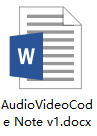
# AudioVideoCode Note v1



承接《AudioVideoCode Note v1》

本文档主要包含：st2110等内容

**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：

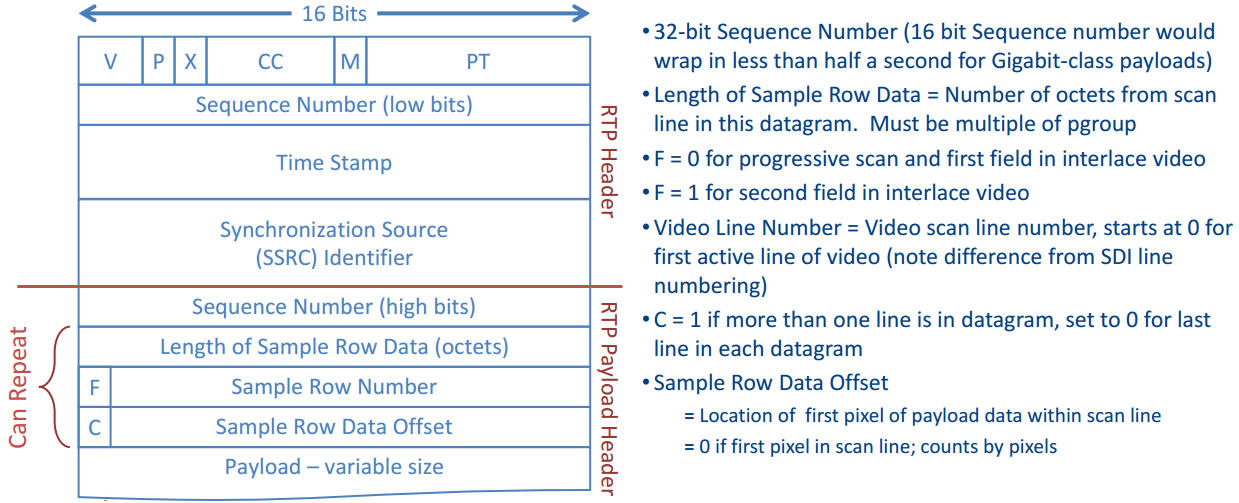
”a=ts-refclk:localmac=7C-E9-D3-1B-9A-AF”

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：

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

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。

在sdp中a=fmtp:开头的行里可以指定像素宽高比sar

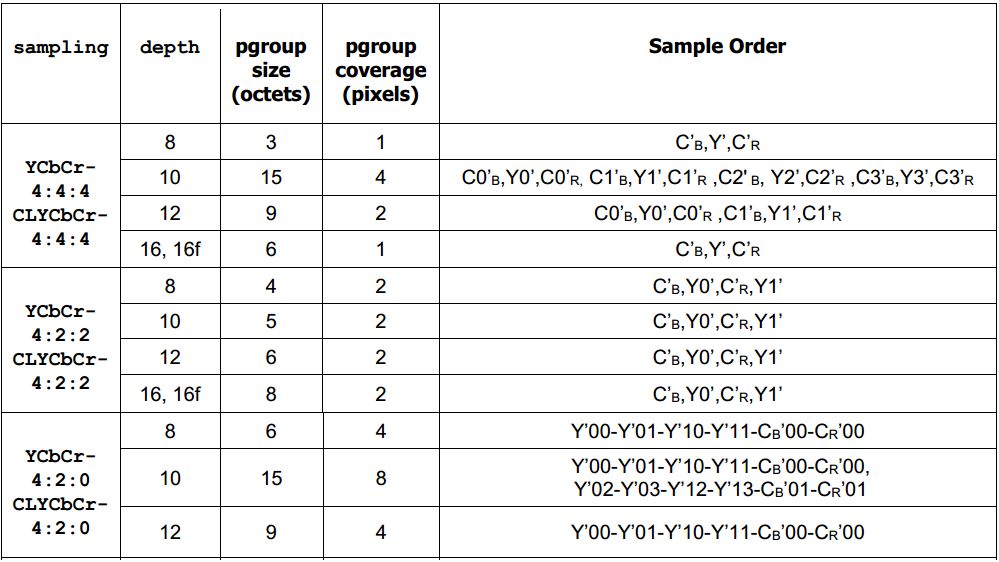
打包模式分为GPM和BPM两种

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。



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

**st2110-30：**

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

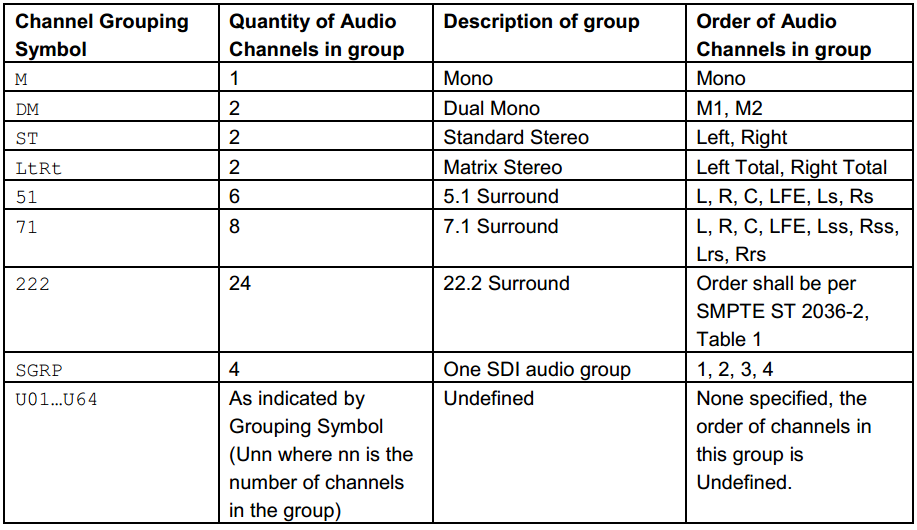
音频必须使用Standard UDP Size Limit（1460）

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



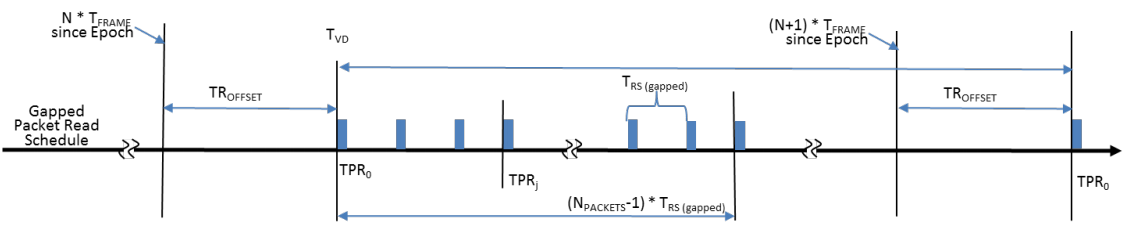
**st2110-21：**

**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

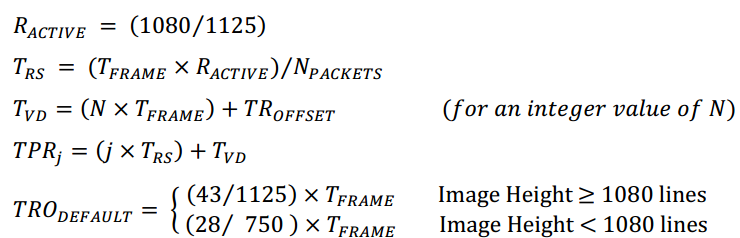
TRS：同一帧内相邻两个rtp包的时间间隔

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

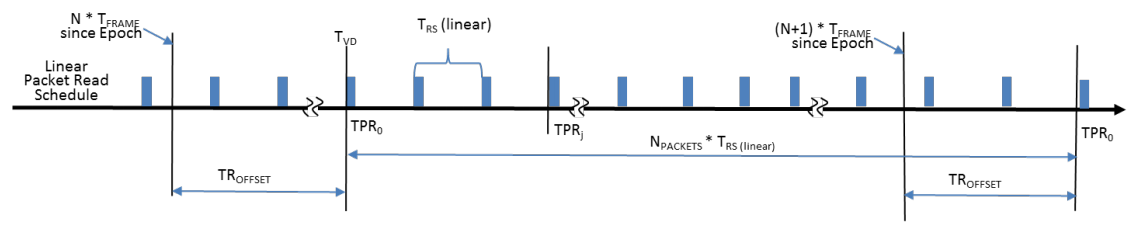
NPACKETS：一帧包含多少个rtp包

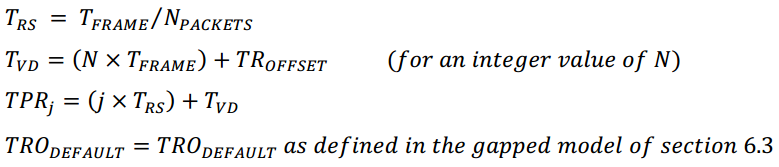
TPRj：帧内j号rtp包的传送时间

TVD：帧内第一个rtp包的传送时间，也即TPR0



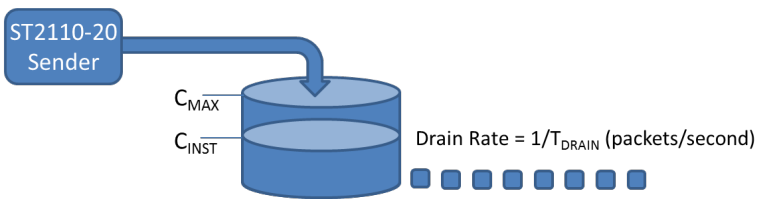
（2）Linear Packet Read Schedule (LPRS)：所有rtp packet均匀发送





**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

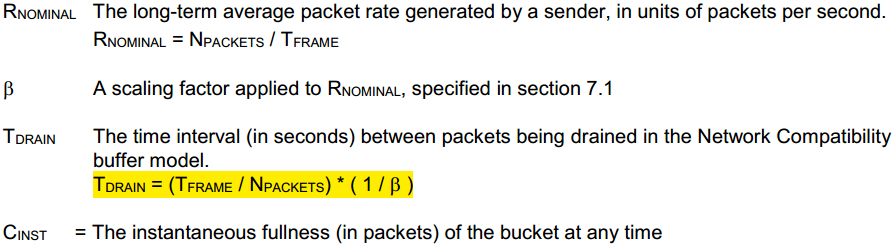


sender后接一个无限容量的漏桶

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

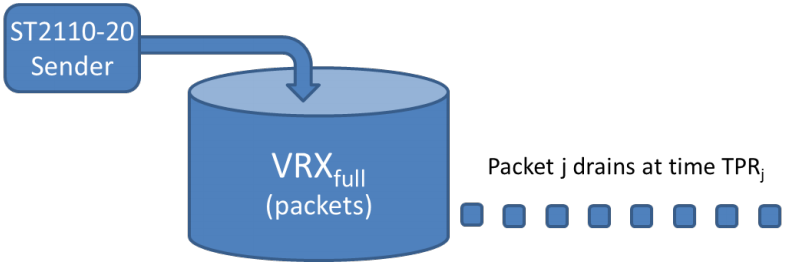
CINST表示任意时刻，漏桶中的包数目

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

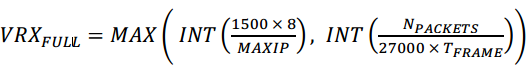
漏桶不能上溢、不能下溢

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

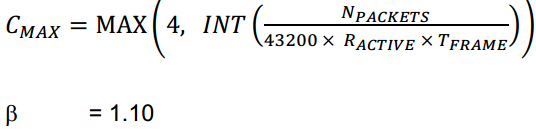
**Narrow Senders (Type N) ：**

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



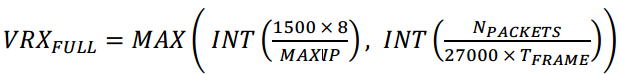
其中，TFrame单位为sec，即等于1/fps

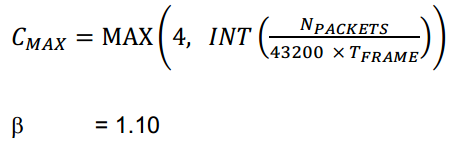
MAXIP为1460（当使用Standard UDP Size Limit时）或8960（当使用Extended UDP Size Limit时）



**Narrow Linear Senders (Type NL)：**

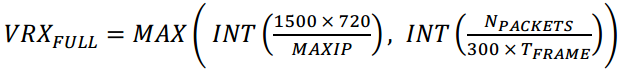
Narrow Linear Senders必须符合LPRS、virtual Receiver Buffer模型、Network Compatibility模型：

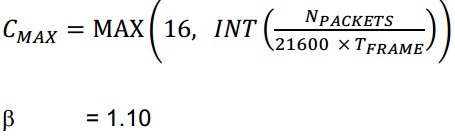




**Wide Senders (Type W)：**

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





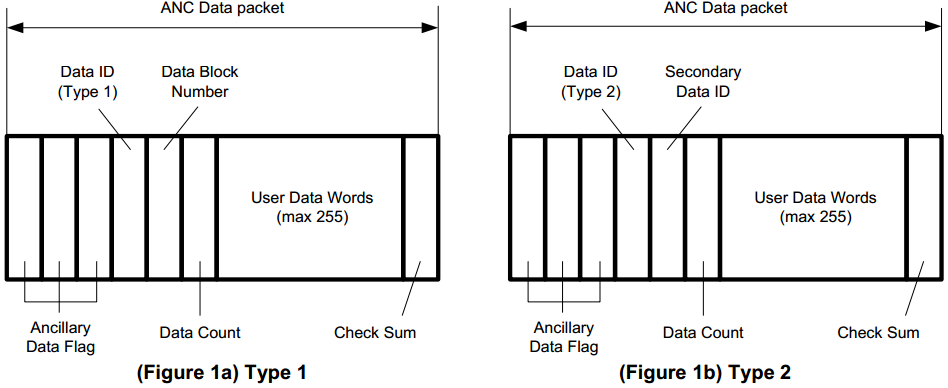
**st2110-40：**

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。

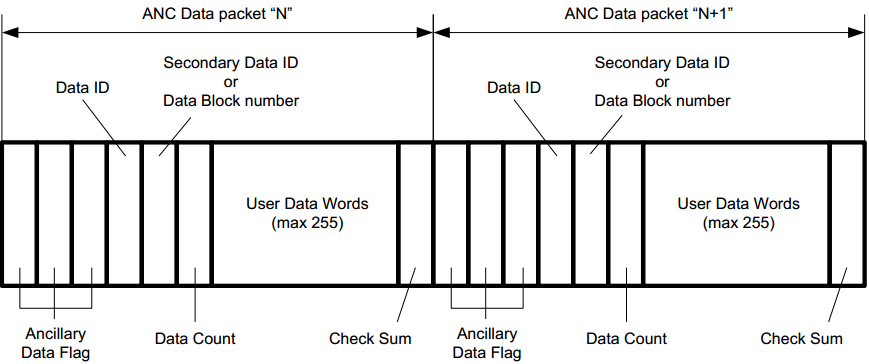
c) A data block number (DBN) word (for Type 1 only) that distinguishes successive ancillary data packets with a common data ID. DBN共10bit，记为b9b8……b0，其中b7~b0是8bit的block number，b8是b7~b0的even parity校验，b9是b8的取反。

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.

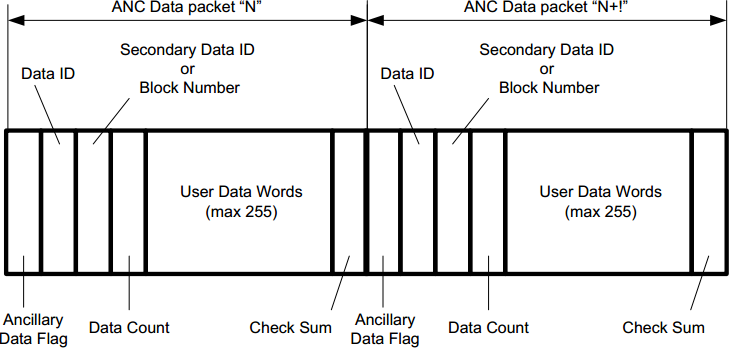
f) A checksum (CS) word.

**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：**

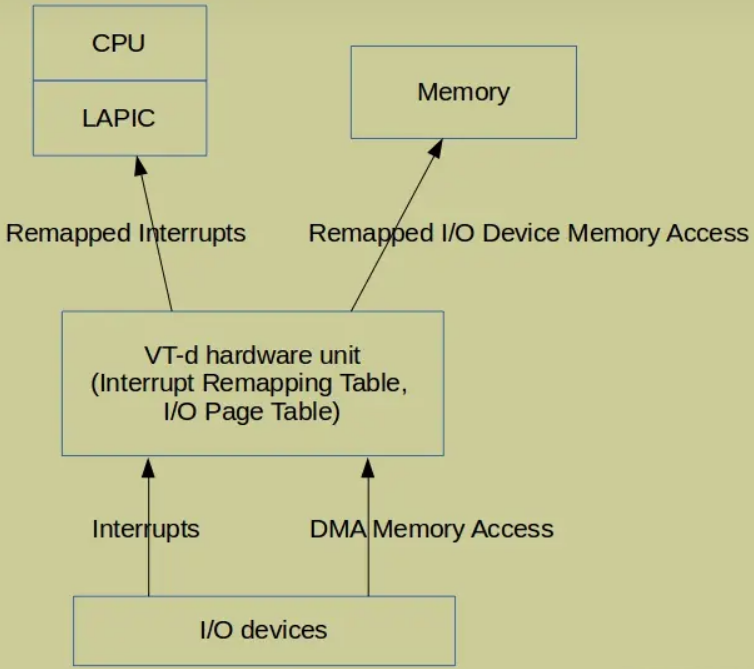


除非另有规定，否则ANC Data Flag占1个字：0x3FC

**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

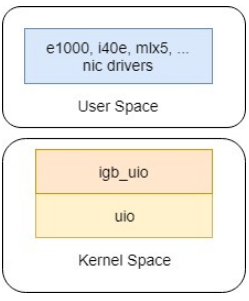
**dpdk：**

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

**工具函数：**

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

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

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

绑核设置：

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

**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

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

在impl->sch\_mgr->sch[]数组中查找空闲的sch（即sch->active为false），并设置sch->type=type、sch->active=false；

找到空闲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

新建会话流程：

1. 调用st20\_get\_bandwidth\_bps计算会话的流量配额
2. 调用st\_sch\_impl\* sch=st\_sch\_get()将流量配额分配给一个sch，确保该sch是active、started的，该sch可能是新的（sch\_request+st\_sch\_add\_quota+sch\_start），也可能是复用旧的
3. 初始化sch->st\_tx\_video\_sessions\_mgr，即创建一个新的tasklet，保存在sch->tx\_video\_mgr->tasklet和sch->tasklet[]数组中，该tasklet的执行函数为tasklet->ops->handler=tvs\_tasklet\_handler
4. 初始化sch->st\_video\_transmitter\_impl，即创建一个新的tasklet，保存在sch->video\_transmitter->tasklet和sch->tasklet[]数组中，该tasklet的执行函数为tasklet->ops->handler=video\_trs\_tasklet\_handler，也即video\_trs\_tsc\_tasklet

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取一帧yuv（app\_tx\_video\_next\_frame），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

2)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

3)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()计算会话的流量配额
2. 调用st\_sch\_impl\* sch=st\_sch\_get()将流量配额分配给一个sch，确保该sch是active、started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：**

回调获取当前ptp时间：ptp\_time

计算当前ptp时间对应的epoch：epochs = ptp\_time / frame\_time

如果当前epoch比上次进帧时的epoch小1或者相等，则强行修改当前epoch为上次epoch+1

计算首个rtp包的发送时间：ptp\_tr\_offset\_time = pacing\_tr\_offset\_time(pacing, epochs);

如果这个时间比当前ptp时间小，说明当前帧来晚了，stat\_epoch\_troffset\_mismatch++记录，同时直接给当前epoch加1，用新的epoch重新调pacing\_tr\_offset\_time计算发送时间

如果重新计算的发送时间依然比当前ptp时间小，则直接报错"error to\_epoch\_tr\_offset"

如果当前epoch减去上个epoch大于1，则说明某些epoch将无帧被发送，记录stat\_epoch\_drop += (epochs - last\_epochs -1)，打印统计日志形如"epoch drop 78"

**pacing\_tr\_offset\_time()源码分析：**

计算当前epoch，首个rtp包应该被发送的时间：

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

{

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

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，越热的时长为pacing->tr\_offset的80% ，发送时间继续提前

**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"：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++，

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

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

回调获取当前ptp时间：ptp\_time

当前ptp时间对应的epoch：ptp\_epochs = ptp\_time / frame\_time

pts对应的epoch：epochs = pts / frame\_time

pts对应的理应发送时间与当前实际时间的差值to\_epoch\_tr\_offset = tx\_audio\_pacing\_time(pacing, epochs) - ptp\_time

设置pts对应的发送时间对应的tsc：pacing->tsc\_time\_cursor = (double)st\_get\_tsc(impl) + to\_epoch\_tr\_offset

计算rtp包头时间戳：pacing->cur\_time\_stamp= epochs \* pacing->frame\_time\_sampling

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)

**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"

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

**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：**

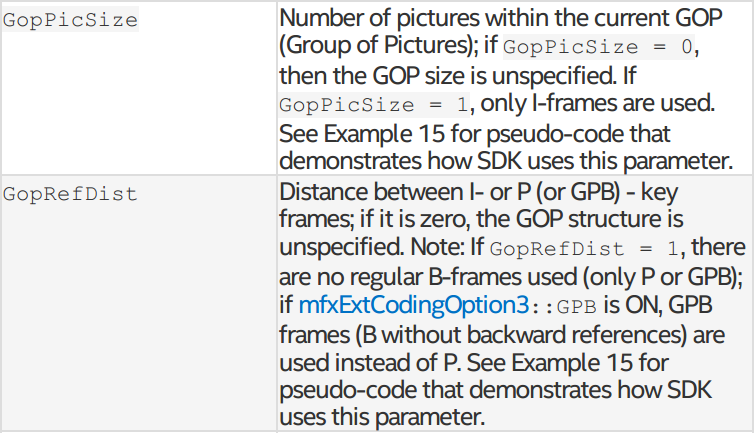
NVIDIA、AMD以及Intel都有自己的视频转码硬件加速技术

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.