control Protocol）两部分组成。RTP 协议基于多播或单播网络为用户提供连续媒体数据的实时传输服务；RTCP 协议是 RTP 协议的控制部分，用于实时监控数据传输质量，为系统提供拥塞控制和流控制。RTP 协议在 RFC3550 中有详细介绍。每一个 RTP 数据包都由固定包头（Header ）和载荷（Payload）两个部分组成，其中包头前 12 个字节的含义是固定的，而载荷则可以是音频或视频数据。RTP 固定包头的格式如图 1 所示：

图

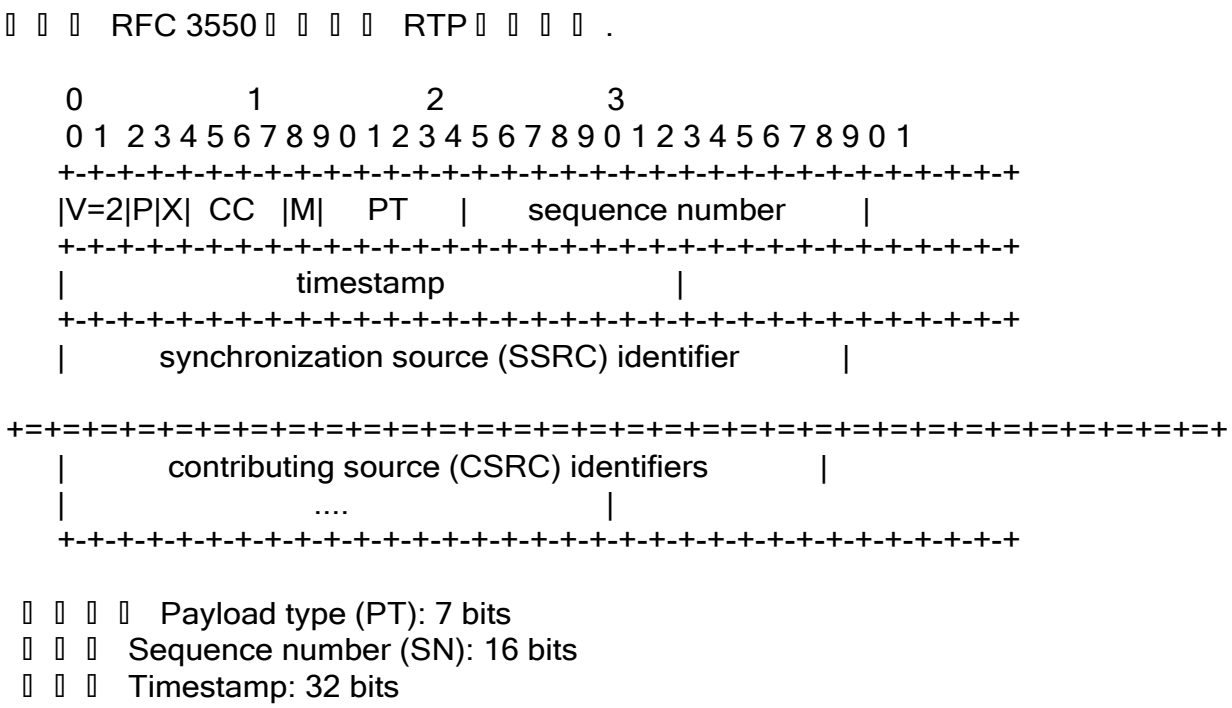


图 1 RTP 包格式

完整的 RTP 固定包头的格式在前面图 1 中已经指出，根据 RFC3984[3]，这里详细给出各个位的具体设置。

(1). V：版本号，2 位。根据 RFC3984，目前使用的 RTP 版本号应设为 0x10。

(2). P：填充位，1 位。当前不使用特殊的加密算法，因此该位设为 0。

(3). X：扩展位，1 位。当前固定头后面不跟随头扩展，因此该位也为 0。

(4). CC：CSRC 计数，4 位。表示跟在 RTP 固定包头后面 CSRC 的数目，对于本文所要实现的基本的流媒体服务器来说，没有用到混合器，该位也设为 0x0。

(5). M：标示位，1 位。如果当前 NALU 为一个接入单元最后的那个 NALU，那么将 M 位置 1；或者当前 RTP 数据包为一个 NALU 的最后的那个分片时（NALU 的分片在后面讲述），M 位置 1。其余情况下 M 位保持为 0。

(6). PT：载荷类型，7 位。对于 H.264 视频格式，当前并没有规定一个默认的 PT 值。因此选用大于 95 的值可以。此处设为 0x60（十进制 96）。

(7). SQ：序号，16 位。序号的起始值为随机值，此处设为 0，每发送一个 RTP 数据包，序号值加 1。

(8). TS：时间戳，32 位。同序号一样，时间戳的起始值也为随机值，此处设为 0。根据 RFC3984，与时间戳相应的时钟频率必须为 90000HZ。

(9). SSRC：同步源标示，32 位。SSRC 应该被随机生成，以使在同一个 RTP 会话期中没有任何两个同步源具有相同的 SSRC 识别符。此处仅有一个同步源，因此将其设为 0x12345678。

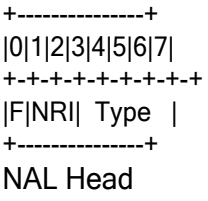
5. H.264 基本流结构及其传输机制

5.1 H.264 基本流的结构

H.264 的基本流（elementary stream，ES）的结构分为两层，包括视频编码层（VCL）和网络适配层（NAL）。视频编码层负责高效的视频内容表示，而网络适配层负责以网络所要求的恰当的方式对数据进行打包和传送。引入 NAL 并使之与 VCL 分离带来的好处包括两方面：其一、使信号处理和网络传输分离，VCL 和 NAL 可以在不同的处理平台上实现；其二、VCL 和 NAL 分离设计，使得在不同的网络环境内，网关不需要因为网络环境不同而对 VCL 比特流进行重构和重编码。

5.2 网络抽象层单元类型 (NALU)

H.264 的基本流由一系列 NALU（Network Abstraction Layer Unit ）组成，不同的 NALU 数据量各不相同。H.264 草案指出[2]，当数据流是储存在介质上时，在每个 NALU 前添加起始码：**0x000001**（或**0x00000001**），用来指示一个 NALU 的起始和终止位置。在这样的机制下，解码器在码流中检测起始码，作为一个 NALU 得起始标识，当检测到下一个起始码时，当前 NALU 结束。每个 NALU 单元由一个字节的 NALU 头（NALU Header）和若干个字节的载荷数据（RBSP）组成。NALU 头由一个字节组成，它的语法如下:



F：1 个比特。

forbidden_zero_bit。 在 H.264 规范中规定了这一位必须为 0。

NRI：2 个比特。

nal_ref_idc。 取 00～11， 似乎指示这个 NALU 的重要性， 如 00 的 NALU 解码器可以丢弃它而不影响图像的回放。 不过一般情况下不太关心这个属性。

Type：5 个比特。

nal_unit_type。 这个 NALU 单元的类型。 简述如下:

0	没有定义	
1-23	NAL单元	单个 NAL 单元包。
24	STAP-A	单一时间的组合包
25	STAP-B	单一时间的组合包
26	MTAP16	多个时间的组合包
27	MTAP24	多个时间的组合包
28	FU-A	分片的单元
29	FU-B	分片的单元
30-31	没有定义	

5.3 打包模式

对于每一个NALU，根据其包含的数据量的不同，其大小也有差异。在IP网络中，当要传输的IP 报文大小超过最大传输单元MTU（Maximum Transmission Unit ）时就会产生IP分片情况。在以太网环境中可传输的最大 IP 报文（MTU）的大小为 1500 字节。如果发送的IP数据包大于MTU，数据包就会被拆开来传送，这样就会产生很多数据包碎片，增加丢包率，降低网络速度。对于视频传输而言，若RTP 包大于MTU 而由底层协议任意拆包，可能会导致接收端播放器的延时播放甚至无法正常播放。因此对于大于MTU 的NALU 单元，必须进行拆包处理。

RFC3984 给出了 3 中不同的RTP 打包方案：

(1). Single NALU Packet：在一个RTP 包中只封装一个NALU，在本文中对于小于 1400 字节的NALU

便采用这种打包方案。

(2). Aggregation Packet：在一个RTP 包中封装多个NALU，对于较小的NALU 可以采用这种打包方案，从而提高传输效率。

(3). Fragmentation Unit：一个NALU 封装在多个RTP包中，在本文中，对于大于 1400 字节的NALU 便采用这种方案进行拆包处理。

a) 单一 NAL 单元模式

即一个 RTP 包仅由一个完整的 NALU 组成。 这种情况下 RTP NAL 头类型字段和原始的 H.264 NALU 头类型字段是一样的。

b) 组合封包模式

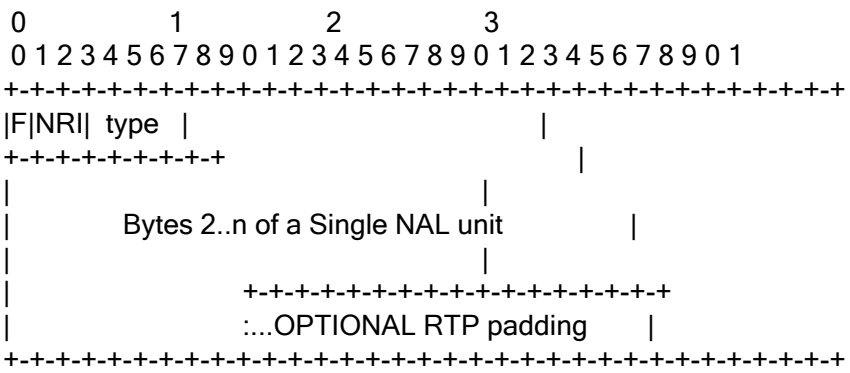
即可能是由多个 NAL 单元组成一个 RTP 包，分别有 4 种组合方式: STAP-A， STAP-B，MTAP16， MTAP24。那么这里的类型值分别是 24， 25， 26 以及 27。

c) 分片封包模式

用于把一个 NALU 单元封装成多个 RTP 包。存在两种类型 FU-A 和 FU-B。类型值分别是 28 和 29。

5.3.1 单一 NAL 单元模式

对于 NALU 的长度小于 MTU 大小的包， 一般采用单一 NAL 单元模式。对于一个原始的 H.264 NALU 单元常由 [Start Code] [NALU Header] [NALU Payload] 三部分组成， 其中 Start Code 用于标示这是一个NALU 单元的开始， 必须是 "00 00 00 01" 或 "00 00 01"， NALU 头仅一个字节， 其后都是 NALU 单元内容。打包时去除 "00 00 01" 或 "00 00 00 01" 的开始码， 把其他数据封包的 RTP 包即可。



如有一个 H.264 的 NALU 是这样的:

[00 00 00 01 67 42 A0 1E 23 56 0E 2F ...]

这是一个序列参数集 NAL 单元. [00 00 00 01] 是四个字节的开始码， 67 是 NALU 头， 42 开始的数据是 NALU 内容。

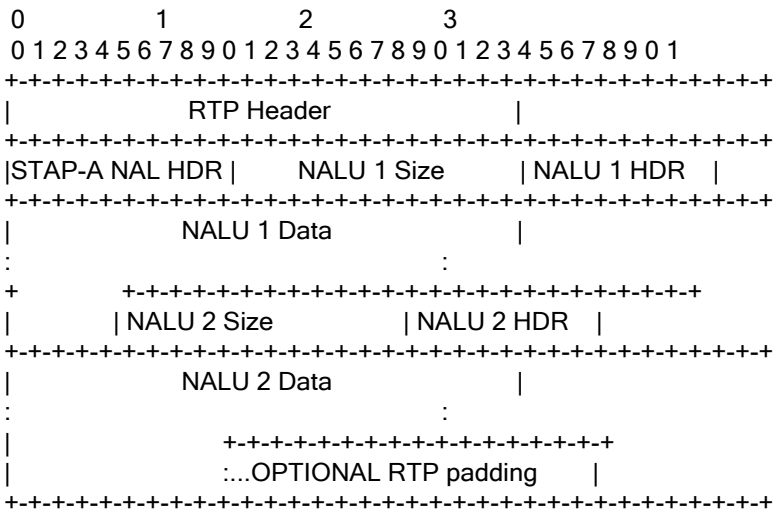
封装成 RTP 包将如下：

[RTP Header] [67 42 A0 1E 23 56 0E 2F]

即只要去掉 4 个字节的开始码就可以了。

5.3.2 组合封包模式

其次， 当 NALU 的长度特别小时， 可以把几个 NALU 单元封在一个 RTP 包中。



5.3.2 分包 Fragmentation Units (FUs).

而当 NALU 的长度超过 MTU 时, 就必须对 NALU 单元进行分片封包。也称为 Fragmentation Units (FUs)。

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| FU indicator  |   FU header   |                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+                               |
|                                                               |
|                  FU payload                                   |
|                                                               |
|                      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                      :...OPTIONAL RTP padding                 |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 14. RTP payload format for FU-A

The FU indicator octet has the following format:

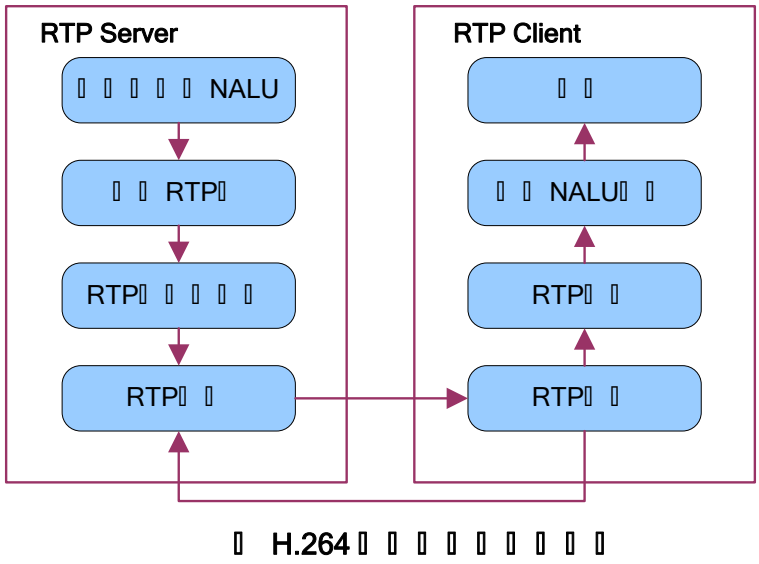
```
+---------------+
|0|1|2|3|4|5|6|7|
+-+-+-+-+-+-+-+-+
|F|NRI|  Type   |
+---------------+
```

The FU header has the following format:

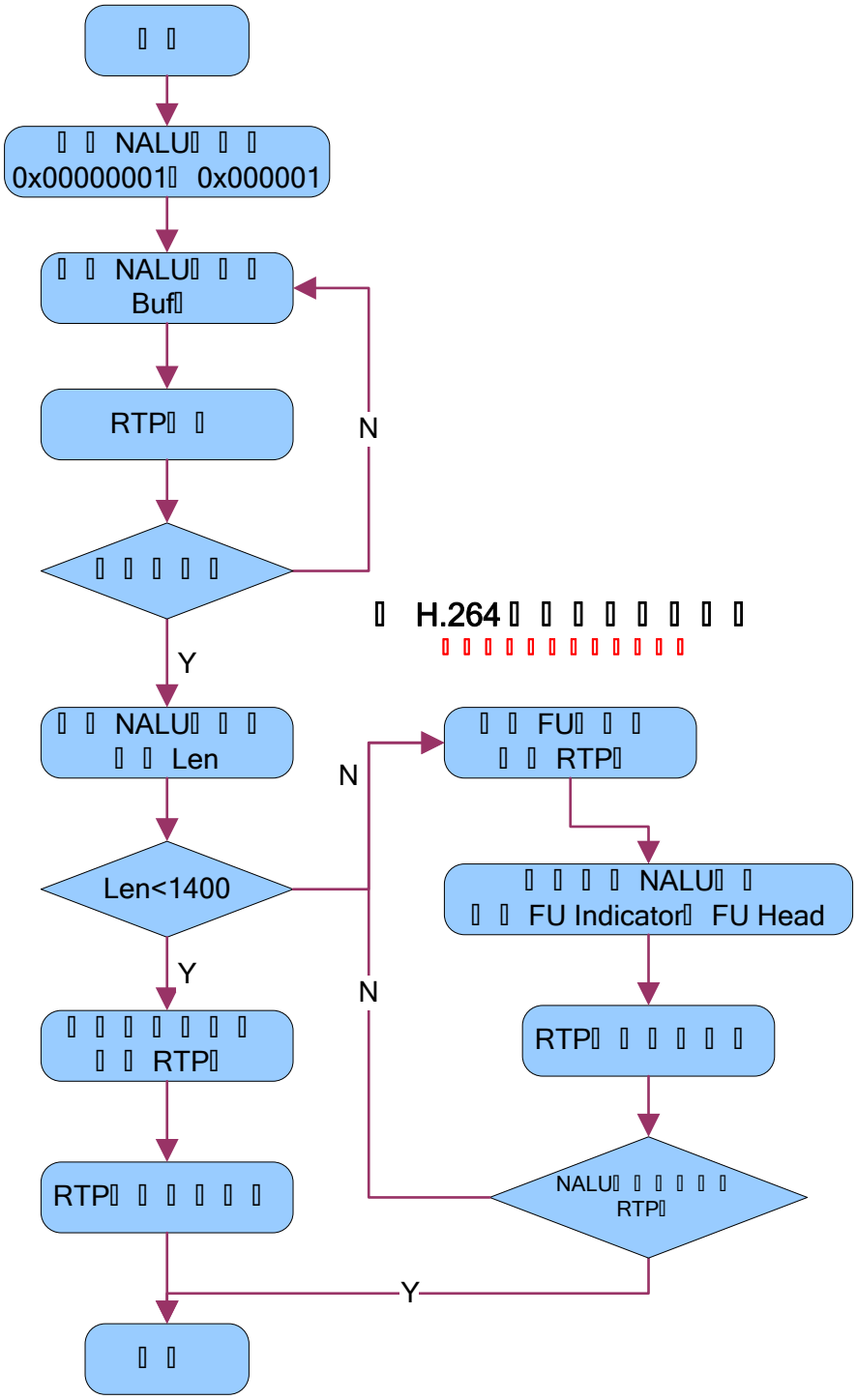
```
+---------------+
|0|1|2|3|4|5|6|7|
+-+-+-+-+-+-+-+-+
|S|E|R|  Type   |
+---------------+
```

5.4 H.264 流媒体传输系统的实现

一个完整的流媒体传输系统包含服务器端和客户端两个部分。对于服务器端，其主要任务是读取H.264 视频，从码流中分离出每个NALU 单元，分析NALU 的类型，设置相应的 RTP 包头，封装 RTP 数据包并发送。而对于客户端来说，其主要任务则是接收 RTP数据包，从RTP 包中解析出NALU 单元，然后送至解码器进行解码播放。该流媒体传输系统的框架如下图所示。



其中服务器端的算法流程如下图所示：



6. 程序示例

/*

ortp 学习笔记

分类： C/C++ 音频编解码 2012-07-16 22:07 309 人阅读 评论(1) 收藏 举报

ortp 版本：ortp-0.18.0.tar.gz

操作系统：window 7 32bit

6.1 windows 下编译 ortp.lib

直接打开 ortp-0.18.0\build\win32native 的工程文件即可，VS2008 下无需任何修改，即可编译出动态链接库 ortp.lib 以及 ortp.dll。

6.2.使用 ortp 提供的测试程序

ortp-0.18.0\src\tests 下的 win_receive 以及 win_sender 目录下程序。

在 windows7 下使用 win_receiver 时会提示下如下错误，这个在错误在 windows XP 下是不存在的 QOSAddSocketToFlow failed to add a flow with error 87 具体问题暂时还没有深究，这个应该是一个系统兼容性问题，需要系统支持 qwave.lib。在查找了错误出处，对比了前几个 ortp 的版本后，对 ortp0.18 的源代码进行了修改

rtp_session_inet.c

/* set socket options (but don't change chosen states) */

/*

rtp_session_set_dscp(session, -1);

rtp_session_set_multicast_ttl(session, -1);

rtp_session_set_multicast_loopback(session, -1);

*/

6.3 发送 H.264 视频帧

ortp 提供的 win_receive 以及 win_sender 一次只发送 160 字节的数据。但我们在发送一帧 H.264 视频帧，每次需要发送 2000-3000 字节的数据，关于 ortp 发送 H.264 视频帧，可以参考：

ortp 编程示例代码

谈谈 RTP 传输中的负载类型和时间戳

除了上面两篇文章提到需要注意的负载类型和时间戳之外，在 ortp 中 win_receive 中还需要显示调用：

rtp_session_set_recv_buf_size(rtp_session_mgr .rtp_session , recv_bufsize);

这里的 recv_bufsize 必须要比 win_sender 中

sended_bytes = rtp_session_send_with_ts (rtp_session_mgr .rtp_session ,

(uint8_t *) send_buffer,

wrapLen,

rtp_session_mgr.cur_timestamp);

的 wrapLen 要大，否则在 receive 端会出现如下错误：

ortp-warning-Error receiving RTP packet: Error code : 10040, err num [10040],error [-1]

6.4 一个结合 H.264 编码，rtp 发送接收，保存 mp4 文件的小程序

ortp_test.rar

编译环境：vs2008

压缩文件中包括了 ffmpeg 以及 ortp 的动态链接库，但你还需要在工程文件中修改它们的路径才能正确链接这两个库。因为这个小程序是一个项目的一部分，源代码中还有些冗余代码，并没有来的及删掉，大家在看的时候需要注意下。

注意：该程序没有分包。

6.5 代码

```
//receiver
//[cpp] view plaincopyprint?
#include <string.h>
#include "ortp/ortp.h"

extern "C"{
	#include <libavformat/avformat.h>
	#include <libswscale/swscale.h>
};

bool m_bExit = FALSE;

struct RtpSessionMgr
{
	RtpSession *rtp_session;
	int timestamp;
};


RtpSessionMgr rtp_session_mgr;
const int timestamp_inc = 3600; // 90000/25

const char recv_ip[] = "127.0.0.1";
const int recv_port = 8008;
const int recv_bufsize = 10240;
unsigned char *recv_buf;

/** 帧包头的标识长度 */
#define CMD_HEADER_LEN 10
```

```
/** 帧包头的定义 */
static uint8_t CMD_HEADER_STR[CMD_HEADER_LEN] = { 0xAA,0xA1,0xA2,0xA3,0xA4,0xA5,0xA6,0xA7,0xA8,0xFF };

/** 帧的包头信息 */
typedef struct _sFrameHeader
{
    /** 命令名称标识 */
    unsigned char cmdHeader[CMD_HEADER_LEN];

    /** 采集的通道号 0~7*/
    unsigned char chId;

    /** 数据类型，音频 或者 视频*/
    unsigned char dataType;

    /** 缓冲区的数据长度 */
    uint32_t len;

    /** 时间戳 */
    uint32_t timestamp;

}FrameHeader;

///////ffmpeg//////////////////////////////////////////////////////////////////
AVOutputFormat *fmt;
AVFormatContext *oc;
AVStream *video_st;
AVCodecContext *codecContext;

const int image_width = 704;
const int image_height = 576;
```

```
const int frame_rate = 25;
static int frame_count;

BOOL ctrlHandlerFunction(DWORD fdwCtrlType)
{
	switch (fdwCtrlType)
	{
		// Handle the CTRL+C signal.
		// CTRL+CLOSE: confirm that the user wants to exit.
	case CTRL_C_EVENT:
	case CTRL_CLOSE_EVENT:
	case CTRL_BREAK_EVENT:
	case CTRL_LOGOFF_EVENT:
	case CTRL_SHUTDOWN_EVENT:
		m_bExit = TRUE;
		return TRUE;

	default:
		return FALSE;
	}
}

void rtpInit()
{
	int ret;
	WSADATA wsaData;

	/**  初始化 winsocket */
	if ( WSAStartup(MAKEWORD(2,2), &wsaData) != 0)
	{
		fprintf(stderr, "WAStartup failed!\n");
		return ;
```

```
    }

    ortp_init();
    ortp_scheduler_init();

    rtp_session_mgr.rtp_session = rtp_session_new(RTP_SESSION_RECVONLY);

    rtp_session_set_scheduling_mode(rtp_session_mgr.rtp_session, 1);
    rtp_session_set_blocking_mode(rtp_session_mgr.rtp_session, 1);
    rtp_session_set_local_addr(rtp_session_mgr.rtp_session, recv_ip, recv_port);

    rtp_session_enable_adaptive_jitter_compensation(rtp_session_mgr.rtp_session, TRUE);
    rtp_session_set_jitter_compensation(rtp_session_mgr.rtp_session, 40);

    rtp_session_set_payload_type(rtp_session_mgr.rtp_session, 34);
    rtp_session_set_recv_buf_size(rtp_session_mgr.rtp_session, recv_bufsize);

    rtp_session_mgr.timestamp = timestamp_inc;
}

int rtp2disk()
{
    int err;
    int havemore = 1;

    while (havemore)
    {
        err = rtp_session_recv_with_ts(rtp_session_mgr.rtp_session,
            (uint8_t *)recv_buf, recv_bufsize,
            rtp_session_mgr.timestamp, &havemore);

        if (havemore)
```

```
            printf("==> Warning: havemore=1!\n");

        if (err > 0)
        {
            FrameHeader *frameHeader;

            printf("receive data is %d\n", err);

            frameHeader = (FrameHeader *)recv_buf;
            printf("frame_len = %d\n", frameHeader->len);

            AVPacket pkt;
            av_init_packet(&pkt);
            pkt.stream_index= video_st->index;
            pkt.data= recv_buf + sizeof(FrameHeader);
            pkt.size = frameHeader->len; // not the video_outbuf_size, note!
            // write the compressed frame in the media file
            err = av_write_frame(oc, &pkt);

            if (err != 0)
            {
                printf("av_write_frame failed\n");
            }
        }
    }

    return 0;
}

AVCodecContext* createCodecContext(AVFormatContext *oc)
{
    AVCodecContext *video_cc = avcodec_alloc_context();
```

```
video_cc = avcodec_alloc_context();
if (!video_cc)
{
	fprintf(stderr, "alloc avcodec context failed\n");
	exit(1);
}

video_cc->codec_id = (CodecID)CODEC_ID_H264;
video_cc->codec_type = AVMEDIA_TYPE_VIDEO;

video_cc->me_range = 16;
video_cc->max_qdiff = 4;
video_cc->qmin = 10;
video_cc->qmax = 51;
video_cc->qcompress = 0.6f;

/* put sample parameters */
video_cc->bit_rate = 400000;

/* resolution must be a multiple of two */
video_cc->width = image_width;
video_cc->height = image_height;

/* time base: this is the fundamental unit of time (in seconds) in terms
of which frame timestamps are represented. for fixed-fps content,
timebase should be 1/framerate and timestamp increments should be
identically 1. */
video_cc->time_base.den = frame_rate;
video_cc->time_base.num = 1;

video_cc->gop_size = 12; /* emit one intra frame every twelve frames at most */
```

```
	video_cc->pix_fmt = PIX_FMT_YUV420P;

	// some formats want stream headers to be separate
	if(!strcmp(oc->oformat->name, "mp4") || !strcmp(oc->oformat->name, "mov") || !strcmp(oc->oformat->name, "3gp"))
		video_cc->flags |= CODEC_FLAG_GLOBAL_HEADER;

	return video_cc;
}

void openVideo(AVFormatContext *oc)
{
	AVCodec *codec;

	/* find the video encoder */
	codec = avcodec_find_encoder(codecContext->codec_id);
	if (!codec) {
		fprintf(stderr, "codec not found\n");
		exit(1);
	}

	/* open the codec */
	if (avcodec_open(codecContext, codec) < 0) {
		fprintf(stderr, "could not open video codec\n");
		exit(1);
	}
}

void ffmpegEncodeInit()
{
	// initialize libavcodec, and register all codecs and formats
	av_register_all();
```

```
char filename[] = "test.mp4";
fmt = av_guess_format(NULL, filename, NULL);

oc = avformat_alloc_context();
oc->oformat = fmt;

fmt->video_codec = (CodecID) CODEC_ID_H264;
_snprintf(oc->filename, sizeof(oc->filename), "%s", filename);

// add the video streams using the default format codecs and initialize the codecs
video_st = NULL;
if (fmt->video_codec != CODEC_ID_NONE) {
	video_st = av_new_stream(oc, 0);
	if (!video_st) {
		fprintf(stderr, "Could not alloc stream\n");
		exit(1);
	}
}

// alloc codecContext
codecContext = createCodecContext(oc);
video_st->codec = codecContext;

// set the output parameters (must be done even if no parameters).
if (av_set_parameters(oc, NULL) < 0) {
	fprintf(stderr, "Invalid output format parameters\n");
	exit(1);
}

dump_format(oc, 0, filename, 1);
```

```
	/* now that all the parameters are set, we can open the audio and
	video codecs and allocate the necessary encode buffers */
	if (codecContext)
		openVideo(oc);

	// open the output file, if needed
	if (!(fmt->flags & AVFMT_NOFILE)) {
		if (url_fopen(&oc->pb, filename, URL_WRONLY) < 0) {
			fprintf(stderr, "Could not open '%s'\n", filename);
			exit(1);
		}
	}

	// write the stream header, if any
	av_write_header(oc);
}

void ffmpegEncodeClose()
{
	int i;

	/* close each codec */
	if (video_st)
		avcodec_close(video_st->codec);

	// write the trailer, if any
	av_write_trailer(oc);

	/* free the streams */
	for(i = 0; i < oc->nb_streams; i++) {
		av_freep(&oc->streams[i]->codec);
		av_freep(&oc->streams[i]);
```

```
    }

    if (!(fmt->flags & AVFMT_NOFILE)) {
        /* close the output file */
        url_fclose(oc->pb);
    }

    /* free the stream */
    av_free(oc);
}

int main()
{
    recv_buf = (uint8_t *)malloc(recv_bufsize);

    rtpInit();
    ffmpegEncodeInit();

    // ================ INSTALL THE CONTROL HANDLER ================
    if (SetConsoleCtrlHandler( (PHANDLER_ROUTINE) ctrlHandlerFunction, TRUE) == 0)
    {
        printf("==> Cannot handle the CTRL-C...\n");
    }

    printf("==> RTP Receiver started\n");

    while (m_bExit == FALSE)
    {
        rtp2disk();

        rtp_session_mgr.timestamp += timestamp_inc;
    }
```

```
	printf("==> Exiting\n");

	free(recv_buf);

	ffmpegEncodeClose();

	rtp_session_destroy(rtp_session_mgr.rtp_session);
	ortp_exit();
}

//sender

#include <ortp/ortp.h>
#include <string.h>

extern "C"{
#include <libavformat/avformat.h>
#include <libswscale/swscale.h>
};

struct RtpSessionMgr
{
	RtpSession *rtp_session;
	uint32_t timestamp_inc;
	uint32_t cur_timestamp;
};

RtpSessionMgr rtp_session_mgr;

const char g_ip[] = "127.0.0.1";
const int g_port = 8008;
```

```
const uint32_t timestamp_inc = 3600; // 90000 / 25

const int image_width = 704;
const int image_height = 576;
const int frame_rate = 25;
static int frame_count, wrap_size;

AVCodecContext *video_cc;

AVFrame *picture;

/** 帧包头的标识长度 */
#define CMD_HEADER_LEN 10

/** 帧包头的定义 */
static uint8_t CMD_HEADER_STR[CMD_HEADER_LEN] = { 0xAA,0xA1,0xA2,0xA3,0xA4,0xA5,0xA6,0xA7,0xA8,0xFF };

/** 帧的包头信息 */
typedef struct _sFrameHeader
{
    /** 命令名称标识 */
    unsigned char cmdHeader[CMD_HEADER_LEN];

    /** 采集的通道号 0~7*/
    unsigned char chId;

    /** 数据类型，音频 或者 视频*/
    unsigned char dataType;

    /** 缓冲区的数据长度 */
    uint32_t len;
```

```
    /** 时间戳 */
    uint32_t timestamp;

}FrameHeader;

// set frame header
FrameHeader frameHeader;

void rtpInit()
{
    char *m_SSRC;

    ortp_init();
    ortp_scheduler_init();
    printf("Scheduler initialized\n");

    rtp_session_mgr.rtp_session = rtp_session_new(RTP_SESSION_SENDONLY);

    rtp_session_set_scheduling_mode(rtp_session_mgr.rtp_session, 1);
    rtp_session_set_blocking_mode(rtp_session_mgr.rtp_session, 1);
    rtp_session_set_remote_addr(rtp_session_mgr.rtp_session, g_ip, g_port);
    rtp_session_set_send_payload_type(rtp_session_mgr.rtp_session, 34); // 34 is for H.263 video frame

    m_SSRC = getenv("SSRC");
    if (m_SSRC != NULL)
    {
        rtp_session_set_ssrc(rtp_session_mgr.rtp_session, atoi(m_SSRC));
    }

    rtp_session_mgr.cur_timestamp = 0;
    rtp_session_mgr.timestamp_inc = timestamp_inc;
```

```
    printf("rtp init success!\n");
}

int rtpSend(unsigned char *send_buffer, int frame_len)
{
    FrameHeader *fHeader = (FrameHeader *)send_buffer;
    fHeader->chId = 0;
    fHeader->dataType = 0; // SESSION_TYPE_VIDEO
    fHeader->len = frame_len;
    fHeader->timestamp = 0;

    printf("frame header len = %d\n", fHeader->len);

    int wrapLen;
    wrapLen = frame_len + sizeof(FrameHeader);

    int sended_bytes;
    sended_bytes = rtp_session_send_with_ts(rtp_session_mgr.rtp_session,
        (uint8_t *)send_buffer,
        wrapLen,
        rtp_session_mgr.cur_timestamp);

    rtp_session_mgr.cur_timestamp += rtp_session_mgr.timestamp_inc;

    return sended_bytes;
}

void createCodecContext()
{
    video_cc = avcodec_alloc_context();
    if (!video_cc)
```

```
    {
        fprintf(stderr, "alloc avcodec context failed\n");
        exit(1);
    }

    video_cc->codec_id = (CodecID)CODEC_ID_H264;
    video_cc->codec_type = AVMEDIA_TYPE_VIDEO;

    video_cc->me_range = 16;
    video_cc->max_qdiff = 4;
    video_cc->qmin = 10;
    video_cc->qmax = 51;
    video_cc->qcompress = 0.6f;

    /* put sample parameters */
    video_cc->bit_rate = 400000;

    /* resolution must be a multiple of two */
    video_cc->width = image_width;
    video_cc->height = image_height;

    /* time base: this is the fundamental unit of time (in seconds) in terms
    of which frame timestamps are represented. for fixed-fps content,
    timebase should be 1/framerate and timestamp increments should be
    identically 1. */
    video_cc->time_base.den = frame_rate;
    video_cc->time_base.num = 1;

    video_cc->gop_size = 12; /* emit one intra frame every twelve frames at most */
    video_cc->pix_fmt = PIX_FMT_YUV420P;
}
```

```
AVFrame *allocPicture(int pix_fmt, int width, int height)
{
    AVFrame *picture;
    uint8_t *picture_buf;
    int size;

    picture = avcodec_alloc_frame();
    if (!picture)
        return NULL;

    size = avpicture_get_size((PixelFormat)pix_fmt, width, height);
    picture_buf = (uint8_t *)av_malloc(size);
    if (!picture_buf) {
        av_free(picture);
        return NULL;
    }
    avpicture_fill((AVPicture *)picture, picture_buf, (PixelFormat)pix_fmt, width, height);
    return picture;
}

void openVideo()
{
    AVCodec *video_codec;

    /* find the video encoder */
    video_codec = avcodec_find_encoder(video_cc->codec_id);
    if (!video_codec) {
        fprintf(stderr, "codec not found\n");
        exit(1);
    }

    /* open the codec */
```

```
    if (avcodec_open(video_cc, video_codec) < 0) {
        fprintf(stderr, "could not open video codec\n");
        exit(1);
    }

    /* allocate the encoded raw picture */
    picture = allocPicture(video_cc->pix_fmt, video_cc->width, video_cc->height);
    if (!picture) {
        fprintf(stderr, "Could not allocate picture\n");
        exit(1);
    }
}

/* prepare a dummy image */
void fill_yuv_image(AVFrame *pict, int frame_index, int width, int height)
{
    int x, y, i;

    i = frame_index;

    /* Y */
    for(y=0;y<height;y++) {
        for(x=0;x<width;x++) {
            pict->data[0][y * pict->linesize[0] + x] = x + y + i * 3;
        }
    }

    /* Cb and Cr */
    for(y=0;y<height/2;y++) {
        for(x=0;x<width/2;x++) {
            pict->data[1][y * pict->linesize[1] + x] = 128 + y + i * 2;
            pict->data[2][y * pict->linesize[2] + x] = 64 + x + i * 5;
```

```
            }
        }
}

void ffmpegInit()
{
        // initialize libavcodec, and register all codecs and formats
        av_register_all();

        // create a codec context
        createCodecContext();

        // open H.264 codec
        openVideo();
}

void getEncodedFrame(unsigned char *buffer, int& len)
{
        int out_size;

        fill_yuv_image(picture, frame_count, video_cc->width, video_cc->height);

        // encode the frame
        out_size = avcodec_encode_video(video_cc, buffer, wrap_size-sizeof(FrameHeader), picture);

        len = out_size;
        frame_count++;
}

int main()
{
        unsigned char *send_outbuf;
```

```
unsigned char *video_part;

frame_count = 0;
wrap_size = 20000;
send_outbuf = (unsigned char *)malloc(wrap_size);

// copy cmdHeader to frameInfo
memcpy(frameHeader.cmdHeader,CMD_HEADER_STR,CMD_HEADER_LEN);

memcpy(send_outbuf, &frameHeader, sizeof(FrameHeader));
video_part = send_outbuf + sizeof(FrameHeader);

ffmpegInit();
rtpInit();

while (1)
{
    int frame_len;
    // get encode frame
    getEncodedFrame(video_part, frame_len);

    printf("encodecFrame length is : %d\n", frame_len);

    if (frame_len > 0)
    {
        rtpSend(send_outbuf, frame_len);
    }
}

rtp_session_destroy(rtp_session_mgr.rtp_session);

free(send_outbuf);
```

```
    // Give us some time
    Sleep(250);

    ortp_exit();
}
```