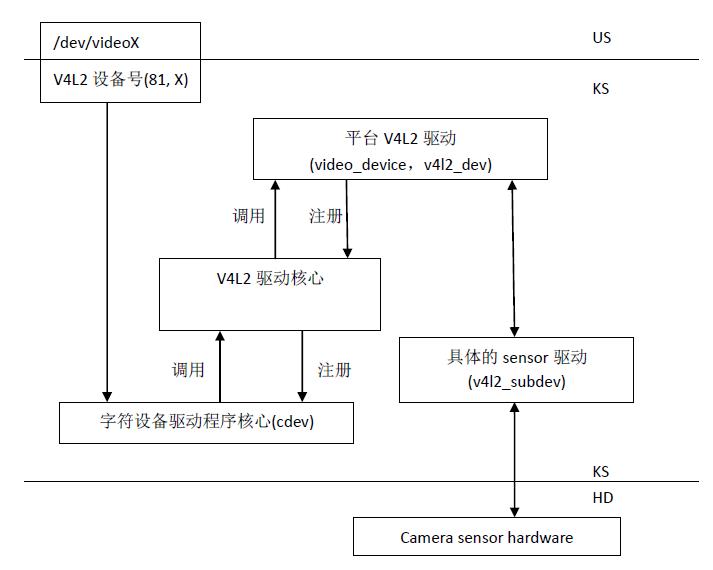
**1、概述**

**V4L2是video for linux version 2的简称**

Video4Linux2是Linux内核中关于视频设备的内核驱动框架，为上层的访问底层的视频设备提供了统一的接口。凡是内核中的子系统都有抽象底层硬件的差异，为上层提供统一的接口和提取出公共代码避免代码冗余等好处。就像公司的老板一般都不会直接找底层的员工谈话，而是找部门经理了解情况，一个是因为底层屌丝人数多，意见各有不同，措辞也不准，部门经理会把情况汇总后再向上汇报；二个是老板时间宝贵。

         V4L2支持三类设备：视频输入输出设备、VBI设备和radio设备(其实还支持更多类型的设备，暂不讨论)，分别会在/dev目录下产生videoX、radioX和vbiX设备节点。我们常见的视频输入设备主要是摄像头，也是本文主要分析对象。下图V4L2在Linux系统中的结构图：



Linux系统中视频输入设备主要包括以下四个部分：

字符设备驱动程序核心：V4L2本身就是一个字符设备，具有字符设备所有的特性，暴露接口给用户空间；

V4L2驱动核心：主要是构建一个内核中标准视频设备驱动的框架，为视频操作提供统一的接口函数；

平台V4L2设备驱动：在V4L2框架下，根据平台自身的特性实现与平台相关的V4L2驱动部分，包括注册video\_device和v4l2\_dev。

具体的sensor驱动：主要上电、提供工作时钟、视频图像裁剪、流IO开启等，实现各种设备控制方法供上层调用并注册v4l2\_subdev。

V4L2的核心源码位于drivers/media/v4l2-core，源码以实现的功能可以划分为四类：

核心模块实现：由v4l2-dev.c实现，主要作用申请字符主设备号、注册class和提供video device注册注销等相关函数；

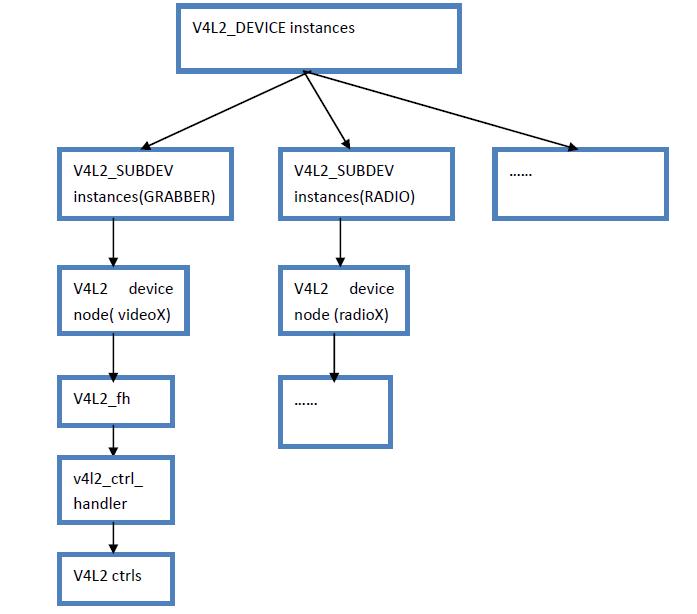
V4L2框架：由v4l2-device.c、v4l2-subdev.c、v4l2-fh.c、v4l2-ctrls.c等文件实现，构建V4L2框架；

Videobuf管理：由videobuf2-core.c、videobuf2-dma-contig.c、videobuf2-dma-sg.c、videobuf2-memops.c、videobuf2-vmalloc.c、v4l2-mem2mem.c等文件实现，完成videobuffer的分配、管理和注销。

Ioctl框架：由v4l2-ioctl.c文件实现，构建V4L2ioctl的框架。

**2、V4L2框架**

         结构体v4l2\_device、video\_device、v4l2\_subdev和v4l2\_fh是搭建框架的主要元素。下图是V4L2框架的结构图：



从上图V4L2框架是一个标准的树形结构，v4l2\_device充当了父设备，通过链表把所有注册到其下的子设备管理起来，这些设备可以是GRABBER、VBI或RADIO。V4l2\_subdev是子设备，v4l2\_subdev结构体包含了对设备操作的ops和ctrls，这部分代码和硬件相关，需要驱动工程师根据硬件实现，像摄像头设备需要实现控制上下电、读取ID、饱和度、对比度和视频数据流打开关闭的接口函数。Video\_device用于创建子设备节点，把操作设备的接口暴露给用户空间，v4l2\_fh是每个子设备的文件句柄，在打开设备节点文件时设置，方便上层索引到v4l2\_ctrl\_handler，v4l2\_ctrl\_handler管理设备的ctrls，这些ctrls(摄像头设备)包括调节饱和度、对比度和白平衡等。

**2.1**

**v4l2\_device**

v4l2\_device在v4l2框架中充当所有v4l2\_subdev的父设备，管理着注册在其下的子设备。以下是v4l2\_device结构体原型(去掉了无关的成员)：

struct v4l2\_device {

         structlist\_head subdevs;    //用链表管理注册的subdev

         charname[V4L2\_DEVICE\_NAME\_SIZE];    //device 名字

         structkref ref;      //引用计数

         ……

};

可以看出v4l2\_device的主要作用是管理注册在其下的子设备，方便系统查找引用到。

V4l2\_device的注册和注销：

int v412\_device\_register(struct device\*dev, struct v4l2\_device \*v4l2\_dev)

static void v4l2\_device\_release(struct kref \*ref)

**2.2**

**V4l2\_subdev**代表子设备，包含了子设备的相关属性和操作。先来看下结构体原型：

struct v4l2\_subdev {

         structv4l2\_device \*v4l2\_dev;  //指向父设备

         //提供一些控制v4l2设备的接口

         conststruct v4l2\_subdev\_ops \*ops;

         //向V4L2框架提供的接口函数

         conststruct v4l2\_subdev\_internal\_ops \*internal\_ops;

         //subdev控制接口

         structv4l2\_ctrl\_handler \*ctrl\_handler;

         /\* namemust be unique \*/

         charname[V4L2\_SUBDEV\_NAME\_SIZE];

         /\*subdev device node \*/

         structvideo\_device \*devnode;

};

每个子设备驱动都需要实现一个v4l2\_subdev结构体，v4l2\_subdev可以内嵌到其它结构体中，也可以独立使用。结构体中包含了对子设备操作的成员

v412\_subdev\_ops和v412\_subdev\_internal\_ops

v4l2\_subdev\_ops结构体原型如下：

struct v4l2\_subdev\_ops {

//视频设备通用的操作：初始化、加载FW、上电和RESET等

         conststruct v4l2\_subdev\_core\_ops        \*core;

//tuner特有的操作

         conststruct v4l2\_subdev\_tuner\_ops      \*tuner;

//audio特有的操作

         conststruct v4l2\_subdev\_audio\_ops      \*audio;

//视频设备的特有操作：设置帧率、裁剪图像、开关视频流等

       const strust v412\_subdev\_video\_ops\* video;

……

};

视频设备通常需要实现core和video成员，这两个OPS中的操作都是可选的，但是对于视频流设备video->s\_stream(开启或关闭流IO)必须要实现。

v4l2\_subdev\_internal\_ops结构体原型如下：

struct  v422\_subdev\_internal\_ops {

    //当subdev注册时被调用，读取IC的ID来进行识别

         int(\*registered)(struct v4l2\_subdev \*sd);

         void(\*unregistered)(struct v4l2\_subdev \*sd);

//当设备节点被打开时调用，通常会给设备上电和设置视频捕捉FMT

         int(\*open)(struct v4l2\_subdev \*sd, struct v4l2\_subdev\_fh \*fh);

         int(\*close)(struct v4l2\_subdev \*sd, struct v4l2\_subdev\_fh \*fh);

};

v4l2\_subdev\_internal\_ops是向V4L2框架提供的接口，只能被V4L2框架层调用。在注册或打开子设备时，进行一些辅助性操作。

Subdev的注册和注销

当我们把v4l2\_subdev需要实现的成员都已经实现，就可以调用以下函数把子设备注册到V4L2核心层：

int v4l2\_device\_register\_subdev(struct v4l2\_device\*v4l2\_dev, struct v4l2\_subdev \*sd)

当卸载子设备时，可以调用以下函数进行注销：

void v4l2\_device\_unregister\_subdev(struct v4l2\_subdev\*sd)

**2.3**

**video\_device**

         video\_device结构体用于在/dev目录下生成设备节点文件，把操作设备的接口暴露给用户空间

struct video\_device

{

         conststruct v4l2\_file\_operations \*fops;  //V4L2设备操作集合

         /\*sysfs \*/

         structdevice dev;             /\* v4l device \*/

         structcdev \*cdev;            //字符设备

         /\* Seteither parent or v4l2\_dev if your driver uses v4l2\_device \*/

         structdevice \*parent;              /\* deviceparent \*/

         structv4l2\_device \*v4l2\_dev;          /\*v4l2\_device parent \*/

         /\*Control handler associated with this device node. May be NULL. \*/

         structv4l2\_ctrl\_handler \*ctrl\_handler;

         /\* 指向video buffer队列\*/

         structvb2\_queue \*queue;

         intvfl\_type;      /\* device type \*/

         intminor;  //次设备号

         /\* V4L2file handles \*/

         spinlock\_t                  fh\_lock; /\* Lock for allv4l2\_fhs \*/

         structlist\_head        fh\_list; /\* List ofstruct v4l2\_fh \*/

         /\*ioctl回调函数集，提供file\_operations中的ioctl调用 \*/

         conststruct v4l2\_ioctl\_ops \*ioctl\_ops;

         ……

};

Video\_device分配和释放，用于分配和释放video\_device结构体：

struct video\_device \*video\_device\_alloc(void)

void video\_device\_release(struct video\_device \*vdev)

video\_device注册和注销，实现video\_device结构体的相关成员后，就可以调用下面的接口进行注册：

static inline int \_\_must\_check video\_register\_device(struct video\_device \*vdev,

                   inttype, int nr)

void video\_unregister\_device(struct video\_device\*vdev);

vdev：需要注册和注销的video\_device；

type：设备类型，包括VFL\_TYPE\_GRABBER、VFL\_TYPE\_VBI、VFL\_TYPE\_RADIO和VFL\_TYPE\_SUBDEV。

nr：设备节点名编号，如/dev/video[nr]。

**2.4**

**v4l2\_fh**

         v412\_fh是用来保存子设备的特有操作方法，也就是下面要分析到的v412\_ctrl\_handler，内核提供一组v4l2\_fh的操作方法，通常在打开设备节点时进行v4l2\_fh注册。

初始化v4l2\_fh，添加v4l2\_ctrl\_handler到v4l2\_fh：

void v4l2\_fh\_init(struct v4l2\_fh \*fh, structvideo\_device \*vdev)

添加v412\_fh到video\_device,方便核心层调用到

void v4l2\_fh\_add(struct v4l2\_fh \*fh)

**2.5**

**v4l2\_ctrl\_handler**

v4l2\_ctrl\_handler是用于保存子设备控制方法集的结构体，对于视频设备这些ctrls包括设置亮度、饱和度、对比度和清晰度等，用链表的方式来保存ctrls，可以通过v4l2\_ctrl\_new\_std函数向链表添加ctrls。

struct v4l2\_ctrl \*v4l2\_ctrl\_new\_std(structv4l2\_ctrl\_handler \*hdl,

                            conststruct v4l2\_ctrl\_ops \*ops,

                            u32id, s32 min, s32 max, u32 step, s32 def)

hdl是初始化好的v4l2\_ctrl\_handler结构体；

ops是v4l2\_ctrl\_ops结构体，包含ctrls的具体实现；

id是通过IOCTL的arg参数传过来的指令，定义在v4l2-controls.h文件；

min、max用来定义某操作对象的范围。如：

v4l2\_ctrl\_new\_std(hdl, ops, V4L2\_CID\_BRIGHTNESS,-208, 127, 1, 0);

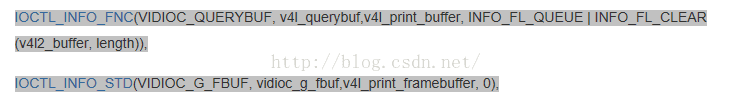
用户空间可以通过ioctl的VIDIOC\_S\_CTRL指令调用到v4l2\_ctrl\_handler，id透过arg参数传递。

3、ioctl框架

         你可能观察到用户空间对V4L2设备的操作基本都是ioctl来实现的，V4L2设备都有大量可操作的功能(配置寄存器)，所以V4L2的ioctl也是十分庞大的。它是一个怎样的框架，是怎么实现的呢？

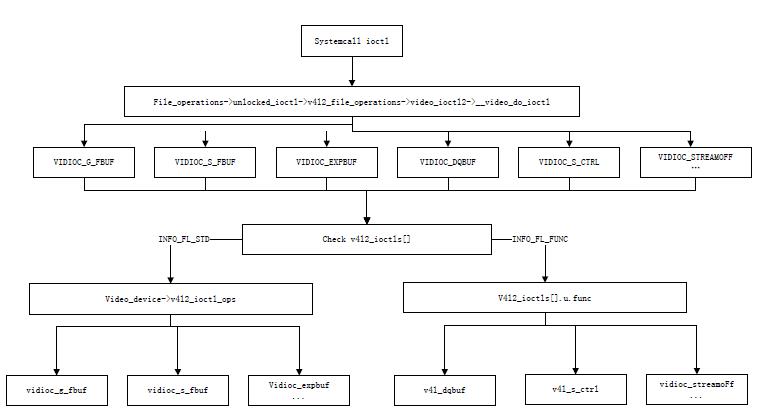
         Ioctl框架是由v4l2\_ioctl.c文件实现，文件中定义结构体数组v4l2\_ioctls，可以看做是ioctl指令和回调函数的关系表。用户空间调用系统调用ioctl，传递下来ioctl指令，然后通过查找此关系表找到对应回调函数。

以下是截取数组的两项：



         内核提供两个宏（IOCTL\_INFO\_FNC和IOCTL\_INFO\_STD)来初始化结构体，参数依次是ioctl指令、回调函数或者v4l2\_ioctl\_ops结构体成员、debug函数、flag。如果回调函数是v4l2\_ioctl\_ops结构体成员，则使用IOCTL\_INFO\_STD；如果回调函数是v4l2\_ioctl.c自己实现的，则使用IOCTL\_INFO\_FNC。

IOCTL调用的流程图如下：



  用户空间通过打开/dev/目录下的设备节点，获取到文件的file结构体，通过系统调用ioctl把cmd和arg传入到内核。通过一系列的调用后最终会调用到\_\_video\_do\_ioctl函数，然后通过cmd检索v4l2\_ioctls[]，判断是INFO\_FL\_STD还是INFO\_FL\_FUNC。如果是INFO\_FL\_STD会直接调用到视频设备驱动中video\_device->v4l2\_ioctl\_ops函数集。如果是INFO\_FL\_FUNC会先调用到v4l2自己实现的标准回调函数，然后根据arg再调用到video\_device->v4l2\_ioctl\_ops或v4l2\_fh->v4l2\_ctrl\_handler函数集。

**4、IO访问**

V4L2支持三种不同IO访问方式(内核中还支持了其它的访问方式，暂不讨论)：

**read和write**，是基本帧IO访问方式，通过read读取每一帧数据，数据需要在内核和用户之间拷贝，这种方式访问速度可能会非常慢；

**内存映射缓冲区(V4L2\_MEMORY\_MMAP)，**是在内核空间开辟缓冲区，应用通过mmap()系统调用映射到用户地址空间。这些缓冲区可以是大而连续DMA缓冲区、通过vmalloc()创建的虚拟缓冲区，或者直接在设备的IO内存中开辟的缓冲区(如果硬件支持)；

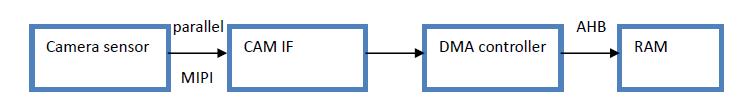
**用户空间缓冲区(V4L2\_MEMORY\_USERPTR)，**是用户空间的应用中开辟缓冲区，用户与内核空间之间交换缓冲区指针。很明显，在这种情况下是不需要mmap()调用的，但驱动为有效的支持用户空间缓冲区，其工作将也会更困难。

Read和write方式属于帧IO访问方式，每一帧都要通过IO操作，需要用户和内核之间数据拷贝，而后两种是流IO访问方式，不需要内存拷贝，访问速度比较快。内存映射缓冲区访问方式是比较常用的方式。

**内存映射缓存区方式**

         硬件层的数据流传输

         Camerasensor捕捉到图像数据通过并口或MIPI传输到CAMIF(camera interface)，CAMIF可以对图像数据进行调整(翻转、裁剪和格式转换等)。然后DMA控制器设置DMA通道请求AHB将图像数据传到分配好的DMA缓冲区。



         待图像数据传输到DMA缓冲区之后，mmap操作把缓冲区映射到用户空间，应用就可以直接访问缓冲区的数据。

**4.1**

**vb2\_queue**

为了使设备支持流IO这种方式，驱动需要实现struct vb2\_queue，来看下这个结构体：

struct vb2\_queue {

         enumv4l2\_buf\_type                  type;  //buffer类型

         unsignedint                        io\_modes;  //访问IO的方式:mmap、userptr etc

         conststruct vb2\_ops                 \*ops;   //buffer队列操作函数集合

         conststruct vb2\_mem\_ops     \*mem\_ops;  //buffer memory操作集合

         structvb2\_buffer              \*bufs[VIDEO\_MAX\_FRAME];  //代表每个buffer

         unsignedint                        num\_buffers;    //分配的buffer个数

……

};

Vb2\_queue代表一个videobuffer队列，vb2\_buffer是这个队列中的成员，vb2\_men\_ops是缓冲内存的操作函数集，vb2\_ops用来管理队列。

**4.1.1**

**vb2\_mem\_ops**

         vb2\_mem\_ops包含了内存映射缓冲区、用户空间缓冲区的内存操作方法：

struct vb2\_mem\_ops {

         void           \*(\*alloc)(void \*alloc\_ctx, unsignedlong size);  //分配视频缓存

         void           (\*put)(void \*buf\_priv);            //释放视频缓存

//获取用户空间视频缓冲区指针

         void           \*(\*get\_userptr)(void \*alloc\_ctx,unsigned long vaddr,

                                               unsignedlong size, int write);

         void           (\*put\_userptr)(void \*buf\_priv);       //释放用户空间视频缓冲区指针

//用于缓存同步

         void           (\*prepare)(void \*buf\_priv);

         void           (\*finish)(void \*buf\_priv);

         void           \*(\*vaddr)(void \*buf\_priv);

         void           \*(\*cookie)(void \*buf\_priv);

         unsignedint     (\*num\_users)(void \*buf\_priv);         //返回当期在用户空间的buffer数

         int              (\*mmap)(void \*buf\_priv, structvm\_area\_struct \*vma);  //把缓冲区映射到用户空间

};

         这是一个相当庞大的结构体，这么多的结构体需要实现还不得累死，幸运的是内核都已经帮我们实现了。提供了三种类型的视频缓存区操作方法：连续的DMA缓冲区、集散的DMA缓冲区以及vmalloc创建的缓冲区，分别由videobuf2-dma-contig.c、videobuf2-dma-sg.c和videobuf-vmalloc.c文件实现，可以根据实际情况来使用。

**4.1.2**

 vb2\_ops

         vb2\_ops是用来管理buffer队列的函数集合，包括队列和缓冲区初始化

struct vb2\_ops {

         //队列初始化

         int(\*queue\_setup)(struct vb2\_queue \*q, const struct v4l2\_format \*fmt,

                               unsigned int \*num\_buffers, unsigned int\*num\_planes,

                               unsigned int sizes[], void \*alloc\_ctxs[]);

         //释放和获取设备操作锁

         void(\*wait\_prepare)(struct vb2\_queue \*q);

         void(\*wait\_finish)(struct vb2\_queue \*q);

         //对buffer的操作

         int(\*buf\_init)(struct vb2\_buffer \*vb);

         int(\*buf\_prepare)(struct vb2\_buffer \*vb);

         int(\*buf\_finish)(struct vb2\_buffer \*vb);

         void(\*buf\_cleanup)(struct vb2\_buffer \*vb);

//开始视频流

         int(\*start\_streaming)(struct vb2\_queue \*q, unsigned int count);

//停止视频流

         int(\*stop\_streaming)(struct vb2\_queue \*q);

//把VB传递给驱动

         void(\*buf\_queue)(struct vb2\_buffer \*vb);

};

**4.1.3**

**vb2\_buffer**

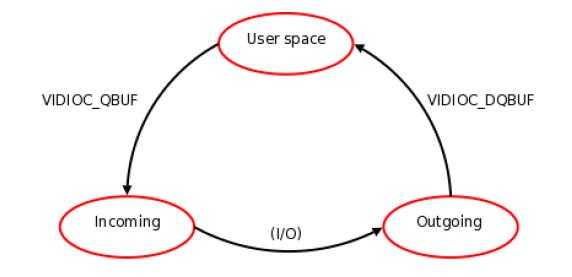
vb2\_buffer是缓存队列的基本单位，内嵌在其中v4l2\_buffer是核心成员。当开始流IO时，帧以v4l2\_buffer的格式在应用和驱动之间传输。一个缓冲区可以有三种状态：

**在驱动的传入队列中，**驱动程序将会对此队列中的缓冲区进行处理，用户空间通过IOCTL:VIDIOC\_QBUF把缓冲区放入到队列。对于一个视频捕获设备，传入队列中的缓冲区是空的，驱动会往其中填充数据；

**在驱动的传出队列中，**这些缓冲区已由驱动处理过，对于一个视频捕获设备，缓存区已经填充了视频数据，正等用户空间来认领；

**用户空间状态的队列，**已经通过IOCTL:VIDIOC\_DQBUF传出到用户空间的缓冲区，此时缓冲区由用户空间拥有，驱动无法访问。

这三种状态的切换如下图所示：



v4l2\_buffer结构如下：

struct v4l2\_buffer {

         \_\_u32                          index;  //buffer 序号

         \_\_u32                          type;   //buffer类型

         \_\_u32                          bytesused;  缓冲区已使用byte数

         \_\_u32                          flags;

         \_\_u32                          field;

         structtimeval           timestamp;  //时间戳，代表帧捕获的时间

         structv4l2\_timecode       timecode;

         \_\_u32                          sequence;

         /\*memory location \*/

         \_\_u32                          memory;  //表示缓冲区是内存映射缓冲区还是用户空间缓冲区

         union {

                   \_\_u32           offset;  //内核缓冲区的位置

                   unsignedlong   userptr;   //缓冲区的用户空间地址

                   structv4l2\_plane \*planes;

                   \_\_s32                 fd;

         } m;

         \_\_u32                          length;   //缓冲区大小，单位byte

};

当用户空间拿到v4l2\_buffer，可以获取到缓冲区的相关信息。Byteused是图像数据所占的字节数，如果是V4L2\_MEMORY\_MMAP方式，m.offset是内核空间图像数据存放的开始地址，会传递给mmap函数作为一个偏移，通过mmap映射返回一个缓冲区指针p，p+byteused是图像数据在进程的虚拟地址空间所占区域；如果是用户指针缓冲区的方式，可以获取的图像数据开始地址的指针m.userptr，userptr是一个用户空间的指针，userptr+byteused便是所占的虚拟地址空间，应用可以直接访问。

**5、用户空间访问设备**

下面通过内核映射缓冲区方式访问视频设备(capturedevice)的流程。

**1>    打开设备文件**

fd = open(dev\_name, O\_RDWR /\* required \*/ | O\_NONBLOCK, 0);

dev\_name[/dev/videoX]

**2>    查询设备支持的能力**

Struct v4l2\_capability  cap;

ioctl(fd, VIDIOC\_QUERYCAP, &cap)

**3>    设置视频捕获格式**

fmt.type= V4L2\_BUF\_TYPE\_VIDEO\_CAPTURE;

fmt.fmt.pix.width       = 640;

fmt.fmt.pix.height      = 480;

fmt.fmt.pix.pixelformat= V4L2\_PIX\_FMT\_YUYV;  //像素格式

fmt.fmt.pix.field       = V4L2\_FIELD\_INTERLACED;

ioctl(fd,VIDIOC\_S\_FMT, & fmt)

**4>    向驱动申请缓冲区**

Struct  v4l2\_requestbuffers req;

req.count= 4;  //缓冲个数

req.type= V4L2\_BUF\_TYPE\_VIDEO\_CAPTURE;

req.memory= V4L2\_MEMORY\_MMAP;

if(-1 == xioctl(fd, VIDIOC\_REQBUFS, &req))

**5>    获取每个缓冲区的信息，映射到用户空间**

structbuffer {

        void  \*start;

        size\_t length;

} \*buffers;

buffers = calloc(req.count, sizeof(\*buffers));

for (n\_buffers= 0; n\_buffers < req.count; ++n\_buffers) {

struct  v4l2\_buffer buf;

buf.type        = V4L2\_BUF\_TYPE\_VIDEO\_CAPTURE;

buf.memory      = V4L2\_MEMORY\_MMAP;

buf.index       = n\_buffers;

if (-1 ==xioctl(fd, VIDIOC\_QUERYBUF, & buf))

                       errno\_exit("VIDIOC\_QUERYBUF");

buffers[n\_buffers].length= buf.length;

buffers[n\_buffers].start=

        mmap(NULL /\* start anywhere \*/,

        buf.length,

        PROT\_READ | PROT\_WRITE /\* required \*/,

        MAP\_SHARED /\* recommended \*/,

        fd, buf.m.offset);

 }

**6>    把缓冲区放入到传入队列上，打开流IO，开始视频采集**

for (i =0; i < n\_buffers; ++i) {

    struct v4l2\_buffer buf;

    buf.type = V4L2\_BUF\_TYPE\_VIDEO\_CAPTURE;

    buf.memory = V4L2\_MEMORY\_MMAP;

    buf.index = i;

    if (-1 == xioctl(fd, VIDIOC\_QBUF, &buf))

          errno\_exit("VIDIOC\_QBUF");

 }

 type = V4L2\_BUF\_TYPE\_VIDEO\_CAPTURE;

 if (-1 == xioctl(fd, VIDIOC\_STREAMON, & type))

**7>  调用select监测文件描述符，缓冲区的数据是否填充好，然后对视频数据**

        for (;;) {

                        fd\_set fds;

                        struct timeval tv;

                        int r;

                        FD\_ZERO(&amp;fds);

                        FD\_SET(fd,&amp;fds);

                        /\* Timeout. \*/

                        tv.tv\_sec = 2;

                        tv.tv\_usec = 0;

                                                        //监测文件描述是否变化

                        r = select(fd + 1,& fds, NULL, NULL, & tv);

                        if (-1 == r) {

                                if (EINTR ==errno)

                                       continue;

                               errno\_exit("select");

                        }

                        if (0 == r) {

                                fprintf(stderr,"select timeout\n");

                               exit(EXIT\_FAILURE);

                        }

                                                        //对视频数据进行处理

                        if (read\_frame())

                                break;

                        /\* EAGAIN - continueselect loop. \*/

               }

**8>    取出已经填充好的缓冲，获取到视频数据的大小，然后对数据进行处理。这里取出的缓冲只包含缓冲区的信息，并没有进行视频数据拷贝。**

buf.type= V4L2\_BUF\_TYPE\_VIDEO\_CAPTURE;

buf.memory= V4L2\_MEMORY\_MMAP;

if (-1 ==ioctl(fd, VIDIOC\_DQBUF, & buf))    //取出缓冲

           errno\_exit("VIDIOC\_QBUF");

process\_image(buffers[buf.index].start,buf.bytesused);   //视频数据处理

if (-1 ==xioctl(fd, VIDIOC\_QBUF, & buf))  //然后又放入到传入队列

     errno\_exit("VIDIOC\_QBUF");

**9>    停止视频采集**

type =V4L2\_BUF\_TYPE\_VIDEO\_CAPTURE;

ioctl(fd,VIDIOC\_STREAMOff, & type);

**10> 关闭设备**

Close(fd);