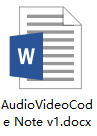
# AudioVideoCode Note v1



承接《AudioVideoCode Note v1》

本文档主要包含：st2110等内容

**SDI：**

**标准版本：**

|  |  |  |  |
| --- | --- | --- | --- |
|  | 标准 | 推出年份 | 视频格式示例 |
| SD-SDI | SMPTE 259M | 1989年 | 480i、576i |
| ED-SDI | SMPTE 344M | 2000年 | 480p、576p |
| HD-SDI | SMPTE 292M | 1998年 | 720p、1080i |
| 双链路HD-SDI | SMPTE 372M | 2002年 | 1080p60 |
| 3G-SDI | SMPTE 424M | 2006年 | 1080p60 |
| 6G-SDI | SMPTE ST 2081 | 2015年 | 1080p120、2160p30 |
| 12G-SDI | SMPTE ST 2082 | 2015年 | 2160p60（真4K） |
| 24G-SDI | SMPTE ST 2083 | 2020年 | 2160p120、4320p30 |

对于1080p60，uyvy，则bps=(1920\*1080\*2)\*60\*8=1,990,656,000≈2Gbps

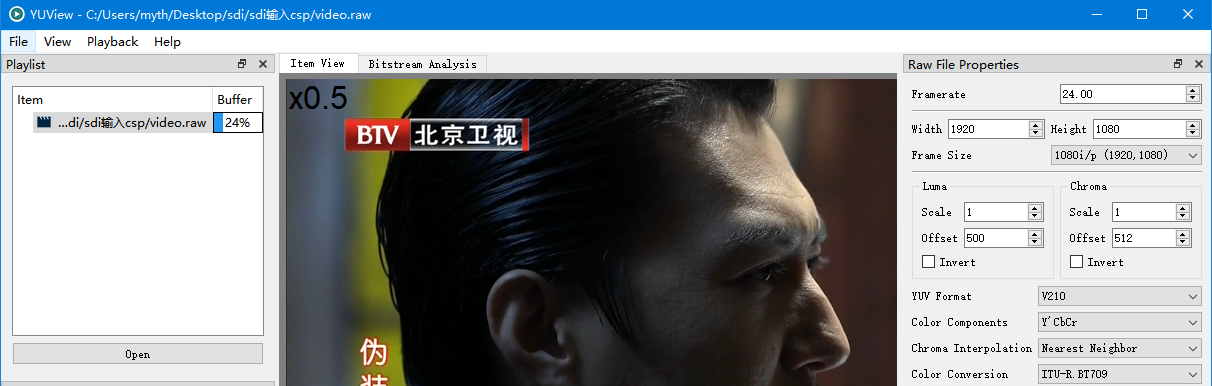
对于1080p60，10bit，则bps=(1920/48\*128x1080)\*60\*8=2,654,208,000≈3Gbps

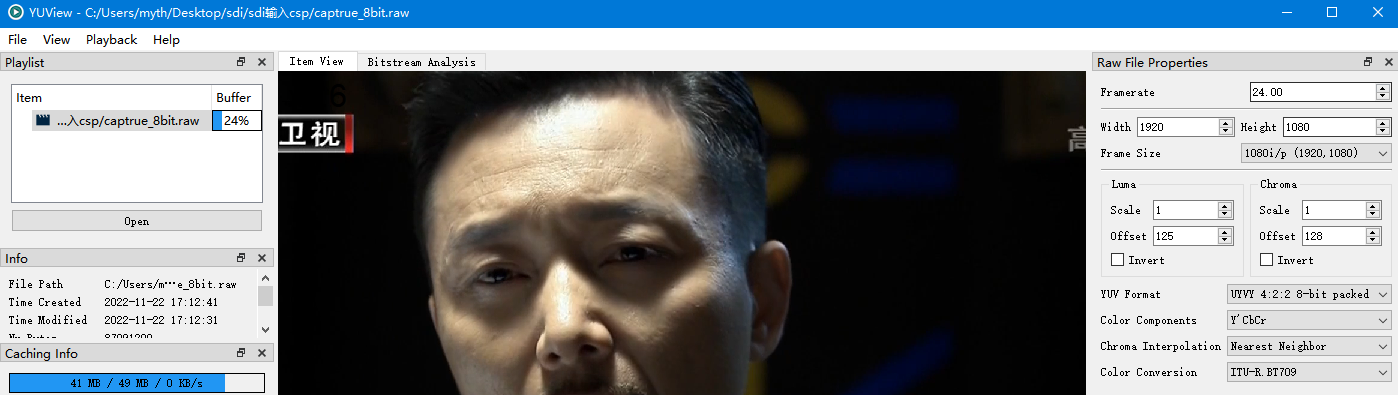
对于2160p60，10bit，则bps=2,654,208,000x4=10,616,832,000≈11Gbps

**csp：**

Blackmagic Design公司生产sdi卡，有blackmagic 8K Pro、blackmagic Intensity Pro 4K等型号

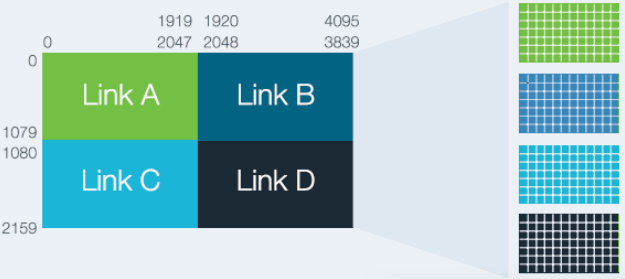
sdi的csp：发送时喂给blackmagic sdi卡uyvy或v210数据，接收时卡根据指定的参数吐出v210或者uyvy格式的数据



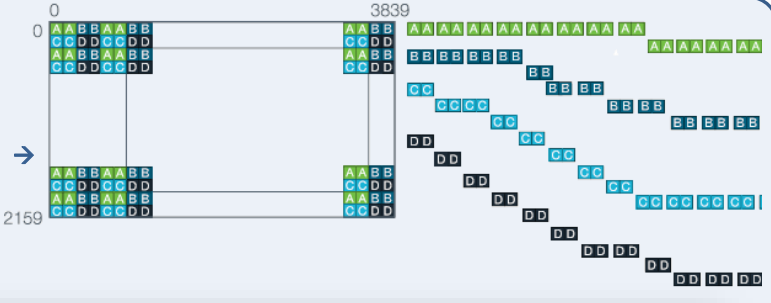


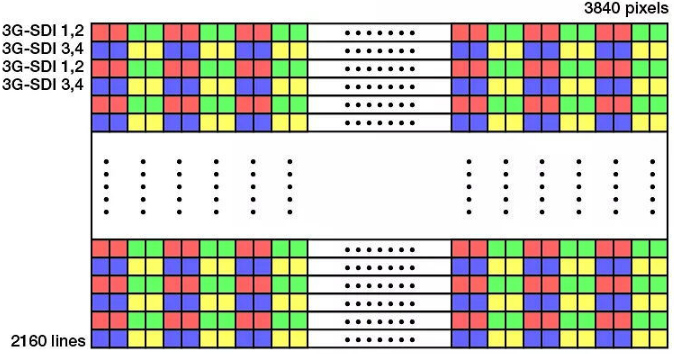
**分隔\交织模式：**

sdi的分割模式（square division mode）:



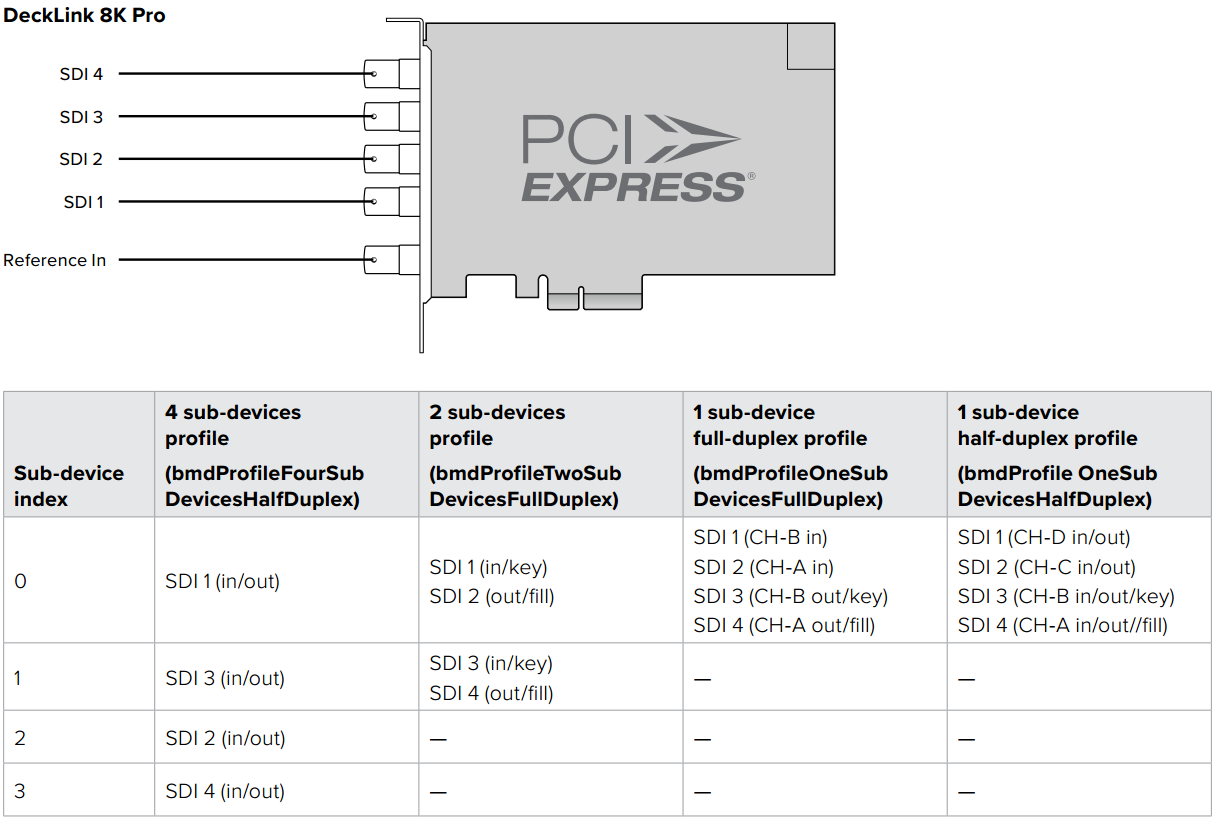
sdi的交织模式（sample Interleave）：





DeckLink 8K Pro卡支持分割（或交织）模式，设置profile为OneSubDeviceHalfDuplex、 bmdDeckLinkConfigQuadLinkSDIVideoOutputSquareDivisionSplit=true（或false）

8K Pro卡分割\交织模式下，1口是右下角，2口是左下角，3口是右上角，4口是左上角，声音在4口



**blackmagic驱动原理：**

调用IDeckLinkOutput::EnableVideoOutput时，必须指定显示模式BMDDisplayMode，这是输出SDI（或HDMI）信号所必需的。

实际如果以低于DisplayMode指定的速率调用DisplayVideoFrameSync()方法，则DeckLink驱动程序仍将以DisplayMode指定的确切速率发送视频，如果新帧未按时进入，驱动程序将简单地重复发送前一帧。如EnableVideoOutput时指定的是1080p50模式，实际以30fps或25fps等速率调用1080p视频帧的DisplayVideoFrameSync。

GetHardwareReferenceLock时间戳基于DeckLink驱动程序捕获的系统时间。它不考虑卡捕获帧的时间延迟，也不考虑通过PCIe的系统延迟。因此，GetHardwareReferenceLock应仅用于长期平均流时间或评估回调函数处理可能产生的潜在瓶颈。

The GetHardwareReferenceClock timestamps are based on the system time captured by the DeckLink driver. It does not account for time delays for the card to capture the frame, nor system delays via PCIe. As such GetHardwareReferenceClock should only be used for long term average stream times or assessing potential bottlenecks that may arise from callback function handling.

IDeckLinkOutput::GetHardwareReferenceClock方法返回一个时钟，该时钟锁定在 DeckLink 硬件输出帧的速率上

**HDCP：**

"So if you are trying to capture from a commercial Blu-ray or TV set-top box then BM devices won't capture direct as the content contains a copy protection flag. There are ways round this by using certain HDMI splitters"

Blackmagic devices have always killed hdcp embedded video.

Typical work around is to find an hdmi splitter that strips out the HDCP, and place it before the blackmagic device.

As far as I understand BM cards send non-HDCP protected signal (as there is no point to protect it as it's not a licensed content)

**PTP时钟同步：**

## 时钟同步含义：

时钟同步包括两个概念：

1. 频率同步（Frequency synchronization）：也叫时钟同步，是指信号之间在频率上保持某种严格的特定关系
2. 相位同步（Phase synchronization）：也叫时间同步，是指信号之间的频率和相位都保持一致，即信号之间的相位差恒为零。



如上图，有两个表Watch A与Watch B，如果这两个表的时间每时每刻都保持一致，这个状态就是时间同步；如果这两个表的时间不一致，但保持一个恒定的差值（如图中的Watch B总比Watch A晚6个小时），这个状态就是频率同步。

## PTP协议：

PTP（Precision Time Protocol，精确时间协议）是一种时间同步（相位同步）协议。

PTP同步精度可以达到us级。NTP（网络时间协议）一般只能达到ms级。

PTP一般采用硬件时间戳。网卡、交换机需要支持硬件时间戳，这种专用硬件可以降低PTP消息传输延迟，大大提高时间同步的准确性。

虽然也可以在网络中使用不支持PTP的硬件设备（如不支持PTP的交换机），但这通常会导致抖动增加或在延迟中引入不对称性，从而导致同步不准确。

1. 硬件支持的PTP： 网卡有自己的板载时钟，用于为接收和传输的PTP消息添加时间戳。这个板载时钟与PTP主机同步，计算机的系统时钟与板载时钟同步。即直接在MAC层进行PTP协议包分析，不经过UDP协议栈，网卡在PTP数据包发送、接收的确切时刻标记数据包，提高同步的精确度
2. 软件支持的PTP：系统时钟用于对PTP消息打时间戳，并直接与主站同步。PTP协议承载在UDP上，软件采用SOCKET收发UDP包，精度较低

## PTP同步原理：

主、从时钟之间交互同步报文并记录报文的收发时间，通过计算报文往返的时间差来计算主、从时钟之间的往返总延时。如果网络是对称的（即两个方向的传输延时相同），则往返总延时的一半就是单向延时，这个单向延时便是主、从时钟之间的时钟偏差。从时钟按照该偏差来调整本地时间，就可以实现其与主时钟的同步。

PTP协议定义了两种传播延时测量机制：请求应答机制（Requset\_Response）和端延时机制（Peer Delay），且这两种机制都以网络对称为前提。

请求应答机制原理如下：



1. 主时钟向从时钟发送Sync报文，并记录发送时间t1；从时钟收到该报文后，记录接收时间t2。
2. 从时钟向主时钟发送Delay\_Req报文，用于发起反向传输延时的计算，并记录发送时间t3；主时钟收到该报文后，记录接收时间t4。
3. 主时钟收到Delay\_Req报文之后，回复一个携带有t4的Delay\_Resp报文。

此时，从时钟便拥有了t1～t4这四个时间戳，由此可计算出主、从时钟间的往返总延时为[(t2–t1)+(t4–t3)]，由于网络是对称的，所以主、从时钟间的单向延时为[(t2–t1)+(t4–t3)]/2。因此，从时钟相对于主时钟的时钟偏差为：Offset=(t2–t1)-[(t2–t1)+(t4–t3)]/2=[(t2–t1)-(t4–t3)]/2。

## PTP域：

我们将应用了PTP协议的网络称为PTP域，如下图：



如上图所示，PTP域中所有的时钟节点都按一定层次组织在一起，整个域的参考时间就是最优时钟（Grandmaster Clock，GM），即最高层次的时钟。通过各时钟节点间PTP协议报文的交互，最优时钟的时间最终将被同步到整个PTP域中，因此也称其为时钟源。

最优时钟可以通过手工配置静态指定，也可以通过BMC（Best Master Clock，最佳主时钟）协议动态选举

PTP域中的节点称为时钟节点，PTP协议定义了三种类型的时钟节点：

1. OC（Ordinary Clock，普通时钟）：该时钟节点在同一个PTP域内只有一个PTP端口参与时间同步，并通过该端口从上游时钟节点同步时间。此外，当时钟节点作为时钟源时，可以只通过一个PTP端口向下游时钟节点发布时间，我们也称其为OC。
2. BC（Boundary Clock，边界时钟）：该时钟节点在同一个PTP域内拥有多个PTP端口参与时间同步。它通过其中一个端口从上游时钟节点同步时间，并通过其余端口向下游时钟节点发布时间。此外，当时钟节点作为时钟源时，可以通过多个PTP端口向下游时钟节点发布时间的，我们也称其为BC。
3. TC（Transparent clock，透明时钟）：TC不与其它时钟节点保持时间同步。TC有多个PTP端口，但它只在这些端口间转发PTP协议报文并对其进行转发延时校正，而不会进行自身的时间同步。TC包括以下两种类型：

* E2ETC（End-to-End Transparent Clock，端到端透明时钟）：直接转发网络中非P2P（Peer-to-Peer，点到点）类型的协议报文，并参与计算整条链路的延时。
* P2PTC（Peer-to-Peer Transparent Clock，点到点透明时钟）：只直接转发Sync报文、Follow\_Up报文和Announce报文，而终结其它PTP协议报文，并参与计算整条链路上每一段链路的延时。

**st2110：**

SMPTE：电影和电视工程师协会，the Society of Motion Picture and Television Engineers，

SMPTE 2110由SMPTE推出，定义了广播电视无压缩信号的IP标准



ST 2022-6 相当于 ST 2110 的前代技术，2110 弥补了 2022-6 的一些不足。

两者最主要的差别在于，ST 2022-6 就是单纯地把SDI信号封装进了IP包里，视频、音频、辅助数据都在同一个包里；而ST 2110把SDI信号中的视频、音频、辅助数据分开了，分别封装进不同的IP包里传输

比起2022-6的单一流，同步三种不同的流要难得多，所以 2110 采用了 PTP协议与RTP时间戳等方式同步

st2110由一系列标准组成，包括总则2110-10、视频2110-20、音频2110-30、辅助区2110-40

• ST 2110-10 System and Timing

• ST 2110-20 Uncompressed Video

• ST 2110-21 Video Stream Packet Shaping

• ST 2110-22 Constant Bit-Rate Compressed Video

• RP 2110-23 Single Video Essence Transport over Multiple ST 2110-20 Streams

• ST 2110-30 Uncompressed Audio

• ST 2110-31 AES3 Audio Streams

• ST 2110-40 Ancillary Data

st2110-10：

media clock与设备内部时钟要频率同步

media clock采样值增加1的速率，是由媒体参数决定的（视频是9k，音频则是采样率，在sdp中指定）

rtp clock和media clock完全一致，rtp报文头中的时间戳字段即对rtp clock的采样

sdp中指定公共参考时钟（common reference clock）：

“a=ts-refclk:ptp=IEEE1588-2008:39-A7-94-FF-FE-07-CB-D0:37”

如果公共参考时钟不可用，则可以指定使用设备内部时钟，这样来自这个设备的流间依旧可以互相同步：

”a=ts-refclk:localmac=7C-E9-D3-1B-9A-AF”

音视频raw数据通过rtp传送，传输层必须使用udp

mtu是1500，ifconfig可以查看；ip报文头固定部分20字节（ipv4）或40字节（ipv6）。因此st2110-10规定Standard UDP Size Limit=1460，即整个udp报文的最大长度为1460字节

Extended UDP Size Limit是8960字节，使用时必须在sdp中指定媒体类型参数MAXUDP

**st2110-20：**



SRD（Sample Row Data）=SRD Header + SRD Segment

一行数据可以被封装在多个SRD中

一行数据可能跨rtp包

一个rtp包最多只能封装一帧或一场

一个rtp包最多只能有三个SRD

pgroup: a pgroup is the minimal group of samples that align to an octet boundary

一个pgroup中的采样点不能跨图像行，封装时可以跨SRD，不能跨rtp包



对帧编，同一帧的所有rtp包的rtp时间戳相同，相邻帧的时间戳差值为90k/fps

对场编，同一场的所有rtp包的rtp时间戳相同，上下两场的时间戳差值是0.5\*90k/fps

sdp示例：

m=video 8888 RTP/AVP 96

a=rtpmap:96 raw/90000

a=fmtp:96 sampling=YCbCr-4:2:2; width=1280; height=720; exactframerate=60000/1001; depth=10; TCS=SDR; colorimetry=BT709; PM=2110GPM; SSN=ST2110-20:2017

其中：

TCS为Transfer Characteristic System，取值SDR、PQ、HLG、LINEAR等，默认SDR

PM为packing mode，取值2110GPM和2110BPM两种

SSN为SMPTE Standard Number，这里固定为ST2110-20:2017

在sdp中a=fmtp:开头的行里可以指定像素宽高比sar

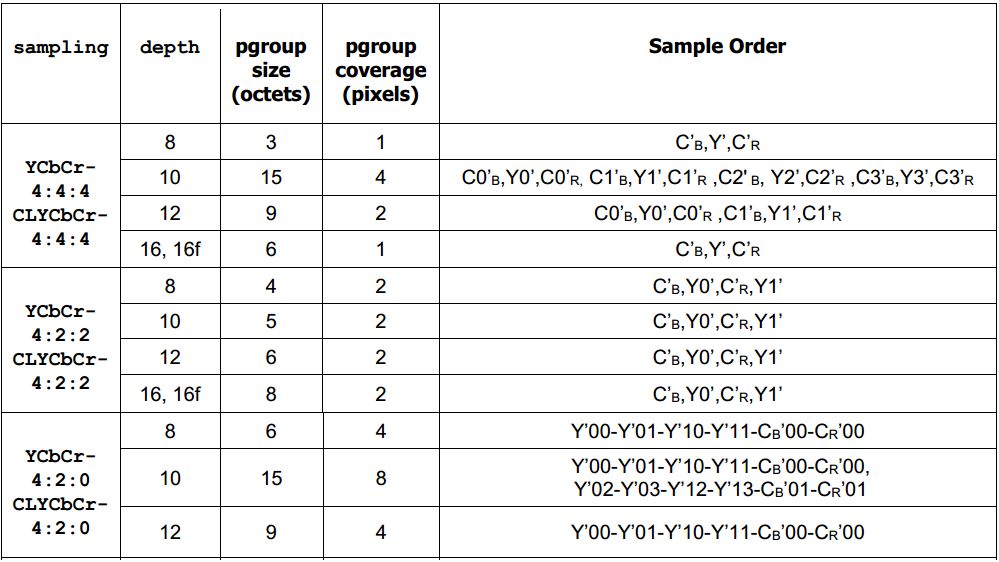
打包模式分为GPM和BPM两种

GPM：可以使用Standard 或Extended UDP Size Limit，rtp包的大小应尽可能接近之

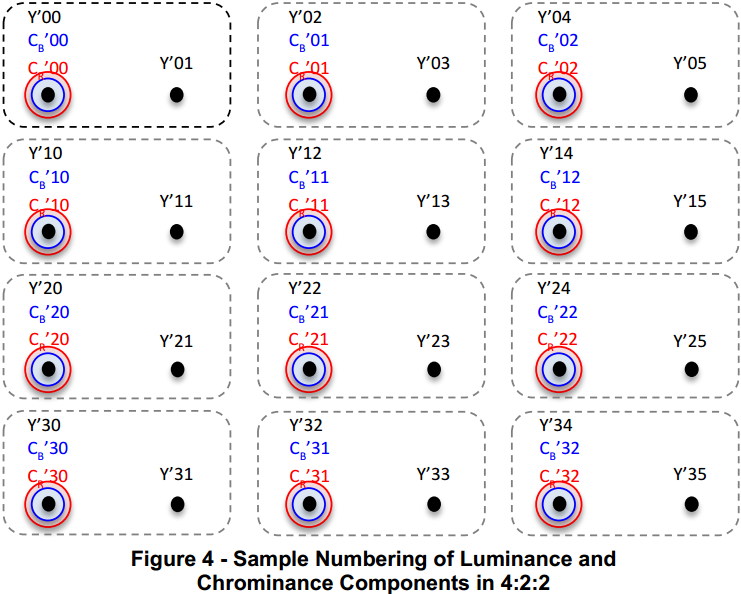
BPM：只能使用Standard UDP Size Limit（1460），rtp包中SRD Segments的总字节数必须是180的整数倍，因此即为180\*7=1260字节

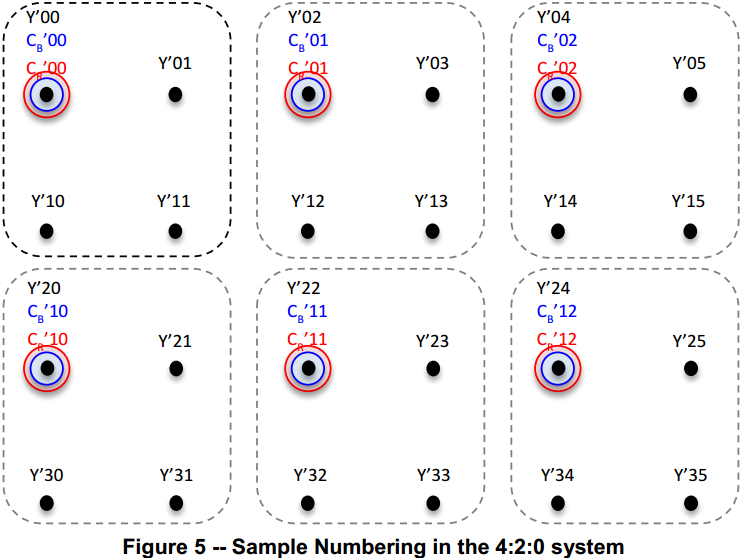
st2110支持rgb、yuv等像素格式

对yuv系列的csp，都是交错模式，相邻像素间无填充，即depth为10bit时，实际占10bit而非16bit。



'Coverage' in the construction tables below refers to a contiguous portion of the active image array in pixels. This may refer to an adjacent line within a field or frame





When packetizing progressive scan 4:2:0 Y’C’BC’R or ICTCP video data segments, samples from two consecutive luminance sample rows shall be included in each pgroup. The sample row number in the Sample Row Data Header is set to that of the first luminance sample row of the pair

示例：

yuv42210bit的高清图像：

2个像素一组，占5字节

每行图像4800字节，共1080行，一帧共5184000字节

一个mac包承载1260字节，加上包头共1260+62=1322字节，一个mac帧中存放一个rtp包

5148000字节可以放入4114个长度为1260的包+1个长为360的包

s->st20\_pkt\_info[ST20\_PKT\_TYPE\_NORMAL]={size = 1322, number = 3086}

s->st20\_pkt\_info[ST20\_PKT\_TYPE\_EXTRA]={size = 1328, number = 1028} //注意：1080-1028=52

s->st20\_pkt\_info[ST20\_PKT\_TYPE\_FRAME\_TAIL]={size = 422, number = 1}

1个mac包里存放252个pg

1行是960个pg，占4个mac包，前3个mac包放满252个pg，最后一个mac只放204个pg，还剩48个pg的剩余空间放第二行，所以这个mac包多了6个字节

每累积21行的数据共100800=21\*4800字节，此时可以恰好存放到100800/1260=80个包里

1080/21=51.4

**st2110场源**

rtp头里的M位，For progressive scan video, the marker bit shall be set to 1 to denote when this RTP packet is the last packet carrying video essence data for a video frame.  
For interlaced video, the marker bit shall be set to 1 to denote when this RTP packet is the last packet carrying video essence data for a video field.

rtp头里的时戳，

All RTP packets which are part of the same progressive frame shall contain the same RTP Timestamp value.  
All RTP packets which are part of the same interlaced field shall contain the same RTP Timestamp value.

rtp头里的Field Identification，

The Field Identification bit shall be set to 0 to signal data in the (temporally) first field and set to 1 to signal data in the second field.

For progressive scan data the Field Identification bit shall be set to zero

rtp头里的SRD Row Number：

shall start at 0 at the top of the image. In the case of interlaced images, each field shall start at 0 at the top of the field。SRD Row Number shall only increase within the field or frame (sample rows are sent in order from top to bottom)。

两场的发送顺序。The fields of an interlaced image are transmitted in time order, first field first

For interlaced systems，if the height is odd, the temporally first field (or segment) shall contain one more line than the temporally second field

In progressive systems, a single RTP packet shall not contain samples from more than one frame.  
In interlaced systems, a single RTP packet shall not contain samples from more than one field.

'Coverage' in the construction tables below refers to a contiguous portion of the active image array in pixels.

This may refer to an adjacent line within a field or frame.

The 4:2:0 sampling system shall only be applied to progressive scan images.而422、444无限制。

**st2110-30：**

L16音频即s16原始pcm

L24音频即s24原始pcm

收发必须支持48kHz，应该支持96和44.1 kHz的采样率

When operating at 48 kHz sampling rate:

* Receivers shall support both L16 and L24 encodings
* Senders shall support either L16, or L24, or both encodings

When operating at 96 kHz sampling rate:

* Both senders and receivers shall support L24 encoding

When operating at 44,1 kHz sampling rate:

* Both senders and receivers shall support L16 encoding

接收端至少需要支持：48kHz + 1~8声道 + 1ms包时长

Packet time is the real-time duration of the media data contained in a media packet.

Given the sampling rate and packet time, the number of samples per packet can be calculated

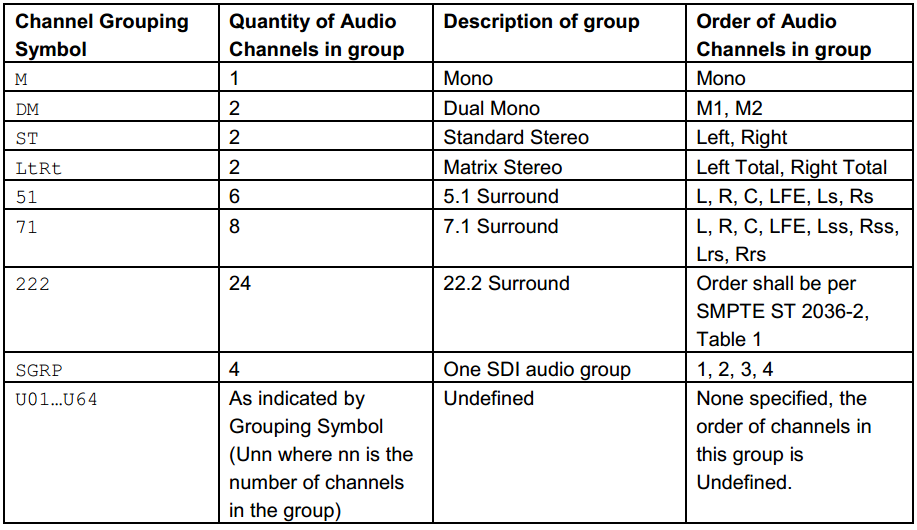
音频必须使用Standard UDP Size Limit（1460）

sdp示例：

a=rtpmap:98 L16/48000

a=fmtp:101 channel-order=SMPTE2110.(51,ST)

如上sdp示例，音频采样格式是s16，采样率48000kHz，含两个音频轨，一个是5.1声道，另一个是双声道



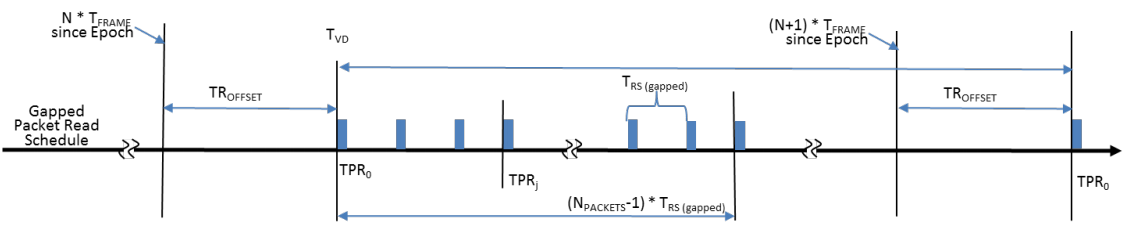
**st2110-21：**

**GPRS和LPRS：**

水平消隐：将光信号扫描为电信号的过程中，当扫描点到达图像右侧边缘时，扫描点快速返回左侧进行下一行扫描，行与行之间的返回过程称为水平消隐；

垂直消隐：将光信号扫描为电信号的过程中，扫描完一帧后，要从图像的右下角返回到左上角开始新一帧的扫描，这一时间间隔，叫做垂直消隐。

（1）Gapped Packet Read Schedule (GPRS)：rtp packet的传输存在gap，对应sdi的垂直消隐



TFrame：相邻帧的时间间隔

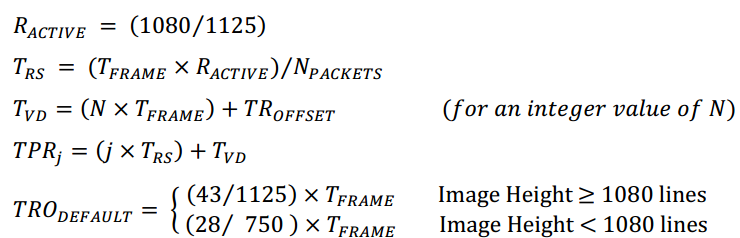
TRS：同一帧内相邻两个rtp包的时间间隔

RACTIVE：the ratio of active time to total time within the frame period

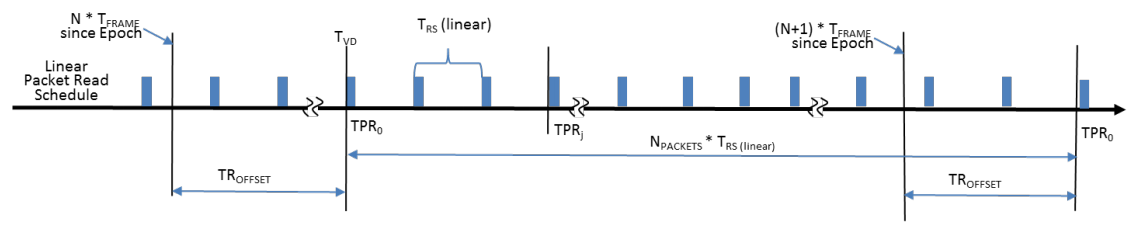
NPACKETS：一帧包含多少个rtp包

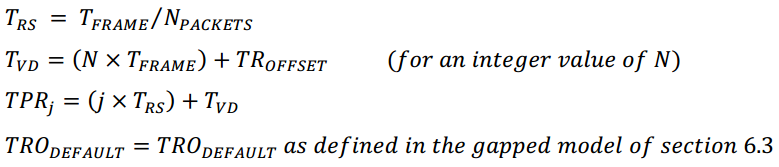
TPRj：帧内j号rtp包的传送时间

TVD：帧内第一个rtp包的传送时间，也即TPR0



（2）Linear Packet Read Schedule (LPRS)：所有rtp packet均匀发送





**Network Compatibility模型：**

A network compatibility model regulates the burst characteristics of senders in order to promote compatibility with a wide variety of switches with varied buffer sizes

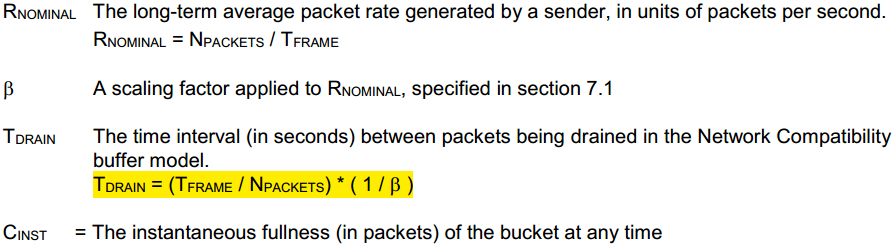


sender后接一个无限容量的漏桶

sender发出的packet进入漏桶，漏桶每TDRAIN秒流完一个packet

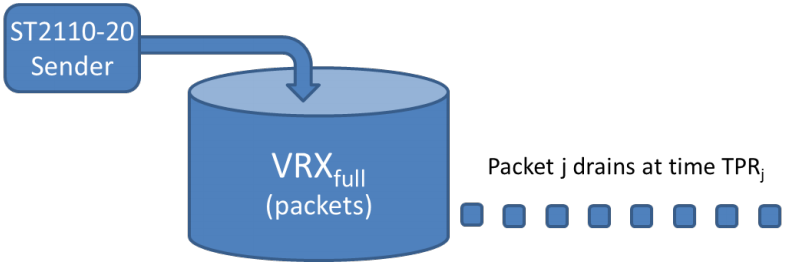
CINST表示任意时刻，漏桶中的包数目

CINST必须小于等于CMAX，其中CMAX由sender类型决定



**Virtual Receiver Buffer模型：**

packets are deposited at the actual moment of transmission, and removed on a specific schedule. Two fundamental Packet Read Schedules (PRS) are defined, a Gapped schedule and a Linear schedule



sender后接一个容量为VRXfull的漏桶

packet进入和流出漏桶不耗时，都是瞬间完成

j号packet离开漏桶的时间为Packet Read Schedules中定义的TPRj

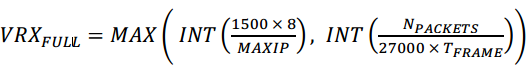
漏桶不能上溢、不能下溢

如果TPRj时刻，j号packet还没有进入漏桶，即下溢

VRXfull表示漏桶的容量，可容纳多少个packet，取值由sender类型决定

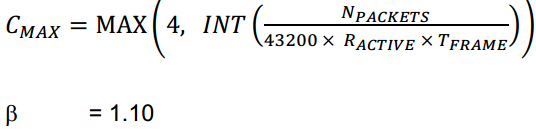
**Narrow Senders (Type N) ：**

narrow senders必须符合GPRS、Virtual Receiver Buffer模型、network compatibility模型：



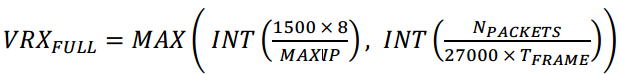
其中，TFrame单位为sec，即等于1/fps

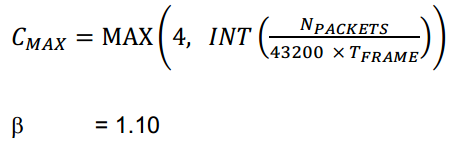
MAXIP为1460（当使用Standard UDP Size Limit时）或8960（当使用Extended UDP Size Limit时）



**Narrow Linear Senders (Type NL)：**

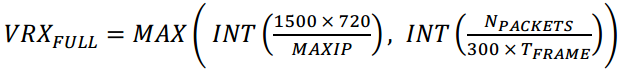
Narrow Linear Senders必须符合LPRS、virtual Receiver Buffer模型、Network Compatibility模型：

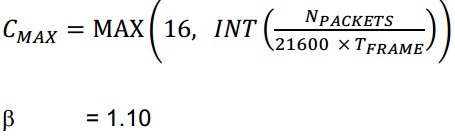




**Wide Senders (Type W)：**

wide senders必须符合LPRS、virtual Receiver Buffer模型、Network Compatibility模型：





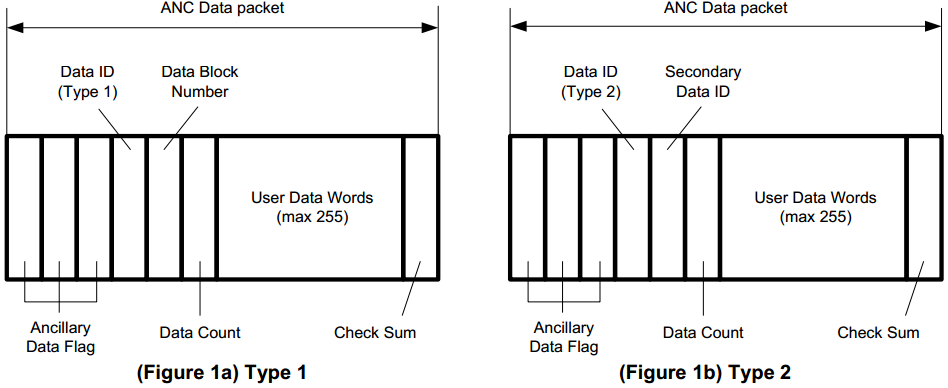
**st2110-40：**

辅助数据包(Ancillary Data Packet)的结构定义于SMPTE ST 291-1中

辅助数据包分为两种类型：Type 1和Type 2

Type 1和Type 2区别在于Type 2用SDID(secondary data id)字段取代了Type1的DBN(data block number)字段

ANC Data Packet由若干个word(字)构成，每个字10bit，具体如下：



**A Type 1 ancillary data packet shall be composed of:**

a) An ancillary data flag (ADF) word that marks the beginning of the ancillary data packet.

b) A data identification (DID) word that defines the user data format carried in user data words (UDW) in the ancillary data packet. DID共10bit，记为b9b8……b0，其中b7~b0是8bit的id，b8是b7~b0的even parity校验，b9是b8的取反。如果是Type 1 anc data packet，则b7为1，如果是Type 2 anc data packet，则b7为0。

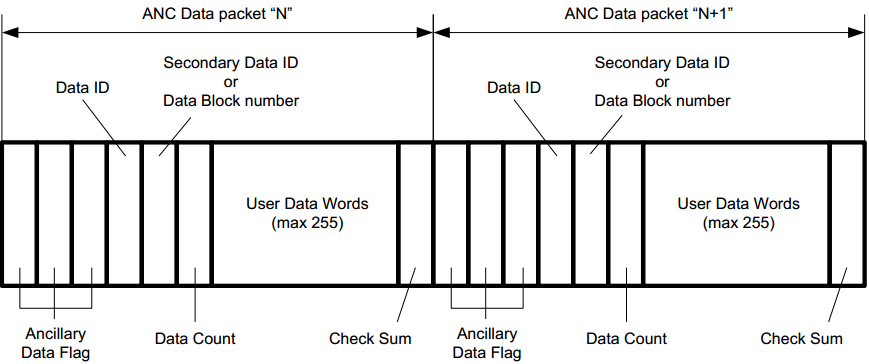
c) A data block number (DBN) word (for Type 1 only) that distinguishes successive ancillary data packets with a common data ID. DBN共10bit，记为b9b8……b0，其中b7~b0是8bit的block number，b8是b7~b0的even parity校验，b9是b8的取反。

d) A data count number (DC) word that defines the quantity of user data words in the ancillary data packet. DC的低8位取值0~255，表示其后UDW域的长度为0~255个字。

e) The user data words (UDW), comprising up to 255 words in each ancillary data packet, where the user data format is defined in a separate, specific application document.

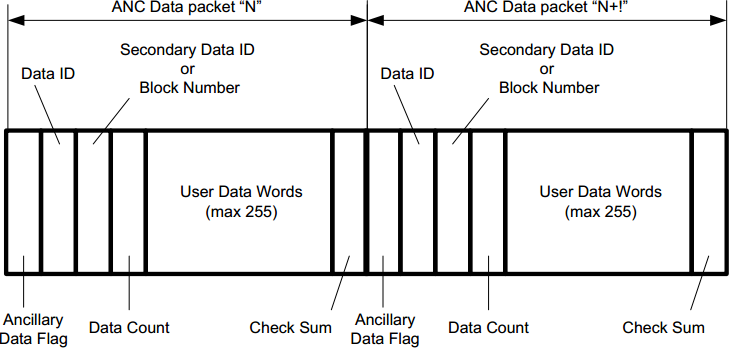
f) A checksum (CS) word.

**Component Interface ANC Data Packet Format：**



除非另有规定，否则ANC Data Flag占3个字：0x000 0x3FFh 0x3FF

**Composite Interface ANC Data Packet Format：**

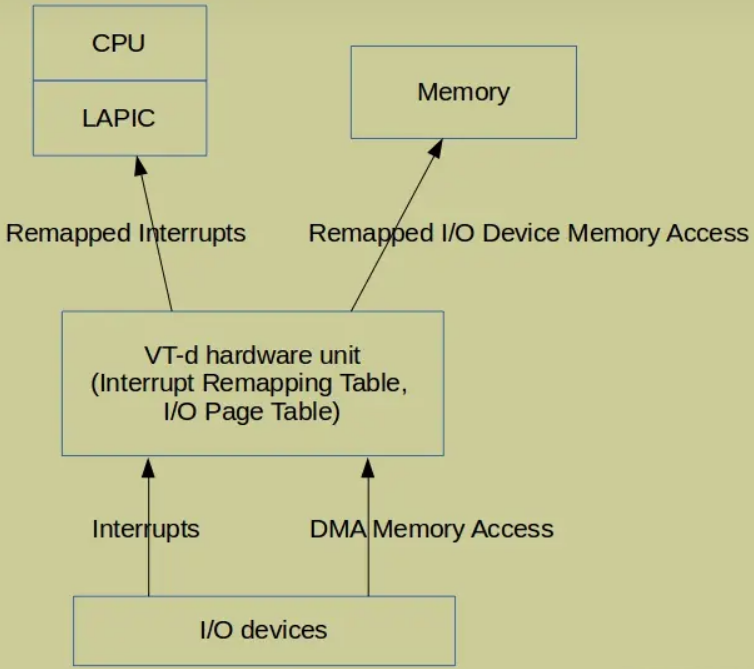


除非另有规定，否则ANC Data Flag占1个字：0x3FC

**Intel VT-d：**

Intel VT-d的全称是Intel Virtualization Technology for Direct I/O，它是Intel的主要针对I/O子系统的虚拟化技术。

VT-d是一个处于CPU、内存和I/O设备之间的硬件设备，通常位于PCI设备树的根部，或者I/O子系统的根部。VT-d会拦截位于它下面的所有I/O设备产生的中断请求和DMA内存访问请求，然后通过查找中断重定向表或者I/O页表的方式来重新定位中断转发的目标LAPIC或者是I/O设备访问的目标主机物理内存地址。如下图所示：



VMM（Virtual Machine Monitor）软件负责I/O设备的分配，即将指定I/O设备和相应的VM对应起来，并且负责建立中断重定向关系表和I/O地址转换页表，并将这些转换关系的配置设置到VT-d硬件设备上。I/O设备发起的中断请求或者DMA内存访问请求中带有相应设备的ID，这样VT-d硬件单元就可以通过硬件查找的方式将请求重定向到相应的VM上，从而达到隔离不同VM的I/O设备的目的。

**中断绑核：**

自系统启动后，每个cpu上运行的中断数目：cat /proc/interrupts

部分inactive interrupts不被/proc/interrupts显示，可以用/proc/irq查看

对中断0和中断2，内核不允许修改其cpu亲和性

示例，将7号中断绑定到0~3核上：echo "15" > /proc/irq/7/smp\_affinity，其中15即mask 0x1111。

中断绑核设置写入/etc/rc.d/rc.local，这样系统启动后就会生效

查看网卡中断的绑核情况：

cat /proc/interrupts | grep [网卡名] | cut -d: -f1 | while read i; do echo -ne irq":$i\t bind\_cpu: "; cat /proc/irq/$i/smp\_affinity\_list; done | sort -n -t' ' -k3

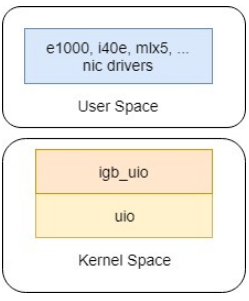
**dpdk：**

DPDK（数据平面开发套件）是INTEL公司开发的一款高性能的网络驱动组件，技术有用户态、轮询取代中断、零拷贝、网卡RSS、访存DirectIO等

DPDK在huge page的基础上实现了特有的内存管理机制。DPDK提供的网卡驱动将收到的报文通过DMA的方式映射到huge page对应的内存中，并抽取报文对应的报文指针，在应用程序和驱动之间维护一套无锁的环形队列来管理这些报文指针。应用程序只需要调用相应的接口，从队列中提取相应的报文进行处理，完成后，再将报文指针放到发送队列中，由驱动程序进行发送。在处理报文的整个过程中，不需要对报文进行任何拷贝，所有操作都依赖于报文指针，可以从整个报文生命周期上，解决内存拷贝问题，实现内存零拷贝。

由于不需要进行频繁的内存拷贝和系统调用，DPDK裸包反弹每个包需要80个时钟周期，而传统Linux内核协议栈每包需要2k~4k个时钟周期

PMD即Poll Mode Driver，即基于用户态的轮询机制的驱动



虽然PMD是在用户态实现设备驱动，但还是依赖于内核提供的策略。其中uio模块，是内核提供的用户态驱动框架，而igb\_uio是DPDK kit中拥有与uio交互，bind指定网卡的内核模块

**修改grub文件：**

修改/etc/default/grub文件，将其中“GRUB\_CMDLINE\_LINUX=”开头的一行，修改为

GRUB\_CMDLINE\_LINUX="crashkernel=auto rd.lvm.lv=centos/root rd.lvm.lv=centos/swap rhgb quiet rcu\_nocbs\_poll audit=0 selinux=0 nmi\_watchdog=0 nosoftlockup transparent\_hugepage=never intel\_iommu=on iommu=pt"

更新grub配置文件：

* 如果是bios启动，则grub2-mkconfig -o /boot/grub2/grub.cfg
* 如果是uefi启动，则grub2-mkconfig -o /boot/efi/EFI/centos/grub.cfg

重启服务器

验证GRUB\_CMDLINE\_LINUX是否生效：dmesg | grep Kernel、cat /proc/cmdline查看

验证iommu=on iommu=pt是否生效：ll /sys/kernel/iommu\_groups/

注意：在启用了UEFI的系统上，CentOS 7的GRUB的有效配置文件是 /boot/efi/EFI/centos/grub.cfg。/boot/grub与/boot/grub2目录下已经不再保存配置文件 。

**stdpdk硬件环境：**

1. 网卡驱动为ice或i40e
2. bios和cpu支持vt-d、vt-x

网卡命令：

ifdown后网卡不能被ping通

查看网卡信息：

service NetworkManager start

查看是否为万兆网卡：ethtool eno1，如果Supported link modes中有10000baseT/Full，则表示支持万兆

查看网卡厂商：lspci |grep -i Ethernet

04:00.0 Ethernet controller: Broadcom Inc. and subsidiaries NetXtreme BCM5720 2-port Gigabit Ethernet PCIe

04:00.1 Ethernet controller: Broadcom Inc. and subsidiaries NetXtreme BCM5720 2-port Gigabit Ethernet PCIe

3b:00.0 Ethernet controller: Broadcom Inc. and subsidiaries NetXtreme II BCM57810 10 Gigabit Ethernet (rev 10)

3b:00.1 Ethernet controller: Broadcom Inc. and subsidiaries NetXtreme II BCM57810 10 Gigabit Ethernet (rev 10)

如上，系统中共四块网卡，Broadcom（博通）公司生产，前两块型号为BCM5720 2-port千兆卡，后两块型号为BCM57810万兆卡

**stdpdk编译：**

修改192.165.153.195的ip：ifconfig em2 192.165.153.200 netmask 255.255.255.0，执行完直接ssh断了，虽然新旧ip在同一网段，但没有了默认网关

dpdk官网，可查看\下载版本\文档：<http://core.dpdk.org/download/>

**stdpdk运行：**

cpupower -c all frequency-set -g performance

tuned-adm profile latency-performance

sysctl -w vm.nr\_hugepages=8192

激活gcc8.3.1：scl enable devtoolset-8 bash

检查内核版本：uname -r

检查gcc版本：gcc -v

检查vt-d是否开启：dmesg | grep DMAR

修改/etc/default/grub：

GRUB\_CMDLINE\_LINUX="crashkernel=auto rd.lvm.lv=centos/root rd.lvm.lv=centos/swap rhgb quiet isolcpus=51,52,55 nohz\_full=51,52,55 rcu\_nocbs=51,52,55 rcu\_nocbs\_poll audit=0 selinux=0 nmi\_watchdog=0 nosoftlockup transparent\_hugepage=never intel\_iommu=on iommu=pt"

更新grub配置文件：grub2-mkconfig -o /boot/grub2/grub.cfg或grub2-mkconfig -o /boot/efi/EFI/centos/grub.cfg

reboot

检查GRUB\_CMDLINE\_LINUX设置是否生效：

cat /proc/cmdline

ll /sys/kernel/iommu\_groups/

绑定网卡到dpdk pmd模式：./script/nicctl.sh bind\_pmd 0000:3d:00.1 （10.10.40.121的eno2网卡bdf）

查看网卡状态：/usr/local/bin/dpdk-devbind.py -s

Network devices using DPDK-compatible driver：

0000:3b:00.0 'NetXtreme II BCM57810 10 Gigabit Ethernet 168e' drv=vfio-pci unused=bnx2x

Network devices using kernel driver：

0000:04:00.0 'NetXtreme BCM5720 2-port Gigabit Ethernet PCIe 165f' if=em1 drv=tg3 unused=vfio-pci \*Active\*

0000:04:00.1 'NetXtreme BCM5720 2-port Gigabit Ethernet PCIe 165f' if=em2 drv=tg3 unused=vfio-pci \*Active\*

0000:3b:00.1 'NetXtreme II BCM57810 10 Gigabit Ethernet 168e' if=p2p2 drv=bnx2x unused=vfio-pci \*Active\*

运行示例

./build/app/RxTxApp --config\_file config/test\_tx\_1port\_1v.json

其中test\_tx\_1port\_1v.json的内容如下：

{

"interfaces": [

{

"name": "0000:3d:00.1",

"name-bak": "eno2",

"ip": "10.10.41.121"

}

],

"tx\_sessions": [

{

"dip": [

"10.10.41.126"

],

"interface": [

0

],

"video": [

{

"replicas": 1,

"type": "frame",

"pacing": "gap",

"packing": "BPM",

"start\_port": 8888,

"payload\_type": 96,

"tr\_offset": "default",

"video\_format": "i1080p25",

"pg\_format": "YUV\_422\_10bit",

"video\_url": "/home/zcj/stdpdk/stdpdk22.09版本/Media-Transport-Library-22.09/yuv422rfc4175be10\_1920x1080\_60sec.yuv"

}

]

}

]

}

其中，

interfaces->name为网卡的bdf端口，也即ethtool -i eno2查询到的bus-info

interfaces->ip指定udp连接的src ip

tx\_sessions->dip为发送udp到的对端ip，stdpdk会发送arp请求查询该ip对应的mac地址

tx\_sessions->video-> video\_format为yuv格式，取值范围见parse\_json.c中：

st\_app\_parse\_json -> st\_json\_parse\_tx\_video -> st\_video\_fmt\_desc数组

{

.fmt = VIDEO\_FORMAT\_1080P\_25FPS,

.name = "i1080p25",

.width = 1920,

.height = 1080,

.fps = ST\_FPS\_P25,

}

gdb执行程序，直接设置main函数参数，并自动加载断点：

gdb -ex "set breakpoint pending on" -x br.txt -ex run --args ./build/app/RxTxApp --config\_file config/test\_tx\_1port\_1v.json

**stdpdk源码分析：**

app\_tx\_video\_start\_source有个app\_tx\_video\_frame\_thread线程：

1)从st\_app\_tx\_video\_session->st\_tx\_video\_session\_handle\_impl (handle)->st\_tx\_video\_session\_impl (impl)->st20\_frames[2]帧队列中取出一帧(st\_frame\_trans类型)

2)从硬盘文件中读取一帧的字节数拷贝到st\_frame\_trans->addr字段，并设置st\_app\_tx\_video\_session->framebuffs[i]，其是st\_tx\_frame类型，包含size、free\ready\tansmiting状态

**st20\_get\_bandwidth\_bps：**

st20\_get\_bandwidth\_bps (int width, int height, enum st20\_fmt fmt, enum st\_fps fps,uint64\_t\* bps)：

先根据宽、高、csp、fps计算码率，然后除以ractive，ractive为1080/1125

st2110-21中Gapped Packet Read Schedule规定p包传输存在gap，对应sdi的垂直消隐，并定义

ractive表示ratio of active time to total time within the frame period，值为1080/1125

示例：

1. 如对于1920x1080, 422\_10BIT, 59.94fps，流量为bps=((uint64\_t)1920x1080x2x10x60000/1001）/(1080.0/1125)= 2589410588bps
2. 如stdpdk中每个schedule默认的流量配额=12 \* st20\_1080p59\_yuv422\_10bit\_bandwidth\_mps()=12\*(uint64\_t)(2589410588/1000/1000)= 31068mbps

**void\* rte\_zmalloc\_socket：**

void\* rte\_zmalloc\_socket(const char \*type, size\_t size, unsigned align, int socket)：

指定numa结点上分配指定大小的内存，并初始化为0

**rte\_pktmbuf\_alloc\_bulk：**

int rte\_pktmbuf\_alloc\_bulk(struct rte\_mempool \*pool, struct rte\_mbuf \*\*mbufs, unsigned count)

从内存池pool中分配count个rte\_mbuf

**lcore设置：**

eal：Environment Abstraction Layer

rte\_eal\_init设置绑核参数，如-l 10,11,12

解析绑核参数：eal\_parse\_args->eal\_parse\_common\_option->update\_lcore\_config

rte\_eal\_init将lcore的配置记录在rte\_config \*cfg = rte\_eal\_get\_configuration();

cfg->lcore\_role[i] = ROLE\_RTE; //i为指定的core，即10,11,12，未指定的核的lcore\_role为ROLE\_OFF

第一个核为main\_lcore，即cfg->main\_lcore=10，对调用rte\_eal\_init的当前线程进行pthread\_setaffinity\_np绑核到cfg->main\_lcore

其余核心即11、12分别创建一个线程，进行绑核，线程名为lcore-worker-<core号>

st\_start时调st\_dev\_get\_lcore获取一个核，11，设置给st\_sch\_impl->lcore

创建会话后，调用st20\_tx\_get\_sch\_idx获取一个scheduler，然后调st\_dev\_get\_lcore获取一个核，此时为12，绑定到用户的生产者线程

(gdb) p \*config

$24 = {main\_lcore = 10, lcore\_count = 1, numa\_node\_count = 2, numa\_nodes = {0, 1, 0 <repeats 30 times>}, service\_lcore\_count = 0, lcore\_role = {ROLE\_OFF,

ROLE\_OFF, ROLE\_OFF, ROLE\_OFF, ROLE\_OFF, ROLE\_OFF, ROLE\_OFF, ROLE\_OFF, ROLE\_OFF, ROLE\_OFF, ROLE\_RTE, ROLE\_OFF <repeats 117 times>},

process\_type = RTE\_PROC\_PRIMARY, iova\_mode = RTE\_IOVA\_VA, mem\_config = 0x7fffc67962c0 <early\_mem\_config>}

/\* internal configuration (per-core) \*/

struct lcore\_config lcore\_config[RTE\_MAX\_LCORE]; 数组下标为core索引

struct lcore\_config

{

pthread\_t thread\_id; /\*\*< pthread id，入口函数为eal\_thread\_loop，线程做了绑核，线程名为lcore-worker-<core索引>\*/

int pipe\_main2worker[2]; /\*\*< communication pipe with main，管道 \*/

int pipe\_worker2main[2]; /\*\*< communication pipe with main，管道 \*/

lcore\_function\_t \* volatile f; /\*\*< function to call \*/

void \* volatile arg; /\*\*< argument of function \*/

volatile int ret; /\*\*< return value of function \*/

volatile enum rte\_lcore\_state\_t state; /\*\*< lcore state \*/

unsigned int socket\_id; /\*\*< physical socket id for this lcore \*/

unsigned int core\_id; /\*\*< core number on socket for this lcore \*/

int core\_index; /\*\*< relative index, starting from 0 \*/

uint8\_t core\_role; /\*\*< role of core eg: OFF, RTE, SERVICE \*/

rte\_cpuset\_t cpuset; /\*\*< cpu set which the lcore affinity to \*/

};

st\_dev\_get\_lcore(st\_main\_impl\* impl, unsigned int\* lcore)：

在用户设置的lcore列表里获取一个核，该核必须满足：

(1)不是main lcore

(2)和ST\_PORT\_P网卡位于同一个numa节点；

(3)处于空闲状态，是否空闲记录在st\_main\_impl->st\_lcore\_shm->lcores\_active[]数组中，数组下标为core号

**sch->cpu\_busy：**

在admin\_tx\_video\_migrate函数中遍历每一个sch，遍历sch中的每一个session，如果某一个session的cpu\_busy\_score大于95，则将所属的sch的cpu\_busy设置为true

session的cpu\_busy\_score的计算方法：cpu\_busy\_score=100 - 100.0 \* nic\_inflight\_cnt / nic\_burst\_cnt

**schedule：**

st\_sch\_add\_quota(struct st\_sch\_impl\* sch, int quota\_mbs)：

为sch添加流量quota\_mbs；

如果添加后总流量超过data\_quota\_mbs\_limit则添加失败

如果添加后总流量超过quota\_mbs\_max\_for\_sleep，则设置sch->allow\_sleep = false禁用sleep模式

st\_sch\_impl\* sch\_request(struct st\_main\_impl\* impl, enum st\_sch\_type type)：

在impl->sch\_mgr->sch[]数组中查找未激活的sch（即sch->active为false），并设置sch->type=type、sch->active=true；

调用sch\_request找到未激活的sch，并调用st\_sch\_add\_quota添加流量配额后，可以调sch\_start(st\_sch\_impl\* sch)启动这个sch

sch\_start(st\_sch\_impl\* sch)：

调用st\_dev\_get\_lcore(st\_main\_impl\*, &sch->lcore)为sch分配一个空闲lcore；

然后调用rte\_eal\_remote\_launch(sch\_tasklet\_func, sch, sch->lcore)让sch->lcore核启动执行函数sch\_tasklet\_func

然后设置sch->started=true

添加第一路视频session时，

impl->sch\_mgr->sch[0]->active为ture，其余sch[i]->active都为false

impl->sch\_mgr->sch[0]->cpu\_busy为false，

impl->sch\_mgr->sch[0]->data\_quota\_mbs\_limit为31068mb，流量配额上限

impl->sch\_mgr->sch[0]->quota\_mbs\_max\_for\_sleep=20812，如果当前流量超过该值，则禁止sleep模式，即设置sch->allow\_sleep = false

impl->sch\_mgr->sch[0]->data\_quota\_mbs\_total为0，当前总流量

st\_sch.c中，scheduler启动的时候，会让该sch绑定的core执行sch\_tasklet\_func函数

1. 遍历sch->tasklet[]数组，执行tasklet关联的任务，即调用tasklet->ops->handler(ops->priv)：
2. 实际上，调tvs\_tasklet\_handler->tv\_tasklet\_frame取一帧yuv（即st20\_tx\_ops->get\_next\_frame回调），然后调tv\_build\_pkt做成一个mac帧，放入环形缓冲区st\_tx\_video\_session\_impl->ring[0]，其是个rte\_ring类型
3. 或者，调video\_trs\_tsc\_tasklet从环形缓冲区取mac帧，然后调用rte\_eth\_tx\_burst发送出去，并回调st20\_tx\_ops->tv\_frame\_free\_cb
4. 或者，调cni\_tasklet\_handlder接收ptp时钟包

**st20\_tx\_create源码分析：**

1. 调用st20\_get\_bandwidth\_bps()计算会话的流量占用
2. 调用st\_sch\_impl\* sch=st\_sch\_get(st\_main\_impl\*, quota\_mbs, ST\_SCH\_TYPE\_DEFAULT)将流量分配给一个sch，确保该sch是active、started、非cpu\_busy的，该sch可能是新的，也可能是复用的旧的
3. 调用st\_tx\_video\_sessions\_sch\_init()，完成两件事：
4. 一是调用tv\_mgr\_init()，初始化sch->st\_tx\_video\_sessions\_mgr。即创建一个新的tasklet(即st\_sch\_tasklet\_impl)，赋值给sch->st\_tx\_video\_sessions\_mgr->tasklet，并保存在sch->tasklet[]数组中，该tasklet的执行函数（即tasklet->ops->handler）为tvs\_tasklet\_handler。
5. 二是调用st\_video\_transmitter\_init()，初始化sch->st\_video\_transmitter\_impl。即创建一个新的tasklet，赋值给sch->st\_video\_transmitter\_impl->tasklet，并保存在 sch->tasklet[]数组中，该tasklet的执行函数为video\_trs\_tasklet\_handler
6. 调用tv\_mgr\_attach()->tv\_attach()，创建并初始化一个st\_tx\_video\_session\_impl，保存于st\_main\_impl->sch\_mgr->sch[i]->st\_tx\_video\_sessions\_mgr->sessions[]数组中。
7. tv\_attach()根据参数st20\_tx\_ops初始化st\_tx\_video\_session\_impl，具体地，依次调用tv\_init\_pkt、tv\_init\_sw、tv\_init\_hw、tv\_init\_hdr、tv\_init\_pacing等函数
8. （以下以bpm模式的YUV422\_10BIT HD为例，位于tv\_init\_pkt函数中）

* s->st20\_bytes\_in\_line = ops->width \* s->st20\_pg.size / s->st20\_pg.coverage=1920\*5/2=4800
* s->st20\_pkt\_len = ST\_VIDEO\_BPM\_SIZE=1260=180\*7 //bpm模式决定了每个rtp包的payload大小为1260字节，即封装的yuv大小，即504个像素
* s->st20\_pkt\_size = s->st20\_pkt\_len + sizeof(struct st\_rfc4175\_video\_hdr)=1260+62=1322; //一个mac帧的总大小，按照一个rtp包只封装一个SRD计算，故rtp包头为20字节数
* s->st20\_total\_pkts = ceil((double)s->st20\_frame\_size / s->st20\_pkt\_len)=4115; //一帧yuv大小为1920x1080\*2\*10/8=5184000字节
* s->st20\_pkt\_info[ST20\_PKT\_TYPE\_NORMAL]={size = 1322, number = 3086}; //一行1920个像素，3个rtp包可以放504\*3=1512个像素，还差408像素
* s->st20\_pkt\_info[ST20\_PKT\_TYPE\_EXTRA]={size = 1328, number = 1028}; //注意：1080-1028=52，第4个rtp包放2个SRD故其包头大小多了6字节
* s->st20\_pkt\_info[ST20\_PKT\_TYPE\_FRAME\_TAIL]={size = 422, number = 1}
* s->st21\_vrx\_narrow = RTE\_MAX(8, s->st20\_total\_pkts / (27000 / fps)); //8个pkt，即对于narrow 类型的sender，漏桶模型中漏桶的容量
* s->st21\_vrx\_wide = RTE\_MAX(720, s->st20\_total\_pkts / (300 / fps)); //720个pkt，对于wide类型的sender，漏桶模型中漏桶的容量
* s->bulk=4;
* s->ops = \*ops; //为st20\_tx\_ops类型
* s->st20\_src\_port[0]=8888
* s->st20\_dst\_port[0]=s->st20\_src\_port[0]=8888
* s->eth\_ipv4\_cksum\_offload[0]=true; //if the eth dev support ipv4 checksum offload
* s->eth\_has\_chain[0]=true; //if the eth dev support chain buff
* s->tx\_mono\_pool=false;
* s->ring\_count = ST\_TX\_VIDEO\_SESSIONS\_RING\_SIZE=512； //必须小于s->st20\_total\_pkts

1. tv\_init\_hdr函数会初始化mac层、ip层、udp层、rtp层报文头设置。该过程中会查询对端mac地址。
2. （以下以bpm模式的YUV422\_10BIT HD为例，位于tv\_init\_pacing函数中）

* double frame\_time = (double)1000000000.0 / fps=40\*1000000; //单位ns
* pacing->frame\_time = frame\_time; //每帧时长，单位ns
* pacing->frame\_time\_sampling = 90000 /fps=3600; //每帧时长，单位90k
* double ractive = 1080.0 / 1125.0; //标准中规定LPRS的参数
* pacing->tr\_offset = s->ops.height >= 1080 ? frame\_time \* (43.0 / 1125.0) : frame\_time \* (28.0 / 750.0); //首个packet相对epoch的偏移，单位ns，标准中规定，对LPRS和GPRS都适用，约为帧时长的3.8%
* pacing->tr\_offset\_vrx = s->st21\_vrx\_narrow; //narrow sender的漏桶容量
* pacing->trs = frame\_time \* ractive / s->st20\_total\_pkts; //GPRS下rtp包的时间间隔，11664ns
* pacing->cur\_epochs = st\_get\_ptp\_time(impl, ST\_PORT\_P) / frame\_time; //这里未修改源码
* pacing->tsc\_time\_cursor = st\_get\_tsc(impl); //tsc时间，单位ns
* pacing->warm\_pkts = RTE\_MIN(pacing->tr\_offset/pacing->trs\*0.8, 128); //80%的包
* pacing->tr\_offset\_vrx += troffset\_warm\_pkts; //time for warm pkts，值为136
* pacing->tr\_offset\_vrx -= 2; //VRX compensate to rl burst(max\_burst\_size=2048)
* pacing->tr\_offset\_vrx -= 2; //leave VRX space for deviation, 值为132
* pacing->pad\_interval = s->st20\_total\_pkts; //值为4115
* s->pacing\_way[port]=ST21\_TX\_PACING\_WAY\_TSC
* s->pacing\_tasklet\_func[port] = video\_trs\_tsc\_tasklet;

st\_init->st\_dev\_init->dev\_eal\_init>rte\_eal\_init

在st\_init(&ctx->para)中，

(1)调dpdk接口rte\_eal\_init(argc，argv)初始化网口，参数为--file-prefix ST\_DPDK --match-allocations --in-memory -a 0000:3d:00.1,max\_burst\_size=2048 -l 10 --log-level info

(2)查询网卡绑定的numa节点(socket\_id)，这里是节点0，即int socket=st\_dev\_get\_socket("0000:3d:00.1")，即rte\_eth\_dev\_get\_port\_by\_name+rte\_eth\_dev\_socket\_id

(3)调numa\_bind绑定进程到numa节点0

(4)创建st\_main\_impl\* impl

(5)st\_dev\_if\_init(impl)，初始化用户态网口，即设置st\_main\_impl->st\_interface[0]，部分信息调dpdk接口获取而来，如inf[0]的socket\_id=0.(即网卡绑定的numa节点)、drv\_type=ST\_DRV\_I40E，部分信息根据业务而来，如ptp\_get\_time\_fn=……、max\_tx\_queues=st\_app\_context->para->tx\_sessions\_cnt\_max+2=3、创建tx\_mbuf\_pool含1024个元素、创建tx\_queues含3个队列

(6)dev\_start\_port：分配创建的3个tx queue、启动网卡即rte\_eth\_dev\_start()、重置网卡io统计即rte\_eth\_stats\_reset()

(7)dev\_detect\_link：检测网卡的link speed，写入st\_main\_impl->st\_interface inf[0]->link\_speed，links peed即指的是百兆网还是千兆网\万兆网，

(8)dev\_if\_init\_pacing：设置网卡的pacing方式，st\_main\_impl->st\_interface inf[0]->tx\_pacing\_way = ST21\_TX\_PACING\_WAY\_TSC

(9)st\_sch\_mrg\_init：设置网卡的st\_main\_impl->sch\_mgr[]->data\_quota\_mbs\_limit=12 \* 1080p@60fps的最大码率，其中12指最多12个session

**tv\_tasklet\_frame源码分析：**

st\_tx\_video\_session\_impl::st20\_pkt\_idx为0时，回调取帧

调用tv\_sync\_pacing，使得，

pacing->cur\_epochs = epochs; //当前帧发送时间所属的epoch

pacing->cur\_time\_stamp = pacing\_time\_stamp(pacing, epochs); //当前帧要发送的90k时戳

pacing->ptp\_time\_cursor = ptp\_tr\_offset\_time; //当前帧要发送的ptp时间

pacing->tsc\_time\_cursor = (double)st\_get\_tsc(impl) + to\_epoch\_tr\_offset; //当前帧要发送的tsc时间

每次封装4个mac帧（bulk=4），每个mac帧一个rtp包：

从s->mbuf\_mempool\_chain中分配rte\_mbuf，存放1260字节的yuv（BPM模式）

从s->mbuf\_mempool\_hdr中每次分配bulk个rte\_mbuf，存放各级协议头（由mac层到rtp层）

然后调用rte\_pktmbuf\_chain，将两个rte\_mbuf链起来

将s->st20\_pkt\_idx、s->pacing. tsc\_time\_cursor、s->pacing. ptp\_time\_cursor都设置给rte\_mbuf

每封装一个rtp包，

pacing->tsc\_time\_cursor += pacing->trs; //下一个rtp包的tsc发送时间

pacing->ptp\_time\_cursor += pacing->trs; //下一个rtp包的ptp发送时间

4个mac帧封装完后，放入环形队列ring中：

n = rte\_ring\_sp\_enqueue\_bulk(s->ring[ST\_SESSION\_PORT\_P], (void\*\*)&pkts[0], bulk, NULL);

返回成功入队的目标数，要么为0，要么是bulk

如果是0，则将这4个pkts标记为inflight状态：

for (unsigned int i = 0; i < bulk; i++) s->inflight[ST\_SESSION\_PORT\_P][i] = pkts[i];

s->has\_inflight[ST\_SESSION\_PORT\_P] = true;

s->inflight\_cnt[ST\_SESSION\_PORT\_P]++;

s->stat\_build\_ret\_code = -STI\_ST22\_PKT\_ENQUEUE\_FAIL;

}

s->st20\_pkt\_idx达到s->st20\_total\_pkts后，说明一个yuv帧已封装完成，此时如果当前tsc时间> pacing->tsc\_time\_cursor，则说明当前帧发送超时，stat\_exceed\_frame\_time++

**tv\_sync\_pacing源码分析：**

函数主要控制视频帧的发送时间，以及rtp包头时戳。

以下针对st20\_tx\_ops->flags为ST20\_TX\_FLAG\_USER\_TIMESTAMP，且st20\_tx\_frame\_meta->tfmt设置为ST10\_TIMESTAMP\_FMT\_MEDIA\_CLK的场景，进行分析：

1. 回调获取当前ptp时间：ptp\_time
2. 计算当前ptp时间对应的epoch：epochs = ptp\_time / frame\_time
3. epochs的期望值为next\_epochs= pacing->cur\_epochs + 1
4. 如果计算出的epochs与期望值相差在[-1sec, 0]范围内，则忽略差值，强行设置epochs为期望值

注意，pacing->max\_onward\_epochs = (double)NS\_PER\_S / pacing->frame\_time

1. 计算发送首个rtp包的ptp时间：start\_time\_ptp = pacing\_start\_time(pacing, epochs)
2. 如果计算出的start\_time\_ptp比当前ptp时间小，则应该放入下一个epoch发送，故epochs++。同时统计值stat\_epoch\_troffset\_mismatch++进行记录。
3. 用新的epochs重新调pacing\_start\_time计算发送时间
4. 统计stat\_epoch\_drop和stat\_epoch\_onward：

if (epochs > next\_epochs) s->stat\_epoch\_drop += (epochs - next\_epochs); //打印的统计日志形如"epoch drop 78"

if (epochs < next\_epochs) s->stat\_epoch\_onward += (next\_epochs - epochs);

1. 更新状态：

pacing->cur\_epochs = epochs;

pacing->cur\_epoch\_time = epochs \* pacing-> frame\_time;

pacing->ptp\_time\_cursor = start\_time\_ptp; //发送首个rtp包的ptp时间

pacing->tsc\_time\_cursor = mt\_get\_tsc(impl) + (start\_time\_ptp - ptp\_time); //发送首个rtp包的tsc时间

pacing->rtp\_time\_stamp = st20\_tx\_frame\_meta->timestamp; //rtp包头时间戳

**pacing\_start\_time()源码分析：**

函数计算首个rtp包应该被发送的ptp时间：

double pacing\_start\_time(struct st\_tx\_video\_pacing\* pacing, uint64\_t epochs)

{

return (epochs \* pacing->frame\_time) + pacing->tr\_offset -

((pacing->vrx + pacing->warm\_pkts) \* pacing->trs);

}

按st2110标准，首个rtp包的发送时间为(epochs \* pacing->frame\_time) + pacing->tr\_offset时刻。

但这样的话，对于Virtual Receiver Buffer模型，每次漏桶中刚有一个rtp，就得发送出去，太临界了。

因此将发送时间提前，提前量为漏桶的容量，即st\_tx\_video\_session\_impl:: st21\_vrx\_narrow

进一步的，stdpdk库使用了预热机制，预热的时长为pacing->tr\_offset的80% ，发送时间继续提前

**tx\_audio\_session\_sync\_pacing源码分析：**

函数主要控制音频帧的发送时间，以及rtp包头时戳。

以下针对st30\_tx\_ops->flags为ST30\_TX\_FLAG\_USER\_TIMESTAMP，且st20\_tx\_frame\_meta->tfmt设置为ST10\_TIMESTAMP\_FMT\_MEDIA\_CLK的场景，进行分析：

1. 回调获取当前ptp时间：ptp\_time
2. 当前ptp时间对应的epoch：epochs = ptp\_time / frame\_time
3. epoch的期望值为next\_epochs= pacing->cur\_epochs + 1
4. 如果计算出的epochs与期望值相差在[-1sec, 10ms]范围内，则强行设置epochs为期望值

pacing->max\_onward\_epochs = (double)(NS\_PER\_S \* 1) / pacing->trs; /\* 1s \*/

pacing->max\_late\_epochs = (double)(NS\_PER\_S \* 1) / pacing->trs / 100; /\* 10ms \*/

1. 计算多久后发送首个rtp包，to\_epoch=epochs \* pacing->trs - ptp\_time，如果小于0，则强制令to\_epoch为0，且统计值stat\_epoch\_mismatch++进行记录
2. 统计stat\_epoch\_drop和stat\_epoch\_onward：

if (epochs > next\_epochs) s->stat\_epoch\_drop += (epochs - next\_epochs);

if (epochs < next\_epochs) s->stat\_epoch\_onward += (next\_epochs - epochs);

1. 更新状态：

pacing->cur\_epochs = epochs;

pacing->cur\_epoch\_time = epochs \* pacing->trs;

pacing->tsc\_time\_cursor = (double)mt\_get\_tsc(impl) + to\_epoch; //首个rtp包发送的tsc时间

pacing->rtp\_time\_stamp = st30\_tx\_frame\_meta->timestamp; //rtp包头时间戳

如果当前tsc小于理应发送时刻的tsc：

差距在200us以内，则返回ST\_TASKLET\_HAS\_PENDING(1)

差距在200us~1sec以内，则返回ST\_TASKLET\_ALL\_DONE(0)

差距大于1sec，则打错误日志提示

core上的sched执行各个tasklet时，如果发现tasklet函数返回的是DONE，且sch->allow\_sleep，则睡眠st\_sleep\_ms(0)

**ptp\_get\_time\_fn源码分析：**

tx\_audio\_session\_sync\_pacing()中获取ptp时间：

uint64\_t ptp\_time = mt\_get\_ptp\_time(impl, MTL\_PORT\_P);

实际调的是struct mt\_interface的成员ptp\_get\_time\_fn，这是个函数指针

mt\_interface->ptp\_get\_time\_fn是在mtl\_init()->mt\_dev\_if\_init()中初始化的，

*//mt\_dev.c*

if (mt\_user\_ptp\_tsc\_source(impl)) {

info("%s(%d), use tsc ptp source\n", \_\_func\_\_, i);

inf->ptp\_get\_time\_fn = ptp\_from\_tsc;

} else if (mt\_user\_ptp\_time\_fn(impl)) {

*/\* user provide the ptp source \*/*

info("%s(%d), use user ptp source\n", \_\_func\_\_, i);

inf->ptp\_get\_time\_fn = ptp\_from\_user;

} else {

info("%s(%d), use mt ptp source\n", \_\_func\_\_, i);

inf->ptp\_get\_time\_fn = ptp\_from\_real\_time;

}

ptp\_from\_user即回调的是用户设置的mtl\_init\_params::ptp\_get\_time\_fn

ptp\_from\_real\_time调用的是mt\_get\_real\_time，

inline uint64\_t mt\_get\_real\_time(void) {

struct timespec ts;

clock\_gettime(CLOCK\_REALTIME, &ts);

return ((uint64\_t)ts->tv\_sec \* NS\_PER\_S) + ts->tv\_nsec;

}

**日志分析：**

stdpdk库打印的日志会保存在/var/log/messages中

* "busy as no ready frame from user 99"：st20\_tx\_ops->get\_next\_frame回调取帧时没取到，则st\_tx\_video\_session\_impl的成员stat\_user\_busy++，日志中99为stat\_user\_busy的值，每打印一次日志后，该值清0
* "mismatch epoch troffset 97"：帧来晚了，应发送时间小于当前ptp时间，stat\_epoch\_troffset\_mismatch++
* “epoch drop 272”：有272个epoch丢失了，即这些epoch时间内没有帧发送
* "build timeout frames 209"：有209个帧由yuv封装为rtp->mac帧时超时了，为stat\_exceed\_frame\_time的值
* "pkts 1029068:1028817"：冒号前者为构建好的mac帧数，后者为rte\_eth\_tx\_burst发送的mac帧数
* "inflight 3104975:2057310"：2057310为inflight\_cnt的值，生成rtp mac帧后会放入ring中，如果入队失败，则inflight\_cnt++，
* “st30 epoch mismatch”：

ST: tv\_init\_pacing[00], trs 9331.713244 trOffset 1528888.888889 warm pkts 128

trs：单位为ns，相邻rtp包发送时间间隔

trOffset：单位为ns，首个rtp包相对epoch（帧时长的整数倍）的偏移

warm pkts：预热rtp包的个数，pacing->tr\_offset\_vrx=漏桶容量加上预热rtp包数再减去4

**st2110查询对端mac地址：**

nicctl.sh bind\_pmd 0000:09:00.0后，ifconfig -a不能查看到指定网卡

nicctl.sh bind\_kernel 0000:09:00.0后ifconfig -a可以查看到指定网卡

bind\_kernel后还需要ifup，才能启用网卡

st20\_tx\_create过程中会查询对端mac地址，否则无法初始化ip报文头，具体调用链路为：

st20\_tx\_create->tv\_mgr\_attach->tv\_attach->tv\_init\_hdr->st\_dev\_dst\_ip\_mac->"st\_arp\_cni\_get\_mac(0), waiting arp from 10.10.80.229"

查询对端mac地址失败时stdpdk日志：

MTL: 2024-11-29 15:51:53, arp\_get\_result(0), cache waiting arp from 10.10.80.229

MTL: 2024-11-29 15:51:54, Error: arp\_get\_result(0), cache fail as timeout to 5000 ms

MTL: 2024-11-29 15:51:54, Error: tv\_init\_hdr(0), get mac fail -5 for 10.10.80.229

如果对端不是dpdk程序，网卡使用的是内核驱动，处于ifup状态，能被ping通，则本机st20\_tx\_create可以查询到对端mac地址

如果对端是dpdk程序，但未启动，则本机st20\_tx\_create无法查询到对端mac地址，一直循环尝试

如果对端是dpdk程序，且已经启动，则本机st20\_tx\_create可以查询到对端mac地址

对端是dpdk程序且已经启动，本机网卡是内核驱动，ping对端，会ping不通，因为dpdk未实现ping协议

**载荷ID：**

在SDI信号中可以包含辅助数据（Ancillary Data），在ST2110系统中，有独立的辅助数据包（ST2110-40数据流）。这些辅助数据中包含有payload id信息。

接收设备可以从辅助数据中读取payload id信息，这也就是为什么有的设备如监视器可以自适应识别SDI的信号格式。

对于不支持解码辅助数据的设备，则必须根据接入的信号格式手动设置设备的系统格式进行格式匹配。

payload id是一串16进制编码，分为4个字节。如对于3840x2160、p50、BT2020色域、HLG曲线的视频样片，监视器解析到的payload id编码为01A0D9CE，含义为：



**多模单模：**

光纤外面的线上有英文缩写，SMF是单模光纤，MMF是[多模光纤](https://www.zhihu.com/search?q=%E5%A4%9A%E6%A8%A1%E5%85%89%E7%BA%A4&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra=%7B%22sourceType%22%3A%22answer%22%2C%22sourceId%22%3A%22158842124%22%7D)

多模光纤允许通过多个光模式，所以多模光纤比单模的更贵些。

但是，单模光纤采用固态激光二极管作为光源，远比多模光纤的光源设备昂贵，所以综合来看，单模光纤的使用成本比多模光纤高得多

单模光纤跳线的护套一般是黄色的，而多模一般是橙色或者所谓的水绿色（就是介于蓝色和绿色之间的颜色）

在光纤通信中，SR、LR、LRM、ER、ZR是光传输中的一种距离术语。

SR表示短距离、LR表示长距离、LRM 表示长度延伸多点模式、ER表示超长距离、ZR则表示最长距离

SFP-10G-SR (短距离) 是一种常见的SFP模块, 它可以在多模光纤中连接高达300米的光纤。一般情况下SR模块多为多模。

SFP-10G-LR(长距离) 用于远程数据传输, 如大型组合或地铁区域网络。它可以是单模或多模式



**msdk：**

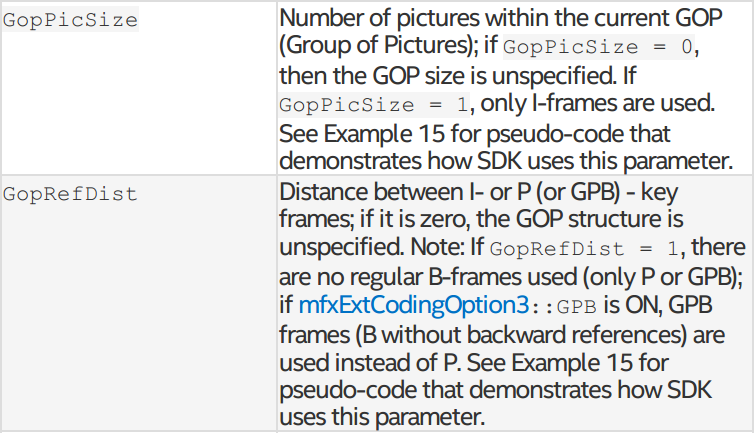
NVIDIA、AMD以及Intel都有自己的视频转码硬件加速技术

NVIDIA的NVENC：Kepler架构新增的功能

Intel的Quick Sync：SNB架构的GPU部分增加了专用的Quick Sync单元

AMD的VCE（Video Encodec Enigine）

NVENC比CUDA编码还要优秀，因为它跟Quick Sync一样属于是专用的编码加速单元，而CUDA加速则是比较通用的，速度上不如专用单元快。



Media SDK HW HEVC编码器使用低延迟B帧(LDB)或通用P/B(GPB)代替传统的P帧。

**Low Delay B-frames (LDB)**  
LDB frames are a special type of B-frame designed to reduce latency. In traditional video encoding, B-frames (bi-directional predictive frames) use both past and future frames for data reference, which can cause delays as future frames need to be encoded before the current frame can be fully processed. On the other hand, LDB frames only reference past frames, functioning more like P frames (predictive frames), which only reference preceding frames. This setup allows for lower latency in video streaming and conferencing applications where real-time performance is critical.

**Generalized P/B frames (GPB)**  
GPB frames are a hybrid approach used in specific video encoders, such as Intel's HEVC encoder. They are encoded with the structure and characteristics of B-frames, but they are used in a predictive manner similar to P-frames. Essentially, a GPB frame can function as a B-frame while only referencing past frames, thus ensuring compatibility with encoders and decoders expecting traditional P-frame behavior.

**Impact and Implementation**  
Using LDB and GPB frames instead of traditional P frames can confuse. Even though these frames are used predictively like P-frames, the encoder may label them as B-frames to improve compression efficiency and reduce latency. However, this approach could confuse video analysis tools and users accustomed to a clear distinction between P and B frames.

# msdk屏蔽核显：

找到核显对应的pci号，echo 1 >> /sys/bus/pci/devices/0000\:00\:02.0/remove

执行完后lspci | grep VGA会查不到核显设备

echo 1 > /sys/bus/pci/rescan，重新扫描PCI总线上的设备，核显会恢复

# 视频工作流程：

color correction：颜色校正

VFX：visual effects，视觉特效

4:4:4色度子采样：用于高端胶片扫描仪和顶级电影后期制作工作流程，这些工作需要最高的图像质量。

4:2:2色度子采样：许多高端数字视频格式和接口（digital video formats and interfaces）都使用此方案，包括流行的ProRes422。

长GOP编解码器：long-GOP codecs，即采用了帧间压缩的编解码器

摄像机捕捉素材（footage）时采用的编解码器，应该具有高位深（deep bit depths）、低色度子采样（low chroma subsampling）、高比特率，并且不使用长GOP压缩

从摄像机中卸载出捕获到的素材后，需要选择一个编解码器用来编辑素材。编辑时速度和易用性优于图像质量，因此选择能将原始素材文件转码为更小体积的文件的编解码器。就在几年前，即使是大预算的好莱坞电影也在480p下进行编辑。但需要确保使用仅空间压缩（如DNxHD或ProRes）的编解码器进行编辑。长GOP编解码器确实生成小文件，但当你在时间线上快速拖动\浏览时，你的计算机将不得不更加努力地处理所有参考的信息，这可能会显著减慢编辑速度。除此之外，一旦你锁定了编辑，最终你会重新链接到更高质量的文件，以替换低质量的编辑编解码器。

编辑完成后，需要选择一个编解码器用来进行色彩校正和视觉特效（color correction and VFX，这两种统称为精加工，即finishing）。通常，您可以使用符合视频时间线（conformed into your video timeline）的原始相机文件，这可以确保您获得尽可能高质量的图像。但如果您决定在这些过程中使用单独的编解码器，您会希望选择一个能够比捕获编解码器保留更多信息的编解码器。

当你准备导出视频以进行交付（export your video for delivery）时，你可能会认为更大的编解码器总是更好。然而，实际上，交付媒介总是决定了编解码器的选择。

完成交付后，你需要选择一个编解码器进行归档。客户有时在交付几年后会要求更高分辨率或更高质量的文件。在这些情况下，打开原始项目文件并重新导出并不总是可行的。因此，你需要为归档文件选择一个好的编解码器。它应该具有极高的质量，尽可能高的比特率，以便在未来需要转码时可以获得最佳的结果。存储一个非常大的文件通常是值得的权衡。

摄像机内会发生光电变换，如gamma编码，Image File gamma

显示器内发生电光变换，如gamma解码，display gamma，gamma校正（Gamma Correct）

与gamma编码一样，log编码将数学函数应用于图像传感器的输出。但与gamma编码的不同之处在于log编码的数学函数是一条对数曲线（由此得名）。这是一条更激进的曲线，它将亮度信息延伸到更广泛的色调范围内。这意味着图像比标准gamma编码平坦得多。

log编码使用更多的数字信息来存储图像的阴影和中间色调，从而保留了更低的色阶和更多细节。这使您可以在后期制作中对图像进行更精细的调整、更具创意的控制。当然，这也意味着当您在大多数显示器上观看图像时，它看起来与现实世界中的样子不同。为什么会这样？因为显示器使用的标准gamma校正不是对数的，所以图像看起来会很模糊。但是，通过反向应用相应的对数函数可以轻松解决这个问题，这正是一些LUT（Look Up Table）所做的。

**"PES packet size mismatch"：**

$ ffmpeg -i test1.ts -hide\_banner -f null -

[mpegts @ 000001d0ebdae580] PES packet size mismatch

[mpegts @ 000001d0ebdae580] Packet corrupt (stream = 0, dts = 3135705481).

[mpegts @ 000001d0ebdae580] PES packet size mismatch

[mpegts @ 000001d0ebdae580] Packet corrupt (stream = 0, dts = 3135727073).

[mpegts @ 000001d0ebdae580] PES packet size mismatch

[mpegts @ 000001d0ebdae580] Packet corrupt (stream = 0, dts = 3135763060).

这里的报错“PES packet size mismatch”来自于ffmpeg的libavformat\mpegts.c文件，

if (pes->total\_size != MAX\_PES\_PAYLOAD &&

pes->pes\_header\_size + pes->data\_index != pes->total\_size + PES\_START\_SIZE) {

av\_log(pes->stream, AV\_LOG\_WARNING, "PES packet size mismatch\n");

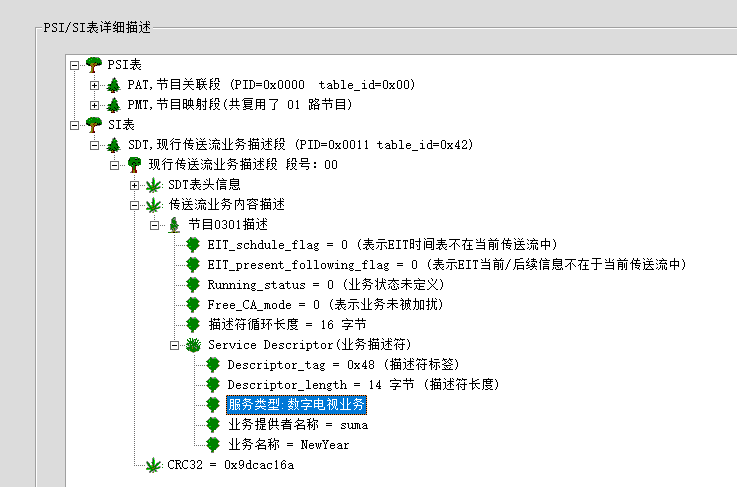
pes->flags |= AV\_PKT\_FLAG\_CORRUPT;

}

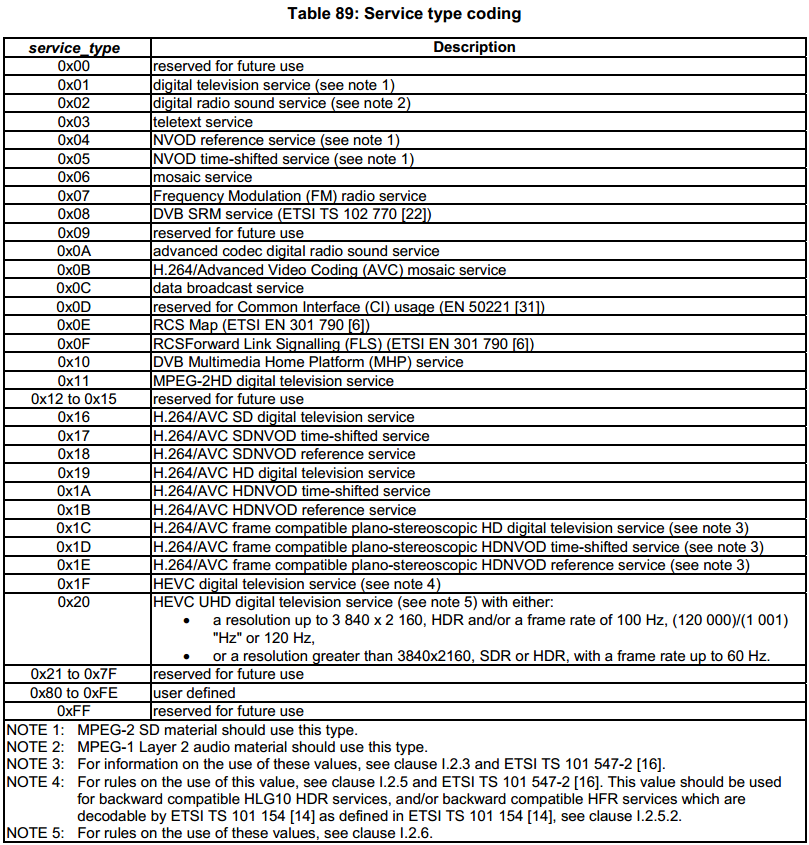
其中，

1. pes->pes\_header\_size为pes头的实际长度，包括3字节的packet start code prefix（0x000001）+1字节的stream id+2字节的PES packet length+2字节+1字节的PES header data length+5字节的pts
2. 解析完14字节的pes头后，pes->data\_index置为0，然后随着解析进行，pes->data\_index增加
3. pes->total\_size是PES packet length字段的取值
4. PES\_START\_SIZE是6，3字节的packet start code prefix+1字节的stream id+2字节的PES packet length字段

**ts流中sdt表里的service\_type域：**



在ETSI 300 468标准中，



在高斯泰克TSOverIP分析处理器中，

service\_type=0显示为用户定义任务

service\_type=1显示为数字电视业务，

service\_type=2显示为数字音频广播业务，

service\_type=3显示为图文电视节目，

service\_type=4显示为NVOD参考业务

service\_type=5显示为NVOD时移业务

service\_type=6显示为马赛克业务

service\_type=25显示为用户定义任务

# pipeline模型下的延迟分析：

考虑三个plugin，demuxer->dec->sdi\_send

其中，demuxer按照25fps帧率均匀出帧；

dec除了第一帧要花费10秒（内存分配、解码器初始化等）外，1秒能处理50帧；

sdi\_send严格按照25fps的帧率向外发送数据

运行流程如下：

0秒时，dec取1帧，花了10秒

10秒时，dec取50帧----------->假设①由此开始sdi发送

11秒时，dec取25帧

11.5秒时，dec取12.5帧

11.75秒时，dec取6.25帧----------->假设②由此开始sdi发送

11.875秒时，dec取3.125帧

11.9375秒时，dec取1.5625帧

11.96875秒时，dec取0.78125帧

……

分析：

（1）如果在10秒时，启动sdi发送，则到了11.75秒时，总共过去了1.75秒，sdi按照帧率发送，25fps，总共是43帧，即此时发送到了第43帧

（2）如果在11.75秒时，才启动sdi发送（前边几次frame函数将输入帧全丢掉），则此时发送的是第50+25+12.5=87帧

所以假设②比假设①能获得更小的延迟

实际测试时的例子：

相邻帧的pts间隔是1920，预处理plugin输出时戳信息如下：

2529ms出35帧

2575ms出4帧

2671ms出4帧

2768ms出4帧

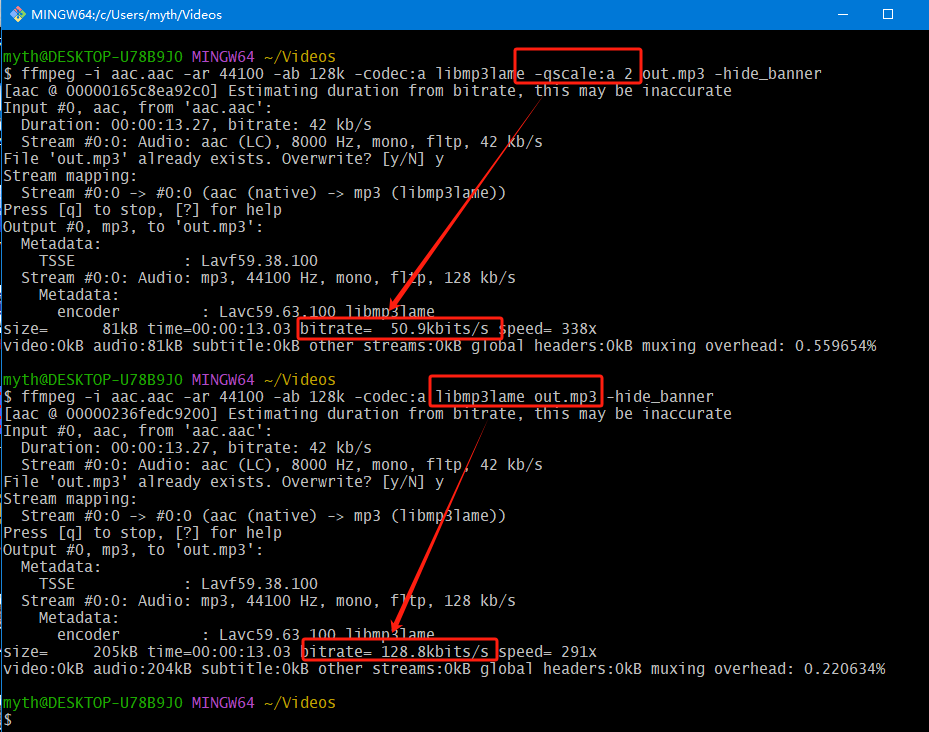
如果2529ms那次出帧，然后进入sdi的frame函数，sdi开始发送，则到了2768ms，过去了239ms，相当于发送到第239\*90/1920=11.2帧

如果2768ms那次出帧然后进入sdi的frame函数，sdi开始发送，则发送的是第35+4+4+4帧

故，形成以下结论：

1. 只要下一个plugin的处理速度大于帧率，则一段时间后肯定是每次只进一帧
2. 对于解码、编码等环节，常是上游第一帧到来时才执行初始化编解码器等操作，耗时过长，因此第二次吐帧都是一次性吐出很多帧，下游plugin开始也是进多帧，最后就开始进1帧了
3. 如果下游plugin是严格按照固定速率处理数据，为了降低延时，最好等其frame函数每次只能取一帧时再开工

**mp3编码的qscale:a选项：**



**mpeg-1 audio layer 2：**

MPEG-1 音频标准的第二层（Layer 2），简称mp2

但mp2也包括mpeg2 version

无论mpeg1还是mpeg2，一个mp2音频帧包含的采样点个数都是1152，即AVCodecContext::frame\_size为1152

audio sequence()

{

    while(true)

    {

        frame()

    }

}

frame()

{

    header()

    error\_check()

    audio\_data()

    ancillary\_data()

}

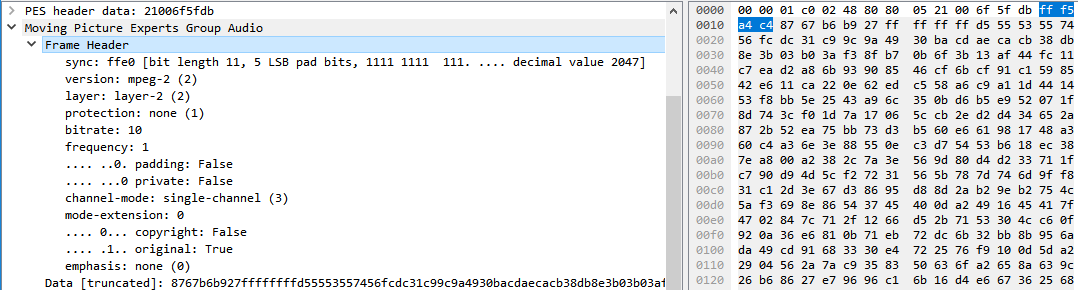
header()，来自ISO/IEC 11172-3标准：

|  |  |  |
| --- | --- | --- |
| 字段 | 比特数 | 含义 |
| syncword | 12 bits | 同步字，'1111 1111 1111'，即0xFFF |
| ID（version） | 1 bit | indicate the ID of the algorithm:   * + 0: ISO/IEC 13818-3 or MPEG-2 Audio extension to lower sampling frequencies（注意该表格下文中各字段含义不考虑ID为0的情况，仅针对ID为1的情况）   + 1: ISO/IEC 11172-3 |
| layer | 2 bits | indicate which layer is used:  "11" Layer I  "10" Layer II  "01" Layer III  "00" reserved  To change the layer, a reset of the decoder is required |
| protection\_bit | 1 bit | 指示是否有CRC校验，0表示有，1表示没有 |
| bitrate\_index | 4 bits | an index to a table, which is different for the different Layers. |
| sampling\_frequency | 2 bits | indicates the sampling frequency:  '00' 44.1 kHz '01' 48 kHz '10' 32 kHz '11' reserved |
| padding\_bit | 1 bit | if this bit equals '1' the frame contains an additional slot to adjust the mean bitrate to the sampling frequency, otherwise this bit will be '0'. Padding is only necessary with a sampling frequency of 44.1 kHz |
| private\_bit | 1 bit | bit for private use. This bit will not be used in the future by ISO |
| mode | 2 bits | indicates the mode according to the following table. In Layer I and II the joint\_stereo mode is intensity\_stereo, in Layer III it is intensity\_stereo and/or ms\_stereo  '00' stereo '01' joint\_stereo (intensity\_stereo and/or ms\_stereo) '10' dual\_channel '11' single\_channel |
| mode\_extension | 2 bits | these bits are used in joint\_stereo mode.  In Layer I and II they indicate which subbands are in intensity\_stereo. All other subbands are coded in stereo  '00' subbands 4-31 in intensity\_stereo, bound==4 '01' subbands 8-31 in intensity\_stereo, bound==8 '10' subbands 12-31 in intensity\_stereo, bound==12 '11' subbands 16-31 in intensity\_stereo, bound==16  In Layer III they indicate which type of joint stereo coding method is applied. The frequency ranges over which the intensity\_stereo and ms\_stereo modes are applied are implicit in the algorithm  intensity\_stereo ms\_stereo '00' off off '01' on off '10' off on '11' on on |
| copyright | 1 bit | if this bit equals '0' there is no copyright on the coded bitstream, '1' means copyright protected |
| original/home | 1 bit | this bit equals '0' if the bitstream is a copy, '1' if it is an original |
| emphasis | 2 bits | indicates the type of de-emphasis that shall be used  '00' no emphasis '01' 50/15 microsec. emphasis '10' reserved '11' CCITT J.17 |

mpeg1\_48khz\_stereo\_128kbps\_s16：



mpeg2\_24khz\_mono\_96kbps\_s16：



**广电行业规范：**

广播电视是有自己的行业规范的，广播有卫星、地面波、有线电视网络、IPTV、互联网等渠道。国内广播电视以中星6号（C波段），中星9号（户户通，Ku波段）传输源码。

卫星直播是最高标准？安全播出，不易受地面网络攻击或自然灾害中断；覆盖全国，无需依赖本地有线网络基础设施。

中星6号（C波段），转发器性能（如36MHz带宽），基本上就决定广电制定标准时规范的视频码率，相当于这个就是广电传输的最高码率标准。

其它渠道（除有线电视专网以外？）的视频源，都是在这个标准的基础上转码或者源码播出。

广电18年制定4K技术规范，定下4K 50P 10bit BT.2020 HDR-HLG\HDR-PQ AVS2 36Mbps视频规格。

至今为止，能正确解码中星6C上的CCTV4K信号的设备少之又少。绝大部分都是经过HDR-SDR转换的信号，如果你去播放源视频，很大概率只能得到一个灰蒙蒙的画面，当然广电自己的播放器，其实还是能够正确解码的，但是现在很少有家庭还在给广电交高昂的费用去看电视。

**分析音视频时间戳乘2源：**

源描述：

在TikTok上看到了一段明显帧率超高的mp4视频，发下以下问题：

1. ffprobe、MediaInfo、PotPlayer都显示帧率仅30fps
2. 用PotPlayer能正常播放，显示的当前帧率约为60fps，远高于源帧率30fps。

进度条显示总时长1分25sec，但实际播放42sec后，整个进度条走完，播放结束。



1. 用ffplay能正常播放，42sec后播放结束
2. 用VLC3.0.21播放，进度条显示总时长1分25sec，实际播放了1分25sec走完进度条，画面有慢放效果，声音断断续续，整体播放效果接近0.5慢速播放加上静音填充

分析：

1. 先用mediainfo，ffprobe查看一下大致信息，二者都显示视频帧率是30fps

$ ffprobe shadow.mp4

Duration: 00:01:25.90, start: 0.000000, bitrate: 3169 kb/s

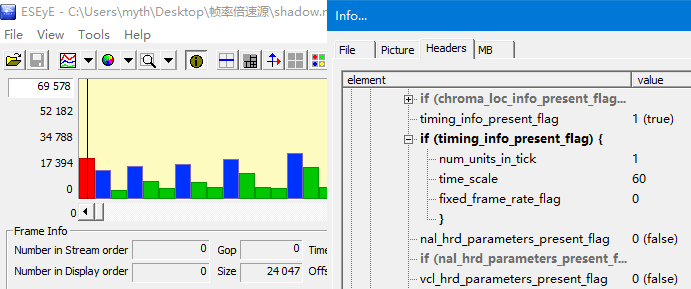
Stream #0:0[0x1](und): Audio: aac (HE-AACv2) (mp4a / 0x6134706D), 44100 Hz, stereo, fltp, 32 kb/s (default)

Stream #0:1[0x2](und): Video: h264 (High) (avc1 / 0x31637661), yuv420p(tv, bt709, progressive), 720x1280 [SAR 1:1 DAR 9:16], 3132 kb/s, 30 fps, 30 tbr, 15360

1. 再析编码码流

对音频，根据he-aacv2标准，一帧aac数据是2048采样点，则对于44100hz的采样率，一帧时长应为46.44ms

对视频，用ESEyE查看码流，sps里指出帧率是60，如下图：



H264帧率解析方法：先检测SPS中timing\_info\_present\_flag标签的值是否为1，如果是1则表示SPS中存在num\_units\_in\_tick，time\_scale和fixed\_frame\_rate\_flag这三个标签，至此可以计算出帧率：

fps = time\_scale / num\_units\_in\_tick;

if ( fixed\_frame\_rate\_flag == 1 ) {fps = fps / 2;}

1. 然后分析下时戳：

时戳单位：

streams[0]->time\_base，44100，音频  
streams[1]->time\_base，15360，视频

用ffprobe查看视频，帧pts间隔=512，转换为毫秒即512/15360\*1000=33.33ms。也即相当于帧率是30fps

echo entry,key\_frame,pts,pkt\_dts,best\_effort\_timestamp,pict\_type,coded\_picture\_number,interlaced\_frame,top\_field\_first > "frame.csv"; ffprobe -v error -hide\_banner -i shadow.mp4 -select\_streams v:0 -print\_format csv -of csv -show\_entries "frame=key\_frame,pts,pkt\_dts,best\_effort\_timestamp,pict\_type,coded\_picture\_number,interlaced\_frame,top\_field\_first" >> frame.csv

用ffprobe查看音频，pts间隔在3929、3930、4870、4871间波动，连续17个间隔里有3个是4870，14个是3930，因此平均间隔为(3\*4870+14\*3930）/17=4095.88，转换为毫秒即4095.88/44100\*1000=92.88ms

echo entry,key\_frame,pts,pkt\_dts,best\_effort\_timestamp,pict\_type,coded\_picture\_number,interlaced\_frame,top\_field\_first > "frame.csv"; ffprobe -v error -hide\_banner -i shadow.mp4 -select\_streams v:0 -print\_format csv -of csv -show\_entries "frame=key\_frame,pts,pkt\_dts,best\_effort\_timestamp,pict\_type,coded\_picture\_number,interlaced\_frame,top\_field\_first" >> frame.csv

1. 对音视频时戳都x2

可以发现，无论音频还是视频，相邻帧的时戳间隔，恰好都是编码码流里描述的2倍

因此推测该源的生成方式为“对原始60fps视频文件的音频时间戳和视频时间戳都乘以2”

1. 对视频时间戳x2，1分钟的视频变成2分钟，60fps的视频看起来是30fps，0.5倍速慢放效果
2. 以音频做主时钟的播放器如ffplay、potplayer，音频时间戳x2，实际上是让时钟变成了2倍速，会以2倍速播放视频，但如果视频时戳也x2，则播放效果又变为60fps，正常倍速，且音画同步
3. 其它不单纯用音频做主时钟的播放器，比如QuickTime player、VLC 3.0、Chrome浏览器，播放音视频时戳都x2的视频会出现音画异常，如声调异常、音画不同步等

对于上述伪造的30fps的源，用ffplay指定用video或system做主时钟，则都是30fps慢放效果

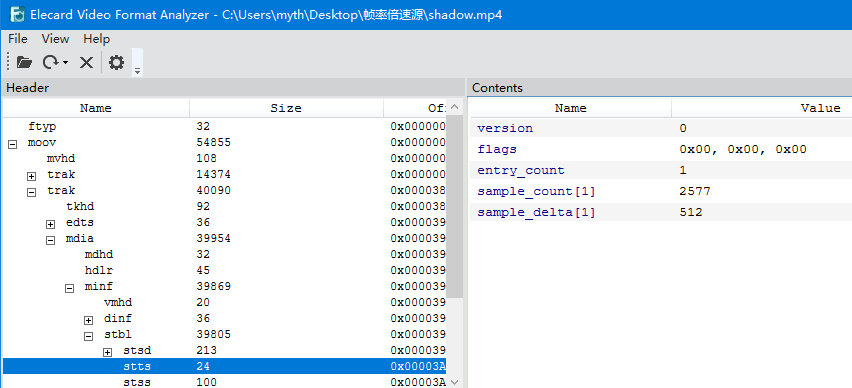
ffplay -sync video shadow.mp4

总共播了1min25sec，画面慢放效果，声音前半段正常，后半段静音

ffprobe计算帧率：

Q：为什么MediaInfo和ffprobe显示的帧率是30？

A：MediaInfo不清楚，ffprobe对于mp4，是从封装层mp4的box里解析的帧率，并不关心h264码流里打的帧率信息。对于.ts文件，则ffprobe从码流中读取帧率信息。具体的，



如上图，mp4中moov->trak->mdia->minf->stbl->stts显示共2577个sample，sample之间的dts差值为512  
moov->trak->mdia->minf->stbl->stsd->VisualSampleEntry显示一个sample包含的帧数frame\_count为1  
因此视频流共2577帧  
moov->trak->mdia->mdhd中指定timescale为15360，因此相邻帧的dts差值为512/15360\*1000=33.33ms  
ffmpeg据此推算出帧率为30fps

复现流程：

1. 制作一个正常的50fps的mp4源：

ffmpeg -i h2641080p50.ts -c copy h2641080p50.mp4

1. ffmpeg命令对音视频时间戳乘2：

ffmpeg -i h2641080p50.ts -c copy -bsf:a setts=ts=2\*PTS -bsf:v setts=pts=2\*PTS:dts=2\*DTS h2641080p50\_x2.ts

ffmpeg -i h2641080p50.mp4 -c copy -bsf:a setts=ts=2\*PTS -bsf:v setts=pts=2\*PTS:dts=2\*DTS h2641080p50\_x2.mp4

1. 播放

ffplay h2641080p50.mp4正常播，2min

Stream #0:0[0x1](und): Video: h264 (Main) (avc1 / 0x31637661), yuv420p(progressive), 1920x1080 [SAR 1:1 DAR 16:9], 4005 kb/s, 49.95 fps, 50 tbr, 90k tbn, Start 1.020000 (default)

ffplay h2641080p50\_x2.mp4正常播，2min

Stream #0:0[0x1](und): Video: h264 (Main) (avc1 / 0x31637661), yuv420p(progressive), 1920x1080 [SAR 1:1 DAR 16:9], 2002 kb/s, 24.98 fps, 25 tbr, 90k tbn, Start 2.040000 (default)

vlc播，进度条4分钟，声音断断续续

ffplay h2641080p50.ts正常播，2min

Stream #0:1[0x101]: Video: h264 (Main) ([27][0][0][0] / 0x001B), yuv420p(progressive), 1920x1080 [SAR 1:1 DAR 16:9], 50 fps, 50 tbr, 90k tbn, Start 15.158256

ffplay h2641080p50\_x2.ts正常播，2min

Stream #0:0[0x100]: Video: h264 (Main) ([27][0][0][0] / 0x001B), yuv420p(progressive), 1920x1080 [SAR 1:1 DAR 16:9], 50 fps, 25 tbr, 90k tbn, Start 3.440089

vlc播，进度条4分钟，声音断断续续

**BT709的YUV转RGB矩阵：**

bt709对应的AVColorSpace值为1

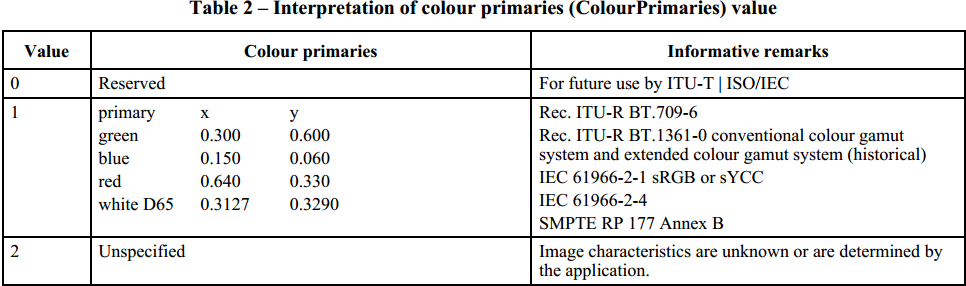
enum AVColorSpace {

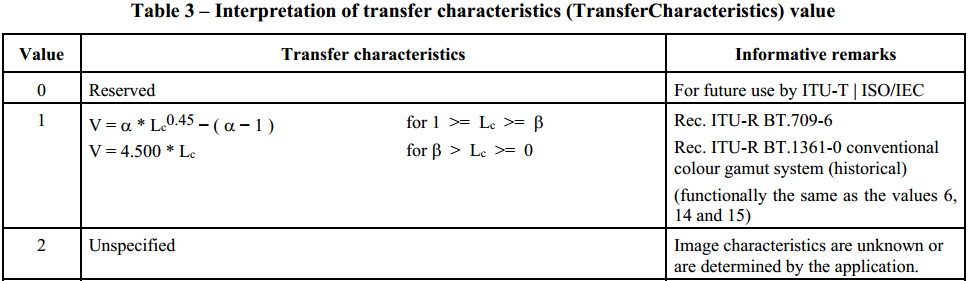
    AVCOL\_SPC\_RGB         = 0,  *///< order of coefficients is actually GBR, also IEC 61966-2-1 (sRGB)*

    AVCOL\_SPC\_BT709       = 1,  *///< also ITU-R BT1361 / IEC 61966-2-4 xvYCC709 / SMPTE RP177 Annex B*

};

ITU-T H.273 (V4) (07/2024)标准中定义了ColourPrimaries、TransferCharacteristics、MatrixCoefficients、VideoFullRangeFlag





Nominal peak white is specified as having ER equal to 1, EG equal to 1 and EB equal to 1.  
Nominal black is specified as having ER equal to 0, EG equal to 0 and EB equal to 0

The application of the transfer characteristics function is denoted by ( x )′ for an argument x

If MatrixCoefficients is not equal to 14 or 15, the signals E′R, E′G and E′B are determined by application of the transfer characteristics function as in equations 11 to 13:



If TransferCharacteristics is equal to 1, E′R, E′G and E′B are real numbers with values in the range of 0 to 1, inclusive

If MatrixCoefficients is not equal to 0, 8, 10, 11, 13, 14, 15, 16 or 17, equations 45 to 47 apply:



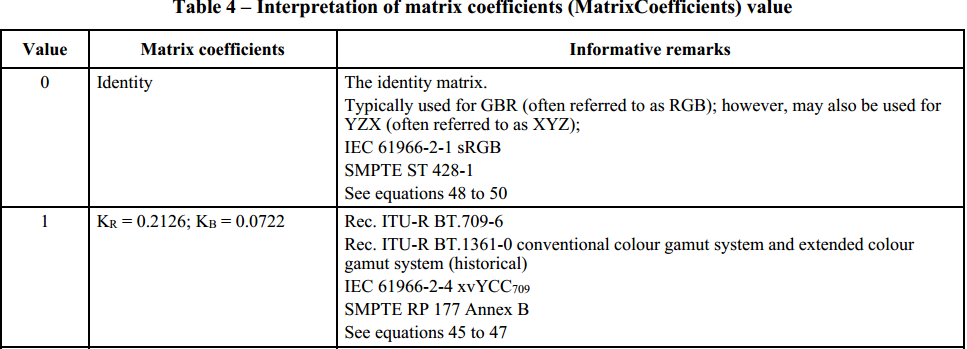
E′Y is a real number with the value 0 associated with nominal black and the value 1 associated with nominal white.

E′PB and E′PR are real numbers with the value 0 associated with both nominal black and nominal white.

When TransferCharacteristics is not equal to 11 or 12, E′Y is a real number with values in the range of 0 to 1, inclusive.

When TransferCharacteristics is not equal to 11 or 12, E′PB and E′PR are real numbers with values in the range of -0.5 to 0.5, inclusive. When TransferCharacteristics is equal to 11 (IEC 61966-2-4) or 12 (Rec. ITU-R BT.1361-0 extended colour gamut system), E′Y, E′PB and E′PR are real numbers with a larger range not specified in this Specification.

If MatrixCoefficients is not equal to 12 or 13, the constants KB and KR are specified in Table 4

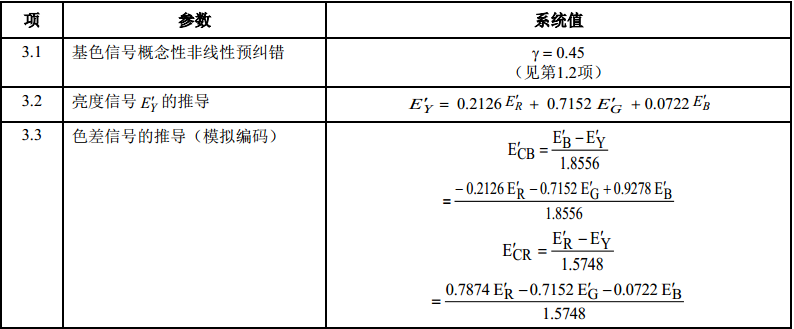


公式45~47写成矩阵形式为：

对矩阵求逆即可得到Y'CbCr转RGB的矩阵：

由表4可知，对于bt709，KR=0.2126，KB=0.0722，代入得：

与ITU-R BT.709-6(06/2015)标准中的描述符合：



用python求逆，得到Y'CBCR转R'G'B'矩阵：

import numpy as np

matrix = np.array([

    [0.2126,  -0.1145721060,  0.5],

    [0.7152,  -0.3854278939,  -0.4541529083],

    [0.0722, 0.5,            -0.0458470916]

])

inverse\_matrix = np.linalg.inv(matrix)

'''

 [[ 1.00000000e+00  1.00000000e+00  1.00000000e+00]

 [-2.73506691e-10  -1.87324273e-01  1.85560000e+00]

 [ 1.57480000e+00  -4.68124273e-01  -1.30501949e-10]]

'''

浮点数有计算误差，如果先公式推导再带入常量值，则得到的Y'CBCR转R'G'B'矩阵为：

字幕：

ffplay播放内置字幕轨的视频，显示字幕：

ffplay -vf subtitles=filename=input.mkv input.mkv

**srt字幕：**

SRT（SubRip Text）

AV\_CODEC\_ID\_SUBRIP

SRT中的每个字幕段由四部分构成：

1. 字幕序号
2. 字幕起止时间，由-->分隔
3. 字幕内容，一或多行
4. 空白行（表示本字幕段的结束）

字幕显示起始时间的格式如下：  
hour:minute:second,millisecond --> hour:minute:second,millisecond  
逗号也可为句号

后面还可以附加用于指定字幕显示位置的信息，以像素为单位，格式如下：

 X1:number Y1:number X2:number Y2:number

一个典型的SRT文件如下：

3

00:00:39,770 --> 00:00:41,880

在经历了一场人生巨变之后

When I was lying there in the VA hospital ...

4

00:00:42,550 --> 00:00:44,690

我被送进了退伍军人管理局医院

... with a big hole blown through the middle of my life,

5

00:00:45,590 --> 00:00:48,120

那段时间我经常会梦到自己在飞翔

... I started having these dreams of flying.

多数SRT支持一些特定格式化，比如斜体、粗体、下划线以及字体颜色。使用时需要基于HTML的标签：

<font color=red>颜色</font>

<i>字体斜体</i>

<u>字体下加划线</u>

<br>换行

<b>字体加粗</b>

mpeg-ts仅支持dvb\_subtitle和dvb\_teletext字幕，不支持srt字幕

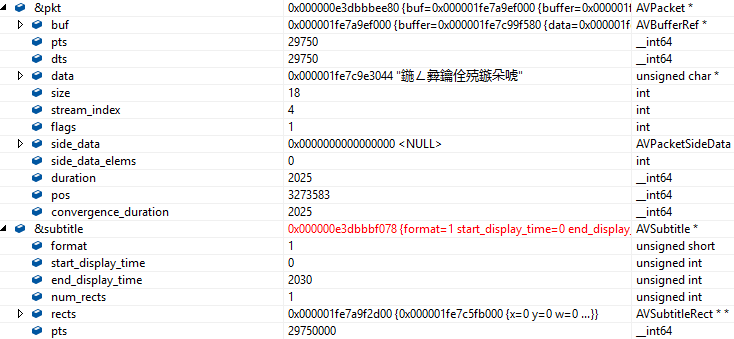
libavformat/srtdec.c

ffmpeg提取dvb\_teletext字幕为.srt文件：ffmpeg -txt\_format text -i INPUT -an -vn -scodec srt test.srt

ffplay显示srt字幕：ffplay test.mp4 -vf subtitles=test.srt

**字幕解码：**

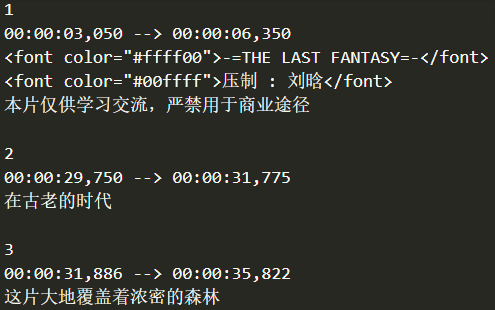
对srt字幕，调用avcodec\_decode\_subtitle2(codecctx, &subtitle, &gotSubtitle, &pkt)前后AVPacket和AVSubtitle的内容：



可以发现，pkt.pts is 29750，对应的时间基为所属AVStream的time\_base（这里为{1,1000}），转为秒即29.750，转换为时:分:秒.毫秒格式即0:0:29,750

pkt.duration is 2025,

pkt.pts + pkt.duration=31775，转换为时:分:秒.毫秒格式即 0:0:31,775，与字幕文件符合



AVSubtitle::pts以{1,AV\_TIME\_BASE}为时间基，即除以AV\_TIME\_BASE得到为秒数

AVSubtitle::start\_display\_time和end\_display\_time以解码参数AVCodecContext的pkt\_timebase为时间基，即乘以pkt\_timebase即为秒数

解码参数AVCodecContext的pkt\_timebase等于被解码的AVStream的time\_base

解码生成的AVSubtitle中

uint32\_t start\_display\_time; //relative to packet pts, in ms

uint32\_t end\_display\_time; //relative to packet pts, in ms

int64\_t pts; //Same as packet pts, in AV\_TIME\_BASE

因此：

start\_display\_time=0，pkt.pts+0即字幕开始显示的时间戳，为29750

end\_display\_time=2030，pkt.ps+2030即字幕结束显示的时间戳，约等于31775

此时，subtitle.rects[0]->ass中内容为：

Dialogue: 0,0:00:29.75,0:00:31.78,Default,,0,0,0,,鍦ㄥ彜鑰佺殑鏃朵唬

**dvb\_teletext和dvb\_subtitle字幕：**

字幕格式包括：

AV\_CODEC\_ID\_DVB\_TELETEXT

AV\_CODEC\_ID\_DVB\_SUBTITLE

AV\_CODEC\_ID\_SRT

其中，

dvb\_teletext、srt字幕是基于文本的

dvd\_subtitle将视频中的每条字幕存储为一个位图

对一个多字幕轨的源nature\_east2.ts进行分析，

ffrobe -i nature\_east2.ts

输出显示共一个节目，13路媒体，前4路媒体为

Stream #0:0[0x23](scr): Subtitle: dvb\_subtitle ([6][0][0][0] / 0x0006)

Stream #0:1[0x24](est,lit,lav): Subtitle: dvb\_teletext ([6][0][0][0] / 0x0006)

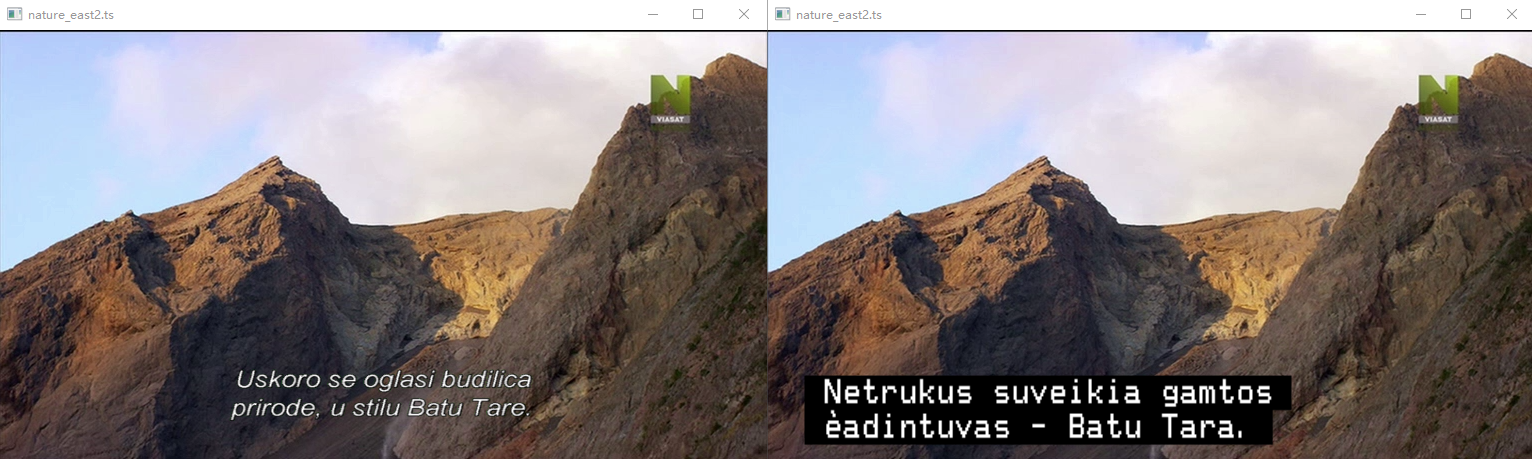
Stream #0:2[0x1f4]: Video: h264, yuv420p, 768x432 [SAR 1:1 DAR 16:9], 25 fps, 25 tbr, 90k tbn, 50 tbc

Stream #0:3[0x2bc](rus): Audio: aac (LC) ([15][0][0][0] / 0x000F), 48000 Hz, stereo, fltp, 96 kb/s

ffplay播放字幕：

ffplay nature\_east2.ts -vst 2 -ast 3 -sst 0（scr字幕，下图左）

ffplay nature\_east2.ts -vst 2 -ast 3 -sst 1（est,lit,lav字幕，下图右）



可以看到不同轨道的字幕在视频中的出现时间、位置是相同的

ffprobe查看字幕时间戳：

ffprobe -i nature\_east2.ts -select\_streams 0 -show\_packets -of xml显示的最后5帧为

<packet codec\_type="subtitle" stream\_index="0" pts="5228912804" pts\_time="58099.031156" flags="K\_"></packet>

<packet codec\_type="subtitle" stream\_index="0" pts="5229414074" pts\_time="58104.600822" flags="K\_"></packet>

<packet codec\_type="subtitle" stream\_index="0" pts="5229474661" pts\_time="58105.274011" flags="K\_"></packet>

<packet codec\_type="subtitle" stream\_index="0" pts="5229757837" pts\_time="58108.420411" flags="K\_"></packet>

<packet codec\_type="subtitle" stream\_index="0" pts="5230174388" pts\_time="58113.048756" flags="K\_"></packet>

可以看到dvd\_subtitle字幕帧的帧率很低

ffprobe -i nature\_east2.ts -select\_streams 0 -show\_packets -of xml显示的最后5帧为：

<packet codec\_type="subtitle" stream\_index="1" pts="5230280283" pts\_time="58114.225367" flags="K\_"></packet>

<packet codec\_type="subtitle" stream\_index="1" pts="5230283505" pts\_time="58114.261167" flags="K\_"></packet>

<packet codec\_type="subtitle" stream\_index="1" pts="5230286673" pts\_time="58114.296367" flags="K\_"></packet>

<packet codec\_type="subtitle" stream\_index="1" pts="5230289951" pts\_time="58114.332789" flags="K\_"></packet>

<packet codec\_type="subtitle" stream\_index="1" pts="5230293118" pts\_time="58114.367978" flags="K\_"></packet>

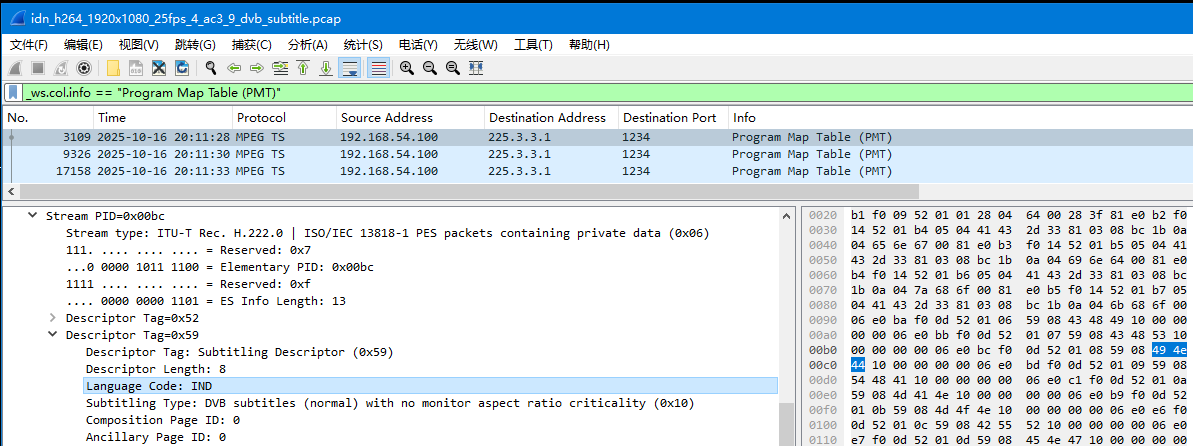
可以发现dvd\_teletext字幕帧的帧率接近视频帧率

**dvb\_teletext、dvb\_subtitle的mpegts打包：**

mpegts打包dvb\_teletext时，标准要求无自适应域、pes header data length固定为36（故常常是5字节的pts+31字节的0xff填充）。

mpegts打包dvb\_subtitle时，无类似“奇怪”的要求

注意：mpegts打包dvb\_subtitle时，pmt表里字幕描述符里Composition Page ID和Ancillary Page ID不能随意设置



上述源，字幕透传时，如果修改Composition Page ID为1、Ancillary Page ID为1，则potplayer播放时能显示字幕，vlc播放时无法显示字幕

**dvb\_teletext字幕标准：**

dvb\_teletext字幕遵循ETSI EN 300472标准（传输）、ETSI EN 300706标准（内容格式）

Teletext即我们常说的“图文电视”，在MPEG-2中图文信息已经成为数字化信息在TS流中传输。但最终数字化的图文信息都将转化为模拟的VBI（垂直消隐间隔）信息插入到正常的电视信号中实现显示

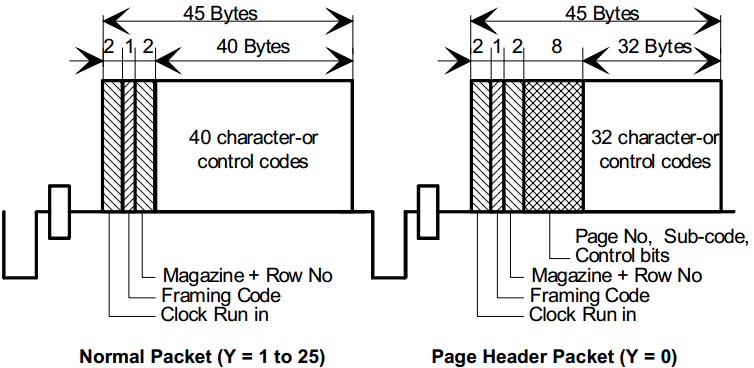
A Teletext data packet occupies one TV line. Each packet starts with a clock run-in and framing code sequence

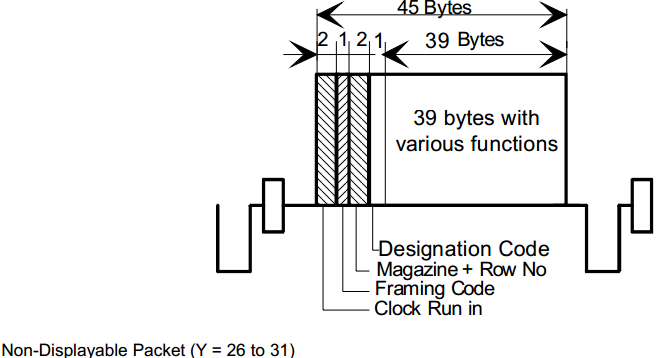
一个Teletext data packet共45字节：

* 前2字节是clock run-in：固定为10101010 10101010
* 第3个字节是framing code：固定为11100100
* 字节4是magazine
* 字节5是packet number Y（Y取值0~31）
* 后40个字节携带address, control, character or data information

packet\_number决定了Teletext data packet的类型，如下：

1. the page header (Y = 0);
2. normal packets intended for direct display (Y = 1 to 25);
3. non-displayable packets (Y = 26 to 31).

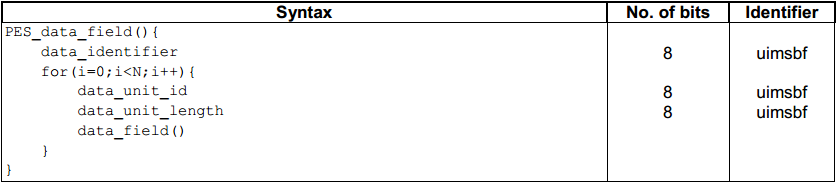




packets with Y = 26 to 31 use a designation code to extend the packet address range

Packets with Y = 0 are referred to as page header packets. They contain unique information for a page including the page address.  
Following the page header packet of a page, all subsequent packets with Y = 1 to Y = 28 inclusive, from the same magazine, relate to that page.  
The transmission of a given page begins with, and includes, its page header packet. It is terminated by and excludes the next page header packet having the same magazine address in parallel transmission mode, or any magazine address in serial transmission mode

根据ETSI EN 300472标准，封装dvb\_teletext字幕的pes包的结构为：



dvb\_teletext字幕所在的mpegts包示例（16进制展示）：

0000 47 40 fa 1e 00 00 01 bd 00 b2 84 80 24 27 2a b5

0010 be 81 ff ff ff ff ff ff ff ff ff ff ff ff ff ff

0020 ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff

0030 ff 10 02 2c e8 e4 a8 57 40 0b 40 a8 a8 a8 0b 00

0040 00 30 60 03 20 e8 6c 40 00 00 00 00 04 04 04 04

0050 04 04 04 04 04 04 04 04 04 04 04 04 04 04 04 04

0060 02 2c c8 e4 a8 57 40 0b 40 a8 a8 a8 0b 00 00 30

0070 60 03 20 e8 6c 40 00 00 00 00 04 04 04 04 04 04

0080 04 04 04 04 04 04 04 04 04 04 04 04 04 04 ff 2c

0090 ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff

00a0 ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff

00b0 ff ff ff ff ff ff ff ff ff ff ff ff

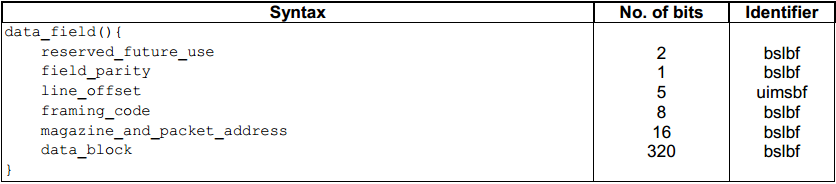
分析：

* 前49字节=4字节的ts头+3字节packet start code prefix+1字节的stream\_id+2字节的PES packet length+2字节+1字节的pes header data length+5字节的pts+31字节的pes填充
* 后139字节为pes data部分，0x10022ce8e4a857400b40a8a8a80b000030600320e86c40000000000404040404040404……（共139字节）

其中pes data部分，

* 0x10，数据标识符（Data Identifier），表示这是 EBU Data
* 0x02，data\_unit\_id，0x02表示EBU Teletext non-subtitle data，0x03表示EBU Teletext subtitle data，0xff表示data\_unit for stuffing
* 0x2c，该字段后的数据长度，data\_unit\_id为2或3时，长度固定为44
* 0xe8~0x04：44字节的data\_field
* 因此，pes data部分可以拆分为3个data\_unit，即图中绿、黄、蓝三部分139=1 + (2+44) + (2+44) + (2+44)

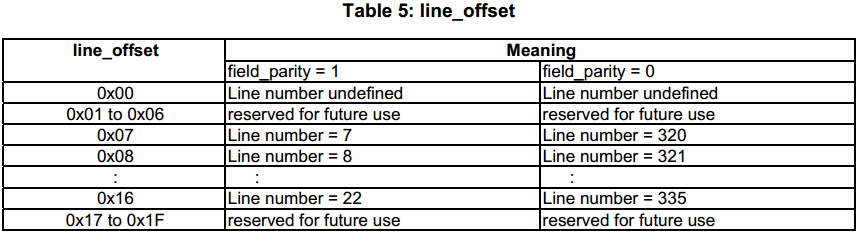
分析data\_unit的data\_field部分，根据ETSI EN 300472标准：



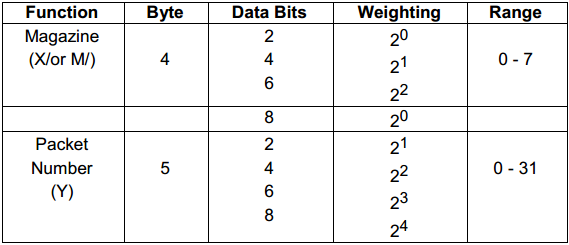
**field\_parity:** this 1-bit flag specifies the field for which the data is intended; the value "1" indicates the first field of a frame, the value "0" indicates the second field of a frame.

**line\_offset:** this 5-bit field specifies the line number on which the Teletext data packet is intended to be presented if it is transcoded into the VBI. Within a field, line\_offset编号应保持递增，但未定义的line\_offset值“0”除外. field\_parity标志的变化indicates a new field. The line\_offset shall be coded according to table 5.

**framing\_code, magazine\_and\_packet\_address, data\_block:** these fields correspond to the 43 bytes following the clock-run-in sequence of an EBU Teletext data packet



packet address包括magazine和packet\_number，各占1字节，采用Hamming 8/4编码：



以第一个data\_unit为例，0xE8 E4 A8 57 40……

0xE8=11101000

* reserved\_future\_use：11
* field\_parity：1
* line\_offset：01000

frame\_code：0xE4，即11100100

magazine：0xA8，即10101000，取比特2、4、6位是000，加权后为0

packet\_number：0x57，即01010111，取比特2、4、6、8位是1111，加权后为30

一个Y=0的示例：

0000 47 40 fa 1c 00 00 01 bd 00 b2 84 80 24 27 2a b9

0010 48 41 ff ff ff ff ff ff ff ff ff ff ff ff ff ff

0020 ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff

0030 ff 10 03 2c e8 e4 a8 a8 57 57 a8 a8 a8 a8 a8 a8

0040 04 04 04 04 04 04 04 04 04 04 04 04 04 04 04 04

0050 04 04 04 04 04 04 04 04 04 04 04 04 04 04 04 04

0060 02 2c c8 e4 a8 57 40 0b 40 a8 a8 a8 0b 00 00 30

0070 60 03 20 e8 6c c0 00 00 00 00 04 04 04 04 04 04

0080 04 04 04 04 04 04 04 04 04 04 04 04 04 04 ff 2c

0090 ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff

00a0 ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff

00b0 ff ff ff ff ff ff ff ff ff ff ff ff

分析：

0xE8=11101000

* reserved\_future\_use：11
* field\_parity：1
* line\_offset：01000

frame\_code：0xE4，即11100100

magazine：0xA8，即10101000，取比特2、4、6位是000，加权后位0

packet\_number：0xA8，即10101000，取比特2、4、6、8位是000，加权后位0

page number：0x57 0x57，0x57即01010111，page units取2、4、6、8bit并加权后是15，page tens是15

page sub-code

control bits

**ff\_h2645\_packet\_split：**

**函数原型：**

ff\_h2645\_packet\_split用于将形参buf+length指向的H.264（AnnexB或avcC）码流中的一个个NALU提取出来，解析NALU Header，分别将每个NALU的NALU Header中的属性，EBSP、RBSP和SODB存贮到形参pkt指向的内存中

int ff\_h2645\_packet\_split(H2645Packet \*pkt, const uint8\_t \*buf, int length,

                          void \*logctx, int is\_nalff, int nal\_length\_size,

                          enum AVCodecID codec\_id, int small\_padding, int use\_ref);

buf+length：一个AVPacket的data

is\_nalff：值为1表示为MP4 (AVCC/HVCC)格式，值为0表示为annex B格式

nal\_length\_size：用于在MP4格式时指定NALU的长度

pkt是输出型参数，H2645Packet类型，定义如下：

typedef struct H2645Packet {

    H2645NAL \*nals;

    H2645RBSP rbsp;

    int nb\_nals;

    int nals\_allocated;

    unsigned nal\_buffer\_size;

} H2645Packet;

调用ff\_h2645\_packet\_split函数后，

指针pkt->nals会指向一个H2645NAL类型的数组，数组共nb\_nals个元素，每个元素都存放着一个从H264码流中提取出来的NALU的信息。比如pkt->nals[0]存放第一个NALU的信息，pkt->nals[1]存放第二个NALU的信息，以此类推。

H2645NAL定义如下：

typedef struct H2645NAL {

    const uint8\_t \*data;

    int size;

*/\*\**

     \* Size, in bits, of just the data, excluding the stop bit and any trailing

     \* padding. I.e. what HEVC calls SODB.

     \*/

    int size\_bits;

    int raw\_size;

    const uint8\_t \*raw\_data;

    GetBitContext gb;

    int type;  *// NAL unit type*

    int ref\_idc;  *// H.264 only, nal\_ref\_idc*

    int temporal\_id;  *// HEVC only, nuh\_temporal\_id\_plus\_1 - 1*

    int nuh\_layer\_id;  *// HEVC only, identifier of layer to which nal unit belongs*

    int skipped\_bytes;

    int skipped\_bytes\_pos\_size;

    int \*skipped\_bytes\_pos;

} H2645NAL;

调用ff\_h2645\_packet\_split函数后，

pkt->nals[idx]->data变为：指向某个缓冲区的指针。该缓冲区存放第idx+1个NALU的"NALU Header + RBSP"。即去掉了startcode或NALUnitLength、去掉了防竞争字节，但保留了NALU Header。

pkt->nals[idx]->size变为：data指向的缓冲区的大小，单位为字节。

pkt->nals[idx]->raw\_data变为：指向某个缓冲区的指针。该缓冲区存放第idx+1个NALU的"NALU Header + EBSP"。即去掉了startcode或NALUnitLength，但保留了NALU Header和防竞争字节。

pkt->nals[idx]->raw\_size变为：raw\_data指向的缓冲区的大小，单位为字节。

pkt->nals[idx]->gb.buffer的值等于：pkt->nals[idx]->data。

pkt->nals[idx]->gb.buffer\_end的值等于：pkt->nals[idx]->data + pkt->nals[idx]->size - 1，有时会再加1

pkt->nals[idx]->gb.index变为8：表示读取完了该NALU的第一个字节（NALU Header，8位）

pkt->nals[idx]->gb.size\_in\_bit的值等于：pkt->nals[idx]->size\_bits。

pkt->nals[idx]->gb.size\_in\_bits\_plus8的值等于：pkt->nals[idx]->gb.size\_in\_bit + 8。

H2645RBSP定义如下：

typedef struct H2645RBSP {

    uint8\_t \*rbsp\_buffer;

    AVBufferRef \*rbsp\_buffer\_ref;

    int rbsp\_buffer\_alloc\_size;

    int rbsp\_buffer\_size;

} H2645RBSP;

rbsp->rbsp\_buffer变为：存贮去掉startcode（或NALUnitLength）和防竞争字节后的H264码流

rbsp->rbsp\_buffer\_size 变为：rbsp->rbsp\_buffer的大小，单位为字节。

注意：

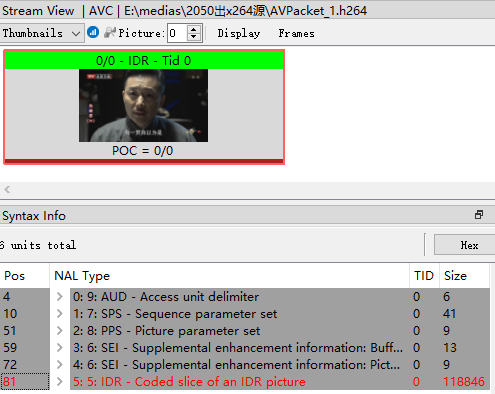
rbsp->rbsp\_buffer并非存放所有NALU的Header+RBSP

部分pkt->nals[idx]->data指向的是rbsp->rbsp\_buffer描述的缓冲区

另外一些pkt->nals[idx]->data的值和raw\_data相同，指向原始码流缓冲区

**示例：**

AVPacket\_1.h264的size为118924字节，包含6个NALU，依次是aud+sps+pps+buffering\_period\_sei+pic\_timing\_sei+idrslice，



如上图，文件总size=(4+2)+(4+37)+(4+5)+(3+10)+(3+6)+( 3+118843) =118924，每组括号内第一个数为起始码长度，第2个数为NALU长度（含NALU Header及防竞争字节，即nalu\_header+ebsp）。

首先，调用av\_read\_frame()返回一个且仅一个AVPacket，其.data=0x555555609660、.size=118924，内容与文件AVPacket\_1.h264的二进制数据完全一致；

然后，调用函数ff\_h2645\_packet\_split(pkt, const uint8\_t \*buf=0x555555609660, length=118924,

logctx=nullptr, is\_nalff=0, nal\_length\_size=0, codec\_id=AV\_CODEC\_ID\_H264, small\_padding=1, use\_ref=1)处理这个AVPacket，函数执行完后，

pkt->nb\_nals=6

pkt->nals[0],data=0x555555721410,size=3,size\_bits=11,raw\_data=0x555555609664,raw\_size=3,type=9(AUD),

gb={buffer=0x555555721410,buffer\_end=0x555555721412,index=8,size\_in\_bits=11}

pkt->nals[1],data=0x555555721413,size=36,size\_bits=279,raw\_data=0x55555560966a,raw\_size=38,type=7(SPS),

gb={buffer=0x555555721413"gz",buffer\_end=0x555555721436"",index=8,size\_in\_bits=279}

pkt->nals[2],data=0x555555609693,size=5,size\_bits=34,raw\_data=0x555555609693,raw\_size=5type=8(PPS),

gb={buffer=0x555555609693,buffer\_end=0x555555609698,index=8,size\_in\_bits=34}

pkt->nals[3],data=0x555555721439,size=9,size\_bits=64,raw\_data=0x55555560969b,raw\_size=10,type=6(SEI),

gb={buffer=0x555555721439,buffer\_end=0x555555721441,index=8,size\_in\_bits=64}

pkt->nals[4],data=0x5555556096a8,size=6,size\_bits=40,raw\_data=0x5555556096a8,raw\_size=6,type=6(SEI),

gb={buffer=0x5555556096a8,buffer\_end=0x5555556096ad,index=8,size\_in\_bits=40}

pkt->nals[5],data=0x555555721443,size=118842,size\_bits=950735,raw\_data=0x5555556096b1,raw\_size=118843,type=5(IDR),

gb={buffer=0x555555721443,buffer\_end=0x55555573e47d,index=8,size\_in\_bits=950735}

对以上数据分析，

* + 1. NALU Header + EBSP

AUD: pkt->nals[0]->raw\_data=0x555555609664,恰好=0x555555609660+4(起始码)

SPS: pkt->nals[1]->raw\_data=0x55555560966a,恰好=pkt->nals[0]+2(aud)+4(起始码)

PPS: pkt->nals[2]->raw\_data=0x555555609693,恰好=pkt->nals[1]+37(sps)+4(起始码)

SEI: pkt->nals[3]->raw\_data=0x55555560969b,恰好=pkt->nals[2]+5(pps)+3(起始码)

SEI: pkt->nals[4]->raw\_data=0x5555556096a8,恰好=pkt->nals[3]+10(sei)+3(起始码)

IDR: pkt->nals[5]->raw\_data=0x5555556096b1,恰好=pkt->nals[4]+6(sei)+3(起始码)

因此pkt->nals[idx]->raw\_data指向的缓冲区实际即输入参数buf描述的缓冲区的一部分。

即pkt->nals[idx]->raw\_data的内容是AVPacket\_1.h264二进制数据的一个子集，是移除了起始码的结果。

注意输出结果里有个奇怪的地方：

pkt->nals[0]->raw\_size=3，实际aud nalu header+ebsp为2字节，末尾多了1字节的0x00

pkt->nals[1]->raw\_size=38，实际sps nalu header+ebsp为37字节，末尾多了1字节的0x00

其余pkt->nals[idx]的raw\_size都与实际相符

* + 1. GetBitContext gb

AUD: pkt->nals[0]->gb.buffer= data，gb.buffer\_end= gb.buffer+ size-1

SPS: pkt->nals[1]->gb.buffer= data，gb.buffer\_end= gb.buffer+ size-1

PPS: pkt->nals[2]->gb.buffer= data，gb.buffer\_end= gb.buffer+ size

SEI: pkt->nals[3]->gb.buffer= data，gb.buffer\_end= gb.buffer+ size-1

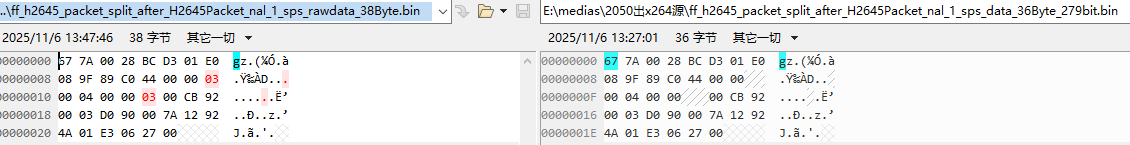
SEI: pkt->nals[4]->gb.buffer= data，gb.buffer\_end= gb.buffer+ size-1

IDR: pkt->nals[5]->gb.buffer= data，gb.buffer\_end= gb.buffer+ size

* + 1. NALU Header + RBSP

pkt->nals[idx]->data与raw\_data的内容相比，移除了防竞争字节0x03

如下图，左边为pkt->nals[1]->raw\_data，38字节，右侧为pkt->nals[1]->data，36字节



pkt->rbsp.rbsp\_buffer= 0x555555721410，rbsp\_buffer\_alloc\_size = 126456, rbsp\_buffer\_size = 118894

pkt->rbsp.rbsp\_buffer\_ref.data=0x555555721410

pkt->rbsp.rbsp\_buffer\_ref.size= 126456

pkt->rbsp.rbsp\_buffer\_ref.buffer= {data = 0x555555721410, size = 126456, refcount = 1, ……}

pkt->nals[2]->data与raw\_data相同，size和raw\_size相同，SPS

pkt->nals[4]->data与raw\_data相同，size和raw\_size相同，pic\_timing SEI

pkt->nals[0]->data= 0x555555721410，AUD

pkt->nals[1]->data= 0x555555721413,恰好= pkt->nals[0]->data+3(raw\_size)，SPS

pkt->nals[3]->data=0x555555721439,恰好=pkt->nals[1]+38(raw\_size)，buffering\_period SEI

pkt->nals[5]->data=0x555555721443,恰好=pkt->nals[3]+10(raw\_size), IDR

因此，

pkt->nals[2]->data、pkt->nals[4]->data未指向pkt->rbsp.rbsp\_buffer描述的缓冲区，而是指向存放原始输入码流的缓冲区

pkt->nals[0\1\3\5]->data指向的是rbsp.rbsp\_buffer描述的缓冲区的一部分

将idx=0、1、3、5四个pkt->nals[i]的raw\_size加起来=3+38+10+118843=118894，恰好是rbsp\_buffer\_size，这说明rbsp.rbsp\_buffer里只包含了AUD、SPS、buffering\_period SEI、IDR四个NALU的RBSP，并未存储PPS、pic\_timing SEI的。

rbsp.rbsp\_buffer\_size并非各个pkt->nals[0\1\3\5]->size之和，而是pkt->nals[0\1\3\5]->raw\_size之和（猜测这样只是为了在确定rbsp\_buffer\_size时更简单，宁多毋少）。因此rbsp.rbsp\_buffer中会有些一些多余的填充字节。

在rbsp.rbsp\_buffer中，

AUD：占0~2，3字节

SPS：占3~0x26，36字节。之后有2字节的0x00 0x00填充

buffering\_period SEI： 占0x29~0x31，9字节。 之后有1字节的0x00填充

IDR：占0x33~0x01D06C，118842字节。之后有1字节的0x00填充

**dts2pts比特流滤波器：**

ffmpeg -i ref\_frames2.h264 -c copy -bsf:v dts2pts output.mp4

滤波器实现代码见源文件dts2pts\_bsf.c

实现思路示例说明：

h264裸流ref\_frames2.h264，B帧数为2，GOP结构为：IBBPBBPBBP……

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 解码序： | 0IDR | 1P | 2B | 3B | 4P | 5B | 6B | 7P | 8B | 9B |
| poc： | 0 | 6 | 2 | 4 | 12 | 8 | 10 | 18 | 14 | 16 |
| dts ： | -48000 | 0 | 48000 | 96000 | 144000 | 192000 | 240000 | 288000 | 336000 | 384000 |

显示序上相邻帧的poc差值是2，记作poc\_diff=2

解码序上：

0号帧的pts取自1号帧的dts

1号帧，poc比0号帧大6，故显示序上相隔了3帧，所以pts应该取自1+3=4号帧的dts

2号帧，poc比0号帧大2，故显示序上相隔了1帧，所以pts应该取自1+1=2号帧的dts

3号帧，poc比0号帧大4，故显示序上相隔了2帧，所以pts应该取自1+2=3号帧的dts

4号帧，poc比0号帧大12，故显示序上相隔了6帧，所以pts应该取自1+6=7号帧的dts

n号帧，poc比0号帧大m，故显示序上相隔了m/poc\_diff帧，所以pts应该取自1+(m/poc\_diff)帧的dts

void \*av\_tree\_find(const struct AVTreeNode \*root, void \*key, int (\*cmp)(const void \*key, const void \*b), void \*next[2]);

当没找到匹配节点时，next 数组的内容是最接近搜索键的两个节点：

next[0]：前驱节点 - 树中比搜索键小的最大节点

next[1]：后继节点 - 树中比搜索键大的最小节点

滤波器源码分析：

av\_read\_frame()读取出AVPacket，

调用ff\_h2645\_packet\_split、cbs\_h264\_read\_sps等函数解析AVPacket，获取sps、pps、slice、poc

滤波器维护了2个数据结构，AVTreeNode和AVFifo

typedef struct DTS2PTSFrame {

    AVPacket    \*pkt;

    int          poc;

    int     poc\_diff;

    int          gop;

} DTS2PTSFrame;

av\_fifo\_write写入fifo中的是DTS2PTSFrame，每个AVPacket，对应写入一个

DTS2PTSFrame{AVPacket\*, poc, poc\_diff, DTS2PTSContext.gop}

这里poc即当前图像的FFMIN(pic\_field\_poc[0], pic\_field\_poc[1]);

每次向fifo中写入一个AVPacket的同时，会向tree里插入1或2个节点

typedef struct DTS2PTSNode {

    int64\_t      dts;

    int64\_t duration;

    int          poc;

    int          gop;

} DTS2PTSNode;

av\_tree\_insert插入树中的是DTS2PTSNode，

poc\_diff是1或者2，如果是2，则每一个AVPacket对应插入2个DTS2PTSNode：

第一个DTS2PTSNode{AVPacket.dts, AVPacket.duration, DTS2PTSContext.nb\_frame, DTS2PTSContext.gop}

第二个DTS2PTSNode{AVPacket.dts+AVPacket.duration/2, AVPacket.duration, DTS2PTSContext.nb\_frame+1, DTS2PTSContext.gop}

使用av\_tree\_insert往树中插入节点时，插入的key是DTS2PTSNode\*，比较函数定义为先比较DTS2PTSNode::poc（实际为DTS2PTSContext.nb\_frame）再比较DTS2PTSNode::gop

给fifo分配的size为H264\_MAX\_DPB\_FRAMES \* 2 \* 2=64，记作fifo\_size

初始时一直向fifo中av\_fifo\_write写数据（同时伴随着av\_tree\_insert插入树节点），直到fifo中缓存的数据个数达到fifo\_size后，才开始一个边av\_fifo\_write写数据，一边av\_fifo\_read读数据。

调av\_fifo\_read读取到一个DTS2PTSFrame，然后使用av\_tree\_find从树中查找对应的DTS2PTSNode节点，查找时比较的是DTS2PTSFrame的poc和DTS2PTSNode的poc，如果poc一样，则继续比较DTS2PTSFrame的gop和DTS2PTSNode的gop

找到对应的DTS2PTSNode节点后，令AVPacket.pts=DTS2PTSNode.dts

从初始到向fifo中写入10个AVPacket后，状态如下：

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| fifo解码序： | 0IDR | 1P | 2B | 3B | 4P | 5B | 6B | 7P | 8B | 9B |
| poc： | 0 | 6 | 2 | 4 | 12 | 8 | 10 | 18 | 14 | 16 |
| gop： | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| dts： | -48000 | 0 | 48000 | 96000 | 144000 | 192000 | 240000 | 288000 | 336000 | 384000 |
|  |  |  |  |  |  |  |  |  |  |  |
| tree |  |  |  |  |  |  |  |  |  |  |
| nb\_frame： | -2 | -1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| gop： | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| dts： | -48000 | -24000 | 0 | 24000 | 48000 | 72000 | 96000 | 120000 | 144000 | 168000 |

注意：DTS2PTSContext.nb\_frame的初值为-(ctx->par\_in->video\_delay << 1)，对ref\_frames2.h264，par\_in->video\_delay为1，因此这里nb\_frame的初值为-2

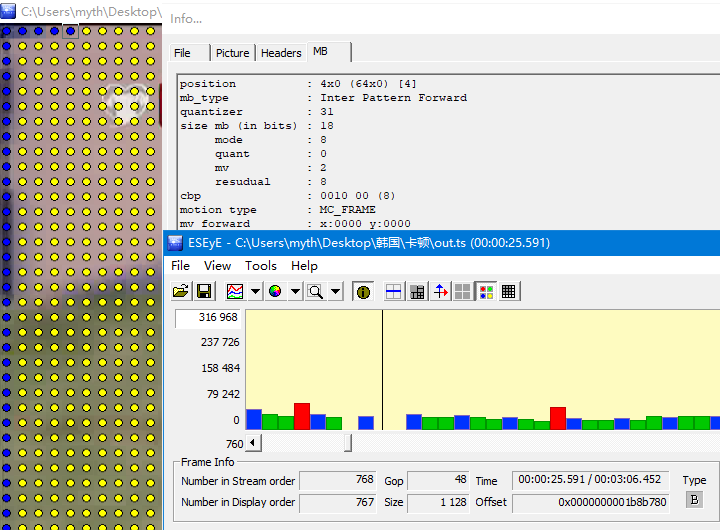
因此，首次av\_fifo\_read并查找时，IDR帧的(poc,gop)=(0,0)，所以在tree中找到的对应节点是(nb\_frame=0,gop=0,dts=0)

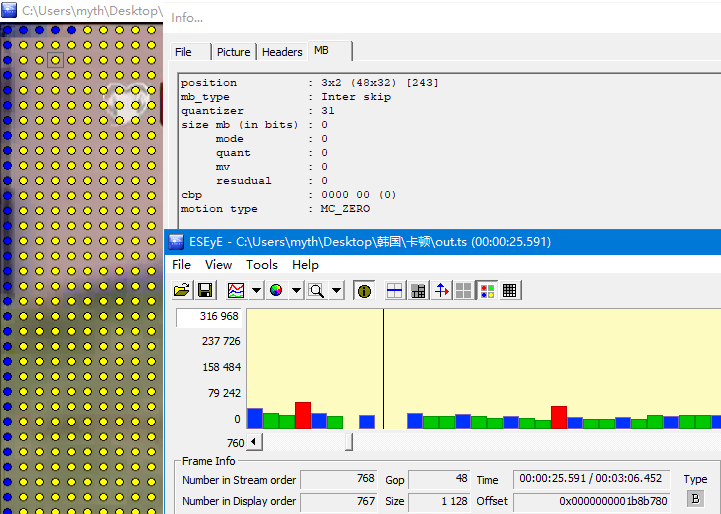
**mpeg2的Inter skip宏块、MC\_ZERO运动矢量：**

一个踢足球的素材，h264 1920x1080 59.94i，用intel media sdk转码为mpeg2 1920x1080 59.94i。

发现编码出的mpeg2码流中意外出现了很多静帧。画面相对于上一副图像是静止的，但编码前都是运动场景。增大码率、修改gop size、修改B帧数是避免这种现象的一些方法。

Elecard StreamEye查看mpeg2码流，发现静帧的大小都很小，约1000byte左右





Elecard StreamEye宏块类型按钮说明：

The **MB types** button  visualizes macroblock types in the video window.

For MPEG-1/2 streams:

* Red circles – Intra.
* Yellow circles – Not Coded (in B- and P-pictures).
* Blue circles – Forward (in B- and P-pictures).
* Green circles – Backward (in B- and P-pictures).
* Green-blue circles – Forward, Backward (in B-pictures).

从Elecard显示可以看到，静帧的都是一些B帧。在该B帧中除了画面边缘一些区域外，都是黄色宏块，即Inter skip宏块。如上图x=3,y=2位置的宏块信息如下：

* mb\_type: Inter skip -宏块类型
* motion type: MC\_ZERO - 运动矢量为零（而非运动矢量残差为0）
* cbp: 0000 00 - 没有残差系数
* size mb: 0 bits - 该宏块不占用任何比特

由于运动矢量为0，也无像素残差数据，因此相对于参考帧，画面是静止的

在MPEG-2标准中，一个宏块的大小是16×16像素。cbp是coded block pattern的缩写。  
cbp一共6bit，高2bit表示cbpc，值为2表示cb、cr中至少一个4x4块的AC系数不全为0，为1表示cb、cr中至少一个2x2的DC系数不全为0，为0表示所有色度系数全0。  
低4bit分别表示4个8x8亮度块，其中从最低一位开始的4位分别对应00，10，01，11位置的8\*8亮度块。如果某位为1，表示该对应8\*8块的4个4\*4块中至少有一个的系数不全为0。