

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/time.h>
#include <fcntl.h>
#include <unistd.h>
#include <linux/types.h>
#include <linux/videodev.h>
#include <malloc.h>

#include <string.h>
#include <sys/time.h>

#define USB_VIDEO_DEV "/dev/ video0"
#define FILE_NAME "/tmp/1.jpg"
#define STILL_IMAGE -1
#define VIDEO_START 0
#define VIDEO_STOP 1
#define VIDEO_PALETTE_RAW_JPEG 20
#define VIDEO_PALETTE_JPEG 21
static int debug = 1;

int get_jpegsize(unsigned char *buf, int size)
{
    int i;
    for(i = 1024; i < size; i++)
    {
        if ( (buf[i] == 0xFF)&&(buf[i+1] == 0xD9)) return i+2;//jpeg 文件格式中是以 0xFF 0xD9 结尾
        的,

        //以此判断文件大小
    }
    return -1;
}

int main(int argc, char *argv[])
{
    int usb_camera_fd = -1, framesize=0, jpegsize=0;
    char *usb_video_dev = USB_VIDEO_DEV; //" /dev/ video0"
    char *filename = FILE_NAME; // "/tmp/1.jpg"
```

```
FILE *fp;
struct video_capability video_caps;
struct video_channel video_chan;
struct video_picture video_pic;
struct video_mbuf video_mbuffer;
struct video_mmap vid_mmap;
unsigned char *mapaddr=NULL,*framebuffer=NULL,*destbuffer=NULL;
usb_camera_fd = open(usb_video_dev,O_RDWR);//打开设备,可读写,也即打开"/dev/ video0"
// usb_camera_fd 是设备号, open 成功则返回设备号
if (usb_camera_fd == -1)
{
    fprintf(stderr,"Can't open device %s",usb_video_dev);
    return 1;
}
/*****video_capability*****/
if (ioctl(usb_camera_fd,VIDIOCGCAP,&video_caps) == -1)//get videodevice capability 获取设备基本信息。
{
    perror("Couldn't get videodevice capability");
    return -1;
}

if (debug)
{
    printf("video name : %s\n",video_caps.name);// struct video_capability video_caps;
    printf("video_caps.channels : %d\n",video_caps.channels);
    printf("video_caps.type : 0x%x\n",video_caps.type);
    printf("video maxwidth : %d\n",video_caps.maxwidth);
    printf("video maxheight : %d\n",video_caps.maxheight);
    printf("video minwidth : %d\n",video_caps.minwidth);
    printf("video minheight : %d\n",video_caps.minheight);
}

/*****video_channel*****/

/*video_chan.channel = video_caps.channels;*/
// struct video_channel video_chan;
if (ioctl(usb_camera_fd,VIDIOCGCHAN,&video_chan) == -1)//获取信号源的属性
{
    perror("ioctl (VIDIOCGCAP)");
    return -1;
}
if (debug)
{
```

```
printf("video channel: %d\n",video_chan.channel);
printf("video channel name: %s\n",video_chan.name);
printf("video channel type: %d\n",video_chan.type);
}*/

/*****video_picture*****/
if (ioctl(usb_camera_fd,VIDIOCGPICT,&video_pic) == -1)//获取设备采集的图象的各种属性
{
perror("ioctl (VIDIOCGPICT)");
return -1;
}

屏蔽了? /*video_pic.palette = VIDEO_PALETTE_JPEG;
video_pic.depth = 8;
video_pic.hue = 50110;
video_pic.whiteness = 41380;

if (ioctl(usb_camera_fd,VIDIOCSPICT,&video_pic) == -1)//设置设备采集的图象的属性
{
perror("ioctl (VIDIOCSPICT)");
//return -1;
}*/屏蔽了?

if (debug)
{
printf("video_pic.brightness : %d\n",video_pic.brightness);
printf("video_pic.colour : %d\n",video_pic.colour);
printf("video_pic.contrast : %d\n",video_pic.contrast);
printf("video_pic.depth : %d\n",video_pic.depth);
printf("video_pic.hue : %d\n",video_pic.hue);
printf("video_pic.whiteness : %d\n",video_pic.whiteness);
printf("video_pic.palette : %d\n",video_pic.palette);
}
/*****video_mbuf*****/
memset(&video_mbuffer,0,sizeof(video_mbuffer));

//初始化 video_mbuf, 以得到所映射的 buffer 的信息
if (ioctl(usb_camera_fd,VIDIOCGMBUF,&video_mbuffer) == -1)//video_mbuf
{
perror("ioctl (VIDIOCGMBUF)");
return -1;
}
if (debug)
```

```
{
printf("video_mbuffer.frames : %d\n",video_mbuffer.frames); // frames 最多支持的帧数
printf("video_mbuffer.offsets[0] : %d\nvideo_mbuffer.offsets[1] : %d\n",video_mbuffer.offsets
[0],video_mbuffer.offsets[1]); //每帧相对基址的偏移
printf("video_mbuffer.size : %d\n",video_mbuffer.size); //每帧大小

}
////等一等，对照 v4l.docx 看到这儿了

//将 mmap 与 video_mbuf 绑定
mapaddr=(unsigned char
*)mmap(0,video_mbuffer.size,PROT_READ,MAP_SHARED,usb_camera_fd, 0);

if (mapaddr < 0)
{
perror("v4l mmap");
return -1;
}
/*****video_mmap*****/

vid_mmap.width = 320;
vid_mmap.height = 240;
vid_mmap.frame = 0;//单帧采集
vid_mmap.format = VIDEO_PALETTE_JPEG;

/*****start capture*****/Mmap 方式下真正开始视频
截取

//若调用成功，开始一帧的截取，是非阻塞的 是否截取完毕留给 VIDIOCSYNC 来判断
if (ioctl(usb_camera_fd,VIDIOCCAPTURE,VIDEO_START) == -1)
{
perror("ioctl (VIDIOCCAPTURE)");
return -1;
}

/*****wait for ready*****/调用 VIDIOCSYNC 等待一
帧截取结束

//若成功，表明一帧截取已完成。可以开始做下一次 VIDIOCMCAPTURE
//frame 是当前截取的帧的序号
/*if (ioctl(usb_camera_fd, VIDIOCSYNC, &vid_mmap.frame) == -1)
{
perror("ioctl (VIDIOCSYNC)");
//return -1;
}
```

```
    }*/

framesize=320*240>>2;//实际采集到的 jpeg 图像的大小最多也就十几 KB

//获取刚刚采集到的图像数据帧的地址

framebuffer = mapaddr + video_mbuffer.offsets[vid_mmap.frame];

//获取图像的大小

//jpegsize = get_jpegsize(framebuffer, framesize);

if (jpegsize < 0)
{
printf("Can't get size of jpeg picture\n");
return 1;
}

//分配空间,准备将 destbuffer 缓冲区中的图像数据写入文件
destbuffer = (unsigned char *)malloc(video_mbuffer.size);
if ( destbuffer == NULL)
{
printf("malloc memory for destbuffer error\n");
return -1;
}
memcpy(destbuffer,framebuffer,video_mbuffer.size);

fp = fopen(filename,"wb");//打开文件
if (!fp)
{
printf("Can't open file %s",filename);
//return -1;
}

fwrite(destbuffer,video_mbuffer.size,1,fp);//写入文件
fclose(fp);//关闭文件

free(destbuffer);//释放空间
munmap(mapaddr,video_mbuffer.size);//取消绑定
close(usb_camera_fd);
return 1;
}
```

v4l2 驱动编写篇——介绍

原文网址: <http://lwn.net/Articles/203924/>

笔者最近有机会写了一个摄像头的驱动, 是“One laptop per child”项目中的摄像头专用的。这个驱动使用了为此目的而设计的内核 API: the Video4Linux2 API。在写这个驱动的过程中, 笔者发现了一个惊人的问题: 这个 API 的文档工作做得并不是很好, 而用户层的文档则写的, 实际上, 相当不错。为了补救现在的状况, LWN 将在未来的内个月里写一系列文章, 告诉大家如何写 V4L2 接口的驱动。

V4L2 有一段历史了。大约在 1998 的秋天, 它的光芒第一次出现在 Bill Dirks 的眼中。经过长足的发展, 它于 2002 年 11 月, 发布 [2.5.46](#) 时, 融入了内核主干之中。然而直到今天, 仍有一部分内核驱不支持新的 API, 这种新旧 API 的转换工作仍在进行。同时, V4L2 API 也在发展, 并在 2.6.18 版本中进行了一些重大的改变。支持 V4L2 的应用依旧相对较少。

V4L2 在设计时, 是要支持很多广泛的设备的, 它们之中只有一部分在本质上是真正的视频设备:

video capture interface (影像捕获接口) 从调谐器或是摄像头上获取视频数据。对很多人来讲, 影像捕获 (video capture) 是 V4L2 的基本应用。由于笔者在这方面的经验是强项, 这一系列文章也趋于强调捕获 API, 但 V4L2 不止这些。

video output interface (视频输出接口) 允许应用使用 PC 的外设, 让其提供视频图像。有可能是通过电视信号的形式。

捕获接口还有一个变体, 存在于 **video overlay interface** (视频覆盖接口) 之中。它的工作是方便视频显示设备直接从捕获设备上获取数据。视频数据直接从捕获设备传到显示设备, 无需经过 CPU。

VBI interfaces (Vertical blanking interval interface, 垂直消隐接口) 提供垂直消隐期的数据接入。这个接口包括 raw 和 sliced 两种接口, 其分别在于硬件中处理的 VBI 数据量。(为什么要在消隐期间接入数据呢? [看这里](#))

radio interface (广播接口) 用于从 AM 或 FM 调谐器中获得音频数据。

也可能出现其它种类的设备。V4L2 API 中还有一些关于编译码和效果设备的 stub, 他们都用来转换视频数据流。然而这块的东西尚未完成确定, 更不说应用了。还有“teletext”和“radio data system”的接口, 他们目前在 V4L1 API 中实现。他们没有移动到 V4L2 的 API 中来, 而且目前也没有这方面的计划。

视频驱动与其他驱动不同之处, 在于它的配置方式多种多样。因此大部分 V4L2 驱动都有一些特定的代码, 好让应用可以知道给定的设备有什么功能, 并配置设备, 使其按期望的方式工作。V4L2 的 API 定义了几十个回调函数, 用来配置如调谐频率、窗口和裁剪、帧速率、视频压缩、图像参数 (亮度、对比度...), 视频标准、视频格式等参数。这一系列文章的很大部分都要用来考察这些配置的过程。

然后, 还有一个小任务, 就是有效地在视频频率下进行 I/O 操作。V4L2 定义了三种方法来在用户空间和外设之间移动视频数据, 其中有些会比较复杂。视频 I/O 和视频缓冲层, 将会分成两篇文章来写, 它们是用来处理一些共性的任务的。

随后的文章每几周发一篇, 共会加入到下面的列表中。

v4l2 驱动编写篇二--注册和打开

原文网址: <http://lwn.net/Articles/204545/>

这篇文章是 LWN 写 V4L2 接口的设备驱动系列文章的第二篇。没看过介绍篇的, 也许可以从那篇开始看。这一期文章将关注 Video for Linux 驱动的总体结构和设备注册过程。

开始之前, 有必要提一点, 那就是对于搞视频驱动的人来说, 有两份资料是非常有价值的。[The V4L2 API Specification](#) (V4L2 API 说明) 这份文档涵盖了用户空间视角下的 API, 但在

很大程度上，V4L2 驱动直接实现的就是那些 API。所以大部分结构体是相同的，而且 V4L2 调用的语义也表述得很明了。打印一份出来（可以考虑去掉自由文本协议的文本内容，以保护树木 [前面是作者原文，节省纸张就是保护树木嘛]），放在容易够到的地方。

内核代码中的 vivi 驱动，即 `drivers/media/video/vivi.c`。这是一个虚拟驱动。它可以用来测试，却不使用任何实际硬件。这样，它就成了一个教人如何写 V4L2 驱动的非常好的实例。

首先，每个 V4L2 驱动都要包含一个必须的头文件：

```
1 #include <linux/videodev2.h>
```

大部分所需的信息都在这里。作为一个驱动作者，当挖掘头文件的时候，你可能也得看看 `include/media/v4l2-dev.h`，它定义了许多你将来要打交道的结构体。

一个视频驱动很可能要有处理 PCI 总线，或 USB 总线的部分。这里我们不会花什么时间还接触这些东西。通常会有一个内部 I2C 接口，我们在这一系列的后续文章中会接触到它。然后还有一个 V4L2 的子系统接口。这个子系统是围绕 `video_device` 这个结构体建立的，它代表的是一个 V4L2 设备。讲解进入这个结构体的一切，将会是这个系列中几篇文章的主题。这里我们先有一个概览。

`video_device` 结构体的 `name` 字段是这一类设备的名字，它会出现于内核日志和 `sysfs` 中出现。这个名字通常与驱动的名字相同。

所表示的设备有两个字段来描述。第一个字段 (`type`) 似乎是从 V4L1 的 API 中遗留下来的，它可以下列四个值之一：

`VFL_TYPE_GRABBER` 表明是一个图像采集设备—包括摄像头、调谐器，诸如此类。

`VFL_TYPE_VBI` 代表的设备是从视频消隐的时间段取得信息的设备。

`VFL_TYPE_RADIO` 代表无线电设备。

`VFL_TYPE_VTX` 代表视传设备。

如果你的设备支持上面提到的不只一种功能，那就要为每个功能注册一个 V4L2 设备。然而在 V4L2 中，注册的每个设备都可以用作它实际支持的各种模式（就是说，你要为一个物理设备创建不多个设备节点，但你却可以调用任意一个设备节点，来实现这个物理设备支持的任意功能）。实质上的问题是，在 V4L2 中，你实际上只需一个设备，注册多个 V4L2 设备只是为了与 V4L1 兼容。

第二个字段是 `type2`，它以掩码的形式对设备的功能提供了更详尽的描述。它可以包含以下值：

`VID_TYPE_CAPTURE` 它可以捕获视频数据

`VID_TYPE_TUNER` 它可以接收不同的频率

`VID_TYPE_TELETEXT` 它可以抓取字幕

`VID_TYPE_OVERLAY` 它可以将视频数据直接覆盖到显示设备的帧缓冲区

`VID_TYPE_CHROMAKEY` 一种特殊的覆盖能力，覆盖的仅是帧缓冲区中像素值为某特定值的区域

`VID_TYPE_CLIPPING` 它可以剪辑覆盖数据

`VID_TYPE_FRAMERAM` 它使用帧缓冲区中的存储器

`VID_TYPE_SCALES` 它可以缩放视频数据

`VID_TYPE_MONOCHROME` 这个是一个纯灰度设备

`VID_TYPE_SUBCAPTURE` 它可以捕获图像的子区域

`VID_TYPE_MPEG_DECODER` 它支持 mpeg 码流解码

`VID_TYPE_MPEG_ENCODER` 它支持编码 mpeg 码流

`VID_TYPE_MJPEG_DECODER` 它支持 mjpeg 解码

VID_TYPE_MJPEG_ENCODER 它支持 mjpeg 编码

V4L2 驱动还要初始化的一个字段是 `minor`, 它是你想要的子设备号。通常这个值都设为 -1, 这样会让 `video4linux` 子系统在注册时自动分配一个子设备号。

在 `video_device` 结构体中, 还有三组不同的函数指针集。第一组只包含一个函数, 那就是 `release()`, 如果驱动没有 `release()` 函数, 内核就会抱怨 (笔者发现一个件有趣的事, 就是这个抱怨涉及到冒犯一篇 LWN 文章的作者)。`release()` 函数很重要: 由于多种原因, 对 `video_device` 的引用可以在最后一个应用关闭文件描述符后很长一段时间依然保持。它们甚至可以在设备已经注销后依然保持。因此, 在 `release()` 函数调用前, 释放这个结构体是不安全的。所以这个函数通常要包含一个简单的 `kfree()` 调用。

`video_device` 的 `file_operations` 结构体包含都是常规的函数指针。视频设备通常都包括 `open()` 和 `release()` 函数。注意: 这里所说的 `release` 函数并非上面所讲到的同名的 `release()` 函数, 这个 `release()` 函数只要设备关闭就要调用。通常都还要有 `read()` 和 `write()` 函数, 这取决于设备的功能是输入还是输出。然而我们要注意的, 对于视频流设备而言, 传输数据还有别的方法。多数处理视频流数据的设备还需要实现 `poll()` 和 `mmap()`; 而且每个 V4L2 设备都要有 `ioctl()` 函数, 但是也可以使用 V4L2 子系统的 `video_ioctl2()`;

第三组函数存在于 `video_device` 结构体本身里面, 它们是 V4L2 API 的核心。这组函数有几十个, 处理不同的设备配置操作、流输入输出和其他操作。

最后, 从一开始就要知道的一个字段就是 `debug`, 可以把它设成是 `V4L2_DEBUG_IOCTL` 或 `V4L2_DEBUG_IOCTL_ARG` (或是两个都设, 这是个掩码), 可以生成很多的调试信息, 它们可以帮助一个迷糊的程序员找到毛病, 知道为什么驱动和应用谁也不知道对方在说什么。

视频设备注册

一旦 `video_device` 已经配置好, 就可以下面的函数注册了:

```
1 int video_register_device(struct video_device *vfd, int type, int nr);
```

这里 `vfd` 是设备的结构体 (`video_device`), `type` 的值与它的 `type` 字段值相同, `nr` 也是一样, 想要的子设备号 (为 -1 则注册时自动分配)。返回值当为 0, 若返回的是负的

出错码, 则表明出错了, 和通常一样, 我们要知道, 设备一旦注册, 它的函数可能就会立即调用, 所以不到一切准备就绪, 不要调用 `video_register_device()`;

设备的注销方法为:

```
1 void video_unregister_device(struct video_device *vfd);
```

请继续关注本系列的下篇文章, 我们将会看看这些函数的具体实现。

`open()` 和 `release()` 每个 V4L2 设备都需要 `open()` 函数, 其原型也与常规的相同。

```
int (*open)(struct inode *inode, struct file *filp);
```

`open()` 函数要做的第一件事是通过给定的 `inode` 找到内部设备, 这是通过找到 `inode` 中存储的子设备号来完成的。这里还可以实现一定数量的初始化, 如果有关闭电源选项的话, 这个时间恰好可以用来开启硬件电源。

V4L2 规范还定义了一些相关的惯例。其一是: 根据其设计, 文件描述符可以在给定的任何时间重复打开。这样设定的目的是当一个应用在显示 (或是产生) 视频信号时, 另一个应用可以改变控制值。所以, 虽然某些操作是独占性质的 (特别是数据读、写等), 但是设备总体本身是要支持描述符复用的。

另一个值得一提的惯例是: `open()` 函数, 总体上讲, 不可以改变硬件中现行的操作参数。有

些时候可能会有这样的情况：通过命令程序，根据一组参数（分辨率，视频格式等）来改变摄像头配置，然后运行一个完全不同的程序来，比如说，从摄像头获取上帧图像。如果摄像头在设置中途改变了，这种模式就不好用。所以除非应用明确表示要改变设置（这种情况当然不包括在 `open` 函数中）V4L2 驱动要尽量保持设定不变。

`release()` 函数做一些必要清理工作。因为文件描述符可以重复打开，所以 `release` 函数中减小引用计数，并在彻底退出之前做检查。如果关闭的文件描述符是用来传输数据的，`release` 函数很可能要关掉 DMA，并做一些其他的清理工作。

本系列的下一篇文章我们将进入查询设备功能和设定系统模式的冗长过程之中，请续断关注。

v4l2 驱动编写篇三--基本 I/O 处理

如果有人 `video for linux` API 规范上花了我时间的话，他肯定已经注意到了一个问题，那就是 V4L2 大量使用了 `ioctl` 接口。视频硬件有大量的可操作旋钮，可能比其它任何处设都要多。视频流要与许多参数相联系，而且有很大一部分处理要通过硬件进行。不使用硬件有良好支持模式可能导致表现不好，甚至根本没有表现。所以我们不得不揭露硬件的许多特性，而对最终应用表现得怪异一点。

传统上来讲，视频驱动中包含的 `ioctl()` 函数一般会长得像一部小说，而函数所得到的结论也往往比小说更令人满意来，他们往往在中间拖了很多（这句话完全不明白什么意思）。所以 V4L2 的 API 在 2.6.18 版本的内核开始做出了改变。冗长的 `ioctl` 函数被替换成了一个大的回调函数的集合，每个回调函数实现自己的 `ioctl` 函数。实际上，在 2.6.19-rc3 中，有 79 个这样的回调函数。而幸运的是，多数驱动并不需实现所有的回调函数，甚至都不是大部分回调函数。

在 `ioctl()` 函数中发生的事情都放到了 `drivers/media/video/videodev.c` 里面。这部分代码处理数据在内核和用户空间之间的传输并把 `ioctl` 调用发送给驱动。要使用它的话，只要把 `video_device` 中的 `video_ioctl2()` 做为 `ioctl()` 来调用就行了。实际上，多数驱动也要把它当成 `unlocked_ioctl()` 来用。Video4Linux2 层的锁可以对其进行处理，而且驱动也应该在合适的地方加锁。（这一段没看明白，乱写的）

v4l2 驱动编写篇第四--输入输出

输入和输出

这是不定期发布的关于写视频驱动程序的 LWN 系统文章的第四篇。没有看过介绍篇的，也许想从这里开始。本周的文章介绍的是应用程序如何确定在特定适配器上哪些输入和输出可用，并且它们之间做出选择。

在很多情况下，视频适配器并不能提供很多的输入输出选项。比如说摄像头控制器，可能只是提供摄像头，而没什么别的功能。然而，在一些其他的情况下，事情将变得很复杂。一个电视卡可能对应板上不用的接头有不同的输入。他甚至可能有可以独立发挥其功能的多路调谐器。有时，那些输入会有不同的特性；有些调谐器可以支持比其他的更广泛的视频标准。对于输出来说，也有同样的问题。

很明显，若想一个应用可以有效地利用视频适配器，它必须有能力找到可用的输入和输出，而且他必须能找到他想操作的那一个。为此，Video4Linux2 API 提供三种不同的 `ioctl()` 调用来处理输入，相应地有三个来处理输出。

这三个（对于硬件支持的每一个功能）驱动都要支持。虽然如此，对于简单的硬件而言，代码还是

很简单的.驱动也要提供一此启动时的默认值.然而,驱动不应该做的是,在应用退出时重置输入输出信息.对于其他视频参数,在多次打开之间,参数应维持不变.

视频标准

在我们进入输入输出的细节之前,我们应该先了解一下视频标准.这些标准描述的是视频为进行传输而做出的格式转换-分辨率,帧频率等.这些标准通常是由每一个国家的监管部门制定的.现在世界上使标准主要的有三个:NTSC(主要是北美使用),PAL(主要是欧洲,非洲和中国),和 SECAM(法,俄和非洲部分地区).然而这在标准在国家之间都有变化,而且有些设备比其他设备能更加灵活,能与更多的标准变种协同工作.

V4L2 使用 `v4l2_std_id` 来代表视频标准,它是一个 64 位的掩码.每个标准变种在掩码中就是一位.所以标准 NTSC 就是 `V4L2_STD_NTSC_M`, 值为 `0x1000`,而日本的变种就是 `V4L2_STD_NTSC_M_JP`

(`0x2000`).如果一个设备可以处理所以有 NTSC 变种,它就可以设为 `V4L2_STD_NTSC`,它可以所有相关位置位.对 PAL 和 SECAM 标准,也存在一组类似的位集. See [this page](#) for a complete list.

对于用户空间而言, V4L2 提供一个 `ioctl()` 命令(`VIDIOC_ENUMSTD`),它允许应用查询设备实现了哪些标准.驱动却无需直接回答查询,而是将 `video_device` 结构体的 `tvnorm` 字段设置为它所支持的所有标准.然后 V4L2 层会向应用输出所支持的标准.`VIDIOC_G_STD` 命令,可以用来查询现在哪种标准是激活的,它也是在 V4L2 层通过返回 `video_device` 结构的 `current_norm` 字段来处理的,驱动程序应在启动时,初始化 `current_norm` 来反应现实情况.有些应用即使他并没有设置过标准,发现标准没有设置也会感到困惑.

当某个应用想要申请某个标准的时候,会发出一个 `VIDIOC_S_STD` 调用,该调用通过下面的函数传到驱动:

```
int (*vidioc_s_std)(struct file *file, void *private_data, v4l2_std_id std);
```

驱动要对硬件编程,以使用给定的标准,并返回 0(或是负的出错编码).V4L2 层需要把 `current_norm` 设为新的值.

应用可能想要知道硬件所看到的是何种信号,答案可以通过 `VIDIOC_QUERYSTD` 找到,它到了驱动里面就是:

```
int (*vidioc_querystd)(struct file *file, void *private_data, v4l2_std_id *std);
```

驱动要尽可能地在這個字段填写详细信息.如果硬件没有提供足够的信息, `std` 字段就会暗示任何可能出现的标准.

这里还有一点值得一提:所以的视频设备必须支持(或是声明支持)至少一种视频标准.视频标准对于摄像头来说没什么意义,它不与任何监管制度绑定.但是也不存一个标准说“我是个摄像头,我什么都能做”,所以 V4L2 层有很多摄像头声明可以返回 PAL 或 NTSC 数据(实际只是如些声明而已).

输入

视频捕获的应用首先要通过 `VIDIOC_ENUMINPUT`

命令来枚举所有可用的输入.在 V4L2 层,这个调用会转换成调用一个驱动中对应的回调函数:

```
int (*vidioc_enum_input)(struct file *file, void *private_data, struct v4l2_input *input);
```

在这个调用中, `file` 对就的是打开的视频设备. `private_data` 是驱动的私有字段, `input` 字段是真正的传递的信息,它有如下几个值得关注的字段:

`__u32 index`:应用关注的输入的索引号;这是惟一个用户空间设定的字段.驱动要分配索

引号给输入，从 0 开始，依次往上增加。想要知道所以可用的输入的应用会调用 VIDIOC_ENUMINPUT

调用过索引号从 0 开始，并开始递增。一旦返回 EINVAL，应用就知道，输入已经用光了。只要有输入，输入索引号 0 就一定要存在的。

__u8 name[32]: 输入的名字，由驱动设定。简单起见，可以设为"Camera"，诸如此类；如果卡上有多个输入，名称就要与接口的打印相符合。

__u32 type: 输入的类型，现在只有两个值可选: V4L2_INPUT_TYPE_TUNER 和 V4L2_INPUT_TYPE_CAMERA。

__u32 audioset: 描述哪个音频输入可以与些视频输入相关联。音频输入与视频输入一样通过索引号枚举（我们会在另一篇文章中关注音频），但并非所有的音频与视频的组合都是可用的。这个字段是一个掩码，代表对于当前枚举出的视频而言，哪些音频输入是可以与之关联的。如果没有音频输入可以与之关联，或是只有一个可选，那么就可以简单地把这个字段置 0。

__u32 tuner: 如果输入是一个调谐器 (type 字段置为 V4L2_INPUT_TYPE_TUNER)，这个字段就是会包含一个相应的调谐设备的索引号。枚举和调谐器的控制也将在未来的文章中讲述。

v4l2_std_id std: 描述设备支持哪个或哪些视频标准。

__u32 status: 给出输入的状态。全整的标识符集合可以在 V4L2 的文档中找到（[即这里](#)）；简而言之，status

中设置的每一位都代表一个问题。这些问题包括没有电源，没有信号，没有同频锁，或 the presence of Macrovision（这个是什么意思？没查到）或是其他一些不幸的问题。

__u32 reserved[4]: 保留字段，驱动应该将其置 0。

通常驱动会设置上面所有的字段，并返回 0。如果索引值超出支持的输入范围，应该返回 -EINVAL。这个调用里可能出现的错误不多。

当应用想改变现行的输入时，驱动会收到一个对回调函数 vidioc_s_input() 的调用。

```
int (*vidioc_s_input) (struct file *file, void *private_data, unsigned int index);
```

index 的值与上面讲到的意义相同 – 它来确定哪个输入是相要的。驱动要对硬件编辑，选择那个输入并返回 0。也有可能要返回 -EINVAL

(索引号不正确时) 或 -EIO

(硬件有问题)。即使只有一路输入，驱动也要实现这个回调函数。

还有另一个回调函数，指示哪一个输入是激活状态的：

```
int (*vidioc_g_input) (struct file *file, void *private_data, unsigned int *index);
```

这里驱动把 index 值设为对应的输入的索引号。

输出

枚举和选择输出的过程与输入的是十分相似的。所以这里的描述就从简。输入枚举的回调函数是这样的：

```
int (*vidioc_enumoutput) (struct file *file, void *private_data, struct v4l2_output *output);
```

v4l2_output

结构的字段是：

__u32 index: 相关输入的索引号。其工作方式与输入的索引号相同：它从 0 开始递增。

__u8 name[32]: 输出的名字。

__u32 type: 输入的类型。支持的输出类型为：V4L2_OUTPUT_TYPE_MODULATOR

用于模拟电视调制器,V4L2_OUTPUT_TYPE_ANALOG

用于基本模拟视频输出,和 V4L2_OUTPUT_TYPE_ANALOGVGAOVERLAY

用于模拟 VGA 覆盖设备.

__u32 audioset: 能与这个视频协同工作的音频集.

__u32 modulator: 与此设备相关的调制器 (对于类型为 V4L2_OUTPUT_TYPE_MODULATOR 的设备而言).

v4l2_std_id std:输出所支持的视频标准.

__u32 reserved[4]: 保留字段,要设置为 0.

也有用于获得和设定现行输入设置的回调函数;他们与输入的具体对称性:

```
int (*vidioc_g_output) (struct file *file, void *private_data, unsigned
int *index);          int (*vidioc_s_output) (struct file *file, void
*private_data,         unsigned int index);
```

即便只有一个输出,设备驱动也要定义所有上述的三个回调函数.

有了这些函数之后,V4L2 应用就可以知道有哪些输入和输出,并在它们间进行选择.然而选择这些输入输出中所传输的是什么视频数据流则是一件更加复杂的事情.本系列的下一期文章,我们将关注视频数据流的格式,以及如何与用户空间协定数据格式.

v4l2 文档第五 A--颜色与格式

颜色与格式这是不定期发布的关于写视频驱动程序的 LWN 系统文章的第五篇.没有看过介绍篇的,也许想从[这里](#)开始.

应用在可以使视频设备工作之前,它必须与驱动达成了解,知道视频数据是何种格式的.这种协商将是一个非常复杂的过程,其原因有二:1、视频硬件所支持的视频格互不相同.2、在内核的格式转换是令人难以接受的.所以应用在找出一种硬件支持的格式,并做出一种大家都可以接受的配置.这篇文章将会讲述格式的基本描述方式;下期文章则会讲述 V4L2 驱动与应用协商格式时所实现的 API.

色域

色域在广义上来讲,就是系统在描述色彩时所使用的坐标.V4L2 规范中定义了好几个,但只有两个使用最为广泛.它们是:

V4L2_COLORSPACE_SRGB.多数开发者所熟悉的 [red,green,blue]数组包含在这个色域之中.它为每一种颜色提供了一个简单的强度值,把它们混合在一起,从而产生了一种广泛的颜色的效果.表示 RGB 值的方法有很多,我们在下面将会有所介绍.

这个色域也包含 YUV 和 YCbCr 的表示方法,这个表示方法最早是为了早期的彩色电视信号可以在黑白电视中的播放,所以 Y (或说亮度)值只是一个简单的亮度值,单独播放时可以产生灰度图像.U 和 V (或 Cb 和 Cr)色度值描述的是色彩中蓝色和红色的分量.绿色可以通过从亮度中减去这些分量而得到.YUV 和 RGB 之间的转换并不简单,但是我们有一些成形的公式可选.

注意: YUV 和 YCbCr 并非完成一样,虽然有时他们的名字会替代使用.

V4L2_COLORSPACE_SMPTE170M 这个是 NTSC 或 PAL 等电视信号的模拟色彩表示方法,电视调谐器通常产生的色域都属于这个色域.

还存在很多其他的色域,他们多数都是电视相关标准的变种.点击查看 [V4L2 规范](#)中的详细列表.

密集存储和平面存储[p=21, null, left]如汝所见，像素值是以数组的方式表示的，通常由 RGB 或 YUV 值组成。要把这数组组织成图像，通常有两种常用的方法。

Packed 格式把一个像素的所有值存领教在一起。


















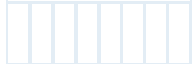



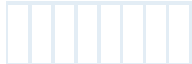











Planar 格式把每一个分量单独存储成一个阵列，这样在 YUV 格式中，所有 Y 值都连续地一起存储在一个阵列中，U 值存储在另一个中,V 值存在第三个中.这些平面常常都存储在一个缓冲区中,但并不一定非要这样。

紧密型存储方式可能使用更为广泛，特别是 RGB 格式，但这两种存储方式都可以由硬件产生并由应用程序请求。如果设备可以产生紧密型和平面型两种，那么驱动就要让两种都在用户空间可见。

四字符码 (four Charactor Code: FourCC)[p=21, null, left]V4L2 API 中表示色彩格式采用的是广受好评的四字符码 (fourcc)机制。这些编码都是 32 位的值，由四个 ASCII 码产生。如此一来，它就有个优点就是，易于传递，对人可读。当一个色彩格式读作，例如，”RGB4”就没有必要去查表了。

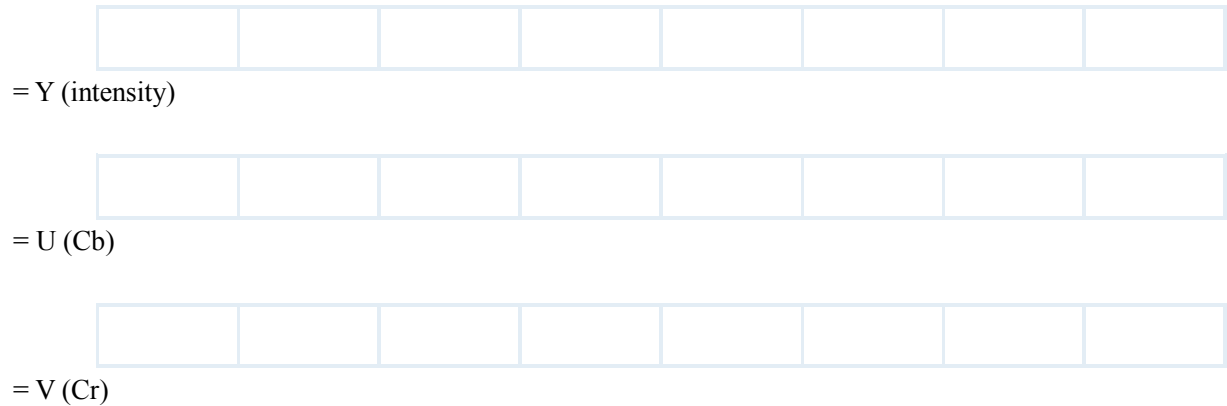
[p=21, null, left]注意：四字符码在很不同的设定中都会使用，有些还是早于 linux 的.Mplayer 中内部使用它们，然而，fourcc 只是说明一种编码机制，并不说明使用何种编码。Mplayer 有一个转换函数，用于在它自己的 fourcc 码和 v4l2 用的 fourcc 码之间做出转换。

RGB 格式[p=21, null, left]在下面的格式描述中，字节是按存储顺序列出的。在小端模式下，LSByte 在前面。每字节的 LSbit 在右侧。每种色域中，轻阴影位是最高有效的。

Name	fourcc	Byte 0	Byte 1	Byte 2	Byte 3
V4L2_PIX_FORMAT_RGB332	RGB1				
V4L2_PIX_FORMAT_RGB444	R444				
V4L2_PIX_FORMAT_RGB555	RGB0				
V4L2_PIX_FORMAT_RGB565	RGBP				
V4L2_PIX_FORMAT_RGB555X	RGBQ				
V4L2_PIX_FORMAT_RGB565X	RGBR				
V4L2_PIX_FORMAT_BGR24	BGR3				
V4L2_PIX_FORMAT_RGB24	RGB3				
V4L2_PIX_FORMAT_BGR32	BGR4				
V4L2_PIX_FORMAT_RGB32	RGB4				
V4L2_PIX_FORMAT_SBGGR8	BA81				
					

当使用有空位(上图中灰色部分)的格式，应用可以使用空位作 **alpha**（透明度）值。
上面的最后一个格式是”Bayer（人名好像是）”格式，此格式与多数摄像机感光器所得到的真实数据非常接近。每个像素都有绿色分量，但蓝和红只是隔一个像素才有分量。本质上讲，绿色带有更重的强度信息，而蓝红色则在丢失时以相隔像素的内插值替换。这种模式我们交在 YUV 格式中再次见到。

YUV 格式[p=21, null, left]YUV 的紧密型模式在下面首先展示，看表的关键处如下：



Name	fourcc	Byte 0	Byte 1	Byte 2	Byte 3
V4L2_PIX_FORMAT_GREY	GREY				
V4L2_PIX_FORMAT_YUYV	YUYV				
V4L2_PIX_FORMAT_UYVY	UYVY				
V4L2_PIX_FORMAT_Y41P	Y41P				

也有几中平面型的 YUV 格式在用，但把它们全画出来并没有什么大的帮助，所以我们只是在下面举一下例子，常用”YUV 4:2:2”(V4L2_PIX_FMT_YUV422, fourcc422P)格式使用三组阵列，一幅 4X4 的图片将如下表示：

Y plane:				

U plane:	<div></div>	<div></div>	<div></div>	<div></div>
	<div></div>	<div></div>	<div></div>	<div></div>
V plane:	<div></div>	<div></div>	<div></div>	<div></div>
	<div></div>	<div></div>	<div></div>	<div></div>

对于 Bayer 格式, YUV 4:2:2 每隔一个 Y 值有一个 U 和一个 V 值, 展示图像需要以内插值替换的丢失的值。其他的平面 YUV 格式有:

V4L2_PIX_FMT_YUV420: YUV 4:2:0 格式, 每四个 Y 值才有一个 U 值一个 V 值. U 和 V 都要在水平和垂直两个方向上都以内插值替换. 平面是以 Y-U-V 的顺序存储的, 与上面的例子一致.

V4L2_PIX_FMT_YVU420: 与 YUV 4:2:0 格式类似, 只是 U, V 值调换了位置.

V4L2_PIX_FMT_YUV410: 每 16 个 Y 值才有一个 U 值和 V 值. 阵列的顺序是 Y-U-V.

V4L2_PIX_FMT_YVU410: 每 16 个 Y 值才有一个 U 值和 V 值. 阵列的顺序是 Y-V-U.

还存在一些其他的 YUV 格式, 但他们的使用极用少。想要了解的话请看[这里](#)。

其他格式[p=21, null, left]还有一些可能对驱动有用的格式如下:

V4L2_PIX_FMT_JPEG: 一种定义模糊的 JPEG 流; 更多信息请看[这里](#)。

V4L2_PIX_FMT_MPEG: MPEG 流. 还有一些 MPEG 流格式的变种; 未来的文章中将讨论流的控制。

[p=21, null, left]还有一些其他的混杂的格式, 其中一些还是有专利保护的;[这里](#) 有一张列表. 格式的描述[p=21, null, left]现在我们已经了解了颜色的格式, 下面将要看看下 V4L2 API 中是如何描述图像格式的了。这里的主要的结构体是 struct v4l2_pix_format (定义于 <linux/videodev2.h>), 它包含如下字段:

__u32 width: 图片宽度, 以像素为单位.

__u32 height: 图片高度, 以像素为单位.

__u32 pixelformat: 描述图片格式的四字符码.

enum v4l2_field field: 很多图片的源会使数据交错 - 先传输奇数行, 然后是偶到行. 真正的摄像头设备是不会做数据的交错的。V4L2 API 允许应用使用很多种方式交错字段. 常用的值为 V4L2_FIELD_NONE (字段不交错), V4L2_FIELD_TOP (只交错顶部字面), 或 V4L2_FIELD_ANY (无所谓). 详情见[这里](#).

__u32 bytesperline: 相临扫描行之间的字节数. 这包括各种设备可能会加入的填充字节. 对于平面格式, 这个值描述的是最大的 (Y) 平面.

__u32 sizeimage: 存储图片所需的缓冲区的大小.

enum v4l2_colorspace colorspace: 使用的色域.

加到一起, 这些参数以合理而完整的方式描述了视频数据缓冲区。应用可以填充 v4l2_pix_format 请求用户空间开发者所能想到的几乎任何格式. 然而, 在驱动层面上, 驱动开发者则要限制在硬件所能支持的格式上. 所以每一个 V4L2 应用都必须经历一个与驱动协定的过程, 以便于使用一个硬件支持并且能满足应用需要的图像格式。下一期文章, 我

们将从驱动角度，描述这种协定是什么进行的。

v4l2 驱动编写篇第五 B--格式的协定

这是不定期发布的关于写视频驱动程序的 LWN 系统文章的一篇续篇。[介绍篇](#)包含了对整个系统的描述，并且包含对本篇的上一篇的链接，在上一集，我们关注了 V4L2 API 是如何描述视频格式的：图片的大小，和像素在其内部的表示方式。这篇文章将完成对这个问题的讨论，它将描述如就硬件所支持的的实际视频格与应用达到协议。

如我们在上一篇中所见，在存储器中表示图像有很多方法。市场几乎找不到可以处理所有 V4L2 所理解的视频格式的设备。驱动不应支持底层硬件不懂的视频格式。实际上在内核中进行格式的转换是令人难以接受的。所以驱动必须可以应用选择一个硬件可以支持的格式。第一步就是简单的允许应用查询所支持的格式。VIDIOC_ENUM_FMT ioctl()就是为此目的而提供的。在驱动内部这个调用会转化为这样一个回调函数（如果查询的是视频捕获设备）。

```
int (*vidioc_enum_fmt_cap)(struct file *file, void *private_data,
                          struct v4l2_fmtdesc *f);
```

这个回返调函数要求视频捕获设备描述其支持的格式。应用会传递一个 v4l2_fmtdesc 结构体：

```
struct v4l2_fmtdesc
{
    __u32          index;
    enum v4l2_buf_type type;
    __u32          flags;
    __u8           description[32];
    __u32          pixelformat;
    __u32          reserved[4];
};
```

应用会设置 index 和 type 字段。index 是用来确定格式的一个简单的整型数；与其他 V4L2 所使用的索引(indexes)一样，这个也是从 0 开始递增，至最大允许的值为止。应用可以通过一直递增索引值(index)直到返回 EINVAL 的方式枚举所有支持的格式。type 字段描述的是数据流类型；对于视频捕获设备来说（摄像头或调谐器就是 V4L2_BUF_TYPE_VIDEO_CAPTURE。

如果 index 对就某个支持的格式，驱动应该填写结构体的其他字段。pixelformat 字段应该是描述视频表现方式的 fourcc 编码，而 description 是对这个格式的一种简短的字符串描述。flags 字段只定义了一个值即 V4L2_FMT_FLAG_COMPRESSED，它表示是一个压缩了的视频格式。上述的函数是视频捕获函数；只有当 type 字段的是值是 V4L2_BUF_TYPE_VIDEO_CAPTURE 时才会调用。VIDIOC_ENUM_FMT 调用将根据 type 字段的值的不同解释成不同的回调函数。

```
/* V4L2_BUF_TYPE_VIDEO_OUTPUT */
```

```
int (*vidioc_enum_fmt_video_output)(file, private_data, f);
```

```
/* V4L2_BUF_TYPE_VIDEO_OVERLAY */
```

```
int (*vidioc_enum_fmt_overlay)(file, private_data, f);
```

```
/* V4L2_BUF_TYPE_VBI_CAPTURE */
int (*vidioc_enum_fmt_vbi)(file, private_data, f);

/* V4L2_BUF_TYPE_SLICED_VBI_CAPTURE */
int (*vidioc_enum_fmt_vbi_capture)(file, private_data, f);

/* V4L2_BUF_TYPE_VBI_OUTPUT */
/* V4L2_BUF_TYPE_SLICED_VBI_OUTPUT */
int (*vidioc_enum_fmt_vbi_output)(file, private_data, f);

/* V4L2_BUF_TYPE_VIDEO_PRIVATE */
int (*vidioc_enum_fmt_type_private)(file, private_data, f);
```

参数类似对于所有的调用都是一样。对于以 V4L2_BUF_TYPE_PRIVATE 开头的解码器，驱动可以支持特殊的缓冲类型是没有意义的。但在应用端却需要一个清楚的认识。对这篇文章的目的而言，我们更加关心的是视频捕获和输出设备；其他的视频设备我们会在将来某期的文章中讲述。

应用可以通过调用 VIDIOC_G_FMT 知道硬件现在的配置如何。这种情况下传递的参数是一个 v4l2_format 结构体：

```
struct v4l2_format
{
    enum v4l2_buf_type type;
    union
    {
        struct v4l2_pix_format      pix;
        struct v4l2_window          win;
        struct v4l2_vbi_format      vbi;
        struct v4l2_sliced_vbi_format sliced;
        __u8      raw_data[200];
    } fmt;
};
```

同样，type 描述的是缓冲区类型；V4L2 层会根据 type 的不同，将调用解释成不同的驱动的回调函数。对于视频捕获设备而言，这个回调函数就是：

```
int (*vidioc_g_fmt_cap)(struct file *file, void *private_data,
                        struct v4l2_format *f);
```

对于视频捕获（和输出）设备，联合体中 pix 字段是我们关注的重点。这是我们在上一期中见过的 v4l2_pix_format 结构体；驱动应该用现在的硬件设置填充那个结构体并且返回。这个调用通常不会失败，除非是硬件出现了非常严重的问题。

其他的回调函数还有：

```
int (*vidioc_s_fmt_overlay)(file, private_data, f);
int (*vidioc_s_fmt_video_output)(file, private_data, f);
```

```
int (*vidioc_s_fmt_vbi)(file, private_data, f);
int (*vidioc_s_fmt_vbi_output)(file, private_data, f);
int (*vidioc_s_fmt_vbi_capture)(file, private_data, f);
int (*vidioc_s_fmt_type_private)(file, private_data, f);
```

vidioc_s_fmt_video_output()与捕获接口一样使用相同的方式使用同一个 pix 字段。

多数应用都想最终对硬件进行配置以使其为应用提供一种符合其目的的格式。改变视频格有两个接口。第一个是 VIDIOC_TRY_FMT 调用,它在 V4L2 驱动中转化为下面的回调函数:

```
int (*vidioc_try_fmt_cap)(struct file *file, void *private_data,
                          struct v4l2_format *f);
int (*vidioc_try_fmt_video_output)(struct file *file, void *private_data,
                                   struct v4l2_format *f);

/* And so on for the other buffer types */
```

要处理这个调用,驱动会查看请求的视频格式,然后断定硬件是否支持这个格式。如果应用请求的格式是不能支持的,就会返回-EINVAL。所以,例如,一个描述了一个不支持的 fourcc 编码或者请求了一个隔行扫描的视频,而设备只支持逐行扫描的就会失败。在另一方面,驱动可以调整 size 字段,以与硬件支持的图像大小相适应。普遍的做法是可能的话就将大小调小。所以一个只能处理 VGA 分辨率的设备驱动会根据情况相应地调整 width 和 height 参数而成功返回。v4l2_format 结构体会在调用后复制给用户空间;驱动应该更新这个结构体以反映改变的参数,这样应用才可以知道它真正得到就是什么。

VIDIOC_TRY_FMT 这个处理对于驱动来说是可选的,但是不推荐忽略这个功能。如果提供了的话,这个函数可以在任何时候调用,甚至时设备正在工作的时候。它不可以对实质上的硬件参数做任何改变,只是让应用知道都可以做什么的一种方式。

如果应用要真正的改变硬件的格式,它使用 VIDIOC_S_FMT 调用,它以下面的方式到达驱动:

```
int (*vidioc_s_fmt_cap)(struct file *file, void *private_data,
                        struct v4l2_format *f);
int (*vidioc_s_fmt_video_output)(struct file *file, void *private_data,
                                 struct v4l2_format *f);
```

与 VIDIOC_TRY_FMT 不同,这个调用是不能随时调用的。如果硬件正在工作,或者有流缓冲器已经开辟了(未来另一篇文章的),改变格式会带来无尽的麻烦。想想会发生什么,比如说,一个新的格式比现在使用的缓冲区大的时候。所以驱动要一直保证硬件是空闲的,如果不空闲就对请求返回失败(-EBUSY)。

格式的改变应该是原子的 – 它或者改变所有的参数以实现请求否则就必须一个也不改变。同样,驱动在必要时是可以改变图像的大小的,通常的回调函数格式与下面的差不多:

```
int my_s_fmt_cap(struct file *file, void *private,
                 struct v4l2_format *f)
{
    struct mydev *dev = (struct mydev *) private;
    int ret;

    if (hardware_busy(mydev))
```

```
    return -EBUSY;
ret = my_try_fmt_cap(file, private, f);
if (ret != 0)
    return ret;
return tweak_hardware(mydev, &f->fmt.pix);
}
```

使用 `VIDIOC_TRY_FMT` 句柄可以避免代码重写而且可以避免任何没有先实现那个函数的借口。如果“try”函数成功返回，结果格式就已知并且可以直接编程进硬件。

还有很多其他调用也用影响视频 I/O 的完成方式。将来的文章将会讨论他们中的一部分。支持设置格式就足以让应用开始传输图像了,而且那也这个结构体的最终目的.所以下一篇文章,（希望会在这次之后的时间不会太久）我们会来关注对视频数据的读和写的支持。

v4l2 驱动编写篇第六 A--基本的帧输入输出

基本的帧输入输出

关于视频驱动的这一系列文章已经更新了好几期，但是我们还没有传输过一帧的视频数据。虽然在这点上，我们已经了解了足够多的关于格式协定方面的细节，我们可以看一下视频帧是如何在应用和设备之间传输的了。

V4L2 API 定义了三种不同的传输视频帧的方法，现在有两种是可以实现的：

`read()` 和 `write()` 系统调用这种普通的方法。根据硬件和驱动的不同,这种方法可能会非常慢-但也不是一定会那样。

将帧直接以视频流的方法送到应用可以访问的缓冲区。视频流这际上是传输视频数据的最有效的方法;这种接口还允许在图像帧中附带一些其他信息。视频流的方法有两种变种,其分别在于缓冲的开辟是在用户空间还是内核空间。

Video4Linux2 API 规范提供一种异频的输入输出机制用于帧的传输。然而这种模式还没有实现，因此不能使用。

这一篇将关注的是简单的 `read()`和 `write()`接口，视频流的方式将在下一期来讲解。

`read()` 和 `write()`

Video4Linux2 规范并没有规定要实现 `read()`和 `write()`,然而很多简单的应用希望这种系统调用可用，所以可能的话，驱动的作者应该使之工作。如果驱动没有实现这些系统调用，它应该在保证 `V4L2_CAP_READWRITE` 置位，来回应 `VIDIOC_QUERYCAP` 调用。然而以笔者的经验，多数的应用在使用调用之前，根本就不会是费心查看调用是否可用。

驱动的 `read()`和/或 `write()`方法必须存在相关的 `video_device` 结构中的 `fops` 字段里。注意：V4L2 规范要求实现这些方法，从而也提供 `poll()`操作。

在一下视频捕获设备上实现 `read()`操作是非常直接的：驱动告诉硬件开始捕获帧，发送一帧到用户空间缓冲，然后关停硬件并返回。如果可能的话，驱动应该安排 DMA 操作直接将数据传送到目的缓冲区，但这种方式只有在控制器可以处理分散/聚集 I/O 的时候可能。否则，驱动应该在内核里启用帧缓冲区。同样，写操作也是尽可能直接传到设备，否则启用帧缓冲

区。

不那么简单的操作也是可以的。例如笔者的“cafe”驱动会在 `read()` 操作后让摄像头控制器工作在一种投机的状态,在一秒钟的下一部分,摄像头中的后续帧将会存储在内核的缓冲区中,如果应用发出了另一个读的调用,它将会更快的反应,无续再次启动硬件。经过一定数目的帧都没有读的话,控制器就会被放回空闲的状态。同理,写操作时,也会延时几十毫秒,意在帮助应用与硬件帧同步。

流参数

`VIDIOC_G_PARM` 和 `VIDIOC_S_PARM` `ioctl()` 系统调用会调整一些 `read()`, `write()` 专用的参数,其中一些更加普便。它看起来像是一个设置没有明显归属的杂项的调用。我们在这里就了解一下,虽然有些参数同时会影响流输入输出的参数。

支持这些调用的 Video4Linux2 驱动提供如下两个方法:

```
int (*vidioc_g_parm) (struct file *file, void *private_data,
                      struct v4l2_streamparm *parms);
int (*vidioc_s_parm) (struct file *file, void *private_data,
                      struct v4l2_streamparm *parms);
```

`v4l2_streamparm` 结构包含下面的联合,这一系列文章的读者到现在应该对它们已经很熟悉了。

```
struct v4l2_streamparm
{
    enum v4l2_buf_type type;
    union
    {
        struct v4l2_captureparm    capture;
        struct v4l2_outputparm     output;
        __u8 raw_data[200];
    } parm;
};
```

`type` 字段描述的是在涉及的操作的类型。对于视频捕获设备,应该为 `V4L2_BUF_TYPE_VIDEO_CAPTURE`。对于输出设备应该为 `V4L2_BUF_TYPE_VIDEO_OUTPUT`。它的值也可以是 `V4L2_BUF_TYPE_PRIVATE`,在这种情况下,`raw_data` 字段用来传递一些私有的,不可移植的,甚至是不鼓励的数据给内核。

对于捕获设备而言, `parm.capture` 字段是要关注的内容,这个结构体如下:

```
struct v4l2_captureparm
{
    __u32 capability;
    __u32 capturemode;
    struct v4l2_fract timeperframe;
```

```

    __u32          extendedmode;
    __u32          readbuffers;
    __u32          reserved[4];
};

```

capability 字段是一组功能标签。目前为止已经定义的只有一个 V4L2_CAP_TIMEPERFRAME, 它代表可以改变帧频率。capturemode 也是一个只定义了一个标签的字段: V4L2_MODE_HIGHQUALITY, 这个标签意在使硬件在高清模式下工作, 实现单帧的捕获。这个模式可以做出任何的牺牲 (包括支持的格式, 曝光时间等), 以达到设备可以处理的最佳图片质量。

timeperframe 字段用于指定想要使用的帧频率, 它又是一个结构体:

```

struct v4l2_fract {
    __u32  numerator;
    __u32  denominator;
};

```

numerator 和 denominator 所描述的系数给出的是成功的帧之间的时间间隔。另一个驱动相关的字段是: extendedmode, 它在 API 中没有明确的意义。readbuffers 字段是 read() 操作被调用时内核应为输入的帧准备的缓冲区数量。

对于输出设备, 其结构体如下:

```

struct v4l2_outputparm
{
    __u32          capability;
    __u32          outputmode;
    struct v4l2_fract timeperframe;
    __u32          extendedmode;
    __u32          writebuffers;
    __u32          reserved[4];
};

```

capability, timeperframe, 和 extendedmode 字段与捕获设备中的意义相同。outputmode 和 writebuffers 与 capturemode 和 readbuffers 对应相同。

当应用想要查询现在的参数时, 它会发出一个 VIDIOC_G_PARM 调用, 因而调用驱动的 vidioc_g_parm() 方法。驱动应该提供现在的设置, 不用的话确保将 extendedmode 设为 0, 并且把 reserved 字段永远设为 0。

设置参数将调用 vidioc_s_parm()。在这种情况下, 驱动将参数设为与应用所提供的参数尽可能近的值, 并调整 v4l2_streamparm 结构体以反应现行使用的值。例如, 应用可以会请求一个比硬件所能提供的更高的帧频率, 在这种情况下, 帧频率会设为最高, 并把 imeperframe 设为这个最高的帧频率。

如果应用提供 timeperframe 为 0, the driver should program the nominal frame rate associated

with the current video norm. (这句不懂)。如果 readbuffers 或 writebuffers 是 0，驱动应返回现行设置而不是删除缓冲区。

到现在为止，我们已经可以写一个支持 read()和 write()方式帧传输的简单的驱动了。然而多数正式的应用要使用流输入输出方式：流的方式使高性能变得更简单；帧可以打包带上附加信息如帧序号，请继续关注本系列的下篇文章，我们将讨论如何在视频驱动中实现流 API。

v4l2 驱动编写篇第六 B--流输入输出

在本系列文章的上一期中，我们讨论了如何通过 read()和 write()的方式实现视频帧的传输，这样的实现可以完成基本的工作，却并不是普遍上用来实现视频输入输出大家偏爱的方法。为了实现最高的性能和最好的信息传输，视频驱动应该支持 V4L2 流输入输出。

使用 read()和 write()方法，每一帧都要通过 I/O 操作在用户和内核空间之间拷贝数据。然而，当使用流输入输出的方式时，这种情况就不会发生。替代的方案是用户与内核空间之间交换缓冲区的指针，这些缓冲区将被映射到应用的地址空间，这也就使零帧复制数成为可能。有两种流输入输出缓冲区：

- * 内存映射缓冲区 (memory-mapped buffers) (type V4L2_MEMORY_MMAP) 是在内核空间开辟的;应用通过 themmap()系统调用将其映射到地址空间。这些缓冲区可以是大而相邻 DMA 缓冲区，通过 vmalloc()创建的虚拟缓冲区，或者（如果硬件支持的话）直接在设备的输入输出存储器中开辟的。

- * 用户空间缓冲区 (V4L2_MEMORY_USERPTR) 是在用户空间的应用中开辟的。很明显，在这种情况下，是不需要 mmap()调用的，但驱动在有效地支持用户空间缓冲区上的工作将会更难一些。

注意：驱动支持流输入输出的方式并非必需，即便做了实现，驱动也不必两种缓冲区类型都做处理。一个灵活的驱动可以支持更多的应用。在实际应用中，似乎多数应用都是使用内存映射缓冲区的。同时使用两种缓冲区是不可能的。

现在，我们将要探索一下支持流输入输出的众多而邈远的细节。任何 Video4Linux2 驱动的作者都要了解这部分 API。然而值得指出的是，有一个更高层次的 API，它能够帮助驱动作者写流驱动。当底层设备可以支持分散\聚集 I/O 的时候，这一层（称为 video-buf）可以使事情变得容易。关于 video-buf API 我们将在将来的某期讨论。

支持流输入输出的驱动应该通知应用这一事实，方法是在 vidioc_querycap()方法中设置 V4L2_CAP_STREAMING 标签。注意：并没有办法来描述支持的是哪一种缓冲区，那是后话。

v4l2_buffer 结构体

当流输入输出是有效的，帧是以 v4l2_buffer 的形式在应用和驱动之间传输的。这个结构体是一个复杂的——畜生？——，要用很长的时间才能描述完。一个很好的开头要大家知道一个缓冲区可以有三种基本的状态：

* 在驱动的传入队列中.如果驱动不用它做什么有用事的话,应用就可以把缓冲区放在这个队列里。对于一个视频捕获设备还讲,传入队列中的缓冲区是空的,等待驱动向其内填入视频数据。对于输入设备来讲,这些缓冲区内是要送入设备的帧数据。

* 在驱动的传出队列中,这些缓冲区已经经过驱动的处理,正等待应用来认领。对于捕获设备而言,传出缓冲区内是新的帧数据;对输出设备而言,这个缓冲区是空的。

* 不在上述两个队列里。在这种状态时,缓冲区是由用户空间拥有的,驱动无法访问。这是应用可以对缓冲区进行操作的时间。我们称其为用户空间状态。

这些状态和造成他们之间传输的操作都放在一起,在下图中示出:

实际上的 v4l2_buffer 结构体如下:

```
struct v4l2_buffer
{
    __u32                index;
    enum v4l2_buf_type    type;
    __u32                bytesused;
    __u32                flags;
    enum v4l2_field        field;
    struct timeval        timestamp;
    struct v4l2_timecode    timecode;
    __u32                sequence;

    /* memory location */
    enum v4l2_memory        memory;
    union {
        __u32                offset;
        unsigned long        userptr;
    } m;
    __u32                length;
    __u32                input;
    __u32                reserved;
};
```

index 字段是鉴别缓冲区的序号;它只在内存映射缓冲区中使用。与其它可以在 V4L2 接口中枚举的对相一样,内存映射缓冲区的 **index** 从 0 开始,依次递增。**type** 字段描述的是缓冲区的类型,通常是 V4L2_BUF_TYPE_VIDEO_CAPTURE 或 V4L2_BUF_TYPE_VIDEO_OUTPUT。

缓冲区的大小是论长度给定的,单位为 byte。缓冲区中的图像数据大小可以在 **bytesused** 字段中找到。很明显, **bytesused** ≤ **length**。对于捕获设备而言,驱动会设置 **bytesused**; 对输出设备而言,应用必须设置这个字段。

field 字段描述的是图像存在缓冲区的那一个区域。这些区域在这系统文章中的 part5a 中可

以找到。

timestamp(时间戳) 字段,对于输入设备来说,代表帧捕获的时间.对输出设备来说,在没有到达时间戳所代表的时间前, 驱动不可以把帧发送出去;时间戳值为 0 代表越快越好。驱动会把时间戳设为帧的第一个字节传送到设备的时间, 或者说是驱动所能达到的最接近的时间。**timecode** 字段可以用来存放时间编码, 对于视频编辑类的应用是非常有用的。关于时间编码详情见表 。

驱动对传过设备的帧维护了一个递增的计数; 每一帧传送时,它都会在 **sequence** 字段中存入现行序号。对于输入设备来讲, 应用可以观察这一字段来检测帧。

memory 字段表示的是缓冲是内存映射的还是用户空间的。对于内存映射的缓冲区, **m.offset** 描述的是缓冲区的位置. 规范将它描述为“从设备存储器开发的缓冲区偏移”,但其实质却是一个 **magic cookie**, 应用可以将其传给 **mmap()**,以确定那一个缓冲区被映射了。然而对于用户空间缓冲区而言, **m.userptr** 是缓冲区的用户空间地址。

input 字段可以用来快速切换捕获设备的输入 – 当然, 这要得设备支持帧与帧音的快速切换才来。**reserved** 字段应置 0。

最后, 还有几个标签定义:

* **V4L2_BUF_FLAG_MAPPED** 暗示缓冲区已映射到用户空间。它只应用于内存映射缓冲区。

* **V4L2_BUF_FLAG_QUEUED**: the buffer is in the driver's incoming queue.

* **V4L2_BUF_FLAG_DONE**: 缓冲区在驱动的传出队列。

* **V4L2_BUF_FLAG_KEYFRAME**: 缓冲区包含一个关键帧,它在压缩流中是非常有用的。
* **V4L2_BUF_FLAG_PFRAME** 和 **V4L2_BUF_FLAG_BFRAME** 也是应用于压缩流中;他们代表的是预测的或者说是差分的帧。

* **V4L2_BUF_FLAG_TIMECODE**: **timecode** 字段有效。

* **V4L2_BUF_FLAG_INPUT**: **input** 字段有效。

缓冲区设定

一旦流应用已经完成了基本的设置, 它将转去执行组织 I / O 缓冲区的任务。第一步就是使用 **VIDIOC_REQBUFS ioctl()** 来建立一组缓冲区, 它将由 **V4L2** 转换成对驱动的 **vidioc_reqbufs()**方法的调用。

```
int (*vidioc_reqbufs) (struct file *file, void *private_data,  
                      struct v4l2_requestbuffers *req);
```

我们要关注的所以内容都在 **v4l2_requestbuffers** 结构体中,如下所示:

```
struct v4l2_requestbuffers
```

```

{
    __u32                count;
    enum v4l2_buf_type    type;
    enum v4l2_memory      memory;
    __u32                reserved[2];
};

```

`type` 字段描述的是完成的 I/O 操作的类型。通常它的值要么是视频获得设备的 `V4L2_BUF_TYPE_VIDEO_CAPTURE`，要么是输出设备的 `V4L2_BUF_TYPE_VIDEO_OUTPUT`。也有其它的类型，但在这里我们不予讨论。

如果应用想要使用内存映射的缓冲区，它将会把 `memory` 字段置为 `V4L2_MEMORY_MMAP`，`count` 置为它想要使用的缓冲区的数目。如果驱动不支持内存映射，它就该返回 `-EINVAL`。否则它将在内部开辟请求的缓冲区并返回 0。返回之后，应用就会认为缓冲区是存在的，所以任何可以失败的任务都在这个阶段进行处理（比如说内存开辟）。

注意：驱动并不一定要开辟与请求的一样数目的缓冲区。在很多情况下，只有一个缓冲区数的最小值有意义。如果应用请求的比最小值小，它可以能实际须要的要多一些。以笔者的经验，`mplayer` 要用两个缓冲区，如果用户空间速度慢下来的话，这将很容易超支（因而丢失帧）。通过强制一个大一点的最小缓冲区数（通过模块参数可以调整），`cafe_ccic` 驱动可以使流输入输出通道更加强壮。`count` 字段应设为方法返回前实际开辟的缓冲区数。

应用可以通过设置 `count` 字段为 0 的方式来释放掉所有已存在的缓冲区。在这种情况下，驱动必须在释放缓冲前停止所有的 DMA 操作，否则会发生非常严重的事情。如果缓冲区已映射到用户空间，则释放缓冲区是不可能的。

相反，如果用的是用户空间缓冲区，则有意义的字段只有缓冲区的 `type` 和只有 `V4L2_MEMORY_USERPTR` 这个值可用的 `memory` 字段。应用不需要指定它想用的缓冲区的数目。因为内存是在用户空间开辟的，驱动无须操心。如果驱动支持用户空间缓冲区，它只须注意应用会使用这一特性，返回 0 就可以了，否则通常的 `-EINVAL` 返回值会被调用到。

`VIDIOC_REQBUFS` 命令是应用得知驱动支持的流输入输出缓冲区类型的唯一方法。
将缓冲区映射到用户空间

如果使用了用户空间，在应用向传入队列放置缓冲区之前，驱动看不到任何缓冲区相关的调用。内存映射缓冲区需要更多的设置。应用通常会查看每一个开辟了的缓冲区，并将其映射到地址空间。第一站是 `VIDIOC_QUERYBUF` 命令，它将转换成驱动中的 `vidioc_querybuf()` 方法：

```

int (*vidioc_querybuf)(struct file *file, void *private_data,
                       struct v4l2_buffer *buf);

```

进入这个方法时，`buf` 字段中要设置的字段有 `type`（在缓冲区开辟时，它将被检查是否与给定的类型相同）和 `index`，它们可以确定一个特定的缓冲区。驱动要保证 `index` 有意义，并添充

buf 中的其余字段。通常来说，驱动内部存储着一个 v4l2_buffer 结构体的数组，所以 vidioc_querybuf() 方法的核心只是一个结构体的赋值。

应用访问内存映射缓冲区的唯一方法就是将其映射到它们的地址空间，所以 vidioc_querybuf() 调用后面通常会跟着一个驱动的 mmap() 方法 - 这个方法，大家要记住，是存储在相关设备的 video_device 结构体中的 fops 字段中的。设备如何处理 mmap() 将依赖于内核中缓冲区是如何设置的。如果缓冲区可以在 remap_pfn_range() 或 remap_vmalloc_range() 之前映射，那就应该在这个时间来做。对于内核空间的缓冲区，页也可以在页错误时通过常规的使用 nopage() 方法的方式单独被映射，对于需要的人来说，在 Linux Device Drivers 可以找到一个关于 handlingmmap() 的一个很好的讨论。

mmap() 被调用时，传递的 VMA 结构应该含有 vm_pgoff 字段中的某个缓冲区的地址 - 当然是经过 PAGE_SHIFT 右移过的。特别是，它应该是你的驱动对于 VIDIOC_QUERYBUF 调用返回值的偏移。请您遍历缓冲区列表，并确保传入地址匹配其中之一。视频驱动程序不应该是一个可以让恶意程序映射内存的任意区域手段。

你所提供的偏移值可几乎所有的东西。有些驱动只是返回(index<<PAGE_SHIFT)，意思是说传入的 vm_pgoff 字段应该正好是缓冲区索引。有一件事你不可以做的就是把缓冲区的内核实际地址存储到 offset 字段，把内核地址泄露给用户空间永远也不是一个好主意。

当用户空间映射缓冲区时，驱动应该在相关的 v4l2_buffer 结构体中调置 V4L2_BUF_FLAG_MAPPED 标签。它也必须设定 open() 和 close() VMA 操作，这样它才可以跟踪映射了缓冲区的进程数。只要缓冲区在哪里映射了，它就不可以在内核里释放掉。如果一个或多个缓冲区的映射计算降为 0，驱动就应该停止正在进行的输入输出，因为没有进程要用它。

流输入输出

到现在为止，我们已经看了很多设置，却没有传输过一帧的数据，我们离这一步已经很近了，但在些之前还有一个步骤要做。当应用通过 VIDIOC_REQBUFS 获得了缓冲区后，那个缓冲区都处于用户空间状态。如果他们是用户空间缓冲区，他们甚至还并不真的存在。在应用可以开始流的输入输出之前，它必须至少将一个缓冲区放到驱动传入队列，对出输出设备，那些缓冲区当然还要先添完有效的帧数据。

要把一个缓冲区加入队列，应用首先要发出一个 VIDIOC_QBUF ioctl() 调用，这个调用 V4L2 会映射为对驱动的 vidioc_qbuf() 方法的调用。

```
int (*vidioc_qbuf) (struct file *file, void *private_data,  
                   struct v4l2_buffer *buf);
```

对于内存映射缓冲而言，还是那样，只有 buf 的 type 和 index 字段有效。驱动只能进行一些明显的检查(type 和 index 有意义,缓冲区还没有在驱动的队列里，缓冲区已映射等。),把缓冲区放在传入队列里 (设置 V4L2_BUF_FLAG_QUEUED 标签),并返回。

在这点上，用户空间缓冲区可能会更加复杂，因为驱动可能从来没看过缓冲区的样子。使用

这个方法时，允许应用在每次向队列传入缓冲区时，传递不同的地址，所以驱动不能提前做任何的设置。如果你的驱动通过内核空间缓冲区将帧送回，它只须记录一下应用提供的用户空间地址就可以了。然而如果你正尝试将通过 DMA 直接将数据传送到用户空间，那将会非常的具有挑战性。

要想把数据直接传送到用户空间，驱动必须先 `fault in` 缓冲区的所有的页，并将其锁定。`get_user_pages()` 可以做这件事。注意这个函数会在开辟很大的内存空间和硬盘 I/O-它可能会卡住很长时间。你得注意要保证重要的驱动功能不能在 `get_user_pages()` 时停止，因为它可能停止很长时间等待许多视频帧通过。

下面就是要告诉设备把图像数据传到用户空间缓冲区（或是相反的方向）了。缓冲区在物理上不是相邻的，相反，它会被分散成很多很多单独的 4096 字节的页（在大部分架构上如此）。很明显，设备得可以实现分散/聚集 DMA 操作才行。如果设备立即传输一个完整的视频帧，它就需要接受一个包含很多页的分散列表（scatterlist）。一个 16 位格式的 VGA 解决方案的图像需要 150 个页，随着图像大小的增加，分散列表的大小也会增加。V4L2 规范说：

如果硬件需要，驱动要与物理内存交换内存页，以产生相邻的内存区。这对内核子系统的虚拟内存中的应用来说是透明的。

然而，笔者却不推荐驱动作者尝试这种深层的虚拟内存技巧。有一个更有前途的方法就是要求用户空间缓冲区分配成大的 tlb 页，但是现在的驱动不那么做。

如果你的设备传输的是小图像（如 USB 摄像头），直接从 DMA 到用户空间的设定就简单些。在任何情况下，不得不支持到用户空间缓冲区的直接 I/O 时，驱动作者都应该（1）确定的确值得这么大的麻烦，因为应用更趋向于使用内存映射缓冲区。（2）使用 `video_buf` 库，它可以帮你解决一些痛苦的难题。

一旦流输入输出开始，驱动就要从它的传入队列里抓取缓冲区，让设备更快地实现转送请求，然后把缓冲区移动到传出队列。转输开始时，缓冲区标签也要相应调整。像序号和时间戳这样的字段必需在这个时候添充。最后，应用会在传出队列中认领缓冲区，让它变回为用户空间状态。这是 `VIDIOC_DQBUF` 的工作，它最终变为如下调用：

```
int (*vidioc_dqbuf) (struct file *file, void *private_data,  
                    struct v4l2_buffer *buf);
```

这里，驱动会从传出队列中移除第一个缓冲区，把相关的信息存入 `buf`。通常，传出队列是空的，这个调用会处于阻塞状态直到有缓冲区可用。然而 V4L2 是用来处理非阻塞 I/O 的，所以如果视频设备是以 `O_NONBLOCK` 方式打开的，在队列为空的情况下驱动就该返回 `-EAGAIN`。当然，这个要求也暗示驱动必须为流 I/O 支持 `poll()`；

剩下最后的一个步骤实际上就是告诉设备开始流输入输出操作。这个任务的 `Video4Linux2` 驱动方法是：

```
int (*vidioc_streamon) (struct file *file, void *private_data,
```



```
enum v4l2_buf_type type);  
int (*vidioc_streamoff)(struct file *file, void *private_data,  
enum v4l2_buf_type type);
```

对 `vidioc_streamon()` 的调用应该在检查类型有意义之后让设备开始工作。如查需要的话，驱动可以请求等传入队列中有一定数目的缓冲区后再开始流的传输。

当应用关闭时，它应发出一个对 `vidioc_streamoff()` 的调用，这个调用要停止设备。驱动还应从传入还传出队列是移除所有的缓冲区，使它们都处于用户空间状态。当然，驱动必须准备好，应用可能在没有停流传输的情况下关闭设备。

v4l2 驱动编写篇第七--控制方法

刚刚完成了这一系列文章的第六部分，我们现在知道如何设置视频设备，并来回传输帧了。然而，有一个众所周知的事实，那就是用户永远也不会满意，不会满足于能从摄像头上看到视频，他们马上就会问我可不可以调参数啊？像亮度、对比度等等。这些参数可以在视频应用中调整，有时也的确会这样做，但是当硬件支持时，在硬件中进行调整有其优势。比如说亮度调整，如果不这样做的话，可能会丢失动态范围，但是基于硬件的调整可以完整保持传感器可以传递的动态范围。很明显，基于硬件的调整也可以让主机的处理器压力轻些。

现代的硬件中通常都可以在运行的时候调整很多参数。然而现在，在不同设备之间这些参数差别很大。简单的亮度调整可以直观地设置一个寄存器，也可以需要处理一个没有头绪的矩阵变换。最好是能尽可能多的把诸多细节对应用隐藏，但能隐藏到什么程序却受到很多限制。一个过于抽象的接口会使硬件的控制无法发挥其极限。

V4L2 的控制接口试图使事情尽可能地简单，同时还能完全发挥硬件的功能。它开始于定义一个标准控制名字的集合，包括 `V4L2_CID_BRIGHTNESS`, `V4L2_CID_CONTRAST`, `V4L2_CID_SATURATION`, 还有许多其他的。对于白平衡、水平，垂直镜像等特性，还提供了一些布尔型的控制。定义的控制 ID 值的完整列表请见 the V4L2 API spec。还有一个驱动程序特定的控制，但这些，显然，一般只能由专用的应用程序使用。私有的控制从 `V4L2_CID_PRIVATE_BASE` 开始往后都是。

一种典型的做法，V4L2 API 提供一种机制可以让应用可以枚举可用的控制操作。为此，他们要发出最终要实现为驱动中的 `vidioc_queryctrl()` 方法的 `ioctl()` 调用。

```
int (*vidioc_queryctrl)(struct file *file, void *private_data,  
struct v4l2_queryctrl *qc);
```

驱动通常会用所关心的控制信息来填充 `qc` 结构体，或是当控制操作不支持时返回 `EINVAL`，这个结构体有很多个字段：

```
struct v4l2_queryctrl  
{  
    __u32 id;  
    enum v4l2_ctrl_type type;
```

```

__u8          name[32];
__s32         minimum;
__s32         maximum;
__s32         step;
__s32         default_value;
__u32         flags;
__u32         reserved[2];
};

```

被查询的控制操作将会通过 `id` 传送。作为一个特殊的情况，应用可以通过设定 `V4L2_CTRL_FLAG_NEXT_CTRL` 位的方式传递控制 `id`。当这种情况发生时，驱动会返回关于下一个所支持的控制 `id` 的信息，这比应用给出的 `ID` 要高。无论在何种情况下，`id` 都应设为实际上被描述的控制操作的 `id`。

其他所有字段都由驱动设定，用来描述所选的控制操作。控制的数据类型在 `type` 字段中给定。这可以是 `V4L2_CTRL_TYPE_INTEGER`、`V4L2_CTRL_TYPE_BOOLEAN`、`V4L2_CTRL_TYPE_MENU` (针对一组固定的择项) 或 `V4L2_CTRL_TYPE_BUTTON` (针对一些设定时会忽略任何给出的值的控制操作)。`name` 字段用来描述控制操作;它可以在展现给用户的应用接口中使用。对于整型的控制来说(仅针对这种控制),`minimum` 和 `maximum` 描述的是控制所实现的值的范围,`step` 给出的是此范围下的粒度大小。`default_value` 顾名思义就是默认值 – 尽管他只对整型,布尔型和菜单控制适用。驱动只应在初始化时将控制参数设为默认。至于其它设备参数,他们应该从 `open()` 到 `close()` 保持不变。结果,`default_value` 很可能不是现在的控制参数值。

不可避免地,还有一组值进一步描述控制操作。`V4L2_CTRL_FLAG_DISABLED` 意为控制操作不可用,就用应该忽略它。`V4L2_CTRL_FLAG_GRABBED` 意为控制暂进不可变,可能会是因为另一个应用正在使用它。`V4L2_CTRL_FLAG_READ_ONLY` 意为可以查看,但不可以改变的控制操作。`V4L2_CTRL_FLAG_UPDATE` 意为调整这个参数可能会对其他控制操作造成影响。`V4L2_CTRL_FLAG_INACTIVE` 意为与当前设备配置无关的控制操作。`V4L2_CTRL_FLAG_SLIDER` 意在暗示应用在表现这个操作的时候可以使用类似于滚动条的接口。

应用可以只是查询几个特别编程过的控制操作,或者他们也想枚举整个集合。对后而来讲,他们会从开始 `V4L2_CID_BASE` 至 `V4L2_CID_LASTP1` 结束,过程中可能会用到 `V4L2_CTRL_FLAG_NEXT_CTRL` 标签。对于菜单型的诸多控制操作(`type=V4L2_CTRL_TYPE_MENU`)而言,应用很可能想要枚举可能的值,相关的回调函数是:

```

int (*vidioc_querymenu)(struct file *file, void *private_data,
                        struct v4l2_querymenu *qm);

```

`v4l2_querymenu` 结构体如下:

```

struct v4l2_querymenu
{

```



```
__u32      id;  
__u32      index;  
__u8       name[32];  
__u32      reserved;  
};
```

在输入中, `id` 是相关的菜单控制操作的 ID 值, `index` 某特定菜单 ID 值的索引值. 索引值从 0 开始, 依次递增到 `vidioc_queryctrl()` 返回的最大值. 驱动会填充菜单项的 `name` 字段. `reserved` 字段恒设为 0.

一旦应用知道了可用的控制操作, 它就很可以开始查询并改变其值. 这种情况下相关的结构体是:

```
struct v4l2_control  
{  
    __u32 id;  
    __s32 value;  
};
```

要查询某一给定控制操作, 应用应将 `id` 字段设为对应的控制的 ID, 并发出一个调用, 这个调最终实现为:

```
int (*vidioc_g_ctrl)(struct file *file, void *private_data,  
                     struct v4l2_control *ctrl);
```

驱动应将值设为当前控制的设定, 还要保证它知道这个特定的控制操作并在应用试图查询不存在的控制操作时返回 `EINVAL`, 试图访问按键控制时也应返回 `EINVAL`.

一个试图改变控制操作的请求实现为:

```
int (*vidioc_s_ctrl)(struct file *file, void *private_data,  
                     struct v4l2_control *ctrl);
```

驱动应该验证 `id`, 保证其值在允许的区间. 如果一切都没有问题的话, 就将新值写入硬件.

最后, 值得注意的是, 还有一个单独的扩展控制接口 也为 `V4L2` 所支持. 这个 API 是一组相当复杂的控制操作. 实际上, 它的主要应用就是 `MPEG` 编解码的参数. 扩展控制可以分门归类, 而且支持 64 位整型值. 其接口与常规的控制接口类似. 详见 API 规范.