# 1 V4L2 驱动

#### 内核文档：kernel/document/video-4linux/v4l2-framework.txt

## v4l2 框架分析

从usb摄像头驱动程序推测出驱动框架

uvc\_driver.c

struct uvc\_driver uvc\_driver = {

.driver = {

.name = "uvcvideo",

.probe = uvc\_probe,

.id\_table = uvc\_ids,

},

};

uvc\_init()函数

usb\_register(&uvc\_driver.driver);

uvc\_probe函数

v4l2\_device\_register(&intf->dev, &dev->vdev)

uvc\_register\_chains(dev)

uvc\_register\_terms

uvc\_register\_video(dev, stream);

vdev = video\_device\_alloc();//video dev分配

vdev->fops = &uvc\_fops;// video dev设置

video\_register\_device(vdev, VFL\_TYPE\_GRABBER, -1);// vedio dev注册设备

\_\_video\_register\_device()//v4l2-dev.c是核心层

vdev->cdev = cdev\_alloc();//分配字符设备

vdev->cdev->ops = &v4l2\_fops;//设置字符设备

cdev\_add(vdev->cdev, MKDEV(VIDEO\_MAJOR, vdev->minor), 1);

//注册字符设备

#### v4l2框架：

核心层： v4l2-dev.c

videobuf层：管理和分配缓冲区

硬件相关层：uvc\_driver.c

#### 视频缓冲区类型

缓冲类型

并不是所有的视频设备都使用相同的缓冲类型。实际上，有三种通用的类型：

–被分散在物理和内核虚拟地址空间的缓冲，几乎所有的用户空间缓冲都是这种类型，

如果可能的话分配内核空间的缓冲也很有意义，但是不幸的是，这个通常需要那些支持离散聚合DMA操作的硬件设备。

–物理上离散的但是虚拟地址是连续的，换句话说，就是用vmalloc分配的内核缓冲。这些缓冲很难用于DMA操作。

– 物理上连续的缓冲。

videobuf可以很好地处理这三种类型的缓冲，但是在此之前，驱动程序作者必须选择一种类型，并且以此类型为基础设计驱动。

---------------------

作者：盘股之

来源：CSDN

原文：https://blog.csdn.net/ramon1892/article/details/8444175

版权声明：本文为博主原创文章，转载请附上博文链接！

## 1.2虚拟视频驱动

### 1.2.1 vivi.c分析

1. 分配video\_device
2. 设置
3. 注册：video\_register\_device

struct vivi\_dev {

struct v4l2\_device v4l2\_dev;

struct v4l2\_ctrl\_handler ctrl\_handler;

struct v4l2\_ctrl \*brightness;

struct video\_device \*vfd;

};

struct video\_device//用来找到真正的设备操作函数和属性

{

const struct v4l2\_file\_operations \*fops;

struct v4l2\_device \*v4l2\_dev;

struct v4l2\_ctrl\_handler \*ctrl\_handler;

const struct v4l2\_ioctl\_ops \*ioctl\_ops;

};

vivi\_init()

vivi\_create\_instance(i)//第i个设备

struct vivi\_dev \*dev; //包含v4l2设备

struct video\_device \*vfd; //包含指向v4l2设备的指针

dev->v4l2\_dev.ctrl\_handler = hdl;

v4l2\_device\_register(NULL, &dev->v4l2\_dev);//不是重要的函数，只是一些必须的初始化

vfd = video\_device\_alloc();//分配video设备

\*vfd = vivi\_template;//设置video设备fops和\*ioctl\_ops

vfd->v4l2\_dev = &dev->v4l2\_dev;//

/\* video\_register\_device(struct video\_device \*vdev,int type, int nr)

type 代表不同的设备VFL\_TYPE\_GRABBER代表摄像头设备，根据不同的type创建不同类型的设备\*/

video\_register\_device(vfd, VFL\_TYPE\_GRABBER, video\_nr);//注册video设备

\_\_video\_register\_device(vdev, type, nr, 1, vdev->fops->owner);

if (vdev->ctrl\_handler == NULL)

//vfd=vdev; vdev->v4l2\_dev= vfd->v4l2\_dev=&dev->v4l2\_dev

vdev->ctrl\_handler = vdev->v4l2\_dev->ctrl\_handler;

vdev->cdev = cdev\_alloc(); //分配

vdev->cdev->ops = &v4l2\_fops;//设置

cdev\_add(vdev->cdev, MKDEV(VIDEO\_MAJOR, vdev->minor), 1);//注册

video\_device[vdev->minor] = vdev;//vdev=vfd

#### 猜测v4l2里面的fops会调用vivi的fops

static const struct file\_operations v4l2\_fops = { //v4l2 的fops

.read = v4l2\_read,

.write = v4l2\_write,

.open = v4l2\_open,

};

static struct video\_device vivi\_template = {

.name = "vivi",

.fops = &vivi\_fops,//用于图像数据的读写

.ioctl\_ops = &vivi\_ioctl\_ops,//用于参数设置

};

static const struct v4l2\_file\_operations vivi\_fops = {// vivi\_fops

.open = v4l2\_fh\_open,

.release = vivi\_close,

.read = vivi\_read,

};

由前面红色代码部分分析：

\*vfd = vivi\_template

vfd->v4l2\_dev=&dev->v4l2\_dev

#### 分析vivi.c的open，read，write，ioctl过程

1. open 函数

app: open(“/dev/video0”)

drv: v4l2\_open

v4l2\_open分析：

vdev = video\_devdata(filp);//根据次设备号从数组中得到video\_device，前面的vfd

video\_device[iminor(file->f\_path.dentry->d\_inode)];

/\*前面紫色代码：vdev=vfd;vfd->fops=& vivi\_fops

vfd->fops->open= &v4l2\_fh\_open \*/

if (vdev->fops->open) // vivi\_fops里面的v4l2\_fh\_open函数

ret = vdev->fops->open(filp);

1. read函数

app：read…

drv: v4l2\_read

struct video\_device \*vdev = video\_devdata(filp);

if (video\_is\_registered(vdev))

ret = vdev->fops->read(filp, buf, sz, off);

1. ioctl

app:ioctl

drv: .unlocked\_ioctl = v4l2\_ioctl

struct video\_device \*vdev = video\_devdata(filp);

if (video\_is\_registered(vdev))

ret = vdev->fops->unlocked\_ioctl(filp, cmd, arg);

.unlocked\_ioctl = video\_ioctl2,//vivi.c

video\_usercopy(file, cmd, arg, \_\_video\_do\_ioctl);//从用户空间复制参数然后调//用\_\_video\_do\_ioctl

\_\_video\_do\_ioctl

struct video\_device \*vfd = video\_devdata(file);

const struct v4l2\_ioctl\_ops \*ops = vfd->ioctl\_ops;

case VIDIOC\_QUERYCAP://根据不同的参数调用v4l2\_ioctl\_ops里面的函数

ret = ops->vidioc\_querycap(file, fh, cap);

总结调用过程：

先调用fops里面的ioctl函数 vdev->fops->unlocked\_ioctl(filp, cmd, arg);

然后转到ioctl\_ops 里面的函数vdev-->ioctl\_ops->vidioc\_querycap(file, fh, cap);

#### 在哪里初始化了CTRL属性：

vivi\_init()

vivi\_create\_instance

hdl = &dev->ctrl\_handler;

v4l2\_ctrl\_handler\_init(hdl, 11);

dev->volume=v4l2\_ctrl\_new\_std(hdl,//音量属性

&vivi\_ctrl\_ops,V4L2\_CID\_AUDIO\_VOLUME, 0, 255, 1, 200);

dev->brightness = v4l2\_ctrl\_new\_std(hdl, //亮度属性

&vivi\_ctrl\_ops,V4L2\_CID\_BRIGHTNESS, 0, 255, 1, 127);

dev->v4l2\_dev.ctrl\_handler = hdl;//dev就是vivi设备结构体

video\_register\_device(vfd, VFL\_TYPE\_GRABBER, video\_nr);

\_\_video\_register\_device(vdev, type, nr, 1, vdev->fops->owner);

if (vdev->ctrl\_handler == NULL)

vdev->ctrl\_handler = vdev->v4l2\_dev->ctrl\_handler

/\* struct vivi\_dev \*dev;

vfd=vdev

vfd-> ctrl\_handler = dev->v4l2\_dev->ctrl\_handler

\*/

v4l2属性使用过程：

\_\_video\_do\_ioctl

case VIDIOC\_QUERYCTRL:

if (vfd->ctrl\_handler)//在哪设置，在video\_register\_device

ret = v4l2\_queryctrl(vfd->ctrl\_handler, p);//根据ID在ctrl\_handler里面找到v4l2\_ctrl,

//返回它的值

## 1.2.2 虚拟摄像头驱动测试

新版本的内核中vivi已经变为vivid

1. sudo apt-get install xawvt//安装xawvt摄像头测试软件
2. uname -a //查看ubuntu内核版本号
3. 下载ubuntu同版本内核，拷贝出vivid文件夹
4. 修改vivid 内部的Makefile添加：

KERN\_DIR = /usr/src/linux-headers-3.19.0-25-generic/

all:

make -C $(KERN\_DIR) M=`pwd` modules

clean:

make -C $(KERN\_DIR) M=`pwd` modules clean

rm -rf modules.order

obj-$(CONFIG\_VIDEO\_VIVID) += vivid.o

1. make //编译处vivid.ko
2. sudo modprobe vivid //安装ubuntu自带vivid，是为了安装依赖的模块
3. sudo rmmod vivid
4. sudo insmod vivid.ko //安装我们自己编译的vivid
5. ls /dev/video\*
6. xawtv -C /dev/video1

### 1.2.3 vivi.c深入分析

strace -o xawtv.log xawtv //追踪应用xawtv涉及的系统调用xawtv是需要运行的命令

strace -o msg.log ./msgsend//追踪msgsend应用涉及的系统调用

wawtv涉及的系统调用：

1. open 1---7 v4l2\_open
2. ioctl(4,VIDIOC\_QUERYCAP //文件描述符fd=4
3. for()

ioctl(4,VIDIOC\_ENUMINPUT//列举输入源

1. for()

ioctl(4, VIDIOC\_ENUMSTD//列举标准制式

1. for()ioctl(4, VIDIOC\_ENUM\_FMT
2. ioctl(4,VIDIOC\_G\_PARM
3. for()

ioctl(4,VIDIOC\_QUERYCTRL //查询属性，亮度等

//8-10都是通过v4l2\_read\_attr来调用

ioctl(4,VIDIOC\_G\_STD //获取标准制式

1. ioctl(4,VIDIOC\_G\_INPUT
2. ioctl(4,VIDIOC\_G\_CTRL
3. ioctl(4,VIDIOC\_TRY\_FMT //尝试能否支持某种格式
4. ioctl(4,VIDIOC\_S\_FMT //设置摄像头使用某种格式
5. ioctl(4，VIDIOC\_REQBUFS //请求系统分配缓冲区
6. for()

ioctl(4，VIDIOC\_QUERYBUF//查询缓冲区

mmap()//映射分配的缓冲区到应用层

1. for()

ioctl(4，VIDIOC\_QBUF //把缓冲区放入队列

1. ioctl(4，VIDIOC\_STREAMON //启动摄像头
2. for()

ioctl(4,VIDIOC\_S\_CTRL //设置属性

ioctl(4,VIDIOC\_S\_INPUT //设置输入源

ioctl(4，VIDIOC\_S\_STD //设置制式

1. for()

{

select(5,[4],NULL,NULL,(5,0))//查询缓冲区是否由数据

ioctl(4,VIDIOC\_DQBUF //把缓冲区从队列取出

ioctl(4,VIDOC\_QBUF//把缓冲区放入队列

}

xawtv的几大函数：

v4l2\_open

v4l2\_read\_artr/v4l2\_write\_attr

v4l2\_start\_streaming

v4l2\_nextframe/v4l2\_waiton

vivi.c中的ioctl函数

static const struct v4l2\_ioctl\_ops vivi\_ioctl\_ops = {

//查询设备设什么类型设备，支持什么功能

.vidioc\_querycap = vidioc\_querycap,

//用于选择输入源，在xawtv的界面里面就是video source，非必须

.vidioc\_enum\_input = vidioc\_enum\_input,

.vidioc\_g\_input = vidioc\_g\_input,

.vidioc\_s\_input = vidioc\_s\_input,

//用于设置制式不是必须的

.vidioc\_s\_std = vidioc\_s\_std, //设置制式VIDIOC\_S\_STD

.tvnorms = V4L2\_STD\_525\_60, //用于VIDIOC\_ENUMSTD

.current\_norm = V4L2\_STD\_NTSC\_M, //获取标准制式VIDIOC\_G\_STD

//列举获得测试设置摄像头数据格式

.vidioc\_enum\_fmt\_vid\_cap = vidioc\_enum\_fmt\_vid\_cap,

.vidioc\_g\_fmt\_vid\_cap = vidioc\_g\_fmt\_vid\_cap,

.vidioc\_try\_fmt\_vid\_cap = vidioc\_try\_fmt\_vid\_cap,

.vidioc\_s\_fmt\_vid\_cap = vidioc\_s\_fmt\_vid\_cap,

//申请，查询，放入，取出队列

.vidioc\_reqbufs = vidioc\_reqbufs,

.vidioc\_querybuf = vidioc\_querybuf,

.vidioc\_qbuf = vidioc\_qbuf,

.vidioc\_dqbuf = vidioc\_dqbuf,

//启动停止摄像头

.vidioc\_streamon = vidioc\_streamon,

.vidioc\_streamoff = vidioc\_streamoff,

};

分析获取数据的过程：linux2.6内核，//核有所不同

1. 请求系统分配缓冲区

/\*队列在open函数里面用videobuf\_queue\_init初始化，这里只是分配缓冲区的头部

真正的缓存还没有分配\*/

ioctl(4，VIDIOC\_REQBUFS //应用层函数

vidioc\_reqbufs //驱动函数

videobuf\_reqbufs(struct videobuf\_queue \*q,struct v4l2\_requestbuffers \*req)

这里会调用buf\_setup()函数

1. 查询缓冲区

ioctl(4，VIDIOC\_QUERYBUF

vidioc\_querybuf//获取缓冲区大小,格式，每一行的长度高度

mmap()//此处才分配缓冲区，映射分配的缓冲区到应用层

vivi\_mmap

videobuf\_mmap\_mapper(&fh->vb\_vidq, vma);

1. 把缓冲区放入队列

ioctl(4，VIDIOC\_QBUF

vidioc\_qbuf

retval = q->ops->buf\_prepare(q,buf,field);//预处理函数

list\_add\_tail(&buf->stream,&q->stream);//把缓冲区放入内核队列尾部

q->ops->buf\_queue(q,buf);//调用驱动程序提供的入队列函数

1. 启动摄像头

ioctl(4，VIDIOC\_STREAMON

videobuf\_streamon

q->ops->buf\_queue(q,buf);

1. 查询缓冲区是否由数据

select(5,[4],NULL,NULL,(5,0))

vivi\_poll()

list\_entry(fh->vb\_vidq.stream.next,struct vivi\_buffer,vb.stream);//有数据获取

poll\_wait(file, &buf->vb.done, wait);//没有数据休眠

休眠了谁来唤醒，谁来构造数据

vivi\_thread\_tick

vivi\_fillbuff(dev,buf);//构造数据然后唤醒应用程序

wake\_up(&buf->vb.done);

谁来调用vivi\_thread\_tick

vivi\_start\_thread

dma\_q->kthread = kthread\_run(vivi\_thread, dma\_q, "vivi");//创建线程vivi，线程函数//vivi\_thread，传入数据dma\_q

vivi\_thread

vivi\_sleep

vivi\_thread\_tick

schedule\_timeout\_interruptible (timeout);//处理完数据休眠timeout

//驱动程序里必定有：产生数据、唤醒进程

1. 把有数据的缓冲区从队列取出

ioctl(4,VIDIOC\_DQBUF

vidioc\_dqbuf

buf = list\_entry(q->stream.next, struct videobuf\_buffer, stream);//获取有数据的缓冲区

retval = videobuf\_waiton(buf, nonblocking, 1);

list\_del(&buf->stream);//从队列里面删除

videobuf\_status(b,buf,q->type);//把这个缓冲区的状态返回给应用程序

1. 应用程序根据VIDIOC\_DQBUF所得到缓冲区状态知道是哪一个缓冲区有数据，就去读取对应的地址里面的数据

8 > 执行步骤3，读取完数据，应用再次把空缓冲区放入队列

ioctl(4,VIDOC\_QBUF

如果使用内核的buffer函数构造这个结构体

static struct videobuf\_queue\_ops vivi\_video\_qops = {

.buf\_setup = buffer\_setup,

.buf\_prepare = buffer\_prepare,

.buf\_queue = buffer\_queue,

.buf\_release = buffer\_release,

}

怎么写摄像头驱动程序：

1. 分配video\_device,video\_device\_alloc()函数分配
2. 设置

.fops

.ioctl\_ops

vivi\_video\_qops//如果使用用内核的buffer函数构造这个结构体

1. 注册

### 1.2.4 自己写虚拟摄像头驱动程序

#### 必须弄清楚的概念

videobuf\_queue只是一个存放缓冲区信息的结构体，里面存放了缓冲区的共性的属性，如一些相关操作函数，类型信息，还有缓冲区的头部信息。里面的struct videobuf\_buffer \*bufs[VIDEO\_MAX\_FRAME]就是存放缓冲区的头部信息的，是缓冲区本身的信息，里面有该缓冲区宽度，高度，内存地址等信息。

## 2 USB摄像头驱动

1. 构造一个usb\_driver ，里面有probe函数和id\_table
2. 注册驱动

有匹配设备调用probe函数

probe函数

1. 分配video\_device
2. 设置.fops和 .ioctl\_ops，如果需要内核提供的缓冲区函数，还要构造一个

videobuf\_queue\_ops

1. video\_register\_device

struct uvc\_driver uvc\_driver = {

.driver = {

.name = "uvcvideo",

.probe = uvc\_probe,

.id\_table = uvc\_ids,

},

};

static const struct file\_operations v4l2\_fops = {

.open = v4l2\_open,

.mmap = v4l2\_mmap,

.unlocked\_ioctl = v4l2\_ioctl,

.poll = v4l2\_poll,

}

const struct v4l2\_file\_operations uvc\_fops = {

.open = uvc\_v4l2\_open,

.unlocked\_ioctl = uvc\_v4l2\_ioctl,

.mmap = uvc\_v4l2\_mmap,

.poll = uvc\_v4l2\_poll,

};

uvc\_init()函数

usb\_register(&uvc\_driver.driver);

uvc\_probe函数

v4l2\_device\_register(&intf->dev, &dev->vdev)

uvc\_register\_chains(dev)

uvc\_register\_terms

uvc\_register\_video(dev, stream);

vdev = video\_device\_alloc();//video dev分配

vdev->fops = &uvc\_fops;// video dev设置

video\_register\_device(vdev, VFL\_TYPE\_GRABBER, -1);// vedio dev注册设备

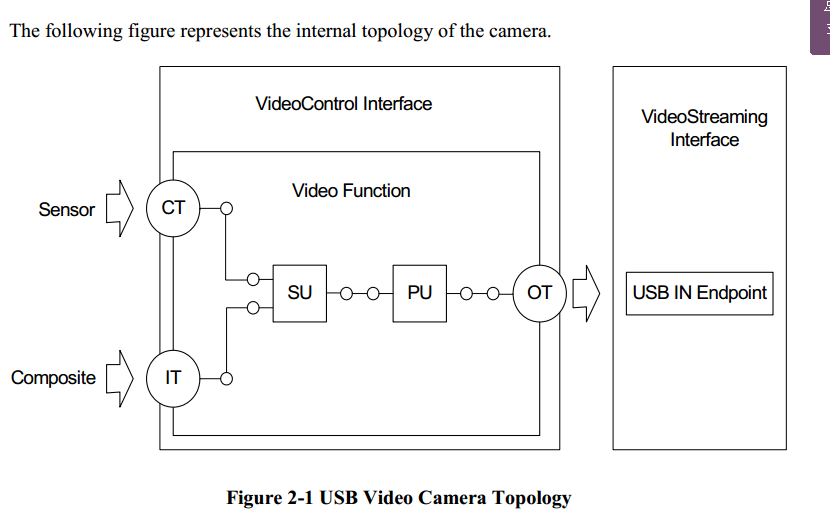
\_\_video\_register\_device()//v4l2-dev.c是核心层

vdev->cdev = cdev\_alloc();//分配字符设备

vdev->cdev->ops = &v4l2\_fops;//设置字符设备

cdev\_add(vdev->cdev, MKDEV(VIDEO\_MAJOR, vdev->minor), 1);

uvc\_fops里面并没有ioctl\_ops，因为在uvc里面没有把它们组织为一个结构体，是一个分散的函数。v4l2\_fops作为用户空间和内核空间交互的接口，最终会调用到uvc\_ops里面的函数。



uvc 协议抽象：

通过Videocontrol Interface来控制

通过VideoStreaming Interface 来传输数据

unit和terminal

CT:camera terminal //相机里面对焦等，普通摄像头没有

IT:input terminal

OT:out terminal

SU: select unit 用于选择哪个输入

PU: process unit 用于控制

#### ioctl 接口：

uvc\_v4l2\_ioctl

video\_usercopy(file, cmd, arg, uvc\_v4l2\_do\_ioctl);

uvc\_v4l2\_do\_ioctl//里面有各种控制函数

1、case VIDIOC\_QUERYCAP: // stream->type应该是设备被枚举时分析描述符时设置的

if (stream->type == V4L2\_BUF\_TYPE\_VIDEO\_CAPTURE)

cap->capabilities = V4L2\_CAP\_VIDEO\_CAPTURE| V4L2\_CAP\_STREAMING;

else cap->capabilities = V4L2\_CAP\_VIDEO\_OUTPUT| V4L2\_CAP\_STREAMING;

1. case VIDIOC\_ENUM\_FMT:

format = &stream->format[fmt->index];// 设备被枚举时分析描述符时设置的

3、case VIDIOC\_G\_FMT:

return uvc\_v4l2\_get\_format(stream, arg);

format = stream->cur\_format;//usb摄像头支持多种格式。每种格式下有多种frame

frame = stream->cur\_frame;// frame不同，分辨率等也不同

4、case VIDIOC\_TRY\_FMT:

return uvc\_v4l2\_try\_format(stream, arg, &probe, NULL, NULL)

/\* Check if the hardware supports the requested format. \*/

for (i = 0; i < stream->nformats; ++i) {

format = &stream->format[i];

if (format->fcc == fmt->fmt.pix.pixelformat)

break;

}

5、case VIDIOC\_S\_FMT://只是把参数保存起来，还没有发个usb摄像头

return uvc\_v4l2\_set\_format(stream, arg)

stream->cur\_format = format;

stream->cur\_frame = frame;

6、case VIDIOC\_REQBUFS:

ret = uvc\_alloc\_buffers(&stream->queue, arg);

vb2\_reqbufs(&queue->queue, rb);

\_\_vb2\_queue\_alloc(q, req->memory, num\_buffers, num\_planes);

7、case VIDIOC\_QUERYBUF:

return uvc\_query\_buffer(&stream->queue, buf);

vb2\_querybuf(&queue->queue, buf);

\_\_fill\_v4l2\_buffer(vb, b);// struct vb2\_queue \*q, struct v4l2\_buffer \*b

8、uvc\_v4l2\_mmap

uvc\_queue\_mmap(&stream->queue, vma);

9、case VIDIOC\_QBUF:

return uvc\_queue\_buffer(&stream->queue, arg);

vb2\_qbuf(&queue->queue, buf);

list\_add\_tail(&vb->queued\_entry, &q->queued\_list);

vb->state = VB2\_BUF\_STATE\_QUEUED;

10、

case VIDIOC\_STREAMON:

return uvc\_video\_enable(video, 1);

/\* Commit the streaming parameters. \*/

ret = uvc\_commit\_video(stream, &stream->ctrl);

uvc\_set\_video\_ctrl(stream, probe, 0);//发送设置给摄像头

ret = \_\_uvc\_query\_ctrl(stream->dev, //哪一个设备

UVC\_SET\_CUR, 0, stream->intfnum,//哪一个接口

probe ? UVC\_VS\_PROBE\_CONTROL : UVC\_VS\_COMMIT\_CONTROL, data,

size, uvc\_timeout\_param);

// Initialize isochronous/bulk URBs and allocate transfer buffers.

uvc\_init\_video(stream, GFP\_KERNEL);

uvc\_init\_video\_isoc(stream, best\_ep, gfp\_flags);

urb->complete = uvc\_video\_complete;//收到数据时调用

uvc\_init\_video\_bulk(stream, ep, gfp\_flags);

usb\_submit\_urb(stream->urb[i], gfp\_flags);

uvc\_video\_complete;

stream->decode(urb, stream, buf);// stream->decode = uvc\_video\_decode\_isoc;

usb\_submit\_urb(urb, GFP\_ATOMIC))

uvc\_video\_decode\_isoc

uvc\_queue\_next\_buffer(&stream->queue,buf);

vb2\_buffer\_done

wake\_up(&q->done\_wq);

11、uvc\_queue\_poll(&stream->queue, file, wait)

vb2\_poll(&queue->queue, file, wait);

poll\_wait(file, &q->done\_wq, wait);

12、case VIDIOC\_DQBUF:

uvc\_dequeue\_buffer(&stream->queue, arg,file->f\_flags & O\_NONBLOCK);

vb2\_dqbuf(&queue->queue, buf, nonblocking);

list\_del(&vb->queued\_entry);

13、case VIDIOC\_STREAMOFF:

return uvc\_video\_enable(stream, 0);

uvc\_uninit\_video(stream, 1);

usb\_kill\_urb(urb);

usb\_free\_urb(urb);

分析设置亮度的过程

case VIDIOC\_S\_CTRL:

ret = uvc\_ctrl\_set(chain, &xctrl);

ret = uvc\_ctrl\_commit(chain);

\_\_uvc\_ctrl\_commit(chain, 0);

uvc\_ctrl\_commit\_entity(chain->dev, entity, rollback);

ret = uvc\_query\_ctrl(dev, //dev在枚举时获得，是控制接口

UVC\_SET\_CUR, ctrl->entity->id,

dev->intfnum, ctrl->info.selector,

uvc\_ctrl\_data(ctrl, UVC\_CTRL\_DATA\_CURRENT),

ctrl->info.size);

总结：

1. UVC设备有2个interface: VideoControl Interface, VideoStreaming Interface

2. VideoControl Interface用于控制，比如设置亮度。它内部有多个Unit/Terminal(在程序里Unit/Terminal都称为entity)

可以通过类似的函数来访问：

ret = uvc\_query\_ctrl(dev /\* 哪一个USB设备 \*/,

SET\_CUR, ctrl->entity->id /\* 哪一个unit/terminal \*/,

dev->intfnum /\* 哪一个接口: VC interface \*/,

ctrl->info->selector,uvc\_ctrl\_data(ctrl, UVC\_CTRL\_DATA\_CURRENT),ctrl->info->size);

3. VideoStreaming Interface用于获得视频数据，也可以用来选择fromat/frame(VS可能有多种format, 一个format支持多种frame， frame用来表示分辨率等信息)

可以通过类似的函数来访问：

ret = \_\_uvc\_query\_ctrl(video->dev /\* 哪一个USB设备 \*/, SET\_CUR, 0,

video->streaming->intfnum /\* 哪一个接口: VS \*/,

probe ? VS\_PROBE\_CONTROL : VS\_COMMIT\_CONTROL, data, size,uvc\_timeout\_param);

4. 我们在设置FORMAT时只是简单的使用video->streaming->format[fmt->index]等数据，

这些数据哪来的？

应是设备被枚举时设置的，也就是分析它的描述符时设置的。

5. UVC驱动的重点在于：

描述符的分析

属性的控制: 通过VideoControl Interface来设置

格式的选择：通过VideoStreaming Interface来设置

数据的获得：通过VideoStreaming Interface的URB来获

#### 分析uvc驱动描述符

**Table A- 2 Video Interface Subclass Codes**

|  |  |
| --- | --- |
| **Video Subclass Code** | **Value** |
| SC\_UNDEFINED | 0x00 |
| SC\_VIDEOCONTROL | 0x01 |
| SC\_VIDEOSTREAMING | 0x02 |
| SC\_VIDEO\_INTERFACE\_COLLECTION | 0x03 |

#define USB\_INTERFACE\_INFO(cl, sc, pr) \

.match\_flags = USB\_DEVICE\_ID\_MATCH\_INT\_INFO, \

.bInterfaceClass = (cl), \

.bInterfaceSubClass = (sc), \

.bInterfaceProtocol = (pr)  
id\_table中设置如下：

{ USB\_INTERFACE\_INFO(USB\_CLASS\_VIDEO, 1, 0) },

匹配类为USB\_CLASS\_VIDEO，子类为SC\_VIDEOCONTROL的接口。因为流接口从属于控制接口，因此只要匹配了控制接口即可找到流接口。

lsusb显示usb设备信息

#### **Class-Specific VC Interface Descriptor**

uvc1.5 class specification47页

这类描述符在接口描述符的当前设置的extra字段里面

**A.5. Video Class-Specific VC Interface Descriptor Subtypes  
Table A- 5 Video Class-Specific VC Interface Descriptor Subtypes**

|  |  |
| --- | --- |
| **Descriptor Subtype** | **Value** |
| VC\_DESCRIPTOR\_UNDEFINED | 0x00 |
| VC\_HEADER | 0x01 |
| VC\_INPUT\_TERMINAL | 0x02 |
| VC\_OUTPUT\_TERMINAL | 0x03 |
| VC\_SELECTOR\_UNIT | 0x04 |
| VC\_PROCESSING\_UNIT | 0x05 |
| VC\_EXTENSION\_UNIT | 0x06 |
| VC\_ENCODING\_ |  |

VideoControl Interface的自定义描述符:

extra buffer of interface 0:

extra desc 0: 0d 24 01 00 01 4d 00 80 c3 c9 01 01 01

VC\_HEADER

extra desc 1: 12 24 02 01 01 02 00 00 00 00 00 00 00 00 03 0e 00 00

VC\_INPUT\_TERMINAL ID

extra desc 2: 09 24 03 02 01 01 00 04 00

VC\_OUTPUT\_TERMINAL ID wTerminalType bAssocTerminal bSourceID

extra desc 3: 0b 24 05 03 01 00 00 02 7f 14 00

VC\_PROCESSING\_UNIT ID bSourceID wMaxMultiplier bControlSize bmControls

extra desc 4: 1a 24 06 04 ad cc b1 c2 f6 ab b8 48 8e 37 32 d4 f3 a3 fe ec 08 01 03 01 3f 00

VC\_EXTENSION\_UNIT ID GUID bNumControls bNrInPins baSourceID

IT(01) ===> PU(03) ===> EU(04) ===> OT(02)

VideoStreaming Interface的自定义描述符:

extra buffer of interface 1:

extra desc 0: 0e 24 01 01 df 00 81 00 02 02 01 01 01 00

VS\_INPUT\_HEADER bNumFormats

extra desc 1: 1b 24 04 01 05 59 55 59 32 00 00 10 00 80 00 00 aa 00 38 9b 71 10 01 00 00 00 00

VS\_FORMAT\_UNCOMPRESSED bFormatIndex bNumFrameDescriptors GUID bBitsPerPixel

extra desc 2: 1e 24 05 01 00 80 02 e0 01 00 00 ca 08 00 00 ca 08 00 60 09 00 15 16 05 00 01 15 16 05 00

VS\_FRAME\_UNCOMPRESSED bFrameIndex bmCapabilities wWidth wHeight

640x480

extra desc 3: 1e 24 05 02 00 60 01 20 01 00 80 e6 02 00 80 e6 02 00 18 03 00 15 16 05 00 01 15 16 05 00

VS\_FRAME\_UNCOMPRESSED

extra desc 4: 1e 24 05 03 00 40 01 f0 00 00 80 32 02 00 80 32 02 00 58 02 00 15 16 05 00 01 15 16 05 00

extra desc 5: 1e 24 05 04 00 b0 00 90 00 00 a0 b9 00 00 a0 b9 00 00 c6 00 00 15 16 05 00 01 15 16 05 00

extra desc 6: 1e 24 05 05 00 a0 00 78 00 00 a0 8c 00 00 a0 8c 00 00 96 00 00 15 16 05 00 01 15 16 05 00

extra desc 7: 1a 24 03 00 05 80 02 e0 01 60 01 20 01 40 01 f0 00 b0 00 90 00 a0 00 78 00 00

VS\_STILL\_IMAGE\_FRAME

extra desc 8: 06 24 0d 01 01 04

VS\_INPUT\_HEADER 0x01

VS\_STILL\_IMAGE\_FRAME 0x03

VS\_FORMAT\_UNCOMPRESSED 0x04

VS\_FRAME\_UNCOMPRESSED 0x05

VS\_COLORFORMAT 0x0D