

---

# **Video for Linux Two API Specification**

**Revision 0.24**

**Michael H Schimek**

[mschimek@gmx.at](mailto:mschimek@gmx.at)

**Bill Dirks**

**Hans Verkuil**

**Martin Rubli**

注:

译: 鱼在飞 (Ch1~4), [fishOnFly@outlook.com](mailto:fishOnFly@outlook.com)

[Tekkaman Ninja](#)

审校:

---

# 目 录

Chapter 1. 通用 API (Common API Elements)	1
1.1. 打开和关闭设备 (Opening and Closing Devices)	1
1.1.1. 设备命名(Device Naming)	1
1.1.2. 相关设备(Related Devices)	2
1.1.3. 并发打开(Multiple Opens)	3
1.1.4. 共享数据流(Shared Data Streams)	3
1.1.5. 函数(Functions)	3
1.2. 功能查询(Querying Capabilities)	3
1.3. 应用程序的优先级(Application Priority)	4
1.4. 视频输入/输出(Video Inputs and Outputs)	4
1.5. 音频输入/输出(Audio Inputs and Outputs)	5
1.6. 高频头和调制器 (Tuners and Modulators)	8
1.6.1. 高频头(Tuners)	8
1.6.2. 调制器 (Modulators)	9
1.6.3. 无线频率(Radio Frequency)	9
1.6.4. 卫星接收器(Satellite Receivers)	10
1.7. 视频标准(Video Standards)	10
1.8. 用户控制(User Controls)	14
1.9. 扩展控制(Extended Controls)	20
1.9.1. 介绍(Introduction)	20
1.9.2. 扩展类控制 API(The Extended Control API)	21
1.9.3. 扩展 control 值的枚举(Enumeration Extended Controls)	21
1.9.4. 创建 control 面板(Creating Control Panels)	22
1.9.5. MPEG Control Reference	22
1.9.6. 摄像头 control 参考(Camera Control Reference)	27
1.10. 数据格式(Data Formats)	27
1.10.1. 协商数据格式(Data Format Negotiation)	27

1.10.2. 枚举图片格式(Image Format Enumeration) .....	28
1.11. 图片修剪, 插入和缩放(Image Cropping, Insertion and Scalling) .....	28
1.11.1. 修剪结构(Cropping Structures) .....	29
1.11.2. 缩放调整(Scaling Adjustments) .....	30
1.11.3. 样例程序(Examples) .....	30
1.12. Streaming 参数(Streaming Parameters) .....	34
Notes .....	34
Chapter 2. 图像格式(Image Formats) .....	35
2.1. 标准图片格式(Standard Image Formats) .....	36
2.2. 色彩空间(Colorspaces) .....	37
2.3. 索引格式(Indexed Format) .....	40
2.4. RGB 格式(RGB Formats) .....	40
2.5. YUV 格式(YUV Formats) .....	43
2.6. 压缩格式(Compressed Formats) .....	44
2.7. 保留的格式标识符(Reserved Format Identifiers) .....	45
Chapter 3. 输入/输出(Input/Output) .....	46
3.1. 读和写(Read/Write) .....	46
3.2. 流式 I/O(内存映射缓冲区)(Streaming I/O(Memory Mapping)) .....	47
3.3. 流 I/O(用户空间缓冲区(用户指针))(Streaming I/O(User Pointers)) .....	50
3.4. 异步 I/O(Asynchronous I/O) .....	52
3.5. 缓冲(Buffers) .....	52
3.5.1. 时码(Timecodes) .....	55
3.6. field 次序(Field Order) .....	55
Notes .....	56
Chapter 4. 接口(Interface) .....	57
4.1. 视频采集接口(Video Capture Interface) .....	57
4.1.1. 查询设备属性(Querying Capabilities) .....	57
4.1.2. 附加功能(Supplemental Functions) .....	57
4.1.3. 图片格式协商(Image Format Negotiation) .....	58

---

4.1.4. 读取图片(Reading Images).....	58
4.2. Video Overlay Interface .....	58
4.2.1. 查询设备属性(Querying Capabilities) .....	59
4.2.2. 附加功能(Supplemental Functions).....	59
4.2.3. 设置(Setup).....	59
4.2.4. Overlay 窗口 (Overlay Window) .....	59
4.2.5. 使能 overlay (Enabling Overlay) .....	60
4.3. 视频输出接口(Video Output Interface) .....	61
4.3.1. 设备功能查询(Querying Capabilities) .....	61
4.3.2. 附加功能 (Supplemental Functions) .....	61
4.3.3. 图片格式协商 (Images Format Negotiation) .....	61
4.3.4. 写图片 (Writing Image) .....	61
4.4. 视频输出 overlay 接口 (Video Output Overlay Interface) .....	62
4.4.1. 设备功能查询 (Querying Capabilities) .....	62
4.4.2. 帧缓冲 (Framebuffer) .....	62
4.4.3. overlay 窗口和缩放.....	63
4.4.4. 使能 overlay (Enabling Overlay) .....	63
4.5. 编码接口 (Codec Interface) .....	64
4.6. 特效设备接口 (Effect Devices Interface) .....	64
4.7. 原始 VBI 数据接口 (Raw VBI Data Interface) .....	64
4.8. 分离 VBI 数据接口 (Sliced VBI Data Interface) .....	64
4.9. 图文电视接口 (Teltext Interface) .....	64
4.10. 无线电接口 (Radio Interface) .....	65
4.11. 无线数据广播系统 (Radio Data Interface) .....	65
Notes .....	65
Chapter 5. 驱动编写.....	66

---

## Chapter 1. 通用 API (Common API Elements)

编程以支持一个 V4L2 的设备包含以下步骤:

- .打开设备
- .修改设备的属性, 选择音频和视频输入, 视频的标准(编码格式?), 图片的亮度等
- .协商数据格式
- .协商输入/输出的方法
- .输入/输出循环
- .关闭设备

### 1.1. 打开和关闭设备 (Opening and Closing Devices)

#### 1.1.1. 设备命名(Device Naming)

V4L2 以模块的方式在内核中实现, 可以被管理员手动的加载或者是当设备第一次打开时被加载。驱动模块被挂在内核模块"videodev"下。在这份文档中, 提供了辅助函数和一个通用的应用程序接口规范。

每个驱动加载时都会用主设备号 81 和次设备号 0-255 来注册一个或多个设备节点。该如何分配次设备号给这些个 V4L2 设备, 完全取决于系统管理员, 这主要是为了解决设备间的冲突(为了区分不同的设备)<sup>1</sup>。当选定次设备号后, 模块将会以"\_nr"为后缀来命名特殊的设备文件名。举个例子来说, 像"video\_nr"就是代表视频采集设备的(在内核中表示为) /dev/video。数字则是和设备类型相关并基于次设备号的一个偏移<sup>2</sup>。

当驱动支持同一型号的多个设备时, 这时就可以赋予多个次设备号了, 并以逗号隔开:

```
> insmod mydriver.o video_nr=0,1 radio_nr=0,1
```

在/etc/modules.conf 文件中, 可能显示如下:

```
alias char-major-81-0 mydriver
alias char-major-81-1 mydriver
alias char-major-81-64 mydriver
```

①

```
Options mydriver video_nr=0,1 radio_nr=0,1
```

②

①当应用程序试图打开主设备号 81, 次设备号 0,1,64 的设备时会加载驱动"mydriver"

②注册前两个视频采集设备时是从主设备号 0 开始的, 接着是 1...一直到 63, 而音频设备则是从次设备号 64 开始的。

如果未显式给定次设备号, 那么模块将提供一个默认值。第四章将阐述不同型号设备的起始次设备号。显然次设备号必须是独一无二的。假如所请求的设备号正在使用时, 很明显起冲突的设备将不会被注册。

按照惯例, 系统管理员会根据主、次设备号在/dev 目录下创建不同功用的特殊设备文件。文件名

---

字则建议有别于列在第四章的 V4L2 设备名。

用命令 `mknod` 创建的字符设备，拥有特权操作并且这些设备是不能通过主、次设备号打开的。这就意味着应用程序是不能可靠的扫描已加载或者是安装的驱动的。用户必须键入设备名，或者应用程序会尝试默认的设备名以打开设备。

在设备文件系统（`devfs`）下，次设备号是被忽略的（明显的，在其下我们只要知道设备的功能，主设备号就提供这样的作用）。V4L2 驱动会自动的在 `/dev/v4l` 目录下，用默认的设备名创建所请求的设备文件。

### 1.1.2. 相关设备(Related Devices)

驱动通常都能支持许多相关的功能。有个例子说，视频采集，直接传输视频，视频间隔消隐信号采集都有着千丝万缕的联系，这是因为这些功能都共享着，起码在他们之间，相同的视频输入（视频采集头，`webcam`）和高频头（可以看作是音视频采集头）。

#### 注（V4L2 所提供的接口）：

1. 视频采集接口(video capture interface): 这种应用的设备可以是高频头或者摄像头。V4L2 的最初设计就是应用于这种功能的。
2. 视频输出接口(video output interface): 可以驱动计算机的外围视频图像设备--像可以输出电视信号格式的设备。
3. 直接传输视频接口(video overlay interface): 它的主要工作是把从视频采集设备采集过来的信号直接输出到输出设备之上，而不用经过系统的 CPU。
4. 视频间隔消隐信号接口(VBI interface): 它使应用可以访问传输消隐期的视频信号。
5. 收音机接口(radio interface): 可用来处理从 AM 或 FM 高频头设备接收来的音频流。

V4L 和早前的 V4L2 版本，视频采集和 `overlay` 是使用相同的设备名，相同的次设备号，但是却有别于 VBI（`Vertical Blanking Interval`）。实践表明，这种方法拥有众多问题<sup>3</sup>，更糟糕的是 V4L 的 `videodev` 模型是禁止同时多次打开设备的。

作为补救措施，当前的 V4L2 版本是通过特定的名字和次设备号放宽了设备类型的概念。为了保持和旧的应用程序的兼容性，驱动必须依旧注册不同的次设备号并制定一个默认的对应该功能给这个设备（`major` 代表一类设备，`minor` 这是用来给内核区分一类设备的不同实例或者功能。这样，`major` 代表一个班级，那么 `minor` 就是用来告诉校长这班级中的某个学生！）。但是，如果驱动支持相关功能，就必须在所注册的次设备号下实现这个功能（就是说为了支持老版应用像 `mjpg-server`，你必须为这类设备提供一个 `minor` 以支持默认的功能；但他有新的作用，你也必须给予支持）。在第四章提及的一些功能可以在打开的时候进行选择。

想象一下，一个驱动支持视频采集，直接视频传输，视频间隔消隐信号和 FM 收音机。他用 0,64,224（这种编码格式是为保兼容性从 V4L API 中继承下来的）次设备号注册了三个设备。不管 `/dev/video(81,0)` 或者 `/dev/vbi(81,224)` 是否处于打开状态，应用程序都可以从这三种设备中选择某项功能。不通过编程（就是说你可以直接用命令 `dd` 和 `cat` 来操作）`/dev/video` 就可采集视频图片，通过 `/dev/vbi` 来获得原始的视频间隔消隐信号。`/dev/radio (81,64)` 是一个不变的无线装置，因而和视频

---

部分的功能并没有联系。但是，没有相互的关联并不意味着你可以同时使用它们。当你用 `open()` 调用时很有可能返回设备忙--EBUSY 的错误代码。

除视频的输入输出外，硬件可能也支持抽样甚至是回放功能（VLC貌似就是这样的）。如果真是这样，那么这些功能会被实现为OSS（Open Sound System）或者ALSA（Advanced Linux Sound Architecture）PCM设备，最终可能是OSS或者ALSA的混合体。V4L2 API并未做任何假设来发现这些有相互关系的设备。

If you have an idea please write to the Video4Linux mailing list:

<https://listman.redhat.com/mailman/listinfo/video4linux-list>.

### 1.1.3. 并发打开(Multiple Opens)

通常，V4L2 设备可被多次打开的。当这被驱动支持时，用户可以，比如，你可以设计一个面板程序，上面有些按钮可以调节亮暗或者音量啊，而与此同时另一个应用程序却正在采集音视频。换句话说，面板程序就被比作了 OSS 或者 ALSA 混合应用了。当一个设备支持多项功能像采集啊且能同时直接接传输啊神马的，Multiple Opens 允许通过 fork 出进程或实现特定的应用程序达到并发的使用设备。

Multiple Opens 是可选的，驱动应该至少允许在没数据交换的情况下支持并发访问，例子就是，上面的面板程序。这意味着当设备正在使用中时 `open()` 可返回 EBUSY 错误代码，`ioctl()` 函数的初始化数据交换（也就是 VIDIOC\_S\_FMT `ioctl`），`read()`，`write()` 函数亦是如此。

只不过打开一个 V4L2 设备并不允许互斥访问<sup>4</sup>。然而数据交换赋予读和写所请求数据类型的权利并做出相应属性的变换。

也可以改变相关属性，文件描述符。应用程序可以通过在 1.3 节描述的优先级机制请求额外的访问权限。

### 1.1.4. 共享数据流(Shared Data Streams)

V4L2 驱动不应该支持在一个设备上，通过拷贝缓冲对同一数据流进行多应用程序的读或写，或者是时分复用亦或是其他相似的方法。非要这么干，可以使用在用户空间的代理程序。如果驱动支持流共享，那么其实现必须是透明的。V4L2 API 并没有列出产生冲突时要如何来解决。

### 1.1.5. 函数(Functions)

应用程序可以使用 `open()` 和 `close()` 函数来打开、关闭 V4L2 设备（都是雷同的）。下面几小节将介绍设备编程所用到的 `ioctl()` 方法。

## 1.2. 功能查询(Querying Capabilities)

这个 V4L2 涉及了各式各样的设备，但是呢，这个 API 接口却并不能都适合这些个繁杂的设备。因此，相同类型的设备就可能拥有完全不一样的能力，所以这份规范也允许忽略一些复杂且不那么重要的 API 接口。

`ioctl` 的 VIDIOC\_QUERYCAP 选项是用来检查内核设备是不是和这份规范相兼容滴，也可以顺便看看这些设备所能够支持的函数和 I/O 方法，一举两得。其他的特性嘛，也可以依葫芦画瓢，调用各自的 `ioctl` 方法，例子么，想知道设备上的视频连接头的名字啊，型号和数量等，就可以调用 `ioctl` 的

---

VIDIOC\_ENUMINPUT 方法。尽管 API 主要功用是--抽象（模糊）底层硬件，但是，ioctl 方法是允许驱动特定的应用程序来可靠的辨识相应的驱动的。

但是，所有的 V4L2 驱动都必须支持 VIDIOC\_QUERYCAP 这个方法（显而易见的）。应用程序都应该在打开设备之后就调用这个方法。

### 1.3. 应用程序的优先级(Application Priority)

当多个程序共享一个设备时，你可能就希望分配给他们不同的优先级（想想为什么封建社会的把人分成三六九等，虽然不人道但有一方面：执政者是希望管理起来方便）。不同于传统"rm -rf /"(移除根目录下的一切，系统需要内核+文件系统，这么做告诉你：不存在阻塞？)思想，视频记录程序可能阻止其他的应用程序，使他们失去对视频的控制,不能换电视台(一个劲的放广告，这搁谁他也受不鸟啊！)。

另一个目的就是，允许被用户控制的应用程序对可工作在后台的低优先级的应用进行抢占，然后（被抢占的）在稍后一点的时间重新获得对设备的控制权。

鉴于这些个特性是不可能都在用户空间中实现，V4L2 就定义了 ioctl 的 VIDIOC\_G\_PRIORITY 和 VIDIOC\_S\_PRIORITY 方法来请求和查询访问文件描述符的优先级。因为有些个驱动并不支持这两个方法，为保持和 V4L2 的早期版本的兼容性，会在打开设备时默认的给他们个中间的优先级。在通过 VIDIOC\_QUERYCAP 后，应用程序通常会调用 VIDIOC\_S\_PRIORITY 请求一个其他的优先级。

如果 ioctls 在其他程序拥有更高的优先级时，用诸如 VIDIOC\_S\_INPUT 方法去改变驱动的属性，会返回 EBUSY 的错误代码。事件机制（如：U 盘的热插拔，他会在用户空间中弹出个东东，告诉你 USB 来了）会通知应用程序有人在别的地方想篡改属性，但是还没添加呢(您老给看看怎么办呗)！

### 1.4. 视频输入/输出(Video Inputs and Outputs)

视频的输入和输出就是设备上的物理连接头。例子是：RF 红外头，复合视频头（不知是啥玩意），I -Video 或者 RGB 连接头。收音机就没有视频输入、输出了。

如果想去了解可用数量的输入和输出属性，应用程序可以分别使用 ioctl 的 VIDIOC\_ENUMINPUT 和 VIDIOC\_ENUMOUTPUT 方法来枚举查看。当当前的视频输入被查询的时候，ioctl 的 VIDIOC\_ENUMINPUT 方法会返回 v4l2\_input 结构体，它包含了可用的信号状态信息。

ioctl 的 VIDIOC\_G\_INPUT 和 VIDIOC\_G\_OUTPUT 方法会返回当前视频的输入、输出索引。为选择不同的输入、输出，应用程序可以使用 ioctl 的 VIDIOC\_S\_INPUT 和 VIDIOC\_S\_OUTPUT 方法。当设备有一个或多个输入时，驱动就必须实现所有的输入类 ioctl 方法，对于输出也是同样的。

#### Example 1-1. 查询当前视频输入设备的信息

```
struct v4l2_input input;
int index;
if(-1 == ioctl(fd, VIDIOC_G_INPUT, &index)){
    perror("VIDIOC_G_INPUT");
    exit(EXIT_FAILURE);
}
```



---

```

memset(&input, 0, sizeof(input));
input.index = index;
if(-1 == ioctl(fd, VIDIOC_ENUMINPUT, &input)){
    perror("VIDIOC_ENUMINPUT");
    exit(EXIT_FAILURE);
}
printf("Current input: %s\n", input.name);

```

### Example 1-2. 切换到第一个视频输入设备

```

int index;
index = 0;
if(-1 == ioctl(fd, VIDIOC_S_INPUT, &index)){
    perror("VIDIOC_S_INPUT");
    exit(EXIT_FAILURE);
}

```

## 1.5. 音频输入/输出(Audio Inputs and Outputs)

音频的输入输出也是一个设备的物理连接头。视频采集设备有输入（必须的），（若有输出就有输出？既是 0 个或多个的意思？）输出设备有输出（这...）。无线接收装置是没有音频输入或输出的（就是个电磁信号）。他们必定拥有一个调频头，这也是他们的音频信号来源！但是对于本 API，调频头只是相对于视频的输入、输出的，音频设备是没有这些个的。TV 卡上的回环接头，就是接收音频信号并将它们发到声卡上变成声音，却没有被当成音频输出。

音视频的输入输出是有联系的。想想也是，视频源通常夹杂音频信号。当视频和音频源是个[高频头](#)时，这就更加明显了。假设存在有两个混合视频输入，两个音频输入，可以想见会有 4 种组合状况。视频和音频接头间的关系，分别定义在结构体 `v4l2_input`（源码在 `Videodev2.h` 中）和 `v4l2_output` 的 `audioset` 域中，其每一位（从零开始）都代表着是输入还是输出的索引。

```

struct v4l2_input{
    __u32    index;           // 输入设备索引，由应用程序设置
    __u8     name[32];        // 设备名

    __u32    type;            /* 输入设备类型 */
    __u32    audioset;         /* 相关联的音频设备 (bitfield)；这样驱动可以最多枚举32对视频和音频设备。如果bit0被置位为1，则则该视频输入设备对应的音频输入即为audio0。具体设置请参见1.5节*/

    __u32    tuner;           /* 相关联的高频头，采集设备可以有 0 个或多个 RF 解调器，当 type 域为 V4L2_INPUT_TYPE_TUNER 时，表示该输入设备是一个高频头。详见 1.6 节 */

```

Capture devices can have zero or more tuners (RF demodulators).  
When the type is set to V4L2\_INPUT\_TYPE\_TUNER this is an RF

---

```

connector and this field identifies the tuner. It corresponds to
struct v4l2_tuner field index. For details on tuners see Section 1.6.

v4l2_std_id    std;        /* 视频输入设备一般都会支持多种视频标准，该域就列举设备
                             所能支持的标准，详见 1.7 节。*/

__u32          status;     /* 当前设备状态信息，只有该设备为当前使用的设备时该域才有
                             效，状态可取值在 Table 3 中 */

__u32          reserved[4]; /* 保留，驱动必需将他们置位为 0 */

};

```

如果想去知道可用的输入、输出数量和属性，应用程序可以分别通过使用 `ioctl` 的 `VIDIOC_ENUMAUDIO` 和 `VIDIOC_ENUMAUDOUT` 方法来枚举他们。当正在使用中的设备被查询时，由 `VIDIOC_ENUMAUDOUT` 方法返回的 `v4l2_audio` 结构中包含了可用的信号状态信息。

```

struct v4l2_audio{
    __u32    index;        音频输入索引，由驱动或应用设置
    __u8     name[32];     音频输入设备名
    __u32    capability;   该音频设备所具属性, see Table 2.
    __u32    mode;        音频设备模式字，由驱动设置(on VIDIOC_S_AUDIO ioctl), see
    Table 3.
    __u32    reserved[2];  保留，驱动和应用必需置他们为 0

};

```

`ioctl` 的 `VIDIOC_G_AUDIO` 和 `VIDIOC_G_AUDOUT` 方法会分别返回当前音频的输入和输出。和 `VIDIOC_G_INPUT`, `VIDIOC_G_OUTPUT` 不同，他们不仅仅返回个索引，而是类似 `VIDIOC_ENUMAUDIO` 和 `VIDIOC_ENUMDOUT` 返回结构体。

意欲选择音频的输入或者是改变其属性，应用程序可以调用 `ioctl` 的 `VIDIOC_S_AUDIO` 方法。同样的如果想选择音频输出（要知道他们是没有可变属性的），则可以调用 `VIDIOC_S_AUDOUT` 方法。

当设备具有一个或多个输入、输出时，驱动就必须实现所有相应的 `ioctl` 方法！如果设备具有任何音频输入或者输出，驱动就必须设置由 `VIDIOC_QUERYCAP` 方法返回的 `v4l2_capability` 结构中 `V4L2_CAP_AUDIO` 标志位。

```

struct v4l2_capability{
    __u8     driver[16];    驱动名。特定驱动的应用程序可以通过该域来验证驱动信息；驱
                             动版本信息存储在 version 域中；用固定长度数组存放字符串虽不明智但在这却不可避免，因而，无
                             论是驱动还是应用都得小心别越界读取。It is also useful to work around known bugs, or to identify
                             drivers in error reports.

    __u8     card[32];     设备名。One driver may support different brands or models of video

```

---

hardware. This information is intended for users, for example in a menu of available devices.

Since multiple TV cards of the same brand may be installed which are supported by the same driver, this name should be combined with the character device file name (e. g. /dev/video2) or the bus\_info string to avoid ambiguities.

    \_\_u8    bus\_info[32];            设备节点位置信息。Location of the device in the system. For example: "PCI Slot 4". This information is intended for users, to distinguish multiple identical devices. If no such information is available the field may simply count the devices controlled by the driver, or contain the empty string (bus\_info[0] = 0).

    \_\_u32    version;                驱动版本号。结合驱动名唯一确定一个驱动。版本号是通过 KERNEL\_VERSION()宏来格式化的:

```
#define KERNEL_VERSION(a,b,c) (((a) << 16) + ((b) << 8) + (c)) __u32
```

```
version = KERNEL_VERSION
```

    \_\_u32    capabilities;            Device capabilities, see Table 2.

    \_\_u32    reserved[4];            保留，驱动和应用必需置他们为 0

```
};
```

```
#define V4L2_CAP_AUDIO                0x00020000 /* has audio support */
```

## Table 2. Device Capabilities Flags

V4L2_CAP_VIDEO_CAPTURE	0x00000001	设备支持视频采集
V4L2_CAP_VIDEO_OUTPUT	0x00000002	设备支持视频输出
V4L2_CAP_VIDEO_OVERLAY	0x00000004	支持视频覆盖接口
V4L2_CAP_VBI_CAPTURE	0x00000010	原始 VBI 数据采集接口
V4L2_CAP_VBI_OUTPUT	0x00000020	原始 VBI 数据输出接口
V4L2_CAP_SLICED_VBI_CAPTURE	0x00000040	Sliced VBI Capture interface.
V4L2_CAP_SLICED_VBI_OUTPUT	0x00000080	Sliced VBI Output interface.
V4L2_CAP_RDS_CAPTURE	0x00000100	未定义

### Example 1-3. 当前音频输入设备信息

```
struct v4l2_audio audio;
memset (&audio, 0, sizeof (audio));
```

---

```

if (-1 == ioctl (fd, VIDIOC_G_AUDIO, &audio)) {
    perror ("VIDIOC_G_AUDIO");
    exit (EXIT_FAILURE);
}

printf ("Current input: %s\n", audio.name);

```

#### **Example 1-4. 切换到第一个音频输入设备**

```

struct v4l2_audio audio;

memset (&audio, 0, sizeof (audio)); /* clear audio.mode, audio.reserved */

audio.index = 0;

if (-1 == ioctl (fd, VIDIOC_S_AUDIO, &audio)) {
    perror ("VIDIOC_S_AUDIO");
    exit (EXIT_FAILURE);
}

```

### **1.6. 高频头和调制器 (Tuners and Modulators)**

#### **1.6.1. 高频头(Tuners)**

视频输入设备可能有一个或多个高频头来解调 RF（射频）信号。VIDIOC\_ENUMINPUT 方法会返回 type 域被设置成 V4L2\_INPUT\_TYPE\_TUNER 的 v4l2\_input 结构体，且其中的 tuner 域也包含着 tuner 的索引号码。

无线发射设备通常只有索引号为 0 的这么一个 tuner，同时没有任何视频输入。

应用程序可以分别通过使用 VIDIOC\_G\_TUNER 和 VIDIOC\_S\_TUNER 方法来查询和改变 tuner 的属性。当正在使用中的设备被查询时，由 VIDIOC\_G\_TUNER 方法返回的 v4l2\_tuner 结构中包含了可用的信号状态信息。需要注意的是，当拥有多个 tuner 时 VIDIOC\_S\_TUNER 并不会切换当前的 tuner。其实这个 tuner 是由当前的视频输入唯一确定的。当设备含有一个或多个 tuner 时，驱动就必须支持 ioctl 方法和提供设置由 VIDIOC\_QUERYCAP 返回的 v4l2\_capability 中 V4L2\_CAP\_TUNER 标志。

```

...

#define VIDIOC_G_TUNER      _IOWR('V', 29, struct v4l2_tuner)

#define VIDIOC_S_TUNER      _IOW('V', 30, struct v4l2_tuner)

...

```

---

### 1.6.2. 调制器 (Modulators)

视频输出设备可以有一个或多个调制器，他可以将视频信号调制成 TV 电线的解说信号，或者是视频记录器的。每个调制器是和一個或多个视频输出相关联的，而关联的数量取决于 RF 射频连接头的数量。

VIDIOC\_ENUMOUTPUT 方法返回的 v4l2\_output 结构中的 type 域会被设置成 V4L2\_OUTPUT\_TYPE\_MODULATOR,而他的 modulator 域包含了 modulator 的索引号。规范并没有定义无线输出设备。

```
struct v4l2_output {
    __u32      index;    /* Which output */
    __u8       name[32]; /* Label */
    __u32      type;     /* Type of output */
    __u32      audioset; /* Associated audios (bitfield) */
    __u32      modulator; /* Associated modulator */
    v4l2_std_id std;
    __u32      reserved[4];
};
```

应用程序可以使用 VIDIOC\_G\_MODULATOR 和 VIDIOC\_S\_MODULATOR 方法来查询和改变 modulator 的属性。需要注意的是，当拥有多个 tuner 时 VIDIOC\_S\_MODULATOR 并不会切换当前的 modulator。其实这个 modulator 是由当前的视频输入唯一确定的。当设备含有一个或多个 modulator 时，驱动就必须支持 ioctl 方法和提供设置由 VIDIOC\_QUERYCAP 返回的 v4l2\_capability 中 V4L2\_CAP\_MODULATOR 标志。

### 1.6.3. 无线频率(Radio Frequency)

```
struct v4l2_frequency {
    __u32      tuner;    The tuner or modulator index number. This is the same value as in
the struct v4l2_input tuner field and the struct v4l2_tuner index field, or the struct v4l2_output modulator
field and the struct v4l2_modulator index field.

    enum v4l2_tuner_type type;    The tuner type. This is the same value as in the struct
v4l2_tuner type field. The field is not applicable to modulators, i.e.ignored by drivers.

    __u32      frequency;    Tuning frequency in units of 62.5 kHz, or if the struct v4l2_tuner
or struct v4l2_modulator capabilities flag V4L2_TUNER_CAP_LOW is set, in units of 62.5 Hz.

    __u32      reserved[8];    保留，同样要置位 0
};
```

---

```
};
```

应用程序可以使用 `VIDIOC_G_FREQUENCY` 和 `VIDIOC_S_FREQUENCY` 方法来获得和设置 tuner 或者 modulator 的无线频率。这些方法同样适用于 TV 和无线设备。当提供 tuner 和 modulator 的 `ioctl` 方法时，驱动就必须实现相关方法。

#### 1.6.4. 卫星接收器(Satellite Receivers)

将要讨论。可以参见 Peter Schlaf 的建议，位于邮件列表：[video4linux-list@redhat.com](mailto:video4linux-list@redhat.com) 上，发表于 2002 年 10 月 23 号，主题是：“Re: [V4L] Re: v4l2 api”。

### 1.7. 视频标准(Video Standards)

视频设备通常支持一个或多个不同的视频标准或标准的不同变种。每个视频输入和输出可以支持另外一套标准。这个集合包含在由方法 `VIDIOC_ENUMINPUT` 返回的结构体 `v4l2_input` 的 `std` 域中，同样的在 `v4l2_output` 的 `std` 域中也包含了。

V4L2 为世界范围内使用的每一种模拟视频标准定义了一个 bit，并且为驱动定义的标准预留了 bit 位，e.g. 在 PAL 电视上观看 NTSC 录像带。应用程序是可以通过预定义的 bit 位来选择特定的标准，但是应该为用户提供一个所有可支持标准的菜单（你决定机制，策略由客户选择）。应用程序可以通过使用 `VIDIOC_ENUMSTD` 方法来枚举和查询所支持标准的属性。

已定义的标准许多都是一些主流标准的变体。事实上硬件可能并不区分它们，或者只是自动的内部做处理和切换。因而，枚举得到的 bit 位也包含着一个或多个标准结合（如下）。

假定有一个可以解调 B/PAL, G/PAL 和 I/PAL 信号的 tuner。枚举得到的第一个标准是 B 和 G/PAL 结合，他们会根据在 UHF 或者 VHF 频带上所选择的无线频率会自动的进行切换（how?）。枚举提供了一个“PAL-B/G”或“PAL- I ”的选择。类似于混合的输入，枚举“PAL-B/G/H/I”, “NTSC-M”和“SECAM-D/K”，<sup>6</sup>可能会混乱标准滴。

应用程序可以分别调用 `VIDIOC_G_STD` 和 `VIDIOC_S_STD` 方法来查询和选择当前视频输入或输出使用的标准。`VIDIOC_QUERYSTD` 方法可以感知所谓公认的标准。鉴于所有这些 `ioctl` 方法只是一个指向 `V4L2_STD_ID` 类型（一个标准集合），而不是标准列举的一个索引。<sup>7</sup>当设备具有一个或多个视频输入或输出时，驱动就必须实现所有视频标准的 `ioctl` 方法。

**Table 3. typedef v4l2\_std\_id**

<code>__u64</code>	<code>v4l2_std_id</code>	This type is a set, each bit representing another video standard as listed below and in Table 4. The 32 most significant bits are reserved for custom (driver defined) video standards.
--------------------	--------------------------	---

```
#define V4L2_STD_PAL_B ((v4l2_std_id)0x00000001)
```

---

```

#define V4L2_STD_PAL_B1    ((v4l2_std_id)0x00000002)
#define V4L2_STD_PAL_G     ((v4l2_std_id)0x00000004)
#define V4L2_STD_PAL_H     ((v4l2_std_id)0x00000008)
#define V4L2_STD_PAL_I     ((v4l2_std_id)0x00000010)
#define V4L2_STD_PAL_D     ((v4l2_std_id)0x00000020)
#define V4L2_STD_PAL_D1    ((v4l2_std_id)0x00000040)
#define V4L2_STD_PAL_K     ((v4l2_std_id)0x00000080)
#define V4L2_STD_PAL_M     ((v4l2_std_id)0x00000100)
#define V4L2_STD_PAL_N     ((v4l2_std_id)0x00000200)
#define V4L2_STD_PAL_Nc    ((v4l2_std_id)0x00000400)
#define V4L2_STD_PAL_60    ((v4l2_std_id)0x00000800)

```

应用于USB摄像头的专用规则，视频标准就没啥意义了。相应地，当出现下列状况的时候：

- incapable of capturing fields or frames at the nominal rate of the video standard, or
- where timestamps refer to the instant the field or frame was received by the driver, not the capture time, or
- where sequence numbers refer to the frames received by the driver, not the captured frames.

驱动应该设置v4l2\_input和v4l2\_output结构中std域为0，同时VIDIOC\_G\_STD, VIDIOC\_S\_STD, VIDIOC\_QUERYSTD和VIDIOC\_ENUMSTD方法应该返回EINVAL错误代码。

### Example 1-5. Information about the current video standard

```

v4l2_std_id std_id;

struct v4l2_standard standard;

if (-1 == ioctl (fd, VIDIOC_G_STD, &std_id)) {
    /* Note when VIDIOC_ENUMSTD always returns EINVAL this
       is no video device or it falls under the USB exception,
       and VIDIOC_G_STD returning EINVAL is no error. */
    perror ("VIDIOC_G_STD");
    exit (EXIT_FAILURE);
}

memset (&standard, 0, sizeof (standard));

standard.index = 0;

```

---

```

while (0 == ioctl (fd, VIDIOC_ENUMSTD, &standard)) {
    if (standard.id &std_id) {
        printf ("Current video standard: %s\n", standard.name);
        exit (EXIT_SUCCESS);
    }
    standard.index++;
}

/* EINVAL indicates the end of the enumeration, which cannot be
   empty unless this device falls under the USB exception. */
if (errno == EINVAL || standard.index == 0) {
    perror ("VIDIOC_ENUMSTD");
    exit (EXIT_FAILURE);
}

```

**Example 1-6. Listing the video standards supported by the current input**

```

struct v4l2_input input;
struct v4l2_standard standard;
memset (&input, 0, sizeof (input));
if (-1 == ioctl (fd, VIDIOC_G_INPUT, &input.index)) {
    perror ("VIDIOC_G_INPUT");
    exit (EXIT_FAILURE);
}
if (-1 == ioctl (fd, VIDIOC_ENUMINPUT, &input)) {
    perror ("VIDIOC_ENUM_INPUT");
    exit (EXIT_FAILURE);
}
printf ("Current input %s supports:\n", input.name);
memset (&standard, 0, sizeof (standard));
standard.index = 0;
while (0 == ioctl (fd, VIDIOC_ENUMSTD, &standard)) {

```



---

```

    if (standard.id & input.std)
        printf ("%s\n", standard.name);
    standard.index++;
}

/* EINVAL indicates the end of the enumeration, which cannot be
   empty unless this device falls under the USB exception. */
if (errno != EINVAL || standard.index == 0) {
    perror ("VIDIOC_ENUMSTD");
    exit (EXIT_FAILURE);
}

```

### **Example 1-7. Selecting a new video standard**

```

struct v4l2_input input;
v4l2_std_id std_id;
memset (&input, 0, sizeof (input));
if (-1 == ioctl (fd, VIDIOC_G_INPUT, &input.index)) {
    perror ("VIDIOC_G_INPUT");
    exit (EXIT_FAILURE);
}
if (-1 == ioctl (fd, VIDIOC_ENUMINPUT, &input)) {
    perror ("VIDIOC_ENUM_INPUT");
    exit (EXIT_FAILURE);
}
if (0 == (input.std & V4L2_STD_PAL_BG)) {
    fprintf (stderr, "Oops. B/G PAL is not supported.\n");
    exit (EXIT_FAILURE);
}

/* Note this is also supposed to work when only B
   or G/PAL is supported. */

```

---

```

std_id = V4L2_STD_PAL_BG;

if (-1 == ioctl (fd, VIDIOC_S_STD, &std_id)) {

    perror ("VIDIOC_S_STD");

    exit (EXIT_FAILURE);

}

```

## 1.8. 用户控制(User Controls)

设备通常具有相当数量的用户配置选项，诸如亮度，饱和度等等，这些个都会显示在图形化的用户界面上。但是，显然不同的设备将会拥有不同的控制选项，更近一步说，就是控制选项的数量和其默认值是随设备的不同而不同的。控制类 `ioctl` 方法提供了一些信息和一种机制来为不同的控制选项创建漂亮用户接口，并且不同的设备也都能很好的工作。

所有的控制选项都是通过 ID 值来进行访问的。`V4L2` 为特定的用途定义了若干个 ID 值。当然，驱动也可以使用 `V4L2_CID_PRIVATE_BASE` 或者更高的值来定制实现自己的控制选项。预定义的控制 ID 都有 `V4L2_CID_` 标识，具体可参见 Table 1-1。这些个 ID 值在查询控制属性、获取和设置它们时会被用到。

通常，应用程序应该直接的把控制选项展现在用户面前，而无需考虑用户的企图。每个控制选项最好有个用户能够见名知意的名字。悲摧的是，如果意图并不显而易见，驱动作者应该提供一个用户手册（来告知），一个用户接口插件或者...。预定义的 ID 值将被介绍如何可编程的来改变控制选项的，例子是当你换频道的时候可以使设备静音。

当切换了视频的输入或者输出、调频头、或者音频的输入或输出时，驱动可能会重新枚举不同的控制选项。区别在于带宽(?)、其他的默认和当前值、步进大小或者其他的主菜单选项。拥有确定的定制 ID 号的控制选项同样是可以改变其名字和类型的。<sup>9</sup> 控制选项的值是全局储存的，当切换到或者保持在某一带宽上的时候，其值是不会改变的。同样的，当设备处于打开或者关闭状态时，当调频收音机的频率变化了其值也不会改变。因为 `V4L2` 规范没有事件触发机制，面板类应用程序趋向于和其他的面板类程序合作(当然了他们得同处一个大的应用程序中了)，[面板类程序可能需要定期的 `poll` 控制选项的值来更新他们的用户界面](#)。<sup>10</sup>

**Table 1-1. Control IDs**

ID	Type	Description
V4L2_CID_BASE		First predefined ID, equal to V4L2_CID_BRIGHTNESS.
V4L2_CID_USER_BASE		Synonym of V4L2_CID_BASE.
V4L2_CID_BRIGHTNESS	integer	Picture brightness, or more precisely, the black level.
V4L2_CID_CONTRAST	integer	Picture contrast or luma gain.
V4L2_CID_SATURATION	integer	Picture color saturation or chroma gain.

---

V4L2_CID_HUE	integer	Hue or color balance.
V4L2_CID_AUDIO_VOLUME	integer	Overall audio volume. Note some drivers also provide an OSS or ALSA mixer interface.
V4L2_CID_AUDIO_BALANCE	integer	Audio stereo balance. Minimum corresponds to all the way left, maximum to right.
V4L2_CID_AUDIO_BASS	integer	Audio bass adjustment.
V4L2_CID_AUDIO_TREBLE	integer	Audio treble adjustment.
V4L2_CID_AUDIO_MUTE	boolean	Mute audio, i. e. set the volume to zero, however without affecting V4L2_CID_AUDIO_VOLUME.  Like ALSA drivers, V4L2 drivers must mute at load time to avoid excessive noise. Actually the entire device should be reset to a low power consumption state.
V4L2_CID_AUDIO_LOUDNESS	boolean	Loudness mode (bass boost).
V4L2_CID_BLACK_LEVEL	integer	Another name for brightness (not a synonym of V4L2_CID_BRIGHTNESS). This control is deprecated and should not be used in new drivers and applications.
V4L2_CID_AUTO_WHITE_BALANCE	boolean	Automatic white balance (cameras).
V4L2_CID_DO_WHITE_BALANCE	button	This is an action control. When set (the value is ignored), the device will do a white balance and then hold the current setting. Contrast this with the boolean V4L2_CID_AUTO_WHITE_BALANCE, which, when activated, keeps adjusting the white balance.
V4L2_CID_RED_BALANCE	integer	Red chroma balance.
V4L2_CID_BLUE_BALANCE	integer	Blue chroma balance.
V4L2_CID_GAMMA	integer	Gamma adjust.
V4L2_CID_WHITENESS	integer	Whiteness for grey-scale devices. This is a synonym for V4L2_CID_GAMMA. This control is deprecated and should not be used in new drivers and applications.
V4L2_CID_EXPOSURE	integer	Exposure (cameras). [Unit?]
V4L2_CID_AUTOGAIN	boolean	Automatic gain/exposure control.

---

V4L2_CID_GAIN	integer	Gain control.
V4L2_CID_HFLIP	boolean	Mirror the picture horizontally.
V4L2_CID_VFLIP	boolean	Mirror the picture vertically.
V4L2_CID_HCENTER_DEPRECATED		
(formerly V4L2_CID_HCENTER)	integer	Horizontal image centering. This control is deprecated. New drivers and applications should use the Camera class controls V4L2_CID_PAN_ABSOLUTE, V4L2_CID_PAN_RELATIVE and V4L2_CID_PAN_RESET instead.
V4L2_CID_VCENTER_DEPRECATED		
(formerly V4L2_CID_VCENTER)	integer	Vertical image centering. Centering is intended to physically adjust cameras. For image cropping see Section 1.11, for clipping Section 4.2. This control is deprecated. New drivers and applications should use the Camera class controls V4L2_CID_TILT_ABSOLUTE, V4L2_CID_TILT_RELATIVE and V4L2_CID_TILT_RESET instead.
V4L2_CID_POWER_LINE_FREQUENCY	integer	Enables a power line frequency filter to avoid flicker. Possible values are: V4L2_CID_POWER_LINE_FREQUENCY_DISABLED (0), V4L2_CID_POWER_LINE_FREQUENCY_50HZ(1) and V4L2_CID_POWER_LINE_FREQUENCY_60HZ(2).
V4L2_CID_HUE_AUTO	boolean	Enables automatic hue control by the device. The effect of setting V4L2_CID_HUE while automatic hue control is enabled is undefined, drivers should ignore such request.
V4L2_CID_WHITE_BALANCE_TEMPERATURE	integer	This control specifies the white balance settings as a color temperature in Kelvin. A driver should have a minimum of 2800 (incandescent) to 6500 (daylight). For more information about color temperature see Wikipedia ( <a href="http://en.wikipedia.org/wiki/Color_temperature">http://en.wikipedia.org/wiki/Color_temperature</a> ).
V4L2_CID_SHARPNESS	integer	Adjusts the sharpness filters in a camera. The minimum value disables the filters, higher values give a sharper picture.
V4L2_CID_BACKLIGHT_COMPENSATION	integer	Adjusts the backlight compensation in a

---

camera. The minimum value disables backlight compensation.

V4L2\_CID\_LASTP1                      End of the predefined control IDs (currently V4L2\_CID\_BACKLIGHT\_COMPENSATION + 1).

V4L2\_CID\_PRIVATE\_BASE                ID of the first custom (driver specific) control.

Applications depending on particular custom controls should check the driver name and version, see Section 1.2.

应用程序可以通过使用 `ioctl` 的 `VIDIOC_QUERYCTRL` 和 `VIDIOC_QUERYMENU` 方法来枚举可使用的控制选项，想获得或者设置这些值可是使用 `VIDIOC_G_CTRL` 和 `VIDIOC_S_CTRL` 方法。当设备拥有一个或者多个控制选项时，驱动就必须实现 `VIDIOC_QUERYCTRL`, `VIDIOC_G_CTRL` 和 `VIDIOC_S_CTRL` 方法，同样的当设备具备一个或多个菜单类的控制选项时，你就得实现 `VIDIOC_QUERYMENU` 方法。

### Example 1-8. Enumerating all controls

```
struct v4l2_queryctrl queryctrl;

struct v4l2_querymenu querymenu;

static void
enumerate_menu (void)
{
    printf (" Menu items:\n");
    memset (&querymenu, 0, sizeof (querymenu));
    querymenu.id = queryctrl.id;
    for (querymenu.index = queryctrl.minimum;
         querymenu.index <= queryctrl.maximum;
         querymenu.index++) {
        if (0 == ioctl (fd, VIDIOC_QUERYMENU, &querymenu)) {
            printf (" %s\n", querymenu.name);
        } else {
            perror ("VIDIOC_QUERYMENU");
            exit (EXIT_FAILURE);
        }
    }
}
```

---

```
memset (&queryctrl, 0, sizeof (queryctrl));

for (queryctrl.id = V4L2_CID_BASE;
     queryctrl.id < V4L2_CID_LASTP1;
     queryctrl.id++) {
    if (0 == ioctl (fd, VIDIOC_QUERYCTRL, &queryctrl)) {
        if (queryctrl.flags & V4L2_CTRL_FLAG_DISABLED)
            continue;

        printf ("Control %s\n", queryctrl.name);
        if (queryctrl.type == V4L2_CTRL_TYPE_MENU)
            enumerate_menu ();
    } else {
        if (errno == EINVAL)
            continue;

        perror ("VIDIOC_QUERYCTRL");
        exit (EXIT_FAILURE);
    }
}

for (queryctrl.id = V4L2_CID_PRIVATE_BASE;;
     queryctrl.id++) {
    if (0 == ioctl (fd, VIDIOC_QUERYCTRL, &queryctrl)) {
        if (queryctrl.flags & V4L2_CTRL_FLAG_DISABLED)
            continue;

        printf ("Control %s\n", queryctrl.name);
        if (queryctrl.type == V4L2_CTRL_TYPE_MENU)
            enumerate_menu ();
    } else {
        if (errno == EINVAL)
            break;

        perror ("VIDIOC_QUERYCTRL");
```

---

```
        exit (EXIT_FAILURE);
    }
}
```

### **Example 1-9. Changing controls**

```
struct v4l2_queryctrl queryctrl;
struct v4l2_control control;
memset (&queryctrl, 0, sizeof (queryctrl));
queryctrl.id = V4L2_CID_BRIGHTNESS;
if (-1 == ioctl (fd, VIDIOC_QUERYCTRL, &queryctrl)) {
    if (errno != EINVAL) {
        perror ("VIDIOC_QUERYCTRL");
        exit (EXIT_FAILURE);
    } else {
        printf ("V4L2_CID_BRIGHTNESS is not supported\n");
    }
} else if (queryctrl.flags & V4L2_CTRL_FLAG_DISABLED) {
    printf ("V4L2_CID_BRIGHTNESS is not supported\n");
} else {
    memset (&control, 0, sizeof (control));
    control.id = V4L2_CID_BRIGHTNESS;
    control.value = queryctrl.default_value;
    if (-1 == ioctl (fd, VIDIOC_S_CTRL, &control)) {
        perror ("VIDIOC_S_CTRL");
        exit (EXIT_FAILURE);
    }
}

memset (&control, 0, sizeof (control));
control.id = V4L2_CID_CONTRAST;
```

---

```

if (0 == ioctl (fd, VIDIOC_G_CTRL, &control)) {
    control.value += 1;

    /* The driver may clamp the value or return ERANGE, ignored here */
    if (-1 == ioctl (fd, VIDIOC_S_CTRL, &control)
        && errno != ERANGE) {
        perror ("VIDIOC_S_CTRL");
        exit (EXIT_FAILURE);
    }

    /* Ignore if V4L2_CID_CONTRAST is unsupported */
} else if (errno != EINVAL) {
    perror ("VIDIOC_G_CTRL");
    exit (EXIT_FAILURE);
}

control.id = V4L2_CID_AUDIO_MUTE;
control.value = TRUE; /* silence */

/* Errors ignored */
ioctl (fd, VIDIOC_S_CTRL, &control);

```

## 1.9. 扩展控制(Extended Controls)

### 1.9.1. 介绍(Introduction)

控制选项机制最初设计的意图是方便用户进行控制设置的（亮度，饱和度，等等）。但是，出乎意料的是，他越发的成为了一种有用的适合实现更多复杂驱动 API 的模型，在那些驱动只需要实现一个大的 API 框架下的一小集合时的情况下，其效果显而易见。

设计和实现扩展型控制机制的强制力来自于 MPEG 编码 API: MPEG 标准是那么大以至于，现行支持 MPEG 标准的编码器，大多数只实现了其标准的一个子集。而且，很多个关乎视频是如何被编码成 MPEG 流的参数是特定于 MPEG 编码芯片的，其原因在于，MPEG 标准只定义了最终的 MPEG 流的格式，而不是（视频）如何被编码成这种格式的（简单说就是：只告诉你结果，却忽略了这个过程）。

不幸的是，最初的控制选项 API 缺少针对某些个新兴使用的功能，最终导致了将这些个新功能扩



---

展到了扩展类的控制选项 API 中去了。

### 1.9.2. 扩展类控制 API(The Extended Control API)

有三种可以使用的 ioctl 方法：VIDIOC\_G\_EXT\_CTRL、VIDIOC\_S\_EXT\_CTRL 和 VIDIOC\_TRY\_EXT\_CTRL。他们扮演着多种控制选项的角色（不同于以往的 VIDIOC\_G\_CTRL 和 VIDIOC\_S\_CTRL 方法，这些只有单一的功用）。当同时需要自动的改变好几个控制选项时，他们就派上用场了。

新的 ioctl 方法都需要一个指向 v4l2\_ext\_controls 结构的指针。这个结构包含一个指向控制选项数组的指针，一些控制选项值和一个控制选项类。这个控制类嘛就是把一些相似的控制给组合在一起。例子说，控制类：V4L2\_CTRL\_CLASS\_USER，看见 user 了吧，它就包含了所有的用户类（自己定制的控制（那些可以通过老式的 VIDIOC\_S\_CTRL 方法设置的）。而 V4L2\_CTRL\_CLASS\_MPEG 类则包含了所有与 MPEG 编码有关的控制选项，等等。

在控制数组中所列的控制选项必须隶属于特定的控制选项类。否则就会出错。

当然你可以使用一个空值数组（count == 0）来检验某个特定控制选项类是否受到支持。

控制选项类就是一个 v4l2\_ext\_control 结构数组，其和结构体 v4l2\_control 非常的相似，区别在于前者允许 64bit 值和可以传递指针（尽管当前并没有使用过该特性）。

有一点很重要，由于控制选项的灵活性，有必要检查一下你要设置的 control 值是不是已经被驱动所支持了，其值的范围是什么。你可以使用 VIDIOC\_QUERYCTRL 和 VIDIOC\_QUERYMENU 方法来进行检查。另一点需要注意的是，

列在菜单目录里的控制类型：V4L2\_CTRL\_TYPE\_MENU 可能并没有得到支持（即 VIDIOC\_QUERYMENU 会返回一个错误）。例子说，所支持的 MPEG 音频比特率列表。一些个驱动只支持一两个比特率，而其他的则可能支持的更宽泛一些。

### 1.9.3. 扩展 control 值的枚举(Enumeration Extended Controls)

推荐的枚举扩展 control 值，是结合 V4L2\_CTRL\_FLAG\_NEXT\_CTRL 标志位来使用

VIDIOC\_QUERYCTRL 方法：

```
struct v4l2_queryctrl qctrl;

qctrl.id = V4L2_CTRL_FLAG_NEXT_CTRL;

while (0 == ioctl (fd, VIDIOC_QUERYCTRL, &qctrl)) {

/* ... */

qctrl.id |= V4L2_CTRL_FLAG_NEXT_CTRL;

}
```

---

使用 `V4L2_CTRL_FLAG_NEXT_CTRL` 标志.会将 `control ID` 置为 0。`VIDIOC_QUERYCTRL` 方法会使用一个比给定的更高的 `ID` 值作为返回值。要是没有这样的 `ID`，将会返回错误。

你要是想得到一个特定的 `control` 类中的 `control` 值的话，你可以将 `qctrl.id` 的值初始化成这个 `control` 类，然后额外的检测，直到检测到另一个 `control` 类时跳出这个循环：

```
qctrl.id = V4L2_CTRL_CLASS_MPEG | V4L2_CTRL_FLAG_NEXT_CTRL;
while (0 == ioctl (fd, VIDIOC_QUERYCTRL, &qctrl)) {
    if (V4L2_CTRL_ID2CLASS (qctrl.id) != V4L2_CTRL_CLASS_MPEG)
        break;
    /* ... */
    qctrl.id |= V4L2_CTRL_FLAG_NEXT_CTRL;
}
```

32bit 的 `qctrl.id` 值被分成了 3 个 bit 区间：高 4 位是为标志位做的保留（e.g.`V4L2_CTRL_FLAG_NEXT_CTRL`）且并不是 `ID` 的一部分。剩下的 28 位中，12 位是相当的重要，用来定义 `control` 类；其余的 16 位是次重要的，用来标识 `control` 类中的成员的（不同的 `control` 值）。且这最后的 16 位是一些个非 0 值。从 `0x1000` 一直到 `0xffff` 都是为驱动相关的 `control` 值所保留的。宏 `V4L2_CTRL_ID2CLASS(id)`，看名字就知道是干啥用的了，我就不介绍了。

假使，驱动并没支持扩展的 `control` 值，结合 `V4L2_CTRL_FLAG_NEXT_CTRL` 标志使用的 `VIDIOC_QUERYCTRL` 方法将会发生错误，此时，就得使用旧版的方法了（参见 1.8 节）。但是，如果他支持的话，就能够保证枚举所有的 `control` 值，包括哪些驱动私有的 `control` 值。

#### 1.9.4. 创建 control 面板(Creating Control Panels)

你也可以为方便图形界面用户选择繁杂的 `control` 而设计一个 `control` 面板。但基本上你不得不用上面介绍过的方法来迭代的枚举 `control` 了。每个 `control` 类都从 `control` 类型为 `V4L2_CTRL_TYPE_CTRL_CLASS` 开始。`VIDIOC_QUERYCTRL` 方法返回的 `control` 类的名字将会显现在 `control` 面板上。

`v4l2_queryctl` 中的 `flags` 域同样包含了 `control` 行为的暗示。可以参见 `VIDIOC_QUERYCTRL` 文档以获得更多的资料。

#### 1.9.5. MPEG Control Reference

下面是在 MPEG 的 `control` 类中的所有 `control`。第一类的是通用的 `control`，接着是特定于硬件类型的 `control`。

##### 1.9.5.1. 通用 MPEG 控制值(Generic MPEG Controls)

**Table 1-2. MPEG Control IDs**

ID	Type	Description
V4L2_CID_MPEG_CLASS	class	The MPEG class descriptor. Calling VIDIOC_QUERYCTRL for this control will return a description of this control
V4L2_CID_MPEG_STREAM_TYPE	enum	The MPEG-1, -2 or -4 output stream type. One cannot assume anything here. Each hardware MPEG encoder tends <b>ENTRYTBL not supported.</b>
V4L2_CID_MPEG_STREAM_PID_PMT	integer	Program Map Table Packet ID for the MPEG transport stream (default 16)
V4L2_CID_MPEG_STREAM_PID_AUDIO	integer	Audio Packet ID for the MPEG transport stream (default 256)
V4L2_CID_MPEG_STREAM_PID_VIDEO	integer	Video Packet ID for the MPEG transport stream (default 260)
V4L2_CID_MPEG_STREAM_PID_PCR	integer	Packet ID for the MPEG transport stream carrying PCR fields (default 259)
V4L2_CID_MPEG_STREAM_PES_ID_AUDIO	integer	Audio ID for MPEG PES
V4L2_CID_MPEG_STREAM_PES_ID_VIDEO	integer	Video ID for MPEG PES
V4L2_CID_MPEG_STREAM_VBI_FMT	enum	Some cards can embed VBI data (e. g. Closed Caption, Teletext) into the MPEG stream. This control selects whether <b>ENTRYTBL not supported.</b>

---

V4L2\_CID\_MPEG\_AUDIO\_SAMPLING\_FREQ    enum

MPEG Audio sampling frequency. Possible values are:

**ENTRYTBL not supported.**

V4L2\_CID\_MPEG\_AUDIO\_ENCODING    enum

MPEG Audio encoding. Possible values are:

**ENTRYTBL not supported.**

V4L2\_CID\_MPEG\_AUDIO\_L1\_BITRATE    enum

Layer I bitrate. Possible values are:

**ENTRYTBL not supported.**

V4L2\_CID\_MPEG\_AUDIO\_L2\_BITRATE    enum

Layer II bitrate. Possible values are:

**ENTRYTBL not supported.**

V4L2\_CID\_MPEG\_AUDIO\_L3\_BITRATE    enum

Layer III bitrate. Possible values are:

**ENTRYTBL not supported.**

V4L2\_CID\_MPEG\_AUDIO\_MODE    enum

MPEG Audio mode. Possible values are:

**ENTRYTBL not supported.**

V4L2\_CID\_MPEG\_AUDIO\_MODE\_EXTENSION    enum

Joint Stereo audio mode extension. In Layer I and II they indicate which subbands are in intensity stereo. All other ENTRYTBL not supported.

V4L2\_CID\_MPEG\_AUDIO\_EMPHASIS    enum

---

Audio Emphasis. Possible values are:

**ENTRYTBL not supported.**

V4L2\_CID\_MPEG\_AUDIO\_CRC      enum

CRC method. Possible values are:

**ENTRYTBL not supported.**

V4L2\_CID\_MPEG\_AUDIO\_MUTE    bool

Mutes the audio when capturing. This is not done by muting audio hardware, which can still produce a slight hiss, V4L2\_CID\_MPEG\_VIDEO\_ENCODING enum

MPEG Video encoding method. Possible values are:

**ENTRYTBL not supported.**

V4L2\_CID\_MPEG\_VIDEO\_ASPECT   enum

Video aspect. Possible values are:

**ENTRYTBL not supported.**

V4L2\_CID\_MPEG\_VIDEO\_B\_FRAMES   integer

Number of B-Frames (default 2)

V4L2\_CID\_MPEG\_VIDEO\_GOP\_SIZE    integer

GOP size (default 12)

V4L2\_CID\_MPEG\_VIDEO\_GOP\_CLOSURE   bool

GOP closure (default 1)

V4L2\_CID\_MPEG\_VIDEO\_PULLDOWN   bool

Enable 3:2 pulldown (default 0)

V4L2\_CID\_MPEG\_VIDEO\_BITRATE\_MODE   enum

Video bitrate mode. Possible values are:

**ENTRYTBL not supported.**

V4L2\_CID\_MPEG\_VIDEO\_BITRATE      integer

---

Video bitrate in bits per second.

V4L2\_CID\_MPEG\_VIDEO\_BITRATE\_PEAK    integer

Peak video bitrate in bits per second. Must be larger or equal to the average video bitrate. It is ignored if the video V4L2\_CID\_MPEG\_VIDEO\_TEMPORAL\_DECIMATION integer

For every captured frame, skip this many subsequent frames (default 0).

V4L2\_CID\_MPEG\_VIDEO\_MUTE    bool

"Mutes" the video to a fixed color when capturing. This is useful for testing, to produce a fixed video bitstream. 0 V4L2\_CID\_MPEG\_VIDEO\_MUTE\_YUV integer

Sets the "mute" color of the video. The supplied 32-bit integer is interpreted as follows (bit 0 = least significant bit):

**ENTRYTBL not supported.**

#### 1.9.5.2. CX2341x 型 MPEG 控制(CX2341x MPEG Controls)

下面表中所列的 MPEG 类 control，是用来处理特定于 Conexant 公司的 CX23415 和 CX23416 编码芯片的编码设置的。

**Table 1-3. CX2341x Control IDs**

ID	Type
Description	
V4L2_CID_MPEG_CX2341X_VIDEO_SPATIAL_FILTER_MODE	enum
Sets the Spatial Filter mode (default MANUAL). Possible values are:	
<b>ENTRYTBL not supported.</b>	
V4L2_CID_MPEG_CX2341X_VIDEO_SPATIAL_FILTER	integer(0-15)
The setting for the Spatial Filter. 0 = off, 15 = maximum. (Default is 0.)	
V4L2_CID_MPEG_CX2341X_VIDEO_LUMA_SPATIAL_FILTER_TYPE	enum
Select the algorithm to use for the Luma Spatial Filter (default 1D_HOR). Possible values:	
ENTRYTBL not supported.	
V4L2_CID_MPEG_CX2341X_VIDEO_CHROMA_SPATIAL_FILTER_TYPE	enum
Select the algorithm for the Chroma Spatial Filter (default 1D_HOR). Possible values are:	
<b>ENTRYTBL not supported.</b>	
V4L2_CID_MPEG_CX2341X_VIDEO_TEMPORAL_FILTER_MODE	enum
Sets the Temporal Filter mode (default MANUAL). Possible values are:	

**ENTRYTBL not supported.**

V4L2\_CID\_MPEG\_CX2341X\_VIDEO\_TEMPORAL\_FILTER integer(0-31)

The setting for the Temporal Filter. 0 = off, 31 = maximum. (Default is 8 for full-scale capturing and 0 for scaled V4L2\_CID\_MPEG\_CX2341X\_VIDEO\_MEDIAN\_FILTER\_TYPE enum

Median Filter Type (default OFF). Possible values are:

**ENTRYTBL not supported.**

V4L2\_CID\_MPEG\_CX2341X\_VIDEO\_LUMA\_MEDIAN\_FILTER\_BOTTOM integer(0-255)

Threshold above which the luminance median filter is enabled (default 0)

V4L2\_CID\_MPEG\_CX2341X\_VIDEO\_LUMA\_MEDIAN\_FILTER\_TOP integer(0-255)

Threshold below which the luminance median filter is enabled (default 255)

V4L2\_CID\_MPEG\_CX2341X\_VIDEO\_CHROMA\_MEDIAN\_FILTER\_BOTTOM integer(0-255)

Threshold above which the chroma median filter is enabled(default 0)

V4L2\_CID\_MPEG\_CX2341X\_VIDEO\_CHROMA\_MEDIAN\_FILTER\_TOP integer(0-255)

Threshold below which the chroma median filter is enabled(default 255)

V4L2\_CID\_MPEG\_CX2341X\_STREAM\_INSERT\_NAV\_PACKETS bool

The CX2341X MPEG encoder can insert one empty MPEG-2 PES packet into the stream between every four vid

1.9.6. 摄像头 control 参考(Camera Control Reference)

Camera 类所包含的 control，是些关乎设备的机械特性（等价的数字式特性）的，诸如可控的焦距或者传感器类型（CMOS、CCD）。

**Table 1-4. Camera Control IDs**

ID	Type
Description	

(详见规范 0.24 版本 P32)

1.10. 数据格式(Data Formats)

1.10.1. 协商数据格式(Data Format Negotiation)

不同的设备和应用程序之间交换不同的数据，例子：视频图片，原始的或者部分处理过的 VBI 数据，RDS 数据报。即使在一类中，有多种不同格式也是可能的，尤其是有着非常丰富的图片格式。

---

虽然驱动必须提供一个默认值，但是选择常交叉着进行关闭、再打开一个设备过程，应用程序应该始终在进行数据交换前，把数据的格式协商好。协商意味着应用程序请求特定的格式然后驱动选择并报告硬件所能够做的最好的满足需求的请求给应用程序。当然应用程序也可以自己查询当前的所选。

通过使用 `v4l2_format` 结构体和 `VIDIOC_G_FMT` 和 `VIDIOC_S_FMT` 方法这一单一机制来实现数据格式协商的。另外，方法 `VIDIOC_TRY_FMT` 也可用来查看硬件到底能干些什么，而不用通过选择某个具体的数据格式来验证。`V4L2 API` 所能支持的所有数据格式会在第四章中特定硬件相关的节中列出。而在第二章中可以进一步查看图片格式。

方法 `VIDIOC_S_FMT` 在初始化顺序中是个主要的转折点。优先在这方面：多面板应用程序在访问同一个设备时可以同时选择当前的输入，改变 `control` 值或者修改其他属性。开始 `VIDIOC_S_FMT` 会将一个逻辑流（包含视频数据，`VBI` 数据等等）赋值给一确定的（`exclusive`）文件描述符（`fd`）。

`exclusive` 意思是说，其他任何应用程序，更准确点说是没有其他任何 `fd`，可以得到这个 `stream` 或者改变已经协商好的设备属性（防竞争？）。例子说，当新的视频标准使用不同的扫描线时，视频标准可以使选定的图片格式失效（？）。因此，只有拥有流的 `fd` 才可以真正的判定某项更改是否有效。也就是说拥有不同流的 `fd` 负责他们之间互不干涉。例子说，当要进行或已在处理视频混叠时，同一时刻，视频采集就可能限制进行同样的修剪和图片尺寸。

当应用程序忽略 `VIDIOC_S_FMT` 方法时，通过 `VIDIOC_REQBUFS` 方法选择 I/O，或者是第一次调用 `read()` 或 `write()` 会暗示它自身的副作用：`locking`。

通常，一个逻辑 `stream` 只能被赋值给一个 `fd`（上面说了），但是为了保持和早期的 `V4L`，早期的 `V4L2` 版本兼容，驱动是使用同一个 `fd` 以允许同时进行视频采集和混叠（`overlay`）的。通过关闭或者重新打开设备可以切换到不同的 `stream` 或者返回到面板模式。通过使用 `VIDIOC_S_FMT`，驱动可能支持切换。

所有驱动要是想和应用程序交换数据就必须支持 `VIDIOC_G_FMT` 和 `VIDIOC_S_FMT` 方法。我们高度建议实现 `VIDIOC_TRY_FMT`，但也只是可选的。

### 1.10.2. 枚举图片格式(Image Format Enumeration)

除了上面提到的通用协商函数外，还有一种特殊的 `ioctl` 来枚举受到视频采集、`overlay` 或者输出设备支持的图像格式。

它就是人见人爱，车见车爆胎的 `VIDIOC_ENUM_FMT`！所有意欲和应用程序交换图像数据的驱动就必须予以实现。

**Important:** 我们并没有假定驱动在内核空间进行图片格式的转换。他们必须枚举那些由硬件直接支持的特定格式。如有必要，驱动作者最好将转换样例或者库集成到应用程序中去。

### 1.11. 图片修剪，插入和缩放(Image Cropping, Insertion and Scalling)

有些个视频采集设备可以采样图片的一部分并且可以任意大小对他们进行缩小或放大。我们称它们为裁剪（`crop`）和缩放（`scall`）。一些个视频输出设备可以将一张图片放大或缩小然后插到一任意扫描线和水平偏移的视频信号中去。

应用程序可以使用下面的 `API` 来在视频信号中选择区域，查询默认的区域和硬件本身的限制。忽



略它们的名字，对于输入和输出设备方法 `VIDIOC_CROPCAP`, `VIDIOC_G_CROP` 和 `VIDIOC_S_CROP` 都是适用的。

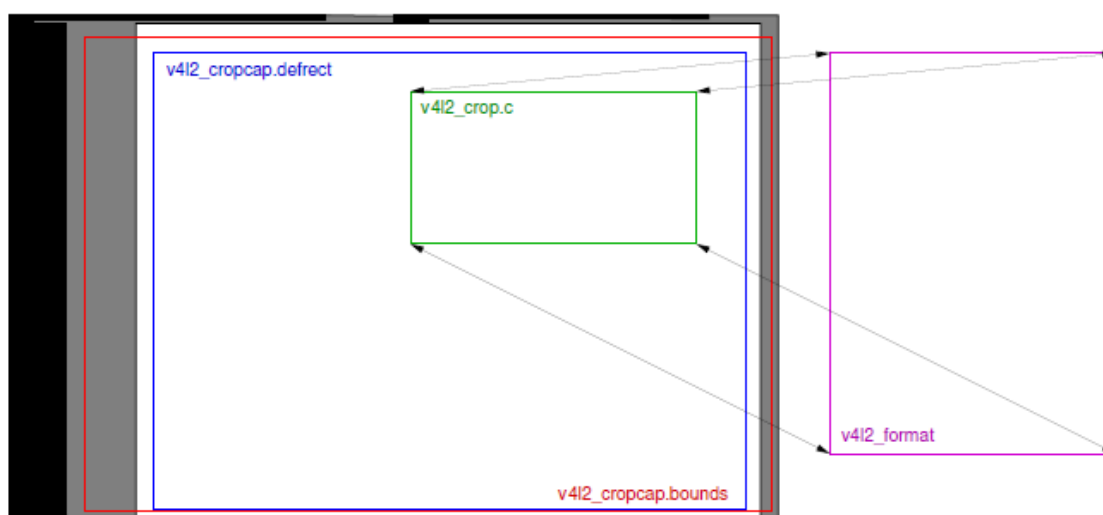
缩放需要一个源（`source`）和一个目标（`target`）。对视频采集或者 `overlay` 设备而言，其源自然是视频信号，修剪 `ioctl` 则决定实际的采样区域。而 `target` 则是应用程序要读取的图片或者需被 `overlay`（重载？No）到图形屏幕上的图片。关于它们的大小（对 `overlay` 而言是位置）是通过 `VIDIOC_G_FMT` 和 `VIDIOC_S_FMT` 进行协商的。

对于一个视频输出设备，其 `source` 是那些有应用程序传进来的或者是编码在视频流中的图片，它们的大小同样是同过 `VIDIOC_G/S_FMT` 协商得到的。而 `target` 则是视频信号，修剪 `ioctl` 决定图片插在那些区域。

即使设备并不支持修剪和缩放或者 `VIDIOC_G/S_CROP` 方法，`source` 和 `target` 也被定义了。在这种情形下，它们的大小（或者位置）会得到修正。所有的采集和输出设备必须支持 `VICIOC_CROPCAP` 方法以帮助应用程序确定缩放功能是否可以使用。

### 1.11.1. 修剪结构(Cropping Structures)

Figure 1-1. Image Cropping, Insertion and Scaling



关于采集设备，左顶角的坐标，可采集区域的宽度和高度由 `VIDIOC_CROPCAP` 返回的 `v4l2_cropcap` 结构中的 `bounds` 结构给出的。为了支持更广范的硬件，本规范并没定义原点或者单位。...

右顶角，可采集区域的宽度和高度的值是和 `v4l2_cropcap` 采用同一坐标系统的 `v4l2_crop` 给出的。应用程序可以使用 `VIDIOC_G/S_CROP` 方法来设定这个矩形框。当然了你必须确保这框框位于边界之内，驱动可能出于硬件本身的限制会调整所请求的大小或者位置。

每个采集设备都有个默认的源框框，这个默认值是由 `v4l2_cropcap` 中的结构体 `defrect` 给出的。这个框框的中心必须和视频信号的采集区对齐，并且要覆盖到驱动作者所考虑到的末了图片。当驱动第一次加载时驱动得重置源框框为默认状态而不是在稍后某个时候。

对于输出设备，定义图片将被插到视频信号那块 `target` 框框时，也会相应的使用这些结构和 `ioctl` 方法。

---

### 1.11.2. 缩放调整(Scaling Adjustments)

视频硬件可以有很多的裁剪，插入和缩放功能限制。他可能只支持放大或者只是缩小，只支持离散的缩放而不是连续可调的，或者在水平和垂直方向上其缩放能力不一样。甚至他压根就不支持缩放功能。与此同时，结构 `v4l2_crop` 描绘的框框可能不得不可虑对齐，并且 `source` 和 `target` 的框框也可能不是任意尺寸的。特别的，`v4l2_crop` 中的最大宽度和高度也必须小于 `v4l2_cropcap.bounds` 定义的区域（废话么）。因此，驱动通常会对所请求的参数进行调整并返回所选的实际值。

应用程序可以先改变 `source` 或 `target` 框框，因为他们有时候更在乎视频信号中的图片尺寸或者确定的区域。假使驱动不得不调整这俩方面以满足硬件的限制，那最后请求的框框（`source` or `target`）可能具有高优先级，所以驱动最好调整相对的那一个。方法 `VIDIOC_TRY_FMT` 是无论如何也不能改变驱动状态的，因此只能调整所请求的框框了。

假设在一个视频采集设备上的缩放在俩方向上都被限制为 1:1 或者 2:1 的比例，并且 `target` 的图片尺寸必须是 16x16 像素的倍数。而 `source` 修剪框框则被设为默认值，在本例中这也是上限值：在 0,0 偏移处的 640x400 像素。这时有个应用程序请求一图片尺寸为 300x225 像素，并假设视频会相应的从满像素尺寸缩放为需要的。驱动会将图片尺寸设置为最为接近的 304x224，接着挑选接近请求尺寸的裁剪框框：608x224（224x2:1 会超过上限 400）。0,0 偏移值并没被修改。`VIDIOC_CROPCAP` 提供的修剪框框默认值，应用程序可以轻松地请求其他的偏移值来居中这个框框。

现在应用程序可能要坚持使用一定的比例来接近原始请求，接着会请求一个 608x456 像素的修剪框框。但有上限限制，所以是 640x384，驱动将会返回修剪尺寸是 608x384，而调整过后的图片尺寸就会是 304x192（请求的是 300x225）。

### 1.11.3. 样例程序(Examples)

`source` 和 `target` 框框在交替进行关闭和重新打开一个设备时应保持不变，这样用管道传递数据就不必进行准备就能很好的工作。当然，更高级的应用程序在开始 I/O 前，应该确保所有参数是合适的。

#### Example 1-10. Resetting the cropping parameters

(P<sub>35</sub>)

(A video capture device is assumed; change `V4L2_BUF_TYPE_VIDEO_CAPTURE` for other devices.)

```
struct v4l2_cropcap cropcap;

struct v4l2_crop crop;

memset (&cropcap, 0, sizeof (cropcap));

cropcap.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;

if (-1 == ioctl (fd, VIDIOC_CROPCAP, &cropcap)) {

    perror ("VIDIOC_CROPCAP");

    exit (EXIT_FAILURE);

}
```

---

```

memset (&crop, 0, sizeof (crop));

crop.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;

crop.c = cropcap.defrect;

    /* Ignore if cropping is not supported (EINVAL). */

if (-1 == ioctl (fd, VIDIOC_S_CROP, &crop)

    && errno != EINVAL) {

    perror ("VIDIOC_S_CROP");

    exit (EXIT_FAILURE);

}

```

### Example 1-11. Simple downscaling

(A video capture device is assumed.)

```

struct v4l2_cropcap cropcap;

struct v4l2_format format;

reset_cropping_parameters ();

    /* Scale down to 1/4 size of full picture. */

memset (&format, 0, sizeof (format)); /* defaults */

format.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;

format.fmt.pix.width = cropcap.defrect.width >> 1;

format.fmt.pix.height = cropcap.defrect.height >> 1;

format.fmt.pix.pixelformat = V4L2_PIX_FMT_YUYV;

if (-1 == ioctl (fd, VIDIOC_S_FMT, &format)) {

    perror ("VIDIOC_S_FORMAT");

    exit (EXIT_FAILURE);

}

    /* We could check the actual image size now, the actual scaling factor

    or if the driver can scale at all. */

```

### Example 1-12. Selecting an output area

---

```

struct v4l2_cropcap cropcap;

struct v4l2_crop crop;

memset (&cropcap, 0, sizeof (cropcap));

cropcap.type = V4L2_BUF_TYPE_VIDEO_OUTPUT;

if (-1 == ioctl (fd, VIDIOC_CROPCAP, &cropcap)) {

    perror ("VIDIOC_CROPCAP");

    exit (EXIT_FAILURE);

}

memset (&crop, 0, sizeof (crop));

crop.type = V4L2_BUF_TYPE_VIDEO_OUTPUT;

crop.c = cropcap.defrect;

    /* Scale the width and height to 50 % of their original size and center the output. */

crop.c.width /= 2;

crop.c.height /= 2;

crop.c.left += crop.c.width / 2;

crop.c.top += crop.c.height / 2;

    /* Ignore if cropping is not supported (EINVAL). */

if (-1 == ioctl (fd, VIDIOC_S_CROP, &crop)

&& errno != EINVAL) {

perror ("VIDIOC_S_CROP");

exit (EXIT_FAILURE);

}

```

### Example 1-13. Current scaling factor and pixel aspect

(A video capture device is assumed.)

```

struct v4l2_cropcap cropcap;

struct v4l2_crop crop;

struct v4l2_format format;

double hscale, vscale;

```

---

```
double aspect;

int dwidth, dheight;

memset (&cropcap, 0, sizeof (cropcap));

cropcap.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;

if (-1 == ioctl (fd, VIDIOC_CROPCAP, &cropcap)) {

    perror ("VIDIOC_CROPCAP");

    exit (EXIT_FAILURE);

}

memset (&crop, 0, sizeof (crop));

crop.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;

if (-1 == ioctl (fd, VIDIOC_G_CROP, &crop)) {

    if (errno != EINVAL) {

        perror ("VIDIOC_G_CROP");

        exit (EXIT_FAILURE);

    }

    /* Cropping not supported. */

    crop.c = cropcap.defrect;

}

memset (&format, 0, sizeof (format));

format.fmt.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;

if (-1 == ioctl (fd, VIDIOC_G_FMT, &format)) {

    perror ("VIDIOC_G_FMT");

    exit (EXIT_FAILURE);

}

/* The scaling applied by the driver. */

hscale = format.fmt.pix.width / (double) crop.c.width;

vscale = format.fmt.pix.height / (double) crop.c.height;

aspect = cropcap.pixelaspect.numerator / (double) cropcap.pixelaspect.denominator;

aspect = aspect * hscale / vscale;
```

---

```
/* Devices following ITU-R BT.601 do not capture  
  
square pixels. For playback on a computer monitor  
  
we should scale the images to this size. */  
  
dwidth = format.fmt.pix.width / aspect;  
  
dheight = format.fmt.pix.height;
```

## 1.12. Streaming 参数(Streaming Parameters)

如同 I/O 一样，**streaming** 参数目的是优化视频采集处理。目前，应用程序可以通过使用 **VIDIOC\_S\_PARM** 方法来请求高质量的采集模式。

当前的视频标准决定了一个名义上的 **fps** 大小。如果采集或者输出低于这个数，应用程序可以在驱动那边请求帧跳过或者复制（重复一帧）。这一点在使用 **read()**或者 **write()**时非常有用，因为读或写并不关心时间戳或者其顺序，从而可避免不必要的数据拷贝。

最终这些个 **ioctl** 可在驱动内部的读/写模式时用来决定要使用的 **buffer** 数量。欲知详情，请看讨论 **read()**函数的那章节。

应用程序可以调用 **VIDIOC\_G/S\_PARM** 方法 来获得和设置 **streaming** 参数。他们使用一个指向 **v4l2\_streamparm** 结构的指针，其中就包含着一个单独于输入和输出设备参数的联合体(就是 **union**)。

驱动并不需要非得实现这些个 **ioctl**，他们是可选的。当然，要是没有实现，使用时只返回简单的 **EINVAL**（Error: Invalid）错误代码。

## Notes

(P<sub>38</sub>)

---

## Chapter 2. 图像格式(Image Formats)

V4L2 API 主要是设计用于设备和应用程序进行图片数据交换的。结构 `v4l2_pix_format` 定义了一张图片在内存中的格式和布局。格式是通过 `VIDIOC_S_FMT` 协商得到的。(这里涉及的重点是关乎视频采集和输出, 对于 `overlay` 帧缓冲格式, 您还是看看 `VIDIOC_G_FBUF`)。

**Table 2-1. struct v4l2\_pix\_format**

<code>__u32</code>	<code>width</code>	Image width in pixels.
<code>__u32</code>	<code>height</code>	Image height in pixels.

应用程序设置这些域是用来请求图片尺寸, 驱动会返回最接近的合适值。在平面格式情形中... (不知是不是我下规范不好怎么地, 在这种列表式中总会有些地方不完整, 当然我看过了, 这不太影响我们的工作)

Applications set these fields to request an image size, drivers return the closest possible values. In case of planar formats

<code>__u32</code>	<code>pixelformat</code>	The pixel format or type of compression, set by  the application. This is a little endian four  character code. V4L2 defines standard RGB  formats in Table 2-1, YUV formats in Section 2.5,  and reserved codes in Table 2-8
--------------------	--------------------------	---

<code>enum v4l2_field</code>	<code>field</code>	Video images are typically interlaced.  Applications can request to capture or output only
------------------------------	--------------------	--

		the top or bottom field, or both fields interlaced or sequentially stored in one buffer or alternating in separate buffers. Drivers return the actual field order selected. For details see Section 3.6.
__u32	bytesperline	Distance in bytes between the leftmost pixels in two adjacent lines.

应用程序和驱动都可以设置这些域以请求在每行的末尾增加填补位。然而驱动却可能予以忽略...

Both applications and drivers can set this field to request padding bytes at the end of each line. Drivers however may ignore

__u32	sizeimage	Size in bytes of the buffer to hold a complete image, set by the driver. Usually this is bytesperline times height. When the image consists of variable length compressed data this is the maximum number of bytes required to hold an image.
-------	-----------	---

enum v4l2_colorspace	colorspace	This information supplements the pixelformat and must be set by the driver, <b>see Section 2.2</b> .
----------------------	------------	--

__u32	priv	Reserved for custom (driver defined) additional information about formats. When not used drivers and applications must set this field to zero.
-------	------	--

## 2.1. 标准图片格式(Standard Image Formats)

为了在驱动和应用程序间进行图片交换，就必须有一个双方都能辨析（统一的）的图片数据格式。V4L2 包含了诸多格式，而本节就来谈谈 V4L2 规范中的标准图片格式。

当然，V4L2 驱动并不局限于这些个格式。相反，可以有定制的格式。在这种情况下，当需要时应用程序就可能依赖解码器来将这种格式转换成标准类型的。但是数据依然可以以专有（定制的）格式来进行数据的存储和重新获取。例子说，假设一个设备支持专有的压缩格式，那么，应用程序仍然可以以这种格式进行采集和保存数据，已达到节省空间的目的。而当要在 X Windows 端显示时，就可以用解码器进行解码并显示。

最后，尽管如此，我们还是需要一些个标准格式的。只有定义了明确的标准格式，才可以保证 V4L2 规范的完整性嘛。

（接下来的这一段主要说标准格式，是未压缩的，其在内存的布局是怎么样的，就不译了）



---

在 V4L2 中，每种格式都有前缀形如 PIX\_FMT\_XXX，定义在 videodev.h 头文件中。这些标识符代表了在下面列出的 4 字节代码，当然了他们和 Windows（相对的是 Linux 或者 Unix）世界里的用法是不一样的。

## 2.2. 色彩空间(Colorspace)

Gamma Correction

[to do]

$$E'R = f(R)$$

$$E'G = f(G)$$

$$E'B = f(B)$$

Construction of luminance and color-difference signals

[to do]

$$E'_Y = \text{Coeff}_R E'_R + \text{Coeff}_G E'_G + \text{Coeff}_B E'_B$$

$$(E'_R - E'_Y) = E'_R - \text{Coeff}_R E'_R - \text{Coeff}_G E'_G - \text{Coeff}_B E'_B$$

$$(E'_B - E'_Y) = E'_B - \text{Coeff}_R E'_R - \text{Coeff}_G E'_G - \text{Coeff}_B E'_B$$

Re-normalized color-difference signals

The color-difference signals are scaled back to unity range [-0.5;+0.5]:

$$K_B = 0.5 / (1 - \text{Coeff}_B)$$

$$K_R = 0.5 / (1 - \text{Coeff}_R)$$

$$P_B = K_B (E'_B - E'_Y) = 0.5 (\text{Coeff}_R / \text{Coeff}_B) E'_R + 0.5 (\text{Coeff}_G / \text{Coeff}_B) E'_G + 0.5 E'_B$$

$$P_R = K_R (E'_R - E'_Y) = 0.5 E'_R + 0.5 (\text{Coeff}_G / \text{Coeff}_R) E'_G + 0.5 (\text{Coeff}_B / \text{Coeff}_R) E'_B$$

Quantization

[to do]

$$Y' = (\text{Lum. Levels} - 1) \cdot E'_Y + \text{Lum. Offset}$$

$$C_B = (\text{Chrom. Levels} - 1) \cdot P_B + \text{Chrom. Offset}$$

$$C_R = (\text{Chrom. Levels} - 1) \cdot P_R + \text{Chrom. Offset}$$

给大伙提供个介绍 RGB 和 YUV 关系的文档（网上找的，在此感谢原作者！）

[JPEG 简易文档 V1.0 - GameRes.com.htm](http://JPEG.简易文档.V1.0-GameRes.com.htm)(如果打不开直接去搜一下吧)

### Example 2-1. ITU-R Rec. BT.601 color conversion

---

## Forward Transformation

```
int ER, EG, EB;      /* gamma corrected RGB input [0;255] */
```

```
int Y1, Cb, Cr;      /* output [0;255] */
```

```
double r, g, b;      /* temporaries */
```

```
double y1, pb, pr;
```

```
int
```

```
clamp (double x)
```

```
{
```

```
int r = x; /* round to nearest */
```

```
if (r < 0)          return 0;
```

```
else if (r > 255)    return 255;
```

```
else                return r;
```

```
}
```

```
r = ER / 255.0;
```

```
g = EG / 255.0;
```

```
b = EB / 255.0;
```

```
y1 = 0.299 * r + 0.587 * g + 0.114 * b;
```

```
pb = -0.169 * r - 0.331 * g + 0.5 * b;
```

```
pr = 0.5 * r - 0.419 * g - 0.081 * b;
```

```
Y1 = clamp (219 * y1 + 16);
```

```
Cb = clamp (224 * pb + 128);
```

```
Cr = clamp (224 * pr + 128);
```

---

```
/* or shorter */
```

```
y1 = 0.299 * ER + 0.587 * EG + 0.114 * EB;
```

```
Y1 = clamp ( (219 / 255.0) * y1 + 16);
```

```
Cb = clamp (((224 / 255.0) / (2 - 2 * 0.114)) * (EB - y1) + 128);
```

```
Cr = clamp (((224 / 255.0) / (2 - 2 * 0.299)) * (ER - y1) + 128);
```

## Inverse Transformation

```
int Y1, Cb, Cr;          /* gamma pre-corrected input [0;255] */
```

```
int ER, EG, EB;          /* output [0;255] */
```

```
double r, g, b;          /* temporaries */
```

```
double y1, pb, pr;
```

```
int
```

```
clamp (double x)
```

```
{
```

```
int r = x; /* round to nearest */
```

```
if (r < 0)          return 0;
```

```
else if (r > 255)   return 255;
```

```
else                return r;
```

```
}
```

```
y1 = (255 / 219.0) * (Y1 - 16);
```

```
pb = (255 / 224.0) * (Cb - 128);
```

```
pr = (255 / 224.0) * (Cr - 128);
```

```
r = 1.0 * y1 + 0 * pb + 1.402 * pr;
```

```
g = 1.0 * y1 - 0.344 * pb - 0.714 * pr;
```

b = 1.0 \* y1 + 1.772 \* pb + 0 \* pr;

ER = clamp (r \* 255); /\* [ok? one should prob. limit y1,pb,pr] \*/

EG = clamp (g \* 255);

EB = clamp (b \* 255);

Table 2-2. enum v4l2\_colorspace

(详见 P<sub>42</sub>)

2.3. 索引格式(Indexed Format)

在这种格式中，每个像素点都用一个 8bit 来表示，用来索引 256 色的调色板。这是特地为视频输出 overlay 准备的。没有用来访问该调色板的 ioctl 方法，但可以且仅可以通过 Linux 的帧缓冲 API（头一回接触）来进行访问。

Table 2-3. Indexed Image Format(P<sub>42</sub>-P<sub>43</sub>)

2.4. RGB 格式(RGB Formats)

Packed RGB Formats

Name

Packed RGB formats – Packed RGB formats

Description

这些格式定义用来匹配传统 PC 的图形帧缓冲的像素格式的。每像素占据 8,16, 24 或者 32 个 bit。这就是组合像素(packed-pixel)格式,意味着每个像素点在内存中是一个挨着一个的。

当使用这些格式中的某一个时，驱动应该报告给色彩空间：V4L2\_COLORSPACE\_SRGB。

Table 2-1. Packed RGB Image Formats

Identifier Code	Byte 0 7654/3210	Byte 1	Byte 2	Byte 3
V4L2_PIX_FMT_RGB 332	b1 b0 g2 g1 g0 r2 r1 r0			

'RGB1'				
V4L2_PIX_FMT_RGB 444 'R444'	g3 g2 g1 g0 b3 b2 b1 b0	a3 a2 a1 a0 r3 r2 r1 r0		
V4L2_PIX_FMT_RGB 555 'RGBO'	g2 g1 g0 r4 r3 r2 r1 r0	a b4 b3 b2 b1 b0 g4 g3		
V4L2_PIX_FMT_RGB 555X 'RGBQ'	a b4 b3 b2 b1 b0 g4 g3	g2 g1 g0 r4 r3 r2 r1 r0		
V4L2_PIX_FMT_RGB 565 'RGBP'	g2 g1 g0 r4 r3 r2 r1 r0	b4 b3 b2 b1 b0 g5 g4 g3		
V4L2_PIX_FMT_RGB 565X 'RGBR'	b4 b3 b2 b1 b0 g5 g4 g3	g2 g1 g0 r4 r3 r2 r1 r0		
V4L2_PIX_FMT_BGR 24 'BGR3'	b7 b6 b5 b4 b3 b2 b1 b0	g7 g6 g5 g4 g3 g2 g1 g0	r7 r6 r5 r4 r3 r2 r1 r0	
V4L2_PIX_FMT_RGB 24 'RGB3'	r7 r6 r5 r4 r3 r2 r1 r0	g7 g6 g5 g4 g3 g2 g1 g0	b7 b6 b5 b4 b3 b2 b1 b0	
V4L2_PIX_FMT_BGR 32 'BGR4'	b7 b6 b5 b4 b3 b2 b1 b0	g7 g6 g5 g4 g3 g2 g1 g0	r7 r6 r5 r4 r3 r2 r1 r0	a7 a6 a5 a4 a3 a2 a1 a0
V4L2_PIX_FMT_RGB 32 'RGB4'	r7 r6 r5 r4 r3 r2 r1 r0	g7 g6 g5 g4 g3 g2 g1 g0	b7 b6 b5 b4 b3 b2 b1 b0	a7 a6 a5 a4 a3 a2 a1 a0

---

一个驱动到底支持哪种RGB格式，你可以去LinuxTV v4l-dvb代码库下个工具测试下。可以访问<http://linuxtv.org/repo/> 来获得更多信息。

## V4L2\_PIX\_FMT\_SBGGR8 ('BA81')

### Name

V4L2\_PIX\_FMT\_SBGGR8 --- Bayer RGB format

### Description

这通常是数字摄像机的本征 (native) 格式，用来表明 CCD 设备上传感器的布局的。每个像素点就一个红色或者绿色或者蓝色。缺失部分必须用邻近像素来内插补充。从左到右，第一行包含一个 blue 和 green，第二行是一个 green 和 red。这种组合会向右向下每隔一行一列进行排布。

#### Example 2-1. V4L2\_PIX\_FMT\_SBGGR8 4 × 4 pixel image

Byte Order. Each cell is one byte.

start + 0:	B <sub>00</sub>	G <sub>01</sub>	B <sub>02</sub>	G <sub>03</sub>
start + 4:	G <sub>10</sub>	R <sub>11</sub>	G <sub>12</sub>	R <sub>13</sub>
start + 8:	B <sub>20</sub>	G <sub>21</sub>	B <sub>22</sub>	G <sub>23</sub>
start + 12:	G <sub>30</sub>	R <sub>31</sub>	G <sub>32</sub>	R <sub>33</sub>

## V4L2\_PIX\_FMT\_SBGGR16 ('BA82')

### Name

V4L2\_PIX\_FMT\_SBGGR16— Bayer RGB format

### Description

这个格式类似于上面提到的 V4L2\_PIX\_FMT\_SBGGR8，不同的在于每个像素有 16bit 深度 (depth, 这...)。最低有效字节存储在低地址空间 (小端格式)。考虑到实际的采样精度可能低于 16bit，比如 10 比特的吧，每个像素的值可能就是 0 到 1023 了。

#### Example 2-1. V4L2\_PIX\_FMT\_SBGGR16 4 × 4 pixel image

Byte Order. Each cell is one byte.

start + 0:	B <sub>00low</sub>	B <sub>00high</sub>	G <sub>01low</sub>	G <sub>01high</sub>	B <sub>02low</sub>	B <sub>02high</sub>	G <sub>03low</sub>
start + 8:	G <sub>10low</sub>	G <sub>10high</sub>	R <sub>11low</sub>	R <sub>11high</sub>	G <sub>12low</sub>	G <sub>12high</sub>	R <sub>13low</sub>
start + 16:	B <sub>20low</sub>	B <sub>20high</sub>	G <sub>21low</sub>	G <sub>21high</sub>	B <sub>22low</sub>	B <sub>22high</sub>	G <sub>23low</sub>
start + 24:	G <sub>30low</sub>	G <sub>30high</sub>	R <sub>31low</sub>	R <sub>31high</sub>	G <sub>32low</sub>	G <sub>32high</sub>	R <sub>33low</sub>

## 2.5. YUV 格式(YUV Formats)

关于 YUV 格式介绍可以看看上面的 JPEG 简易文档...

### Packed YUV formats

#### Name

Packed YUV formats ---Packed YUV formats

#### Description

类似于 packed RGB 格式，这些格式用 16 或者 32bit 将每个像素点的 Y, Cb 和 Cr 存储起来。

Table 2-1. Packed YUV Image Formats (P<sub>49</sub>)

### V4L2\_PIX\_FMT\_GREY ('GREY')

#### Name

V4L2\_PIX\_FMT\_GREY— Grey-scale image

#### Description

这是一个灰度图。这是个弱化的 YCbCr 格式，其中简化掉了 Cb 或者 Cr 数据。

Example 2-1. V4L2\_PIX\_FMT\_GREY 4 × 4 pixel image

Byte Order. Each cell is one byte.

start + 0:	Y'00	Y'01	Y'02	Y'03
start + 4:	Y'10	Y'11	Y'12	Y'13
start + 8:	Y'20	Y'21	Y'22	Y'23
start + 12:	Y'30	Y'31	Y'32	Y'33

### V4L2\_PIX\_FMT\_Y16 ('Y16')

#### Name

V4L2\_PIX\_FMT\_Y16— Grey-scale image

#### Description

这也是个灰度图，只是其每像素点有 16bit 的深度。最低有效字节存储在低地址空间（小端格式）。考虑到实际的采样精度可能低于 16bit，比如 10 比特的吧，每个像素的值可能就是 0 到 1023 了。

Example 2-1. V4L2\_PIX\_FMT\_Y16 4 × 4 pixel image

Byte Order. Each cell is one byte.

start + 0:	Y'00low	Y'00high	Y'01low	Y'01high	Y'02low	Y'02high	Y'03low	Y'03high
------------	---------	----------	---------	----------	---------	----------	---------	----------

start + 8:	Y'10low	Y'10high	Y'11low	Y'11high	Y'12low	Y'12high	Y'13low	Y'13high
start + 16:	Y'20low	Y'20high	Y'21low	Y'21high	Y'22low	Y'22high	Y'23low	Y'23high
start + 24:	Y'30low	Y'30high	Y'31low	Y'31high	Y'32low	Y'32high	Y'33low	Y'33high

## V4L2\_PIX\_FMT\_YUYV ('YUYV')

### Name

V4L2\_PIX\_FMT\_YUYV— Packed format with ½ horizontal chroma resolution, also known as YUV4:2:2

### Description

此格式，每4个byte来表征2个像素。也即每4个byte就有2个Y，一个Cb和一个Cr。因为人眼对色彩度不是很敏感，所以就没必要有那么多Cb和Cr了。V4L2\_PIX\_FMT\_YUYV在Windows环境中是YUY2。

Example 2-1. **V4L2\_PIX\_FMT\_YUYV** 4 × 4 pixel image

Byte Order. Each cell is one byte.

start + 0:	Y'00	Cb00	Y'01	Cr00	Y'02	Cb01	Y'03	Cr01
start + 8:	Y'10	Cb10	Y'11	Cr10	Y'12	Cb11	Y'13	Cr11
start + 16:	Y'20	Cb20	Y'21	Cr20	Y'22	Cb21	Y'23	Cr21
start + 24:	Y'30	Cb30	Y'31	Cr30	Y'32	Cb31	Y'33	Cr31

下面有一堆 YUV 格式，详见 P<sub>53</sub>–P<sub>59</sub>

## 2.6. 压缩格式(Compressed Formats)

Table 2-7. Compressed Image Formats

Identifier	Code	Description
V4L2_PIX_FMT_JPEG	'JPEG'	TBD. See also VIDIOC_G_JPEGCOMP,  VIDIOC_S_JPEGCOMP.
V4L2_PIX_FMT_MPEG	'MPEG'	MPEG stream. The actual format is determined by extended control



## 2.7. 保留的格式标识符(Reserved Format Identifiers)

这些格式不是由本规范定义的，列出来只是为了参考和避免可能的命名冲突（你知道Linux绝大部分是开源的，其命名方式完全由项目作者决定，没有统一的标准，所以怕撞车）。接下来的部分主要是说你可以把你自己定制的格式上交给一个邮件列表，供大家参考...

Table 2-8. Reserved Image Formats

Identifier	C ode	Description
V4L2_PIX_FMT_DV	'dv sd'	unknown
V4L2_PIX_FMT_ET61 X251	'E 625'	Compressed format of the ET61X251 driver.
V4L2_PIX_FMT_HI240	'HI 24'	8 bit RGB format used by the BTTV driver, <a href="http://bytesex.org/bttv/">http://bytesex.org/bttv/</a>
V4L2_PIX_FMT_HM12	'H M12'	YUV 4:2:0 format used by the IVTV driver, <a href="http://www.ivtvdriver.org/">http://www.ivtvdriver.org/</a> The format is documented in the kernel sources in the file  Documentation/video4linux/cx2341x/README.hm12
Identifier	C ode	Description
V4L2_PIX_FMT_MJPE G	'M JPG'	Compressed format used by the Zoran driver
V4L2_PIX_FMT_PWC 1	'P WC1'	Compressed format of the PWC driver.
V4L2_PIX_FMT_PWC 2	'P WC2'	Compressed format of the PWC driver.
V4L2_PIX_FMT_SN9C	'S	Compressed format of the SN9C102 driver.

10X	910'	
V4L2_PIX_FMT_WNV	'W	Used by the Winnov Videum driver, <a href="http://www.thedirks.org/winnov/">http://www.thedirks.org/winnov/</a>
A	NVA'	
V4L2_PIX_FMT_YYUV	'Y	unknown
	YUV'	

To be continued...

## Chapter 3. 输入/输出(Input/Output)

V4L2 API 规定了若干个不同读写设备的方法。驱动和应用程序间进行数据交换必须支持其中的一种。

传统的 I/O 方法是打开 V4L2 设备后会自动的调用 `read()` 和 `write()` 函数。如果驱动没有支持这种方法，任何读写设备的尝试都将失败。

用其他的方法必须事先协商好。选用通过 `mmap` 或者用户空间缓冲区的流式 I/O 方法，应用程序可以调用 `VIDIOC_REQBUFS` 方法。目前，异步 I/O 方法还没定义呢（如果你有好的建议可以随时关注 V4L2 的邮件列表）。

`video overlay` 也是一种值得考虑的 I/O 方法，尽管应用程序并不会直接接收图像数据。调用 `VIDIOC_S_FMT` 可以初始化 `video overlay`。查看 4.2 节获得更多信息。

通常，确切的说任何一种 I/O 方法，当然也包括 `overlay`，都有他们对应的 `fd`。唯一相同的是应用程序并不直接和驱动进行数据交换（面板应用，详见 1.1 节），并且驱动允许使用同一个 `fd` 进行连续不间断的视频采集和 `overlay`，主要是为了和 V4L 以及 V4L2 的早期版本保持兼容。

`VIDIOC_S_FMT` 和 `VIDIOC_REQBUFS` 的允许(切换方法)程度是有限的，但为了简洁，驱动除了通过关闭和重新打开设备来切换 I/O 方法外无需再支持其它的切换方式。

接下来的几小节会详细讨论各种 I/O 方法。

### 3.1. 读和写(Read/Write)

`VIDIOC_QUERYCAP` 方法会返回结构 `v4l2_capability`，而其 `capabilities` 域中有个

---

V4L2\_CAP\_READWRITE 标志，如果被设置了，那么输入和输出设备会分别支持 read()和 write()函数。

驱动可能会借助 cpu 来拷贝数据，当然他们也可能支持 DMA 方式，因此该 I/O 方法并不一定没有其他通过交换缓冲区指针交换数据的方法来的低效。但这仍然属于低级的，原因在于没有传递过像帧计数器或者时间戳这样的元信息（meta-information）。这些个信息是相当必要的，因为他们可以用来确定帧是否被丢弃了或者用来同其他应用程序进行数据流同步。尽管如此，这也是一种最简单的进行 I/O 的方法，只需要请求少量或者不用进行设置就可交换数据。可以使用如下的命令行技巧来操作（其中 vidctrl 工具是假想的）：

```
> vidctrl /dev/video --input=0 --format=YUYV --size=352x288
```

```
> dd if=/dev/video of=myimage.422 bs=202752 count=1
```

使用 read() 函数进行读设备，write() 则进行写。假使驱动要和应用程序交换数据，它必须实现一种 I/O 方法，但这也不是必要的<sup>1</sup>。当读或者写可用时，驱动也必须支持 select() 和 poll ()<sup>2</sup>。

### 3.2. 流式 I/O(内存映射缓冲区)(Streaming I/O(Memory Mapping))

**内存映射缓冲区**(type 为 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 I/O 内存中开辟的缓冲区（如果硬件支持）。驱动编写见第五章。

就是说当标志位 V4L2\_CAP\_STREAMING 置 1 时，就会支持这种 I/O 方法。有俩种 streaming 方法，应用程序可通过调用 VIDIOC\_REQBUFS 来确定是不是支持内存映射。

streaming 是一种 I/O 方法，它主要是通过和应用程序交换缓冲区指针来交换数据，即不用拷贝数据了。内存映射主要是将设备内存缓冲映射到应用程序的地址空间去。设备内存可以是，就像独立显卡它有自己的存储区吧，这存储区就是设备内存。当然了最要效的还属 DMA 了。

一个驱动可能支持多集合 buffer。通过一独一无二的 buffer 类型值来辨识每一集合。它们是相互独立的并且可以拥有一个不同类型的数据。必须使用不同的 fd 来同时访问不同的集合<sup>3</sup>。

应用程序可以调用 VIDIOC\_REQBUFS 来分配想要的设备内存，只要提供 buffer 数和 buffer 类型（例子：V4L2\_BUF\_TYPE\_VIDEO\_CAPTURE）。当然上面的 ioctl 方法也可以用来更改 buffer 数或者释放分配的内存，如果仍有处于 mapped 状态，它可能不做什么改变。

应用程序在可以访问这些 buffer 之前还必须通过 mmap() (这驱动编程中很常见的) 将这些 buffer 映射到应用程序的地址空间去。可以用 VIDIOC\_QUERYBUF 来获得设备内存中的这些个 buffer 地址。mmap() 第六和第二个参数是 m.offset 和 length（由结构 v4l2\_buffer 返回），这俩参数可别瞎改。谨记一点，这些个 buffer 地址是物理地址而不是虚拟地址（内核空间，用户空间；物理地址，虚拟地址，进程地址... 不同架构范围是不同的）。资源有限，不用的就 munmap() 掉吧。

#### Example 3-1. Mapping buffers

```
struct v4l2_requestbuffers reqbuf;
```

```
struct {
```

```
void *start;
```

---

```

size_t length;

} *buffers;

unsigned int i;


memset (&reqbuf, 0, sizeof (reqbuf));

reqbuf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;

reqbuf.memory = V4L2_MEMORY_MMAP;

reqbuf.count = 20;


if (-1 == ioctl (fd, VIDIOC_REQBUFS, &reqbuf)) {

if (errno == EINVAL)

    printf ("Video capturing or mmap-streaming is not supported\n");

else

    perror ("VIDIOC_REQBUFS");

exit (EXIT_FAILURE);

}

/* We want at least five buffers. */

if (reqbuf.count < 5) {

/* You may need to free the buffers here. */

printf ("Not enough buffer memory\n");

exit (EXIT_FAILURE);

}


buffers = calloc (reqbuf.count, sizeof (*buffers));

assert (buffers != NULL);

for (i = 0; i < reqbuf.count; i++) {

struct v4l2_buffer buffer;

memset (&buffer, 0, sizeof (buffer));

buffer.type = reqbuf.type;

```

---

```

buffer.memory = V4L2_MEMORY_MMAP;

buffer.index = i;

if (-1 == ioctl (fd, VIDIOC_QUERYBUF, &buffer)) {

    perror ("VIDIOC_QUERYBUF");

    exit (EXIT_FAILURE);

}

buffers[i].length = buffer.length;          /* remember for munmap() */

buffers[i].start = mmap (  NULL, buffer.length,

                          PROT_READ | PROT_WRITE, /* recommended */

                          MAP_SHARED,             /* recommended */

                          fd, buffer.m.offset);

if (MAP_FAILED == buffers[i].start) {

    /* If you do not exit here you should unmap() and free()

       the buffers mapped so far. */

    perror ("mmap");

    exit (EXIT_FAILURE);

}

}

/* Cleanup. */

for (i = 0; i < reqbuf.count; i++)

munmap (buffers[i].start, buffers[i].length);

```

概念上来说 streaming 驱动会维护两个缓冲队列：传出队列和传入队列。这样可以减少因为其他进程抢占或者网络延迟带来数据丢失的可能性。队列被组织成 FIFO，输出用传入 FIFO，采集用传出 FIFO。

驱动可能请求一个最小数量的 buffer，除此别无数量上的限制，应用程序可以提前入列、出列或者数据处理。驱动也可以不同于 buffer 的出列顺序入列，并且驱动可以任意顺序填充入列的空

---

buffer<sup>4</sup>。结构 v4l2\_buffer 中的 index，在这里就起个标识 buffer 的作用。

驱动将所有 mapped 的 buffer 初始化为清空状态，且不可访问。对于采集应用程序来说通常的做法是，先填满所有的 buffers，接着开始采集并进入读循环。这里应用程序会一直等待，直到有 buffer 可以入列为止，并且当这些数据不再需要时，重新清除 buffer。输出应用程序会填充、入列 buffer，当有足够的 buffer 被堆满，输出就会启用 VIDIOC\_STREAMON。在写循环中，当应用程序消耗光 buffer 时，它必须等待直到有空的 buffer 可以出列和可被重新利用。

应用程序可以使用 VIDIOC\_QBUF 和 VIDIOC\_DQBUF 方法来入列和出列一个 buffer。buffer 的状态到底是 mapped，enqueued，full 还是 empty 的，可使用 VIDIOC\_QUERYBUF 来确定。有两种方法可以推迟执行应用程序直到有一个或更多的 buffer 可供 dequeued。通常当没有 buffer 处于外出队列时，VIDIOC\_DQBUF 方法会被阻塞。当设置 O\_NONBLOCK 标志位时，遇到没 buffer，VIDIOC\_DQBUF 会立即返回 EAGAIN (Error AGAIN，就是提醒你再试一次) 错误代码。这同样适用于 select() 和 poll() 函数。

可以调用 VIDIOC\_STREAMON 和 VIDIOC\_STREAMOFF 来开启采集或者进行输出。注意：VIDIOC\_STREAMOFF 有个副作用，就是会将俩队列中的所有 buffer 清除掉。在多任务系统中不知道那个进程正在运行，所以假若应用程序想和其他的时间保持同步时，最好检查一下结构 v4l2\_buffer 中关乎采集 buffer 的 timestamp 域，或者在为输出入列 buffer 时设置该域。

实现内存映射机制 I/O 的驱动必须支持如下方法：

VIDIOC\_REQBUFS, VIDIOC\_QUERYBUF, VIDIOC\_QBUF,  
VIDIOC\_DQBUF, VIDIOC\_STREAMON 和 VIDIOC\_STREAMOFF,

函数：

mmap(), munmap(), select() 和 poll。[capture example]

### 3.3. 流 I/O(用户空间缓冲区(用户指针))(Streaming I/O(User Pointers))

**用户空间缓冲区**(type 为 V4L2\_MEMORY\_USERPTR) 是在用户空间的应用中开辟缓冲区。很明显，在这种情况下是不需要 mmap() 调用的，但驱动为有效地支持用户空间缓冲区，其工作将会更困难。

当设置了 V4L2\_CAP\_STREAMING 标志位时，设备就支持该方法了。你必须通过调用 VIDIOC\_REQBUFS 方法来确定是否支持使用用户空间缓冲区的方式而不是内存映射方式。

这种方法结合了 read/write 和内存映射方法优点。buffer 是由应用程序自己申请的，且可以来自虚拟或者是共享内存。只有指针指向的数据才可被交换，这些个指针和元数据在结构 v4l2\_buffer 中传递。驱动只能通过调用 VIDIOC\_REQBUFS 方法来切换到用户指针模式的 I/O 操作。事先并没有 buffer (不同于 memory mapping 会提前申请俩个 buffer)，这样它就不可以索引或者通过使用 VIDIOC\_QUERYBUF 方法来查询。

#### Example 3-2. Initiating streaming I/O with user pointers

```
struct v4l2_requestbuffers reqbuf;  
  
memset (&reqbuf, 0, sizeof (reqbuf));
```

---

```

reqbuf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;

reqbuf.memory = V4L2_MEMORY_USERPTR;

if (ioctl (fd, VIDIOC_REQBUFS, &reqbuf) == -1) {

if (errno == EINVAL)

    printf ("Video capturing or user pointer streaming is not supported\n");

else

    perror ("VIDIOC_REQBUFS");

exit (EXIT_FAILURE);

}

```

buffer 的地址和大小都是通过 VIDIOC\_QBUF 来传递的。虽然 buffer 一般都是循环利用的，但是应用程序也可以在每次调用 VIDIOC\_QBUF 时传递不同的地址和大小。如果硬件需要，驱动可以在物理内存中交换内存页以达到内存区域的连续性。在内核的虚拟内存子系统中，这些变化对于应用程序来说都是透明的。当 buffer 页被交换到磁盘时，他们还会被交换回来最终为了 DMA（DMA 的内存申请不是相对固定的么：DMA 区，常规内存区，高端内存区）而被锁在物理内存中。

方法 VIDIOC\_DQBUF，正如其中的 D 预示的是 dequeued 那些填满的或者是要显示的 buffer。驱动可以在完成 DMA 和这个方法之间的任何时候解锁内存页。解锁内存也发生在如下情形中：调用 VIDIOC\_STREAMOFF 或者 VIDIOC\_REQBUFS 或者是关闭设备时。应用程序必须注意在未进行 dequeuing 前不要释放 buffer。第一点在于，这些 buffer 会浪费物理内存的。第二点是，当这些为了随后的复用或者其他目的被返回到应用程序的自由列表时，可能并没有通知驱动这些个更改。

通常对于采集程序来说会 enqueue 一些数量的空 buffer，然后启动采集接着进入到读循环。应用程序会一直等待直到填满的 buffer 可被 dequeued，当数据不再需要时会进行 re-enqueue。输出应用程序会填充和 enqueue buffer，当有足够的 buffer 被堆满时，会启动输出。在写循环中，当应用程序耗尽自由列表中的 buffer 时会等待，一直到有可 dequeued 和 reused 空的 buffer 为止。默认情况下，当没有 buffer 在输出队列时，VIDIOC\_DQBUF 会被阻塞。但要是 open() 以 O\_NONBLOCK 打开时，VIDIOC\_DQBUF 会立即返回 EAGAIN 的错误代码。这同样适用于 select() 和 poll() 函数。

可以调用 VIDIOC\_STREAMON 和 VIDIOC\_STREAMOFF 来开启或者停止采集或者输出。注意：VIDIOC\_STREAMOFF 有个副作用，就是会将俩队列中的所有 buffer 清除掉。在多任务系统中没有正在做的概念，所以假若应用程序向和其他的时间保持同步时，就好检查一下结构 v4l2\_buffer 中关乎采集 buffer 的 timestamp 域，或者在为输出入列 buffer 时设置该域。

实现内存映射机制 I/O 的驱动必须支持如下方法：

```

VIDIOC_REQBUFS, VIDIOC_QBUF,

VIDIOC_DQBUF, VIDIOC_STREAMON 和 VIDIOC_STREAMOFF,

```

函数：

```

select() 和 poll7。

```

### 3.4. 异步 I/O(Asynchronous I/O)

不好意思该方法等着优秀的你来定义呢...

### 3.5. 缓冲(Buffers)

一个 buffer 包含着被应用程序和驱动使用流式 I/O 方法来进行交换的数据。只有指针指向的 buffer 会被交换并且数据本身没有被复制。这些指针，包括元信息象时间戳什么的，都储存在结构 v4l2\_buffer 中，参见 VIDIOC\_QUERYBUF, VIDIOC\_QBUF 和 VIDIOC\_DQBUF 方法。

通常所谓的时间戳就是第一个传输的数据字节。但实际上，V4L2 API 覆盖的广泛的硬件对时间戳要求很精确。

类似的限制也体现在输出的时间戳上。

但需要注意的是，尽管对时间戳的要求较高，系统时间本身不是非常的稳定，会受到很多因素的影响<sup>8</sup>。

Table 3-1. struct v4l2\_buffer

__u32	index	Number of the buffer, set by the application. This field is only used for memory mapping I/O and can range from zero to the number of buffers allocated with the VIDIOC_REQBUFS ioctl (struct v4l2_requestbuffers count) minus one.
enum v4l2_buf_type		Type of the buffer, same as struct v4l2_format type or struct v4l2_requestbuffers type, set by the application.
__u32	bytesused	The number of bytes occupied by the data in the buffer. It depends on the negotiated data format and may change
	<input type="checkbox"/>	ith each buffer for compressed variable size data like JPEG images. Drivers must set this field when type



---

		refers to an input stream, applications when an output stream.
<code>__u32</code>	flags	Flags set by the application or driver, see Table 3-3.
<code>enum v4l2_field</code>	field	Indicates the field order of the image in the buffer, see Table 3-8. This field is not used when the buffer contains VBI data. Drivers must set it when type refers to an input stream, applications when an output stream.
<code>struct timeval</code>	timestamp	For input streams this is the system time (as returned by the <code>gettimeofday()</code> function) when the first data byte was captured. For output streams the data will not be displayed before this time, secondary to the nominal frame rate determined by the current video standard in enqueued order. Applications can for example zero this field to display frames as soon as possible. The driver stores the time at which the first data byte was actually sent out in the timestamp field. This permits applications to monitor the drift between the video and system clock.
<code>struct v4l2_timecode</code>	timecode	When type is <code>V4L2_BUF_TYPE_VIDEO_CAPTURE</code> and the <code>V4L2_BUF_FLAG_TIMECODE</code> flag is set in flags, this structure

---

		contains a frame timecode. In
		V4L2_FIELD_ALTERNATE mode the
		top and bottom field contain the same
		timecode. Timecodes are intended to
		help video editing and are typically
		recorded on video tapes, but also
		embedded in compressed formats like
		MPEG. This field is independent of the
		timestamp and sequence fields.
__u32	sequence	Set by the driver, counting the frames in
		the sequence.
		In V4L2_FIELD_ALTERNATE mode the top and bottom field
		have the same sequence number. The count starts at zero
enum v4l2_memory	memory	This field must be set by applications
		and/or drivers in accordance with the
		selected I/O method.
union	m	
	__u32	offset When memory is V4L2_MEMORY_MMAP
		this is the offset of the buffer from the
		start of the device memory. The value is
		returned by the driver and apart of
		serving as parameter to the mmap()
		function not useful for applications. See
		Section 3.2 for details.
	unsigned long	userptr When memory is
		V4L2_MEMORY_USERPTR this is a
		pointer to the buffer (casted to unsigned
		long type) in virtual memory, set by the
		application. See Section 3.3 for details.

__u32	length	Size of the buffer (not the payload) in bytes.
__u32	input	Some video capture drivers support rapid and synchronous video input changes, a function useful for example in video surveillance applications. For this purpose applications set the V4L2_BUF_FLAG_INPUT flag, and this field to the number of a video input as in struct v4l2_input field index.
__u32	reserved	A place holder for future extensions and custom (driver defined) buffer types V4L2_BUF_TYPE_PRIVATE and higher.

Table 3-2. enum v4l2\_buf\_type

(详见 P<sub>68</sub>)

Table 3-3. Buffer Flags

(详见 P<sub>68</sub>)

Table 3-4. enum v4l2\_memory

(详见 P<sub>69</sub>)

3.5.1. 时码(Timecodes)

结构 v4l2\_timecode 被设计用来保存 SMPTE 12M 或类似时码的。(struct timeval timestamps 存储在 v4l2\_buffer 的 timestamp 域中)。

Table 3-5. struct v4l2\_timecode

(详见 P<sub>70</sub>)

3.6. field 次序(Field Order)

需要分清逐行扫描视频还是隔行扫描视频。逐行扫描视频是一行接着一行的传送视频图像的。隔行扫描视频是将一张图片分成奇偶两个部分。交替的传送着一奇一偶。...

需要明白的是视频摄像头可不是一次曝光一张图片的,且很少分开传送帧的。事实上这里的 field 指的是设备会一次采集的两个不同实例。想想看有可能这前一 field 和下一 field 之间有个物体发生

---

移动了。这对应用程序分析一帧中的哪个 field 是旧的非常重要。这便是 temporal order。

当驱动以 field 接着 field 提供或者接受图片而不是逐行扫描时，对程序来说这些个 field 是如何关联帧的也是非常重要的。我们分成顶层和底层 field，这便是 spatial order。顶层 field 来自隔行(interlace)帧的第一行，第二行便是底层 field 的第一行。

但是因为这些 field 采集时是一个接着一个，那到底第一个属于 top 还是 bottom 是无关紧要的。任何俩个连续的 top/bottom 或者 bottom/top 都算作有效。仅当源来自逐行，比方说你正在将一个电影转变成视频，那俩中 field 就可能来自于同一个帧，这种情况下就按自然顺序办。

没必要凭感觉断定是 top field 还是 older field。到底 older field 中是否包含有 top 或 bottom 行，按照约定这是由视频标准决定的。在下面的图中会说明 temporal 和 spatial order 之间的区别。

所有视频采集和输出设备必须报告各自当前的 field 顺序。一些驱动可能支持切换不同的 field order，为此应用程序在调用 VIDIOC\_S\_FMT 之前应该初始化结构 v4l2\_pix\_format 中的 field 域。如果不希望变更 field order，就设置成 V4L2\_FIELD\_ANY(0)。

#### Table 3-8. enum v4l2\_field

(详见 P<sub>71</sub>)

#### Figure 3-1. Field Order, Top Field First Transmitted

(详见 P<sub>73</sub>)

#### Figure 3-2. Field Order, Bottom Field First Transmitted

(详见 P<sub>74</sub>)

### Notes

(比较重要！详见 P<sub>75</sub>)

---

## Chapter 4. 接口(Interface)

### 4.1. 视频采集接口(Video Capture Interface)

视频采集设备采样模拟信号并把数字化后的图片储存到内存中去。现在，几乎所有的设备都能进行 25fps 或者 30fps 的采集。利用这些个接口应用程序可以控制采集的进程或者把驱动中的图片移动到用户空间中去。

依照惯例，是通过字符设备文件来访问 V4L2 视频采集设备的 `/dev/video` 和 `/dev/video0 ~ /dev/video63`，其主设备号为 81，次设备号为 0 到 63。前者 `/dev/video` 一般是首选视频设备的一个符号链接。注意，输出设备同样使用这些个字符设备文件。

#### 4.1.1. 查询设备属性(Querying Capabilities)

支持视频采集接口的设备会设置结构 `v4l2_capability` 中 `capabilities` 域的 `V4L2_CAP_VIDEO_CAPTURE` 标志，其中 `v4l2_capability` 结构由 `VIDIOC_QUERYCAP` 返回。作为设备次要功能他们也可能支持 `video overlay`(标志: `V4L2_CAP_VIDEO_OVERLAY`)和 `raw VBI` 采集(flag:`V4L2_CAP_VBI_CAPTURE`)接口。他们必须至少支持一种 I/O 方法或者 `read/write` 或者 `streaming I/O`。`tuner` 和 `audio` 是可选的。

#### 4.1.2. 附加功能(Supplemental Functions)

视频采集设备最好按需支持音频输入，`tuner`，`control`，裁剪和缩放，`streaming` 等的 `ioctl` 方法。所有采集设备必须支持视频输入和标准的视频 `ioctl` 方法。

---

### 4.1.3. 图片格式协商(Image Format Negotiation)

采集结果是由修剪和图片格式参数决定的。前者选定要采集的区域，后者则决定图片的存储格式，i.e.是 RGB 还是 YUV 的，每像素占的 bit 数等等。他们一起定义了在处理中图片是如何进行缩放的。

为了允许 Unix 的工具链可以编程然后像读取普通文件一样读取设备，这些参数在 `open()`后是不会被重置的。好的 V4L2 应用程序能够确保它们可以得到它们想要的功能，包括修剪和缩放。

修剪的初始化至少应该将参数重置为默认值。在 1.11 节中已给出样例。

要查看当前的图片格式，应用程序只要用 `V4L2_BUF_TYPE_VIDEO_CAPTURE` 填充 `v4l2_format` 中的 `type` 字段，并将它传给 `VIDIOC_G_FMT` 方法即可。驱动会负责填充结构 `v4l2_format` 中 `fmt` 联合体的 `pix` 成员的。

可以通过像上面的方法来请求不同的参数，初始化 `v4l2_format` 中 `vbi` 成员结构或者只用 `VIDIOC_G_FMT` 返回的值通过 `VIDIOC_S_FMT` 来设置相应字段。原因在于驱动会返回恰当的用于初始化的参数。如同 `VIDIOC_S_FMT`，方法 `VIDIOC_TRY_FMT` 可以在不禁用 I/O 或者可能的因硬件准备而耗费的时间的情况下来了解硬件到底有哪些限制。

结构 `v4l2_pix_format` 在第二章中。可以参见规范以获得更多关于 `VIDIOC_G_FMT`, `VIDIOC_S_FMT` 和 `VIDIOC_TRY_FMT` 的讨论。视频采集设备必须实现 `VIDIOC_G_FMT` 和 `VIDIOC_S_FMT`，即使你将后者实现为忽略所有请求或者和前者一个样也行，`VIDIOC_TRY_FMT` 则是可选的。

### 4.1.4. 读取图片(Reading Images)

一个视频采集设备可能支持 `read()`函数和/或者 `streaming`(内存映射或者用户指针式)I/O，详见第 3 章的讨论。

## 4.2. Video Overlay Interface

首先介绍一下 video overlay，video overlay 不同于 video capture，是指不需要对 video 信号的帧进行 copy，直接将视频信号转化成显卡的 VGA 信号或者将捕获到的视频帧直接存放在显卡的内存中，具体过程就是将视频帧直接写入 framebuffer 中，不需要经过平台的处理。实际上它就是直接将视频数据写入 framebuffer，而没有经过处理。Video overlay 需要硬件的支持，必须是支持 video overlay 的 camera 才能使用这套 overlay interface。

因为 video overlay 直接使用 linux 的 framebuffer（关于 framebuffer 的相关驱动编写可参考链接：[http://www.360doc.com/content/07/0321/19/1880\\_407424.shtml](http://www.360doc.com/content/07/0321/19/1880_407424.shtml)）来显示捕获到的 image，所以和 capture 相比它更有效率，而不是将捕获到的 image 拷贝以后通过其他方式(android surfaceflinger)来显示。Video overlay 只用来 preview，又被称为 framebuffer overlay 或 previewing。<sup>link</sup>

Video overlay 和 Video capture 使用同样的字符设备文件：`/dev/video`，默认情况下对该文件的操作就是个 Video Capture，overlay 的功能只有在调用 `VIDIOC_S_FMT` 后才会有效。

如果驱动允许通过 `read/write`、`streaming IO` 同时进行 capture 和 overlay，那么这就不能保证能够按照原标准帧率来处理视频了。有可能这一帧被 capture 使用，下一帧就被 overlay 使用了。

规范建议应用程序应该使用不同的 fd 分别处理 capture 和 overlay，能同时进行 capture 和 overlay

---

的设备一般都是支持的（使用不同 fd）。当然了为了保持向前兼容，驱动也是允许俩个操作使用同一个 fd 的。

#### 4.2.1. 查询设备属性(Querying Capabilities)

支持 video overlay 接口的设备会设置结构 `v4l2_capability` 中 `capabilities` 域的 `V4L2_CAP_VIDEO_CAPTURE` 标志，其中 `v4l2_capability` 结构由 `VIDIOC_QUERYCAP` 返回。因而驱动必需能够支持 4.2.2 节中列出的功能，和可选支持功能。

#### 4.2.2. 附加功能(Supplemental Functions)

视频采集设备最好按需支持音频输入，tuner，control，裁剪和缩放，streaming 的 ioctl。所有采集设备必须支持视频输入和视频标准 ioctl。

#### 4.2.3. 设置(Setup)

在 overlay 操作可进行之前，应用程序必须提供必要的参数，比如 frame buffer 的地址及大小，图片格式—RGB 5:6:5，来编程驱动。通常这是通过 `VIDIOC_G_FBUF` 和 `VIDIOC_S_FBUF` 方法来分别获得和设置这些参数。需要注意的是后者由于可设置 DMA 到物理地址，出于保护内核机制，该操作需要超级用户权限。一般而言不会使用 root 账号或者具有粘着位的用户（让本没有权限访问某资源的用户可以访问该资源）来运行类似 TV 这般娱乐应用的。协议建议使用一个具有合适权限的中间件（辅助）程序在必要的时候对 V4L2 驱动进行参数设置。

有些设备会直接覆盖掉显卡的输出信号（缓存区）。这类视频设备是不会修改 frame buffer 的，并且驱动也不需要 frame buffer 的地址及像素格式。应用可以通过 `VIDIOC_G_FBUF` 来判断一个设备是否属于这种类型。

一个驱动可能支持下列一个或多个方法（请修改以下不合适的地方）：

1. 色度键控只显示初始图形像素有特定颜色的 overlaid image。
2. 可指定一位图，其每个 bit 代表在 overlaid image 中的一个像素。设置该 bit 位则显示相应的像素，否则不予显示。
3. 可指定一个修剪矩形框。在这个框内不显示视频，则可在该区域内看见 Graphic surface。
4. Framebuffer 有一个可以被用来修剪或粘合 framebuffer 和 video 的 alpha 通道。
5. 有一个全局的 alpha 值可以用来粘合 framebuffer 和 video images。

当驱动支持同时采集和 overlay，硬件却禁止不同的图像和 framebuffer 格式时，最先请求的格式具有更高的优先级。尝试采集或者 overlay 时有可能返回 EBUSY 错误或者直接返回修改过的参数。

#### 4.2.4. Overlay 窗口 (Overlay Window)

需要 overlay 的图片是由修剪及 overlay 窗口参数决定的。前者决定区域，后者决定如何 overlay 以及修剪修剪参数至少也是初始化时的默认值。在 1.11 节有示例程序。

规范使用 `v4l2_window` 结构描述 overlay 窗口。其定义了要被 overlay 图片的尺寸、图形表面位置以及该如何修剪。应用程序可以通过设置 `v4l2_format` 结构中 `type` 域为

---

V4L2\_BUF\_TYPE\_VIDEO\_OVERLAY, 然后调用 VIDIOC\_G\_FMT 方法来获得当前的一些参数。驱动负责填充 v4l2\_window 中的结构 *win*。一般不会返回上一次修改过的参数。

类似的, 可以通过填充相应的结构, 调用 VIDIOC\_S\_FMT 方法来设置你想要的参数。驱动会对所请求的参数进行相应的更改以适应硬件本身所具有的限制, 然后像 VIDIOC\_G\_FMT 一样返回实际的参数。和 VIDIOC\_S\_FMT 一样, 版本 VIDIOC\_TRY\_FMT 方法用于获得硬件所能提供的特性, 而不会改变驱动的状态(当前的参数), 如其名: TRY, 只是试试某个参数硬件是否支持而已。

overlay 图片的缩放比例 v4l2\_window 结构中相应宽与高以及修剪矩形框来表示。更多信息参看 1.11 节。

当驱动支持同时采集和 overlay, 硬件却禁止不同的图像和窗口大小时, 最先请求的格式具有更高的优先级。尝试采集或者 overlay 时有可能返回 EBUSY 错误或者直接返回修改过的参数(这个参数不行)。

**Table 4-1. struct v4l2\_window** (各个域的具体解释参见规范)

```
struct v4l2_window{
    struct v4l2_rect    w;
    struct v4l2_field field;
    __u32               chromakey;
    struct v4l2_clip* clips;
    __u32               clipcount;
    void *              bitma;
    __u8                global_alpha;
}
```

**Table 4-2. struct v4l2\_clip**

```
struct v4l2_clip{
    struct v4l2_rect c;
    struct v4l2_clip* next;
}
```

**Table 4-3. struct v4l2\_rect**

```
struct v4l2_rect{
    __s32    left;
    __s32    top;
    __s32    width;
    __s32    height;
}
```

#### 4.2.5. 使能 overlay (Enabling Overlay)

overlay 应用程序可以简单的通过 VIDIOC\_OVERLAY 方法来开启或者关闭 frame buffer 功能。



---

### 4.3. 视频输出接口(Video Output Interface)

视频输出设备会将静态或者图片序列编程模拟视频信号。通过该接口应用程序可以控制编码进程以及将图片从用户空间移动给驱动。按照惯例 V4L2 视频输出接口通过字符设备文件：/dev/video, /dev/video0~dev/video63, 主设备号为 81, 次设备号为 0~63。对于/dev/video 通常是一个符号链接。需要注意的是视频采集设备也使用同样的字符文件。

#### 4.3.1. 设备功能查询(Querying Capabilities)

通过调用 VIDIOC\_QUERYCAP 方法, 支持视频输出接口的设备驱动会用 V4L2\_CAP\_VIDEO\_OUTPUT 来设置 v4l2\_capability 结构中的 capabilities 域。这些设备通常也会支持 VBI 输出 (V4L2\_CAP\_VBI\_OUTPUT) 接口作为第二功能。这样的设备至少得支持一种读写或者流式 IO 方法。调制和音频输出则是可选的。

#### 4.3.2. 附加功能 (Supplemental Functions)

视频输出设备可按需支持音频输出, 调制, 控制, 修剪及缩放和流式参数等 ioctl 方法。但是, 视频输出和视频标准 ioctl 方法必须得到支持。

#### 4.3.3. 图片格式协商 (Images Format Negotiation)

输出结果是由修剪和图片格式参数决定的。前者选定图片要显示的视频图片区域, 后者则决定图片的存储格式, i.e.是 RGB 还是 YUV 的, 每像素占的 bit 数等等。他们一起定义了在处理中图片是如何进行缩放的。

为了允许 Unix 的工具链可以编程然后像读取普通文件一样读取设备, 这些参数在 open()后是会被重置的。好的 V4L2 应用程序能够确保它们可以得到它们想要的功能, 包括修剪和缩放。

修剪的初始化至少应该将参数重置为默认值。在 1.11 节中已给出样例。

要查看当前的图片格式, 应用程序只要用 V4L2\_BUF\_TYPE\_VIDEO\_OUTPUT 填充 v4l2\_format 中的 type 字段, 并将它传给 VIDIOC\_G\_FMT 方法即可。驱动会负责填充结构 v4l2\_format 中 fmt 联合体的 pix 成员的。

可以通过像上面的方法来请求不同的参数, 初始化 v4l2\_format 中 vbi 成员结构或者只用 VIDIOC\_G\_FMT 返回的值通过 VIDIOC\_S\_FMT 来设置相应字段。原因在于驱动会返回恰当的用于初始化的参数。如同 VIDIOC\_S\_FMT, 方法 VIDIOC\_TRY\_FMT 可以在不禁用 I/O 或者可能的因硬件准备而耗费的时间的情况下来了解到底有哪些硬件上的限制。

结构 v4l2\_pix\_format 在第二章中。可以参见规范以获得更多关于 VIDIOC\_G\_FMT, VIDIOC\_S\_FMT 和 VIDIOC\_TRY\_FMT 的讨论。视频输出设备必须实现 VIDIOC\_G\_FMT 和 VIDIOC\_S\_FMT, 即使你将后者实现为忽略所有请求或者和前者一个样也行, VIDIOC\_TRY\_FMT 则是可选的。

#### 4.3.4. 写图片 (Writing Image)

视频输出接口可能支持写函数以及流式 IO 操作。详情请参考第三章。

---

## 4.4. 视频输出 overlay 接口 (Video Output Overlay Interface)

注意：这是一项实验性功能接口！

至于和 Video Output 、 Video Overlay 的区别请参考：  
<http://blog.csdn.net/kickxxx/article/details/7755127>（似乎和以上的章节有出入，翻译问题）

有些视频输出设备可以使用一个 framebuffer 图片覆盖到正在往外发送的视频信号上。应用程序可以借助视频 overlay 接口的结构以及 ioctl 方法实现这样的功能。

OSD

### 4.4.1. 设备功能查询 (Querying Capabilities)

通过调用 VIDIOC\_QUERYCAP 方法，支持视频输出接口的设备驱动会用 V4L2\_CAP\_VIDEO\_OUTPUT\_OVERLAY 来设置 v4l2\_capability 结构中的 capabilities 域。

### 4.4.2. 帧缓冲 (Framebuffer)

不同于视频 overlay 接口，帧缓冲一般都用在 TV 卡而不是视频卡上。在 Linux 中，他被抽象成了帧缓冲设备--/dev/fbN。给定一个 V4L2 设备，应用可以通过调用 VIDIOC\_GET\_FBUF 方法来获得该设备对应的帧缓冲区。该 ioctl 方法会返回帧缓冲的物理地址，存储在 v4l2\_framebuffer 结构的 base 域中。帧缓冲设备的 FBIOGET\_FSCREENINFO 方法可以返回同样的地址，只是其存储在 fb\_fix\_screeninfo 结构的 smem\_start 域中。他们都定义在头文件 linux/fb.h 中。

帧缓冲去的大小有当前的视频标准决定。V4L2 驱动只会当所有使用帧缓冲区的应用关闭后才可能更改视频标准，其余情况只会返回 EBUSY 错误代码。

#### Example 4-1. Finding a framebuffer device for OSD

```
#include <linux/fb.h>
struct v4l2_framebuffer fbuf;
unsigned int i;
int fb_fd;
if (-1 == ioctl (fd, VIDIOC_G_FBUF, &fbuf)) {
    perror ("VIDIOC_G_FBUF");
    exit (EXIT_FAILURE);
}
for (i = 0; i < 30; ++i) {
    char dev_name[16];
    struct fb_fix_screeninfo si;
    snprintf (dev_name, sizeof (dev_name), "/dev/fb%u", i);
    fb_fd = open (dev_name, O_RDWR);
    if (-1 == fb_fd) {
        switch (errno) {
            case ENOENT: /* no such file */
            case ENXIO: /* no driver */
```

---

```

        continue;
    default:
        perror ("open");
        exit (EXIT_FAILURE);
    }
}

if (0 == ioctl (fb_fd, FBIOGET_FSCREENINFO, &si)) {
    if (si.smem_start == (unsigned long) fbuf.base)
        break;
    } else {
        /* Apparently not a framebuffer device. */
    }
    close (fb_fd);
    fb_fd = -1;
}

/* fb_fd is the file descriptor of the framebuffer device
   for the video output overlay, or -1 if no device was found. */

```

#### 4.4.3. overlay 窗口和缩放

**overlay** 操作由源和目标矩形框定义的。源框选择帧缓冲去的子区域用于图片进行 **overlay**，目标框则圈定图片要出现在输出视频信号的位置。驱动可也许支持也许不支持缩放，矩形块的任意大小和位置。更进一步，驱动可能支持或者不支持视频 **overlay** 接口中定义的一些方法。

**v4l2\_window** 结构定义了源矩形框，在帧缓冲中的位置以及 **overlay** 会调用的一些方法。应用程序可以通过设置 **v4l2\_format** 结构中 **type** 域为 **V4L2\_BUF\_TYPE\_VIDEO\_OUTPUT\_OVERLAY**，然后调用 **VIDIOC\_G\_FMT** 方法来获得当前的一些参数。驱动负责填充 **v4l2\_window** 中的结构 **win**。一般不会返回上一次修改过的参数。

类似的，可以通过填充相应的结构，调用 **VIDIOC\_S\_FMT** 方法来设置你想要的参数。驱动会对所请求的参数进行相应的更改以适应硬件本身所具有的限制，然后像 **VIDIOC\_G\_FMT** 一样返回实际的参数。和 **VIDIOC\_S\_FMT** 一样，版本 **VIDIOC\_TRY\_FMT** 方法用于获得硬件所能提供的特性，而不会改变驱动的状态(当前的参数)，如其名：**TRY**，只是试试某个参数硬件是否支持而已。不同于 **VIDIOC\_S\_FMT**，后者还可以工作在 **overlay** 已经使能的情形下。

结构 **v4l2\_crop** 则定义了目标矩形块的大小及位置。缩放比例由 **v4l2\_window** 结构中相应宽与高以及 **v4l2\_crop** 来表示。正如修剪 API 以应用在视频采集和视频 **overlay** 设备一样，API 也以相同方式应用到视频输出和视频输出 **overlay** 设备上了，且极少会反转数据流。更多信息参看 1.11 节。

#### 4.4.4. 使能 overlay (Enabling Overlay)

**V4L2** 并未定义使能或者禁用该 **overlay** 功能的 **ioctl** 方法，但是，驱动的帧缓冲接口可能会提供 **FBIOLBLANK** 方法。

---

## 4.5. 编码接口 (Codec Interface)

**Suspended:** 自 2.6 内核起, 该接口由于种种原因而被暂停。

Codec 接口 api 可以提供在内存中的压缩, 解压缩, 转换或者视频数据格式转变。应用程序可以通过 `write()` 向驱动发送要转变的数据, 或者通过 `read()` 从驱动处接受已被转杯的数据。出效率考虑, 驱动很可能支持流式 I/O。

## 4.6. 特效设备接口 (Effect Devices Interface)

**Suspended:** 自 2.6 内核起, 该接口由于种种原因而被暂停。

## 4.7. 原始 VBI 数据接口 (Raw VBI Data Interface)

VBI 是 Vertical Blanking Interval (场消隐区) 的缩写。在模拟显示设备上 (例如: CRT 显示器), 一场的显示是从屏幕上方到下方一行一行绘制而成。一旦一场数据绘制完毕, 电子束会由下方转移到上方, 以备开始下一场数据的显示。在这个电极复位的时间间隙内, CRT 和信号间是没有数据传输的, 因此称作场消隐区。在场消隐区时间内, 不需要传输任何有用的视频数据, 所以显示器不接受和处理任何的输入视频信号。在这个间隔内, 设备间可以利用空闲的基带进行额外的数据通讯。为了让这个问题简单化, 水平扫描线在场消隐区时继续传输, 但是扫描线上不附带任何可供显示的视频数据。此时, 每一个 VBI 扫描线, 或者一组 VBI 数据扫描线可能包含有按照某种标准进行编码后的信息。最常用的 VBI 应用就是在闭路电视上, 用于显示字幕。

不管有没有电子耦合视频信号, VBI 数据持续传输, 并且编码成为整个视频信号数据的一部分。VBI 数据可以通过任何视频信号的媒体载体传输 (比如: 无线电波, 光纤, 卫星, 线缆等) [\[vbi\]](#)。

## 4.8. 分离 VBI 数据接口 (Sliced VBI Data Interface)

接收到 VBI 数据的硬件设备, 只需要去完完整整的接纳数据, 并且对接收到的数据不做任何的数据处理, 这个称之为原始数据。这些原始数据可能直接提交给软件解析其内容。

另外一种方式是用 VBI 硬件按照某种标准或者类型去解析 VBI 数据, 在这种方式下, 信号的输出就被分离成为闭路电视字幕, 或者其他类型的数据包。

典型的 VBI 数据分离电路支持都支持这两种模式。自从 VBI 硬件处理电路中添加了数据分离, 错误处理等功能后, Sliced 模式变得更加有用。Raw VBI Data (原始 VBI 数据) 主要用于没有任何数据标准, 且信号直接送递显示设备的场合 [\[vbi\]](#)。

## 4.9. 图文电视接口 (Teltext Interface)

图文电视是在模拟电视系统中, 电视屏幕每秒显示 25 帧电视信号, 每帧 625 行, 每行从屏幕左侧扫到右侧, 每帧分两场从屏幕上边扫到下边。每帧 625 行中实际显示在屏幕上的只有 575 行, 还有 50 行是逆程, 是看不到的。逆程通常除了用来传输测试信号外还可用来传输额外的数据信息, 包括图形、文字。在接收端观众使用专用的图文电视解码器可以在屏幕上收看到所传送的信息 [\[Teltext\]](#)。

linux 提供了专门的 Teltext API, 其由 Martin Buck 设计提交。内核中的头文件为 `linux/videotext.h`,

---

可以从 <http://home.pages.de/~videotext> (已经打不开了)。Linux 提供的图文电视接口设备文件是 `/dev/vtx` 和 `/dev/vttuner`，其相应的主、次设备号是(83, 0)，次设备号是(83, 16)。

Teltext 的 API 也被集成在了 V4L 中了使用的设备文件为 `/dev/vtx0~dev/vtx31`，而其主设备号为 81，次设备号为 192~223。在 2.6 中将会移除为 VTX 提供的 API。

#### 4.10. 无线电接口 (Radio Interface)

该接口适用于 AM 和 FM (模拟) 无线接收器。

Linux 中 V4L2 提供的无线电接口设备文件是 `/dev/radio`，`/dev/radio0~dev/radio63`，其主设备号是 81，次设备号是 64~127。

Page 95

#### 4.11. 无线数据广播系统 (Radio Data Interface)

它是在调频广播发射信号中利用副载波把电台名称、节目类型、节目内容及其它信息以数字形式发送出去。通过具有 RDS 功能的调谐器就可以识别这些数字信号，变成字符显示在显示屏上。在收到节目的同时，通过 RDS 可知道接收到的是那个电台，它的发射频率，并给出该电台其余的频率，由此再使用“切换频率”钮来保证所接收的信号为最强的频率 [\[rds\]](#)。

#### Notes

Page 96

---

## Chapter 5. 驱动编写

本规范所附关乎 V4L2 驱动编写的第五章节来自大神 [Tekkaman Ninja](#)，/膜拜。



# Video for Linux Two (V4L2)

## ——驱动编写指南

原始翻译稿: [Video4Linux2 \(usr 技术社区 译文组\)](#)

原文地址: [The Video4Linux2](#)

整理: [Tekkaman Ninja](#)

2012-8-17



声明: 本文是基于 <http://lwn.net/> 网站上的经典系列文章 [《Video4Linux2》](#) 的翻译整理。

原始翻译来自 [usr 技术社区 译文组](#) 和 [雷宏亮的博文](#)。

本文因个人学习需要顺手做整理、修正, 发布此文档仅为方便广大 Linux 爱好者。

## 目录

一、	API 介绍 .....	3
二、	注册和 open() .....	4
1.	视频设备注册 .....	5
2.	open() 和 release() .....	5
三、	基本 ioctl()处理 .....	7
四、	输入和输出 .....	9
1.	视频标准 .....	9
2.	输入 .....	10
3.	输出 .....	11
五、	颜色与格式 .....	12
1.	色域 .....	12
2.	密集存储和平面存储 .....	12
3.	四字符码 .....	13
4.	RGB 格式 .....	13
5.	YUV 格式 .....	14
6.	其他格式 .....	15
7.	格式描述 .....	15
六、	格式协商 .....	16
七、	基本的帧 I/O .....	19
1.	read() 和 write() .....	19
2.	流参数 .....	19
八、	流 I/O .....	22
1.	v4l2_buffer 结构体 .....	22
2.	缓冲区设定 .....	24
3.	将缓冲区映射到用户空间 .....	25
4.	流 I/O .....	26
九、	控制 .....	28



# 一、API 介绍

笔者最近有机会写了一个用于“One Laptop Per Child”项目中的摄像头驱动。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, 而实际上[用户层的文档](#)则写得相当不错。为了补救这一现状, 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, 在发布 [2.5.46](#) 时, 融入了主线内核之中。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 旨在支持多种设备, 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最有经验, 这一系列文章也趋于强调视频捕获 API, 但 V4L2 的应用不限于此。
- **video output interface (视频输出接口)** 允许驱动 PC 外设, 使其向外输出视频图像, 而这可能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。
- **radio interface (广播接口)** 用于从 AM 或 FM 调谐器中获取音频数据。

也可能出现其它种类的设备。V4L2 API 中还有一些关于编解码器和效果设备的桩函数 (stub), 他们都用来转换视频数据流。然而这部分还尚未形成规范, 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, 它们是用来处量一些共性的任务的。

随后的文章每几周发一篇, 都会加入到下面的列表中。

- [Part 2: 注册和 open\(\)](#)
- [Part 3: 基本 ioctl\(\)处理](#)
- [Part 4: 输入和输出](#)
- [Part 5a: 颜色与格式](#)
- [Part 5b: 格式协商](#)
- [Part 6a: 基本的帧 I/O](#)
- [Part 6b: 流 I/O](#)
- [Part 7: 控制](#)

## 二、注册和 open()

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](#)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：

- [V4L2 API 规范](#)。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），放在触手可及的地方。
- 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，它就成了一个教人编写 V4L2 驱动的极佳实例。

首先，每个 V4L2 驱动都要包含一个必须的头文件：

```
#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 或 USB 总线，这里我们不会花时间来触及这些东西。通常会有一个内部 I2C 接口，我们在本系列的后续文章中会接触到它。然后还有一个 V4L2 子系统接口，这个子系统是围绕 `video_device` 结构体建立的，它代表一个 V4L2 设备。讲解这个结构体中的一切，将会是这个系列中几篇文章的主题。这里我们先对这个结构体有一个概览。

`video_device` 结构体的 `name` 字段是这类设备的名字，它将出现在内核日志和 `sysfs` 中出现。这个名字通常与驱动名称相同。

设备类型则由两个字段来描述。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。

如果你的设备支持不只一个上面提到的功能，那就要为每个功能注册一个 V4L2 设备。而在 V4L2 中，每个注册的设备都可以按照它实际支持的各种模式来调用（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何功能）。实质上，在 V4L2 中，你实际上只需一个设备，而注册多个 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。它可包含如下值：

- `VID_TYPE_CAPTURE` 视频数据捕获
- `VID_TYPE_TUNER` 不同的频率调整
- `VID_TYPE_TELETEXT` 字幕抓取
- `VID_TYPE_OVERLAY` 可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` 剪辑覆盖数据
- `VID_TYPE_FRAMERAM` 使用帧缓冲区中的内存空间
- `VID_TYPE_SCALES` 视频缩放
- `VID_TYPE_MONOCHROME` 纯灰度设备
- `VID_TYPE_SUBCAPTURE` 捕获图像子区域

- **VID\_TYPE\_MPEG\_DECODER** 支持 mpeg 码流解码
- **VID\_TYPE\_MPEG\_ENCODER** 支持编码 mpeg 码流
- **VID\_TYPE\_MJPEG\_DECODER** 支持 mjpeg 解码
- **VID\_TYPE\_MJPEG\_ENCODER** 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(笔者发现一件有趣的事,就是这个抱怨涉及到冒犯一篇 LWN 文章作者)。**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**<sup>[8]</sup>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),可以生成很多调试信息,它们可以帮助迷糊的程序员找到毛病,知道驱动和应用之间的沟通障碍在哪。

译者注:新内核中, **video\_device** 结构体中的这个 **file\_operations** 应该指的是 **v4l2\_file\_operations**。

## 1. 视频设备注册

一旦 **video\_device** 已经配置好,就可以用下面的函数注册了:

```
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** 是期望的子设备号(为-1 则注册时自动分配)。成功则返回值为 0。若出错,则返回的是负的出错码。我们必须意识到:通常设备一旦注册,它的函数可能就会被立即调用,所以在一切准备就绪前,不要调用 **video\_register\_device()**。

设备注销函数为:

```
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。

## 2. 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,就是通过找到 **inode** 中存存储的子设备号来完成的。这里还可以实现一定程度的初始化,如果有掉电选项的话,此时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的（特别是数据读、写等），但是整个设备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，总的说来，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（例如）从摄像头获取上帧图像。如果摄像头的设置在中途改变了，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()`函数中要减小引用计数，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，则函数很可能要关掉 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，请继续关注。

### 三、基本 ioctl() 处理

如果在 [video4linux2 API 规范](#) 上花点时间的话，你肯定已经注意到了一个问题：V4L2 大量使用 ioctl() 接口。视频硬件有大量的可操作旋钮（配置寄存器），可能比其它任何外设都要多。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。在硬件支持良好的模式范围外操作轻则可能导致性能不佳，但通常是根本无法使用。所以我们不得不揭露许多硬件特性，而对最终使应用表现得有点怪异。

传统上来说，视频驱动中包含的 ioctl() 函数可能会长得像一部 [尼尔·斯蒂芬森](#) 的小说，而函数的结果也往往比小说更令人满意，但他们通常在代码文件中占了大量篇幅<sup>[1]</sup>。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，多数驱动并不须实现所有回调，甚至不须实现大部分回调。

*译者注：ioctl() 函数通常用 switch-case 语句实现。如果不使用一个大回调函数集来替换 ioctl() 中的处理功能，会导致 ioctl() 变得冗长，对于维护和阅读极为不利。*

现在的情况是，冗长的 ioctl() 函数已经被移到了 drivers/media/video/videodev.c 文件中。这部分代码处理数据在内核和用户空间之间的传输并把单个 ioctl() 调用传递给驱动处理。若要使用它，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，而驱动也应该在适当的地方加锁。

驱动要实现的第一个回调函数可能是：

```
int (*vidioc_querycap)(struct file *file, void *fh, struct v4l2_capability *cap);
```

这个函数处理 **VIDIOC\_QUERYCAP** 的 ioctl()，只是简单问问“你是谁？你能干什么？”实现它是 V4L2 驱动的责任。和所有其他 V4L2 回调函数一样，这个函数中的参数 **priv** 是 **file->private\_data** 域的内容，通常的做法是在 open() 的时候把它指向驱动中表示设备的内部结构体。

驱动应该负责填充 **cap** 结构并且返回“0 或负的错误码”值。如果成功返回，则 V4L2 层会负责把回复拷贝到用户空间。

**v4l2\_capability** 结构（定义在 <linux/videodev2.h> 中）是这样的：

```
struct v4l2_capability {
    __u8 driver[16];          /* i.e. "bttv" */
    __u8 card[32];           /* i.e. "Hauppauge WinTV" */
    __u8 bus_info[32];       /* "PCI:" + pci_name(pci_dev) */
    __u32 version;           /* should use KERNEL_VERSION() */
    __u32 capabilities;      /* Device capabilities */
    __u32 reserved[4];
};
```

其中 **driver** 域应该被填充设备驱动的名字，**card** 域应该被填充这个设备的硬件描述信息。并不是所有的驱动都消耗精力去处理 **bus\_info** 域，这些驱动通常使用下面这个方法：

```
springf(cap->bus_info, "PCI:%s", pci_name(&my_dev));
```

**version** 域用来保存驱动的版本号。**capabilities** 域是一个位掩码用来描述驱动能做的不同事情：

```
/* Values for 'capabilities' field */
#define V4L2_CAP_VIDEO_CAPTURE      0x00000001 /* Is a video capture device */
#define V4L2_CAP_VIDEO_OUTPUT      0x00000002 /* Is a video output device */
#define V4L2_CAP_VIDEO_OVERLAY     0x00000004 /* Can do video overlay */
```

```

#define V4L2_CAP_VBI_CAPTURE          0x00000010      /* Is a raw VBI capture device */
#define V4L2_CAP_VBI_OUTPUT           0x00000020      /* Is a raw VBI output device */
#define V4L2_CAP_SLICED_VBI_CAPTURE  0x00000040      /* Is a sliced VBI capture device */
#define V4L2_CAP_SLICED_VBI_OUTPUT    0x00000080      /* Is a sliced VBI output device */
#define V4L2_CAP_RDS_CAPTURE          0x00000100      /* RDS data capture */
#define V4L2_CAP_VIDEO_OUTPUT_OVERLAY 0x00000200      /* Can do video output overlay */
#define V4L2_CAP_HW_FREQ_SEEK         0x00000400      /* Can do hardware frequency seek */
#define V4L2_CAP_TUNER                 0x00010000      /* has a tuner */
#define V4L2_CAP_AUDIO                 0x00020000      /* has audio support */
#define V4L2_CAP_RADIO                 0x00040000      /* is a radio device */
#define V4L2_CAP_READWRITE             0x01000000      /* read/write systemcalls */
#define V4L2_CAP_ASYNCIO              0x02000000      /* async I/O */
#define V4L2_CAP_STREAMING             0x04000000      /* streaming I/O ioctls */

```

最后一个域 **reserved** 是保留的。V4L2 规则要求 reserved 被置为 0。但因为 video\_ioctl2() 设置了整个的结构体为 0，所以这个就不用我们操心了。

可以在“vivi”这个驱动中找到一个典型的应用：

```

static int vidioc_querycap(struct file *file, void *priv, struct v4l2_capability *cap)
{
    struct vivi_fh *fh = priv;
    struct vivi_dev *dev = fh->dev;
    strcpy(cap->driver, "vivi");
    strcpy(cap->card, "vivi");
    strncpy(cap->bus_info, dev->v4l2_dev.name, sizeof(cap->bus_info));
    cap->version = VIVI_VERSION;
    cap->capabilities = V4L2_CAP_VIDEO_CAPTURE |
                      V4L2_CAP_STREAMING |
                      V4L2_CAP_READWRITE;

    return 0;
}

```

考虑到这个回调函数的出现，我们希望应用程序使用它，避免要求设备完成它们不可能完成的功能。然而，在编程中，应用程序不会花太多的精力来关注 **VIDIOC\_QUERYCAP** 调用。

另一个可选而又不常被实现的回调函数是：

```
int (*vidioc_log_status)(struct file *file, void *fh);
```

这个函数用来实现 **VIDIOC\_LOG\_STATUS** 调用，作为视频应用程序编写者的**调试助手**。当调用时，它应该打印描述驱动及其硬件的当前状态信息。这个信息应该足够充分以便帮助迷糊的应用程序开发者弄明白为什么视频显示一片空白。

下一部分开始介绍剩下的 77 个回调函数。尤其是我们要开始看看与硬件协商一系列操作模式的冗长过程。



## 四、 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](#)的，也许想从那里开始。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，并且在它们之间做出选择。

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。比如摄像头控制器，可能只是提供摄像头信号输入，而没什么别的功能；然而，在一些其他的情况下，事情就变得复杂了。一个电视卡板上不同的接口可能对应不同的输入。他甚至可能拥有可独立发挥功能的多路调谐器。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，一个应用若想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，驱动不应在应用退出时重置输入输出信息。在多次打开之间，对于其他视频参数也应维持不变。

### 1. 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如何为传输而进行格式化——分辨率、帧率等。这些标准通常是由每一个国家的监管部门制定的。现在世界上使用的标准主要有三个: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)。然而这些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 的定义为 `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有 NTSC 变种，它就可以设为 `V4L2_STD_NTSC`，它将所有相关位置位。对 PAL 和 SECAM 标准，也存在一组类似的位集。[这个网页](#)有完整的列表。

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 层会向应用回复所支持的标准。`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` 来反映现实情况。有些应用即使他并没有设置过标准，发现标准没有被设置也会感到困惑。

当某个应用想要申请某个特定标准时，会发出一个 `VIDIOC_S_STD` 调用，该调用传到驱动时通过下面的回调函数实现：

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

驱动要对硬件编程，以使用特定的标准，并返回 0(或是负的错误码)。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。

注意：所有视频设备都必须支持（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，它不与任何特定规范绑定。但是也不存在一个标准说“我是个摄像头，我什么都能做”，所以 V4L2 层有很多摄像头声明返回 PAL 或 NTSC 数据（译者注：实际只是如些声明而已）。

## 2. 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 对应要打开的视频设备。private\_data 是驱动的私有字段。input 字段是传递的真正信息，它有如下几个值得关注的字段：

- **\_\_u32 index:** 应用关注的输入索引号；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，依次增加。想要知道所有可用的输入，应用会调用 VIDIOC\_ENUMINPUT，索引号从 0 开始，并开始递增。一旦驱动返回-EINVAL，应用就知道：输入已经枚举完了。只要有输入，输入索引号 0 就一定存在。
- **\_\_u8 name[32]:** 输入的名称，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消息相符合。
- **\_\_u32 type:** 输入类型，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哪些视频输入相关联。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今后的文章中讲述。
- **v4l2\_std\_id std:** 描述设备支持哪个或哪些视频标准。
- **\_\_u32 status:** 给出输入状态。完整的标识符集合可以在 [V4L2 文档](#) 中找到；简而言之，status 中设置的每一位都代表一个问题。这些问题包括掉电、无信号、失锁、存在 Macrovision 模拟防拷贝系统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，应返回-EINVAL。这个调用里可能出现的错误不多。

当应用想改变当前输入时，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——它用来确定哪个输入是想要的。驱动要对硬件操作，选择指定输入并返回 0。也有可能要返回-EINVAL(索引号不正确) 或-EIO(硬件故障)。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，指示哪一个输入处在激活状态：

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

这里驱动把\*index 值设为当前激活输入的索引号。



### 3. 输出

枚举和选择输出的过程与输入十分相似。所以这里描述从简。

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 结构的字段成员如下：

- \_\_u32 index: 相关输入索引号。其工作方式与输入的索引号相同：它从 0 开始递增。
- \_\_u8 name[32]: 输出的名称。
- \_\_u32 type: 输出类型。支持的输出类型：
  - 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，
  - V4L2\_OUTPUT\_TYPE\_ANALOGVGAOVERLAY 用于模拟 VGA 覆盖设备。
- \_\_u32 audioset: 能与视频协同工作的音频集。
- \_\_u32 modulator: 与此设备相关的调制器(仅对类型为 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，他们与输入的回调对应：

```
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。然而选译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商数据格式。

## 五、 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](#)的，也许想从那里开始。

应用在视频设备可以工作之前，它必须与驱动达成一致，知道视频数据是何种格式。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各不相同。
- 2、 在内核的格式转换是令人难以接受的。

所以应用要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。

### 1. 色域

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

- V4L2\_COLORSPACE\_SRGB

多数开发者所熟悉的 [red、green、blue]数组就包含在这个色域中。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，从而产生了丰富的颜色。表示 RGB 值的方法有很多，我们在下面将会介绍。

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“亮度”）值只是一个简单的亮度值，单独显示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 之间的转换并不那么直接，但是我们有一些[公式](#)可用。

**注意：YUV 和 YCbCr 并非完全一样，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。

### 2. 密集存储和平面存储

如上所述，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分量值连续存放在一起。
- Planar 格式把每一个分量单独存储成一个阵列。例如在 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分中,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。

### 3. 四字符码

（译者注：four Charactor Code: FourCC）

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。例如，当一个色彩格式读作“RGB4”就没有必要去查表了。

注意：四字符码在很多不同的配置中都会使用，有些还是早于 linux。Mplayer 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。

### 4. RGB 格式

在下面的格式描述中，字节按存储顺序列出的。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				
bayer pattern	bayer pattern				

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贝尔图(bayer pattern)，此格式与多数摄像机传感器所得到的真实数据(raw data)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。这种模式我们将在 YUV 格式中再次见到。










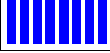











## 5. YUV 格式

下面首先展示 YUV 的紧密型模式，表中的图示如下：



















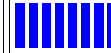













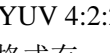
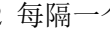
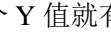
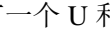

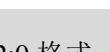

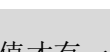
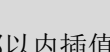
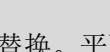
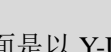

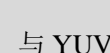

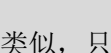

 = Y (intensity)

 = U (Cb)

 = V (Cr)

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				
					
					

也有几种平面型的 YUV 格式在使用，但把它们全画出来并没有什么意义，所以我们只是在下面举个例子，常用“YUV 4:2:2”(V4L2\_PIX\_FMT\_YUV422, fourcc422P)格式使用三组阵列，一幅 4X4 的图片如下表示：

Y plane:				
				
				
				
U plane:				
				
				
				
V plane:				
				
				
				

对于贝尔格式 (bayer pattern), YUV 4:2:2 每隔一个 Y 值就有一个 U 和一个 V 值，显示图像需要以内插值替换丢失值。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，但他们极少使用。完整列表请看[这里](#)。

## 6. 其他格式

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；今后的文章中将讨论流的控制。

还有一些其他各种各样的格式，其中一些有专利保护；[这里](#)有一张列表。

## 7. 格式描述

现在我们已经了解了颜色格式，接下来将要看下 V4L2 API 中是如何描述图像格式。这里主要的结构体是 **struct v4l2\_pix\_format** (定义于 `<linux/videodev2.h>`)，它包含如下字段：

- **\_\_u32 width:** 图像宽度，以像素为单位。
- **\_\_u32 height:** 图像高度，以像素为单位。
- **\_\_u32 pixelformat:** 描述图像格式的四字符码。
- **enum v4l2\_field field:** 很多图像源会使数据交错——先传输奇数行，然后是偶数行。真正的摄像头设备是不会做数据交错的。V4L2 API 允许应用使用多种交错方式。常用的值为 V4L2\_FIELD\_NONE (非交错)、V4L2\_FIELD\_TOP (仅顶部交错)或 V4L2\_FIELD\_ANY (忽略)。详情见[这里](#)。
- **\_\_u32 bytesperline:** 相邻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:** 存储图像所需的缓冲区的大小。
- **enum v4l2\_colorspace colorspace:** 使用的色域。

这些参数加到一起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，驱动开发者则要将其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商的过程，以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，描述这种协定是怎么进行的。

## 六、 格式协商

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[上一篇文章](#)的链接。在上一篇中，我们关注了 V4L2 API 是如何描述视频格式：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，它将描述如何就硬件所支持的实际视频格式与应用协商。

如上篇所述，在存储器中表示图像的方法有很多种。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理解的视频格式。实际上在内核中进行格式转换是令人难以接受的。所以驱动必须能让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，这个调用会转化为如下的回调函数（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);
```

这个回调函数要求视频捕获设备描述其支持的格式。应用会传入一个 `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` 成员：

- `index` 是用来确定格式的一个简单整型数；与其他 V4L2 所使用的索引一样，这个也是从 0 开始递增，至最大允许值为止。应用可以通过一直递增索引值直到返回 `-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（摄像头或调谐器）来说就是 `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成员：

- `pixelformat` 应是描述视频表现方式的四字符码，
- `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`，表示一个压缩的视频格式。

上述函数是针对视频捕获函数，只有当 `type` 值为 `V4L2_BUF_TYPE_VIDEO_CAPTURE` 时才会调用。

`VIDIOC_ENUM_FMT` 调用将根据 `type` 值解释为不同的回调函数。

```
/* 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);

```

参数对于以上所有调用都一样。对于 **V4L2\_BUF\_TYPE\_PRIVATE** 类型的解码器，驱动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。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** 成员是我们关注的重点。这是我们在上一篇中见过的 **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，除非硬件出现了非常严重问题。

其他回调函数还有：

```

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，以相同的方式使用 `pix` 成员。

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合适的格式。改变视频格式有两个函数接口，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被支持的，就会返回 `-EINVAL`。例如，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。通常的做法是尽量将大小调小。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的是什么。

**VIDIOC\_TRY\_FMT** 这个处理对于驱动来说是可选的，但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，这个回调是不能随时调用的。如果硬件正在工作或已经开辟了流缓冲区(今后的文章将介绍)，改变格式会带来无尽的麻烦。想想会发生什么：比如，一个新的格式比现在使用的缓冲区大的时候。所以驱动总是要保证硬件是空闲的，否则就对请求返回失败(`-EBUSY`)。

格式的改变应该是原子的——它要么改变所有的参数以实现请求，要么一个也不改变。同样，驱动在必要时是可以改变图像大小的，常用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** 处理可以避免代码重写，而且可以避免任何没有预先实现那个函数的借口。如果“try”函数成功返回，最终格式就已经在工作了并可以直接编程进硬件。

还有很多其他调用也会影响视频 I/O 的完成方式。今后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，而且那也是这个结构体的最终目的。所以下一篇文章，(希望会在不久之后)我们会来关注对视频数据的读和写支持。



## 七、基本的帧 I/O

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视频数据。虽然在这一点上，我们已经了解了足够的关于格式协商方面的细节，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()系统调用可以在通常情况下使用。**根据硬件和驱动的不同，这种方法可能会非常慢——但也不是一定的。
- **将帧数据以视频流的方法直接送到应用可以访问的缓冲区。**视频流实际上是传输视频数据最有效的方法，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，其区别在于缓冲的开辟是在用户空间还是内核空间。
- **Video4Linux2 API 规范提供一种异步 I/O 机制用于帧传输。**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讲解。

### 1. 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使用这种系统调用，所以驱动作者应尽可能实现这两个系统调用。如果驱动确实实现了这些系统调用，它应保证 **V4L2\_CAP\_READWRITE** 置位，来回应 **VIDIOC\_QUERYCAP** 调用。然而以笔者的经验，多数应用在使用调用之前，根本就不会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 **fops** 成员里。注意：V4L2 规范要求实现这些方法同时也提供 **poll()**操作。

在一个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，然后停止硬件并返回。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 **DMA** 控制器可以处理分散/聚集 I/O 的时候才有可能。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。

稍微复杂点的操作也是可以的。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状态，在接下来的一秒内，摄像头的后续帧将会存储在内存缓冲区中，如果应用发出了另一个 read()，它将会有更快的反应，无需再次启动硬件。经过一定数目的帧都没有读的话，控制器就会回到空闲状态。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。

### 2. 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，其中一些也较常用。它看起来像是一个设置没有明显归属的杂项调用。我们在这里就了解一下，虽然有些参数

同时会影响流 IO 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体，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** 成员描述的是所涉及操作的类型。

对于视频捕获设备，应为 **V4L2\_BUF\_TYPE\_VIDEO\_CAPTURE**。对于输出设备，应为 **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，甚至是不鼓励使用的数据给内核。

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** 成员是一组功能标签。目前为止仅有一个被定义：**V4L2\_CAP\_TIMEPERFRAME**，它代表可以改变的帧频率。

**capturemode** 成员也只定义了一个标签：**V4L2\_MODE\_HIGHQUALITY**，这个标签意在使硬件在适于单帧捕获的高清模式下工作。这种模式为达到设备可处理的最佳图片质量，能做出任何牺牲（包括支持的格式、曝光时间等）。

**timeperframe** 成员用于指定想要使用的帧率，它也是一个结构体：

```
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。另一个驱动相关的成员是：**extendedmode**，它在 API 中没有明确意义。

**readbuffers** 字段是 read() 操作被调用时内核应为输入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** 也是相对类似的。

当应用想要查询当前参数时，它可以发出一个 **VIDIOC\_G\_PARM** 调用，从而调用驱动的 `vidioc_g_parm()` 方法。驱动应该提供当前设置，如不使用 **extendedmode** 的话应确保其为 0，并永远把 **reserved** 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** 结构体以反应当前值。例如，应用可以请求一个比硬件所能提供的更高的帧率，在这种情况下，帧率会设为最高，并把 **imeperframe** 设为这个最高的帧率。

如果应用提供 **timeperframe** 为 0，驱动应设置为与当前视频制式相对应的帧率。如果 **readbuffers** 或 **writebuffers** 是 0，驱动应返回现行设置而非删除缓冲区。

至此，我们已经能写一个支持 `read()` 和 `write()` 方式帧传输的简单驱动了。然而多数正式的应用都期望使用流 IO 方式：流方式使提高性能变得更简单，并允许帧在打包时带上附加信息（metadata），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。

## 八、 流 I/O

在本系列文章的[上一篇](#)中，我们讨论了如何通过 `read()`和 `write()`方式实现视频帧传输，这种实现可以完成基本的工作，却并非最常用的视频 I/O 实现方法。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 I/O。

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，使用流 I/O 方式，这种情况就不会发生。而是用户与内核空间之间交换缓冲区指针，这些缓冲区将被映射到用户地址空间，使帧的零拷贝成为可能。

有两种流 I/O 缓冲区：

- 内存映射缓冲区(type 为 `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 I/O 内存中开辟的缓冲区（如果硬件支持）。
- 用户空间缓冲区(type 为 `V4L2_MEMORY_USERPTR`) 是用户空间的应用中开辟缓冲区。很明显，在这种情况下是不需要 `mmap()`调用的，但驱动为有效地支持用户空间缓冲区，其工作将会更困难。

注意：驱动支持流 I/O 方式并非必需，即便实现了，也不必对两种缓冲区类型都做处理。虽然一个灵活的驱动应该支持更多的应用，但在实际应用中，似乎多数应用都是使用内存映射缓冲区，且不可能同时使用两种缓冲区。

现在，我们将要探索一下支持流 I/O 的众多蹩脚的细节。任何 Video4Linux2 驱动作者都要了解这部分 API。然而值得一提的是，有一个更高层次的 API，它能够帮助驱动作者完成流驱动。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`）可以使工作变得容易。关于 `video-buf` API 我们将在今后的某期讨论。

支持流 I/O 的驱动应通过在 `vidioc_querycap()`方法中设置 `V4L2_CAP_STREAMING` 标签通知应用。注意：我们无法描述支持哪种缓冲区，那是后话。

### 1. v4l2\_buffer 结构体

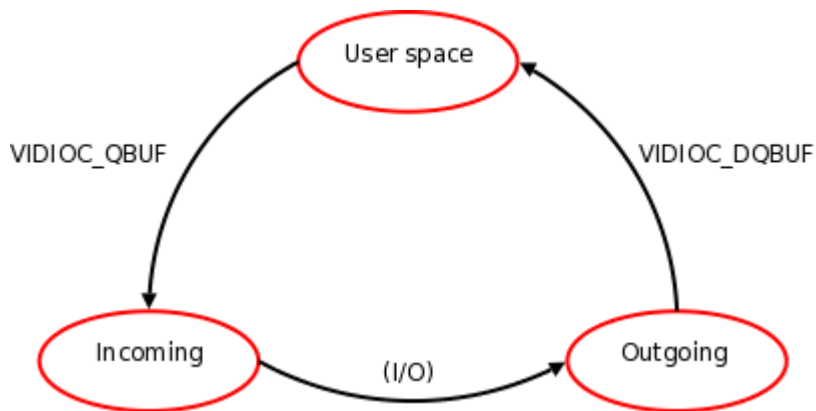
当使用流 I/O 时，帧以 `v4l2_buffer` 的格式在应用和驱动之间传输。这个结构体是一个复杂的野兽，需要点时间才能描述完。

首先大家要知道一个缓冲区可以有三种基本状态：

- **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，传入队列中的缓冲区是空的，等待驱动向其中填入视频数据；对于输出设备来讲，这些缓冲区是要设备发送的帧数据。
- **在驱动的传出队列中。**这些缓冲区已由驱动处理过，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，缓冲区由用户空间拥有，驱动无法访问。这是应用可对缓

缓冲区进行操作的唯一时间。我们称其为用户空间状态。

将这些状态和他们之间传输的操作放在一起，如下图所示：



译者注：其实缓冲区的三种状态是以驱动为中心的，可以理解为传入队列为要给驱动处理的缓冲区，传出队列为驱动处理完毕的缓冲区，而第三种状态是脱离驱动控制的缓冲区。

实际的 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** 成员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 接口中枚举的对象一样，内存映射缓冲区的 index 从 0 开始，依次递增。

**type** 成员描述的是缓冲区类型，通常是 V4L2\_BUF\_TYPE\_VIDEO\_CAPTURE 或 V4L2\_BUF\_TYPE\_VIDEO\_OUTPUT。

缓冲区的大小由 **length** 给定，单位为 byte。缓冲区中的图像数据大小可以在 **bytesused** 成员中找到。显然，bytesused ≤ length。对于捕获设备而言，驱动会设置 bytesused；对输出设备而言，应用必须设置这个成员。

**field** 字段描述存在缓冲区中的图像属于哪一个域。关于域的解释在这系列文章中的 [part5a](#) 中可以找到。

**timestamp(时间戳)** 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，对于视频编辑类应用是非常有用的。关于时间编码详情见[此处](#)。

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** 成员中存入当前序号。对于输入设备来讲，应用可以观察这一成员来检测帧。

**memory** 字段表示缓冲是内存映射缓冲区还是用户空间缓冲区。对于内存映射缓冲区，**m.offset** 描述的是缓冲区的位置。规范将它描述为“从设备内存基址开始计算的缓冲区偏移”，但其实质却是一个 magic cookie，应用可以将其传给 **mmap()**，以确定哪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——如果设备支持帧间的快速切换。

**reserved** 字段置 0。

最后，还有几个 **flags** 的位掩码定义：

- **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:** 缓冲区在驱动的传入队列。
- **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、差分帧。
- **V4L2\_BUF\_FLAG\_TIMECODE:** timecode 字段有效。
- **V4L2\_BUF\_FLAG\_INPUT:** input 字段有效。

## 2. 缓冲区设定

当流应用已经完成了基本设置，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，它由 V4L2 转换成对驱动 **vidioc\_reqbufs()** 方法的调用。

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

我们要关注的所有内容都在 **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** 字段描述的是所完成的 I/O 操作类型。通常它的值是 **V4L2\_BUF\_TYPE\_VIDEO\_CAPTURE** (视频捕获设备), 或者 **V4L2\_BUF\_TYPE\_VIDEO\_OUTPUT** (视频输出设备)。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缓冲区, 它会把 **memory** 字段置为 **V4L2\_MEMORY\_MMAP**, **count** 置为期望使用的缓冲区数。如果驱动不支持内存映射, 它应返回 **-EINVAL**。否则它将在内部开辟请求的缓冲区并返回 0。返回后, 应用就会认为缓冲区是存在的, 所以任何可能失败的操作都应在这个阶段处理 (比如说内存申请)。

注意: 驱动并不一定要开辟与请求数目一样的缓冲区。在多数情况下, 只有最小缓冲区数是有意义的。如果应用请求的比这个最小值小, 它可能得到比实际申请的多一些。以笔者的经验, **mplayer** 要用两个缓冲区, 如果用户空间速度变慢, 这将很容易超支 (并导致丢帧)。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 I/O 通道更加强壮。**count** 字段应设为驱动函数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问题。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, 如果使用用户空间缓冲区, 则有意义的字段只有缓冲区的 **type** 以及仅 **V4L2\_MEMORY\_USERPTR** 这个值可用的 **memory** 字段。应用无须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, 否则返回 **-EINVAL**。

**VIDIOC\_REQBUFS** 命令是应用得知驱动所支持的流 I/O 缓冲区类型的唯一方法。

### 3. 将缓冲区映射到用户空间

如果使用用户空间缓存, 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的相关调用。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的缓冲区, 并将其映射到自己的地址空间。第一步是 **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);
```

进入此方法时, **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符) 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** 结构体数组, 所以 **vidioc\_querybuf()** 方法的核心只是一个结构体赋值。

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自己的地址空间, 所以 **vidioc\_querybuf()** 调用后面通常会跟着一个驱动的 **mmap()** 方法——要记住, 这个方法指针是存储在相关设备的 **video\_device** 结构体中 **fops** 字段中的。设备如何处理 **mmap()** 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, 那就应该在此时完成。对于内核空间的缓冲区, 内存页也可以在 **page-fault** 时通过使用常规的 **nopage()** 方法被单独映射。如果必要, 在 [Linux Device Drivers](#) 可以找到一个关于处理 **mmap()** 的一个很好的讨论。

译者注: 可参考《Linux 设备驱动程序 (第三版)》第十五章 内存映射和 DMA。

**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地址——当然是经过 **PAGE\_SHIFT** 右移过的。尤其是它应为驱动对于 **VIDIOC\_QUERYBUF** 调用返回的那个偏移值。请遍历缓冲区列表, 并确保传入的地址匹配其中一个。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的手段。

你所提供的偏移值几乎可以是任何值。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。你应该尽量避免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不是一个好主意。

当用户空间映射缓冲区时，驱动应在相关的 `v4l2_buffer` 结构体中调置 `V4L2_BUF_FLAG_MAPPED` 标签。它也必须在 `open()` 和 `close()` 中设定 VMA 操作，这样它才能跟踪映射了缓冲区的进程数。只要缓冲区在任何地方被映射了，它就不能在内核中释放。如果一个或多个缓冲区的映射计数为 0，驱动就应该停止正在进行的 I/O 操作，因为没有进程需要它。

## 4. 流 I/O

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，但在此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，他们甚至还不存在。在应用开始流 I/O 之前，它必须至少将一个缓冲区放到驱动传入队列中。对于输出设备，那些缓冲区当然还要先填完有效的视频帧数据。

要把一个缓冲区放进传入队列，应用首先要发出一个 `VIDIOC_QBUF` ioctl() 调用，V4L2 会将其映射为对驱动 `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，还是只有 `buf` 的 `type` 和 `index` 成员有效。驱动只能进行一些显式的检查(`type` 和 `index` 是否有效、缓冲区是否在驱动队列中、缓冲区已映射等)，把缓冲区放进传入队列里（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，因为驱动可能从来不知道缓冲区的情况。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，那将会非常具有挑战性。

要想把数据直接传送到用户空间，驱动必须先对缓冲区中的所有内存页产生异常，并将其锁定。`get_user_pages()` 可以完成这个操作。注意这个函数可能会做大量内存空间申请和硬盘 I/O 操作——它可能会长时间阻塞。你必须小心，并保证重要的驱动函数不会在 `get_user_pages()` 阻塞时停止，因为它可能因等待许多视频帧数据通过而长时间阻塞。

接下来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连续的，相反，它会被分散成许多单独的页（大部分架构上是 4096 字节/页）。显然，设备必须实现分散/聚集 DMA 操作才行。如果设备需要传输一个完整的视频帧，它就必须接受一个包含很多分散页的列表（scatterlist）。一个 16 位格式的 VGA 图像需要 150 个页，随着图像大小的增加，分散页列表的大小也会增加。V4L2 规范描述如下：

如果硬件需要，驱动要在物理内存中完成页交换，以产生连续的内存区。这发生在内核的虚拟内存管理子系统中，对应用来说是透明的。

然而，笔者却不推荐驱动作者尝试这种底层的虚拟内存技巧。有一个更好的方法就是申请用户空间缓冲区分成 Hugetlb 页，但是现在的驱动都不那么做。

如果你的设备是以更小的单位来传输图像（如 USB 摄像头），数据直接通过 DMA 到用户空间的配置就简单些。在任何情况下，当面对支持直接 I/O 到用户空间缓冲区的挑战时，驱动作者都应该：

- (1) 确定去解决这样大的麻烦是值得的，因为应用更趋向于使用内存映射缓冲区。
- (2) 使用 `video_buf` 层，它可以帮你解决一些痛苦的难题。

一旦流 I/O 开始，驱动就要从它的传入队列里获取缓冲区，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填充。最后，应用会在传出队列中认领缓冲区，让它变回为用户态。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 I/O 操作。完成这个任务的 **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才让设备开始工作。如果需要的话，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传输。

当应用关闭时，它应发出一个 **vidioc\_streamoff()** 调用，此调用要停止设备。驱动还应从传入和传出队列中移除所有的缓冲区，使它们都处于用户空间状态。当然，驱动必须意识到：应用可能在没有停止流传输的情况下关闭设备。

## 九、 控制

刚刚完成了这一系列文章的[第六部分](#)，现在我们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：用户永远也不会满意，并不会满足于仅能从摄像头上看到视频，他们马上就会问是否可以调整参数？例如亮度、对比度等等。这些参数可在视频应用中调整，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传递的动态范围。很明显，基于硬件的调整也可以减轻主机处理器的压力。

现代硬件中通常都可以在运行时调整很多参数。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，也可能需要处理一个非常复杂的矩阵变换。最好是尽可能多的把诸多细节对应用隐藏，但能隐藏到什么程度却受到很多限制。一个过于抽象的接口会使硬件的控制无法发挥到极限。

V4L2 的控制接口试图使事情尽可能地简单化，同时还能完全发挥硬件的功能。它始于定义一个标准控制名的集合，包括 **V4L2\_CID\_BRIGHTNESS**、**V4L2\_CID\_CONTRAST**、**V4L2\_CID\_SATURATION**，还有许多其他定义。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 值的完整列表详见 [V4L2 API 规范](#)。还有一些驱动程序特定的控制，但显然这些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能枚举可用的控制操作。为此，他们要发出最终由驱动 `vidioc_queryctrl()` 方法实现的 `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** (针对一组固定的选项) 或 **V4L2\_CTRL\_TYPE\_BUTTON** (针对一些设定时会忽略任何给出值的控制操作)。

**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——仅管他只对整型、布尔型和菜单控制适用。驱动仅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。

还有一组定义值进一步描述控制操作:

- **V4L2\_CTRL\_FLAG\_DISABLED** : 控制操作不可用, 应用应忽略它。
- **V4L2\_CTRL\_FLAG\_GRABBED** ; 控制暂时不可变, 可能是因为另一个应用正在使用它。
- **V4L2\_CTRL\_FLAG\_READ\_ONLY** : 可查看, 但不可改变的控制操作。
- **V4L2\_CTRL\_FLAG\_UPDATE**: 调整这个参数可以会对其他控制操作造成影响。
- **V4L2\_CTRL\_FLAG\_INACTIVE** : 与当前设备配置无关的控制操作。
- **V4L2\_CTRL\_FLAG\_SLIDER** : 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。

应用可以只查询几个特定编程过的控制操作, 或者他们也可枚举整个集合。对后者来讲, 他们会从 **V4L2\_CID\_BASE** 开始至 **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, 应用很可能希望枚举可能的值, 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** 是相关菜单控制操作的 ID 值, **index** 为某特定菜单 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** 字段恒为 0。

一旦应用知道了可用的控制操作, 它就很可能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, 这个调用最终实现为:

```
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：V4L2 还支持一个独立的[扩展控制接口](#)。这个 API 是一组相当复杂的控制操作。实际上，它的主要应用是 MPEG 编解码参数。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。