编写基于V4L2视频驱动主要涉及到以下几个知识点：

●    摄像头方面的知识  
                要了解选用的摄像头的特性，包括访问控制方法、各种参数的配置方法、信号输出类型等。

●    Camera解码器、控制器  
                如果摄像头是模拟量输出的，要熟悉解码器的配置。最后数字视频信号进入camera控制器后，还要熟悉camera控制器的操作。

●    V4L2的API和数据结构  
                编写驱动前要熟悉应用程序访问V4L2的方法及设计到的数据结构。

●    V4L2的驱动架构  
                最后编写出符合V4L2规范的视频驱动。

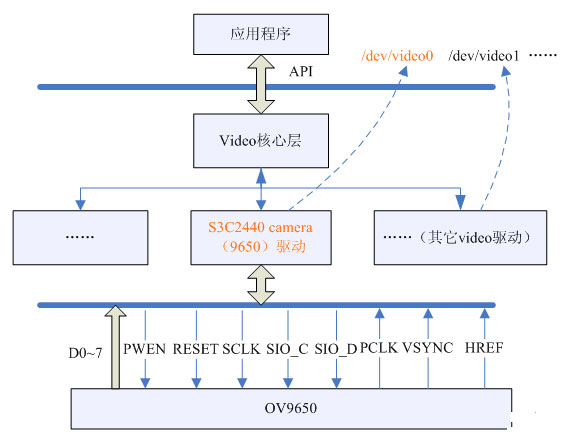
本文介绍基于S3C2440硬件平台的V4L2视频驱动开发。摄像头采用OmniVision公司的OV9650和OV9655。主要包含以下几个方面的内容：

视频驱动的整体驱动框架

●    3C2440 camera控制器+ov9650（ov9655）  
        ●    V4L2 API及数据结构  
        ●    V4L2驱动框架  
        ●    ov9650（ov9655）+s3c2440+V4L2实例

一、 视频驱动的整体框架

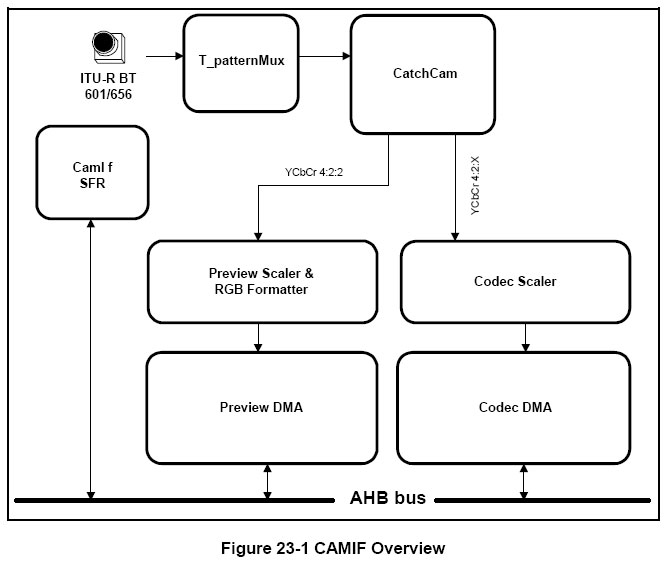
视频驱动的整体框架见下图：



二、S3C2440 camera控制器+ov9650（ov9655）

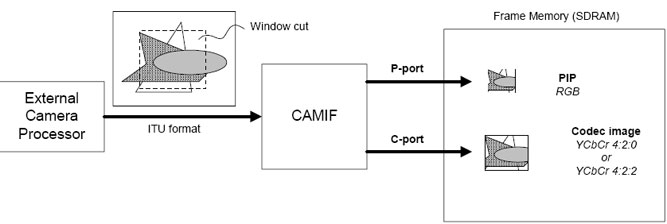
（1）S3C2440 camera控制器介绍

S3C2440支持ITU-R BT601/656格式的数字图像输入，支持的2个通道的DMA，Preview通道和Codec通道，参见下图。

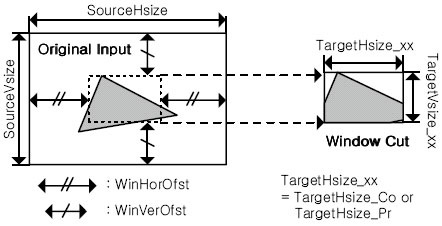


Preview通道可以将YCbCr4:2:2格式的图像转换为RGB（16bit或24bit）格式的数据，并存放于 为Preview DMA分配的内存中，最大分辨率为640\*480。主要用于本地液晶屏显示。如果将Preview DMA的内存和Framebuffer内存重叠的话，就可以实现采集直接输出到液晶屏上了。

Codec通道可以输出YCbCr4:2:0或YCbCr4:2:2格式到为Codec DMA分配的内存中。最大分辨率为4096\*4096。主要用于图像的编解码处理。

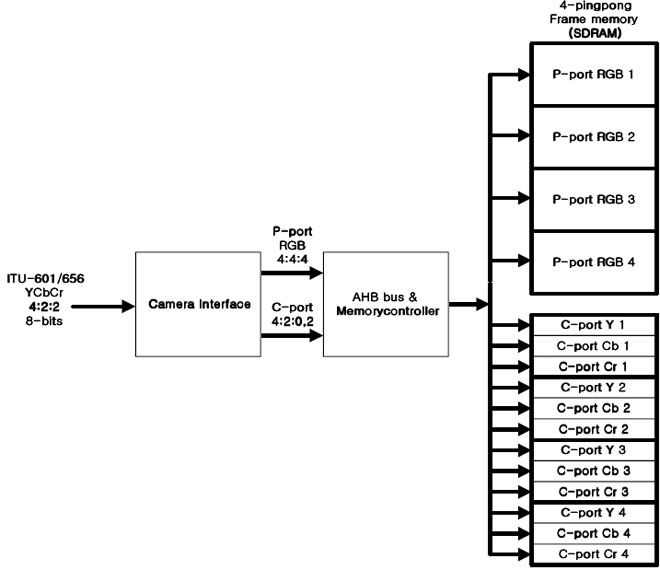


上图中的window cut功能是指在图像可以先做一个裁剪。通过设置CIWDOFST完成此功能，见下图。图像进入P、C通道后，各自的scaler单元还可以对其进行缩放、旋转等处理。

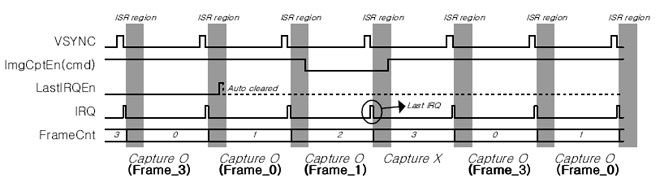


S3C2440 camera控制器支持乒乓存储。为了防止采集和输出之间的冲突，采用了乒乓存储方式。每次采集一帧后，自动转到下一个存储区。如果你因为内存空间不足，不想使用此功能的话，可以将四个区域设置到同一块空间。

在做图像处理时，需要关注到最后存储区中的图像格式，如codec通道硬件自动把Y、Cb、Cr分离存储。



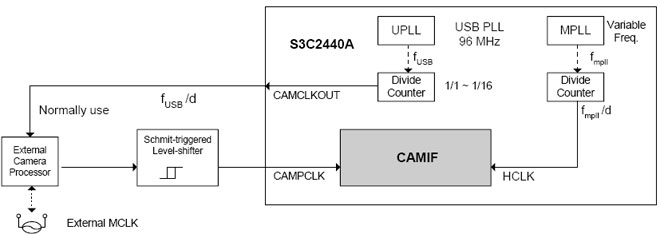
S3C2440 camera 控制器Last IRQ功能的使用，也是需要掌握的。如果处理不好，输出的图像效果会受影响。



控制器会在每个VSYNC下降沿判断ImgCptEn信号等命令。如果在下降沿发现ImgCptEn信号有效，则产生 IRQ中断。然后才开始一帧图像的真正采集。而如果在VSYNC下降沿判断到ImgCptEn为低电平且之前LastIRQEn没有使能，则不会产生任何 中断，且不会再进行下一帧的采集。如果你想在ImgCptEn关闭后，一帧采集完后产生一个中断通知你，那么就需要在最后一次中断产生前（stop capturing后的vysnc下将沿）使能lastirq就可以了。

我在移植linux驱动时就遇到了一个Last IRQ的问题。现象是输出图像上面总是有一条比其它部分反应慢。采集运动图像，就能看出现象。查看代码是因为没有设立lastirq，因为每次如果不在 lastirq产生的情况下读取，图像缓冲中的数据是不稳定的，可能照成图像不完整。修改代码支持lastirq后，问题解决。

Camera控制器时钟设置也是需要注意的，ov9650需要Camera控制器为其提供时钟。

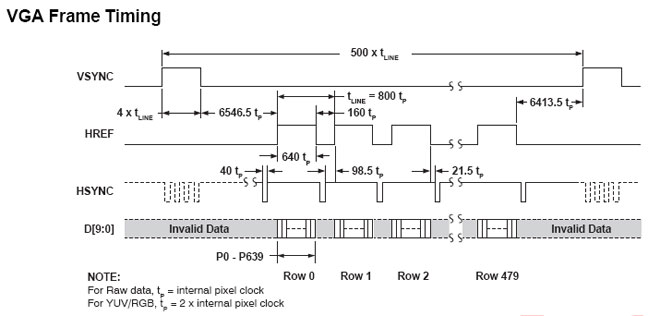


提供给外部摄像头的时钟是由UPLL输出时钟分频得到的。而CAMIF的时钟是由HCLK提供的。本例中，提供给ov9650的时钟为24M。

（2）ov9650（ov9655）设置方法

OV9650是OmniVision公司的COMS摄像头，130万像素，支持SXVGA、VGA、QVGA、CIF等图像输出格式。 最大速率在SXVGA时为15fps，在VGA时为30fps。

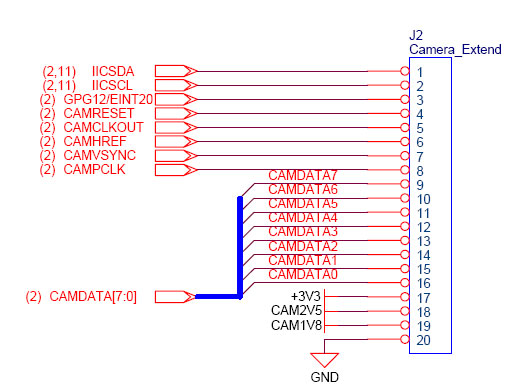
OV9650摄像头时序如下图：



上图中D[9:2]用于8-bitYUV或者RGB565/RGB555(D[9]MSB、D[2]LSB)。D[9:0]用于10-bit RGB。本例中使用8-bit YUV模式。

我手边开发板的Camera和S3C2440的接线原理图如下（对应camera中具体的信号名称参见前文的驱动整体架构图）。

注：GPG12用于PWEN信号

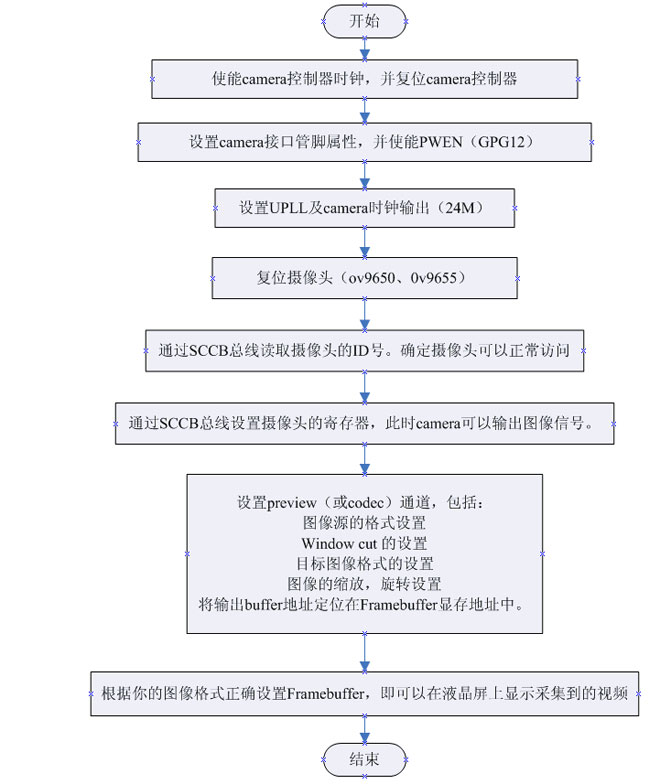


OV9650摄像头设置方法是通过SCCB总线设置

SCCB可以看作是一种简化的I2C总线，可以使用IO模拟SCCB时序。

（3）编写ARM测试代码测试camera功能

在Keil环境下编写一个测试代码完成从摄像头采集图像输出到液晶屏。下面列出程序的流程。



（4）编写测试代码过程中常见的问题

●    摄像头寄存器的配置

因为摄像头有很多寄存器，可能一下无法理解里面所有的配置含义，所以开始时希望得到一份可用的配置。但往往从别人的测试 代码中拿到配置后，仍然无法使用。我这里列出几个可能的原因：（1）摄像头中的图像输出格式和你在camera控制器中设置的不一致，同一个摄像头可以设 置多种输入格式，如：YCbYCr或CbYCrY。（2）图像输出的一些时序和你的camera控制器设置不一致，摄像头可以设置一些时序，如：图像数据 在CAMPCLK的上升沿有效还是下降沿有效。（3）注意输出图像的格式和Framebuffer控制器的匹配，如字节顺序等问题。

●    Ov9650和ov9655的使用区别

这里主要列出两者之间在复位信号上有差别，ov9650是高电平复位，而ov9655是低电平复位。

三、 V4L2 API及数据结构（camera的测试程序）

V4L2是V4L的升级版本，为linux下视频设备程序提供了一套接口规范。包括一套数据结构和底层V4L2驱动接口。

1、常用的结构体在内核目录include/linux/videodev2.h中定义

struct v4l2\_requestbuffers        //申请帧缓冲，对应命令VIDIOC\_REQBUFS   
        struct v4l2\_capability        //视频设备的功能，对应命令VIDIOC\_QUERYCAP   
        struct v4l2\_input        //视频输入信息，对应命令VIDIOC\_ENUMINPUT  
        struct v4l2\_standard        //视频的制式，比如PAL，NTSC，对应命令VIDIOC\_ENUMSTD   
        struct v4l2\_format        //帧的格式，对应命令VIDIOC\_G\_FMT、VIDIOC\_S\_FMT等  
        struct v4l2\_buffer        //驱动中的一帧图像缓存，对应命令VIDIOC\_QUERYBUF   
        struct v4l2\_crop        //视频信号矩形边框  
        v4l2\_std\_id        //视频制式

2、常用的IOCTL接口命令也在include/linux/videodev2.h中定义

VIDIOC\_REQBUFS //分配内存   
        VIDIOC\_QUERYBUF         //把VIDIOC\_REQBUFS中分配的数据缓存转换成物理地址   
        VIDIOC\_QUERYCAP        //查询驱动功能   
        VIDIOC\_ENUM\_FMT        //获取当前驱动支持的视频格式   
        VIDIOC\_S\_FMT        //设置当前驱动的频捕获格式   
        VIDIOC\_G\_FMT        //读取当前驱动的频捕获格式   
        VIDIOC\_TRY\_FMT        //验证当前驱动的显示格式   
        VIDIOC\_CROPCAP        //查询驱动的修剪能力   
        VIDIOC\_S\_CROP        //设置视频信号的矩形边框   
        VIDIOC\_G\_CROP        //读取视频信号的矩形边框  
        VIDIOC\_QBUF        //把数据从缓存中读取出来   
        VIDIOC\_DQBUF        //把数据放回缓存队列   
        VIDIOC\_STREAMON        //开始视频显示函数   
        VIDIOC\_STREAMOFF        //结束视频显示函数   
        VIDIOC\_QUERYSTD         //检查当前视频设备支持的标准，例如PAL或NTSC。

3、操作流程

V4L2提供了很多访问接口，你可以根据具体需要选择操作方法。需要注意的是，很少有驱动完全实现了所有的接口功能。所以在使用时需要参考驱动源码，或仔细阅读驱动提供者的使用说明。

下面列举出一种操作的流程，供参考。

（1）打开设备文件  
                  int fd = open(Devicename,mode);  
                  Devicename：/dev/video0、/dev/video1 ……  
                  Mode：O\_RDWR [| O\_NONBLOCK]

如果使用非阻塞模式调用视频设备，则当没有可用的视频数据时，不会阻塞，而立刻返回。

（2）取得设备的capability

          struct v4l2\_capability capability；  
                  int ret = ioctl(fd, VIDIOC\_QUERYCAP, &capability);

看看设备具有什么功能，比如是否具有视频输入特性。

（3）选择视频输入

          struct v4l2\_input input；  
                  ……初始化input  
                  int ret = ioctl(fd, VIDIOC\_QUERYCAP, &input);

一个视频设备可以有多个视频输入。如果只有一路输入，这个功能可以没有。

（4）检测视频支持的制式

          v4l2\_std\_id std;  
                  do {  
                                ret = ioctl(fd, VIDIOC\_QUERYSTD, &std);  
                   } while (ret == -1 && errno == EAGAIN);  
                switch (std) {  
                case V4L2\_STD\_NTSC:   
                                //……  
                case V4L2\_STD\_PAL:  
                                //……  
                }

（5）设置视频捕获格式

          struct v4l2\_format fmt;  
                  fmt.type = V4L2\_BUF\_TYPE\_VIDEO\_OUTPUT;  
                  fmt.fmt.pix.pixelformat = V4L2\_PIX\_FMT\_UYVY;  
                  fmt.fmt.pix.height = height;  
                  fmt.fmt.pix.width = width;  
                  fmt.fmt.pix.field = V4L2\_FIELD\_INTERLACED;  
                  ret = ioctl(fd, VIDIOC\_S\_FMT, &fmt);  
                  if(ret) {  
                          perror("VIDIOC\_S\_FMT\n");  
                          close(fd);  
                          return -1;  
                  }

（6）向驱动申请帧缓存

          struct v4l2\_requestbuffers req;  
                   if (ioctl(fd, VIDIOC\_REQBUFS, &req) == -1) {  
                          return -1;  
                   }

v4l2\_requestbuffers结构中定义了缓存的数量，驱动会据此申请对应数量的视频缓存。多个缓存可以用于建立FIFO，来提高视频采集的效率。

（7）获取每个缓存的信息，并mmap到用户空间

          typedef struct VideoBuffer {  
                          void \*start;  
                          size\_t length;  
                  } VideoBuffer;

          VideoBuffer\* buffers = calloc( req.count, sizeof(\*buffers) );  
                  struct v4l2\_buffer buf;

          for (numBufs = 0; numBufs < req.count; numBufs++) {//映射所有的缓存  
                          memset( &buf, 0, sizeof(buf) );  
                          buf.type = V4L2\_BUF\_TYPE\_VIDEO\_CAPTURE;  
                          buf.memory = V4L2\_MEMORY\_MMAP;  
                          buf.index = numBufs;  
                          if (ioctl(fd, VIDIOC\_QUERYBUF, &buf) == -1) {//获取到对应index的缓存信息，此处主要利用length信息及offset信息来完成后面的mmap操作。  
                                  return -1;  
                          }

                  buffers[numBufs].length = buf.length;  
                          // 转换成相对地址  
                          buffers[numBufs].start = mmap(NULL, buf.length,  
                                  PROT\_READ | PROT\_WRITE,  
                                  MAP\_SHARED,  
                                  fd, buf.m.offset);

                  if (buffers[numBufs].start == MAP\_FAILED) {  
                                  return -1;  
                          }

（8）开始采集视频

          int buf\_type= V4L2\_BUF\_TYPE\_VIDEO\_CAPTURE；  
                  int ret = ioctl(fd, VIDIOC\_STREAMON, &buf\_type);

（9）取出FIFO缓存中已经采样的帧缓存

          struct v4l2\_buffer buf;  
                  memset(&buf,0,sizeof(buf));  
                  buf.type=V4L2\_BUF\_TYPE\_VIDEO\_CAPTURE;  
                  buf.memory=V4L2\_MEMORY\_MMAP;  
                  buf.index=0;//此值由下面的ioctl返回  
                  if (ioctl(fd, VIDIOC\_DQBUF, &buf) == -1)  
                  {  
                          return -1;  
                  }

根据返回的buf.index找到对应的mmap映射好的缓存，取出视频数据。

（10）将刚刚处理完的缓冲重新入队列尾，这样可以循环采集

          if (ioctl(fd, VIDIOC\_QBUF, &buf) == -1) {  
                          return -1;  
                  }

（11）停止视频的采集

          int ret = ioctl(fd, VIDIOC\_STREAMOFF, &buf\_type);

（12）关闭视频设备

          close(fd);

四、 V4L2驱动框架

上述流程的各个操作都需要有底层V4L2驱动的支持。内核中有一些非常完善的例子。

比如：linux-2.6.26内核目录/drivers/media/video//zc301/zc301\_core.c 中的ZC301视频驱动代码。上面的V4L2操作流程涉及的功能在其中都有实现。

1、V4L2驱动注册、注销函数

Video核心层（drivers/media/video/videodev.c）提供了注册函数  
              int video\_register\_device(struct video\_device \*vfd, int type, int nr)  
                    video\_device: 要构建的核心数据结构  
                    Type: 表示设备类型，此设备号的基地址受此变量的影响  
                    Nr: 如果end-base>nr>0 ：次设备号=base（基准值，受type影响）+nr；  
                    否则：系统自动分配合适的次设备号

具体驱动只需要构建video\_device结构，然后调用注册函数既可。

如：zc301\_core.c中的  
                    err = video\_register\_device(cam->v4ldev, VFL\_TYPE\_GRABBER,  
                              video\_nr[dev\_nr]);  
          Video核心层（drivers/media/video/videodev.c）提供了注销函数  
                    void video\_unregister\_device(struct video\_device \*vfd)

2、struct video\_device 的构建

      video\_device结构包含了视频设备的属性和操作方法。参见zc301\_core.c

      strcpy(cam->v4ldev->name, "ZC0301[P] PC Camera");  
              cam->v4ldev->owner = THIS\_MODULE;  
              cam->v4ldev->type = VID\_TYPE\_CAPTURE | VID\_TYPE\_SCALES;  
              cam->v4ldev->fops = &zc0301\_fops;  
              cam->v4ldev->minor = video\_nr[dev\_nr];  
              cam->v4ldev->release = video\_device\_release;  
              video\_set\_drvdata(cam->v4ldev, cam);

大家发现在这个zc301的驱动中并没有实现struct video\_device中的很多操作函数，如：vidioc\_querycap、vidioc\_g\_fmt\_cap等。主要原因是struct file\_operations zc0301\_fops中的zc0301\_ioctl实现了前面的所有ioctl操作。所以就不需要在struct video\_device再实现struct video\_device中的那些操作了。

另一种实现方法如下：

static struct video\_device camif\_dev =  
        {  
                .name = "s3c2440 camif",  
                .type = VID\_TYPE\_CAPTURE|VID\_TYPE\_SCALES|VID\_TYPE\_SUBCAPTURE,  
                .fops = &camif\_fops,  
                .minor = -1,  
                .release = camif\_dev\_release,  
                .vidioc\_querycap = vidioc\_querycap,  
                .vidioc\_enum\_fmt\_cap = vidioc\_enum\_fmt\_cap,  
                .vidioc\_g\_fmt\_cap = vidioc\_g\_fmt\_cap,  
                .vidioc\_s\_fmt\_cap = vidioc\_s\_fmt\_cap,  
                .vidioc\_queryctrl = vidioc\_queryctrl,  
                .vidioc\_g\_ctrl = vidioc\_g\_ctrl,  
                .vidioc\_s\_ctrl = vidioc\_s\_ctrl,  
        };  
        static struct file\_operations camif\_fops =  
        {  
                .owner = THIS\_MODULE,  
                .open = camif\_open,  
                .release = camif\_release,  
                .read = camif\_read,  
                .poll = camif\_poll,  
                .ioctl = video\_ioctl2, /\* V4L2 ioctl handler \*/  
                .mmap = camif\_mmap,  
                .llseek = no\_llseek,  
        };

注意：video\_ioctl2是videodev.c中是实现的。video\_ioctl2中会根据ioctl不同的cmd来调用video\_device中的操作方法。

3、Video核心层的实现

参见内核/drivers/media/videodev.c

（1）注册256个视频设备

static int \_\_init videodev\_init(void)  
        {  
                int ret;  
                if (register\_chrdev(VIDEO\_MAJOR, VIDEO\_NAME, &video\_fops)) {  
                        return -EIO;  
                }  
                ret = class\_register(&video\_class);  
                ……  
        }

上面的代码注册了256个视频设备，并注册了video\_class类。video\_fops为这256个设备共同的操作方法。

（2）V4L2驱动注册函数的实现

int video\_register\_device(struct video\_device \*vfd, int type, int nr)  
        {  
                int i=0;  
                int base;  
                int end;  
                int ret;  
                char \*name\_base;

        switch(type) //根据不同的type确定设备名称、次设备号  
                {  
                        case VFL\_TYPE\_GRABBER:  
                                base=MINOR\_VFL\_TYPE\_GRABBER\_MIN;  
                                end=MINOR\_VFL\_TYPE\_GRABBER\_MAX+1;  
                                name\_base = "video";  
                                break;  
                        case VFL\_TYPE\_VTX:  
                                base=MINOR\_VFL\_TYPE\_VTX\_MIN;  
                                end=MINOR\_VFL\_TYPE\_VTX\_MAX+1;  
                                name\_base = "vtx";  
                                break;  
                        case VFL\_TYPE\_VBI:  
                                base=MINOR\_VFL\_TYPE\_VBI\_MIN;  
                                end=MINOR\_VFL\_TYPE\_VBI\_MAX+1;  
                                name\_base = "vbi";  
                                break;  
                        case VFL\_TYPE\_RADIO:  
                                base=MINOR\_VFL\_TYPE\_RADIO\_MIN;  
                                end=MINOR\_VFL\_TYPE\_RADIO\_MAX+1;  
                                name\_base = "radio";  
                                break;  
                        default:  
                                printk(KERN\_ERR "%s called with unknown type: %d\n",  
                                        \_\_func\_\_, type);  
                                return -1;  
                }

        /\* 计算出次设备号 \*/  
                mutex\_lock(&videodev\_lock);  
                if (nr >= 0 && nr < end-base) {  
                        /\* use the one the driver asked for \*/  
                        i = base+nr;  
                        if (NULL != video\_device[i]) {  
                                mutex\_unlock(&videodev\_lock);  
                                return -ENFILE;  
                        }  
                } else {  
                        /\* use first free \*/  
                        for(i=base;i<end;i++)  
                                if (NULL == video\_device[i])  
                                        break;  
                        if (i == end) {  
                                mutex\_unlock(&videodev\_lock);  
                                return -ENFILE;  
                        }  
                }  
                video\_device[i]=vfd; //保存video\_device结构指针到系统的结构数组中，最终的次设备号和i相关。  
                vfd->minor=i;  
                mutex\_unlock(&videodev\_lock);  
                mutex\_init(&vfd->lock);

        /\* sysfs class \*/  
                memset(&vfd->class\_dev, 0x00, sizeof(vfd->class\_dev));  
                if (vfd->dev)  
                        vfd->class\_dev.parent = vfd->dev;  
                vfd->class\_dev.class = &video\_class;  
                vfd->class\_dev.devt = MKDEV(VIDEO\_MAJOR, vfd->minor);  
                sprintf(vfd->class\_dev.bus\_id, "%s%d", name\_base, i - base);//最后在/dev目录下的名称  
                ret = device\_register(&vfd->class\_dev);//结合udev或mdev可以实现自动在/dev下创建设备节点  
                ……  
        }

从上面的注册函数中可以看出V4L2驱动的注册事实上只是完成了设备节点的创建，如：/dev/video0。和video\_device结构指针的保存。

（3）视频驱动的打开过程

当用户空间调用open打开对应的视频文件时，如：

int fd = open(/dev/video0, O\_RDWR);

对应/dev/video0的文件操作结构是/drivers/media/videodev.c中定义的video\_fops。

static const struct file\_operations video\_fops=  
        {  
                .owner = THIS\_MODULE,  
                .llseek = no\_llseek,  
                .open = video\_open,  
        };

奇怪吧，这里只实现了open操作。那么后面的其它操作呢？还是先看看video\_open吧。

static int video\_open(struct inode \*inode, struct file \*file)  
        {  
                unsigned int minor = iminor(inode);  
                int err = 0;  
                struct video\_device \*vfl;  
                const struct file\_operations \*old\_fops;

        if(minor>=VIDEO\_NUM\_DEVICES)  
                        return -ENODEV;  
                mutex\_lock(&videodev\_lock);  
                vfl=video\_device[minor];  
                if(vfl==NULL) {  
                        mutex\_unlock(&videodev\_lock);  
                        request\_module("char-major-%d-%d", VIDEO\_MAJOR, minor);  
                        mutex\_lock(&videodev\_lock);  
                        vfl=video\_device[minor]; //根据次设备号取出video\_device结构  
                        if (vfl==NULL) {  
                                mutex\_unlock(&videodev\_lock);  
                                return -ENODEV;  
                        }  
                }  
                old\_fops = file->f\_op;  
                file->f\_op = fops\_get(vfl->fops);//替换此打开文件的file\_operation结构。后面的其它针对此文件的操作都由新的结构来负责了。也就是由每个具体的video\_device的fops负责。  
                if(file->f\_op->open)  
                        err = file->f\_op->open(inode,file);  
                if (err) {  
                        fops\_put(file->f\_op);  
                        file->f\_op = fops\_get(old\_fops);  
                }  
                ……  
        }