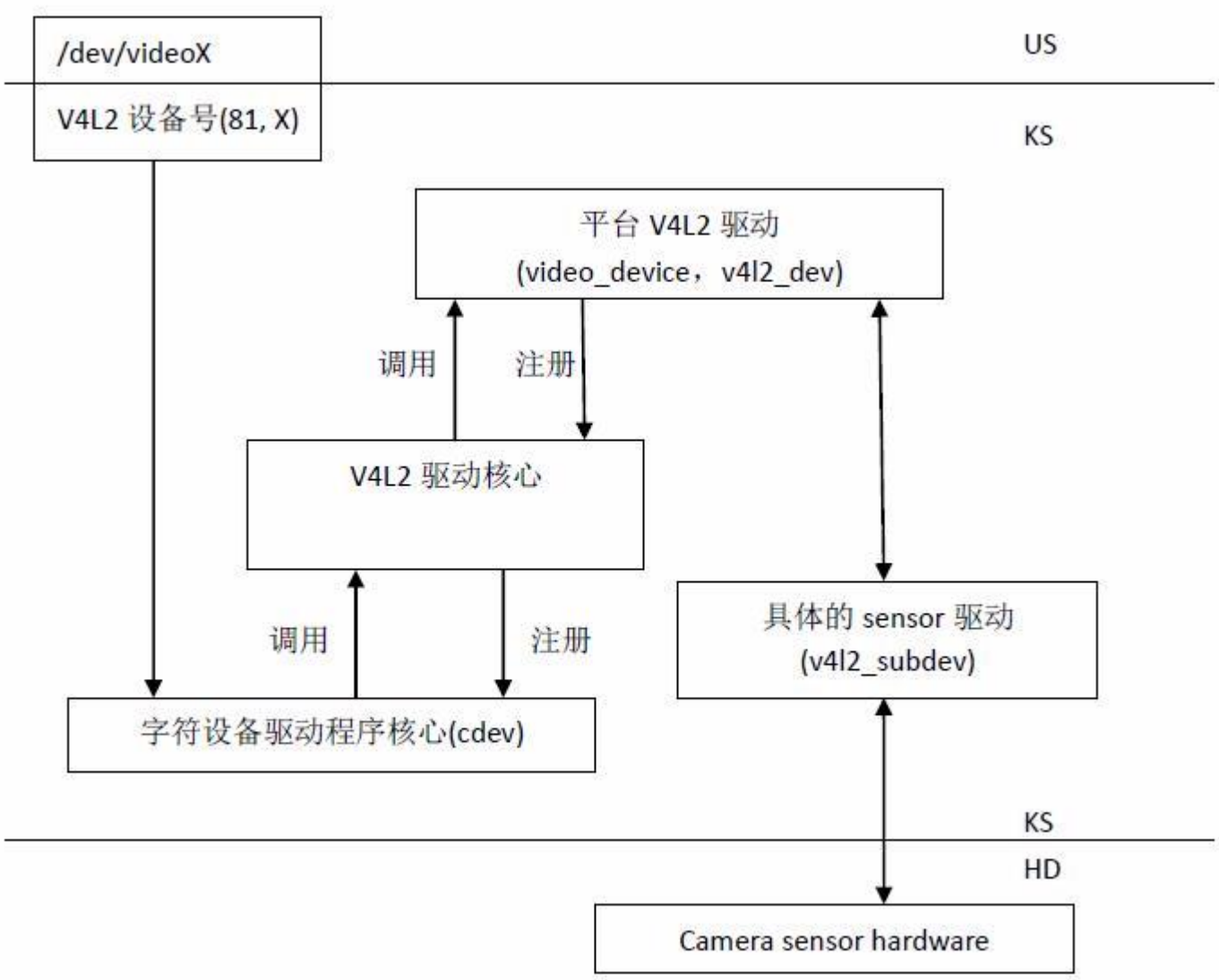
V4L2 框架分析学习

Author：CJOK
Contact：cjok.liao#gmail.com
SinaWeibo：@廖野 cjok

1、概述

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

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



Linux 系统中视频输入设备主要包括以下四个部分：

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

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

平台 V4L2 设备驱动：在 V4L2 框架下，根据平台自身的特性实现与平台相关的 V4L2 驱动部分，包括注册 video_device 和 v4l2_dev。

具体的 sensor 驱动：主要上电、提供工作时钟、视频图像裁剪、流 IO 开启等，实现各种设备控制方法供上层调用并注册 v4l2_subdev。

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

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

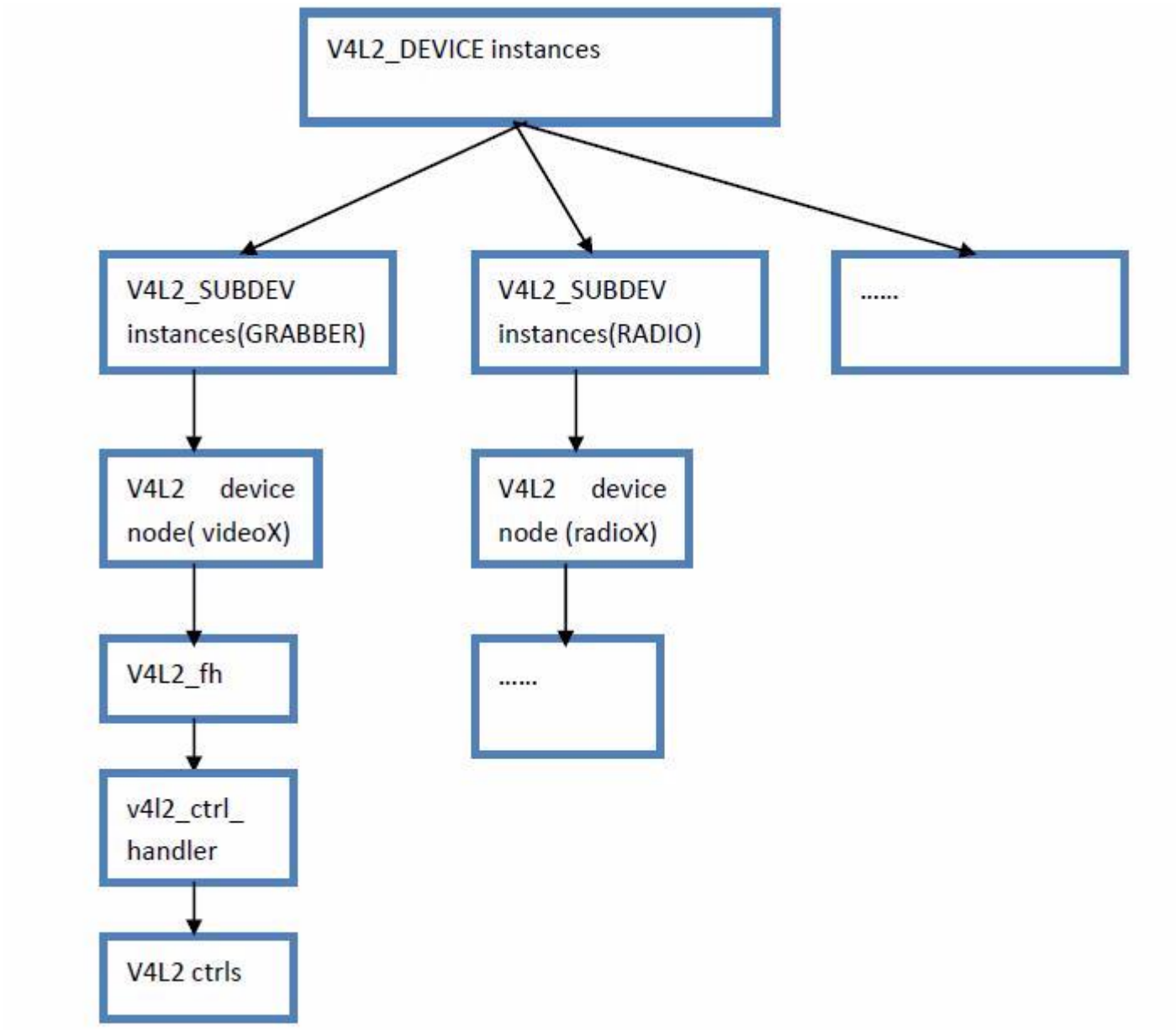
V4L2 框架：由 v4l2-device.c、v4l2-subdev.c、v4l2-fh.c、v4l2-ctrls.c 等文件实现，构建 V4L2 框架；

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

Ioctl 框架：由 v4l2-ioctl.c 文件实现，构建 V4L2 ioctl 的框架。

2、V4L2 框架

结构体 v4l2_device、video_device、v4l2_subdev 和 v4l2_fh 是搭建框架的主要元素。下图是 V4L2 框架的结构图：



从上图可以看出 V4L2 框架是一个标准的树形结构，v4l2_device 充当了父设备，通过链表把所有注册到其下的子设备管理起来，这些设备可以是 GRABBER、VBI 或 RADIO。V4l2_subdev 是子设备，v4l2_subdev 结构体包含了对设备操作的 ops 和 ctrls，这部分代码和硬件相关，需要驱动工程师根据硬件实现，像摄像头设备需要实现控制上下电、读取 ID、饱和度、对比度和视频数据流打开关闭的接口函数。Video_device 用于创建子设备节点，把操作设备的接口暴露给用户空间。V4l2_fh 是每个子设备的文件句柄，在打开设备节点文件时设置，方便上层索引到 v4l2_ctrl_handler，v4l2_ctrl_handler 管理设备的 ctrls，这些 ctrls(摄像头设备)包括调节饱和度、对比度和白平衡等。

v4l2_device

v4l2_device 在 v4l2 框架中充当所有 v4l2_subdev 的父设备，管理着注册在其下的子设备。以下是 v4l2_device 结构体原型(去掉了无关的成员)：

```
struct v4l2_device {
        struct list_head subdevs;    //用链表管理注册的 subdev
        char name[V4L2_DEVICE_NAME_SIZE];    //device 名字
        struct kref ref;      //引用计数
        ……
};
```

可以看出 v4l2_device 的主要作用是管理注册在其下的子设备，方便系统查找引用到。

V4l2_device 的注册和注销：

```
int v4l2_device_register(struct device* dev, struct v4l2_device *v4l2_dev)
static void v4l2_device_release(struct kref *ref)
```

V4l2_subdev

V4l2_subdev 代表子设备，包含了子设备的相关属性和操作。先来看下结构体原型：

```
struct v4l2_subdev {
    struct v4l2_device *v4l2_dev;  //指向父设备
    //提供一些控制 v4l2 设备的接口
    const struct v4l2_subdev_ops *ops;
    //向 V4L2 框架提供的接口函数
    const struct v4l2_subdev_internal_ops *internal_ops;
    //subdev 控制接口
    struct v4l2_ctrl_handler *ctrl_handler;
    /* name must be unique */
    char name[V4L2_SUBDEV_NAME_SIZE];
    /*subdev device node */
```

```
    struct video_device *devnode;
};
```

每个子设备驱动都需要实现一个 v4l2_subdev 结构体，v4l2_subdev 可以内嵌到其它结构体中，也可以独立使用。结构体中包含了对子设备操作的成员 v4l2_subdev_ops 和 v4l2_subdev_internal_ops。

v4l2_subdev_ops 结构体原型如下：

```
struct v4l2_subdev_ops {
    //视频设备通用的操作：初始化、加载 FW、上电和 RESET 等
    const struct v4l2_subdev_core_ops       *core;
    //tuner 特有的操作
    const struct v4l2_subdev_tuner_ops      *tuner;
    //audio 特有的操作
    const struct v4l2_subdev_audio_ops      *audio;
    //视频设备的特有操作：设置帧率、裁剪图像、开关视频流等
    const struct v4l2_subdev_video_ops      *video;
    ……
};
```

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

v4l2_subdev_internal_ops 结构体原型如下：

```
struct v4l2_subdev_internal_ops {
    //当 subdev 注册时被调用，读取 IC 的 ID 来进行识别
    int (*registered)(struct v4l2_subdev *sd);
    void (*unregistered)(struct v4l2_subdev *sd);
    //当设备节点被打开时调用，通常会给设备上电和设置视频捕捉 FMT
    int (*open)(struct v4l2_subdev *sd, struct v4l2_subdev_fh *fh);
    int (*close)(struct v4l2_subdev *sd, struct v4l2_subdev_fh *fh);
};
```

v4l2_subdev_internal_ops 是向 V4L2 框架提供的接口，只能被 V4L2 框架层调用。在注册或打开子设备时，进行一些辅助性操作。

Subdev 的注册和注销

当我们把 v4l2_subdev 需要实现的成员都已经实现，就可以调用以下函数把子设备注册到 V4L2 核心层：

```
int v4l2_device_register_subdev(struct v4l2_device *v4l2_dev, struct v4l2_subdev *sd)
```

当卸载子设备时，可以调用以下函数进行注销：

```
void v4l2_device_unregister_subdev(struct v4l2_subdev *sd)
```

video_device

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

```
struct video_device
{
    const struct v4l2_file_operations *fops;  //V4L2 设备操作集合

    /*sysfs */
    struct device dev;        /* v4l device */
    struct cdev *cdev;       //字符设备

    /* Set either parent or v4l2_dev if your driver uses v4l2_device */
    struct device *parent;         /* device parent */
    struct v4l2_device *v4l2_dev;     /*v4l2_device parent */

    /*Control handler associated with this device node. May be NULL. */
    struct v4l2_ctrl_handler *ctrl_handler;

    /* 指向 video buffer 队列*/
    struct vb2_queue *queue;

    int vfl_type;      /* device type */
    int minor;  //次设备号

    /* V4L2 file handles */
    spinlock_t        fh_lock; /* Lock for all v4l2_fhs */
```

```
    struct list_head       fh_list; /* List of struct v4l2_fh */

    /*ioctl 回调函数集，提供 file_operations 中的 ioctl 调用 */
    const struct v4l2_ioctl_ops *ioctl_ops;
    ……
};
```

Video_device 分配和释放，用于分配和释放 video_device 结构体：

```
struct video_device *video_device_alloc(void)
void video_device_release(struct video_device *vdev)
```

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

```
static inline int __must_check video_register_device(struct video_device *vdev, int type, int nr)
void video_unregister_device(struct video_device *vdev);
```

vdev：需要注册和注销的 video_device；

type：设备类型，包括 VFL_TYPE_GRABBER、VFL_TYPE_VBI、VFL_TYPE_RADIO 和 VFL_TYPE_SUBDEV。

nr：设备节点名编号，如/dev/video[nr]。

v4l2_fh

v4l2_fh 是用来保存子设备的特有操作方法，也就是下面要分析到的 v4l2_ctrl_handler，内核提供一组 v4l2_fh 的操作方法，通常在打开设备节点时进行 v4l2_fh 注册。

初始化 v4l2_fh，添加 v4l2_ctrl_handler 到 v4l2_fh：

```
void v4l2_fh_init(struct v4l2_fh *fh, struct video_device *vdev)
```

添加 v4l2_fh 到 video_device，方便核心层调用到：

```
void v4l2_fh_add(struct v4l2_fh *fh)
```

v4l2_ctrl_handler

v4l2_ctrl_handler 是用于保存子设备控制方法集的结构体，对于视频设备这些 ctrls 包括设置亮度、饱和度、对比度和清晰度等，用链表的方式来保存 ctrls，可以通过 v4l2_ctrl_new_std 函数向链表添加 ctrls。

```
struct v4l2_ctrl *v4l2_ctrl_new_std(struct v4l2_ctrl_handler *hdl, const struct v4l2_ctrl_ops *ops, u32 id, s32 min,
s32 max, u32 step, s32 def)
```

hdl 是初始化好的 v4l2_ctrl_handler 结构体；

ops 是 v4l2_ctrl_ops 结构体，包含 ctrls 的具体实现；

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

min、max 用来定义某操作对象的范围。如：

```
v4l2_ctrl_new_std(hdl, ops, V4L2_CID_BRIGHTNESS,-208, 127, 1, 0);
```

用户空间可以通过 ioctl 的 VIDIOC_S_CTRL 指令调用到 v4l2_ctrl_handler，id 透过 arg 参数传递。

3、ioctl 框架

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

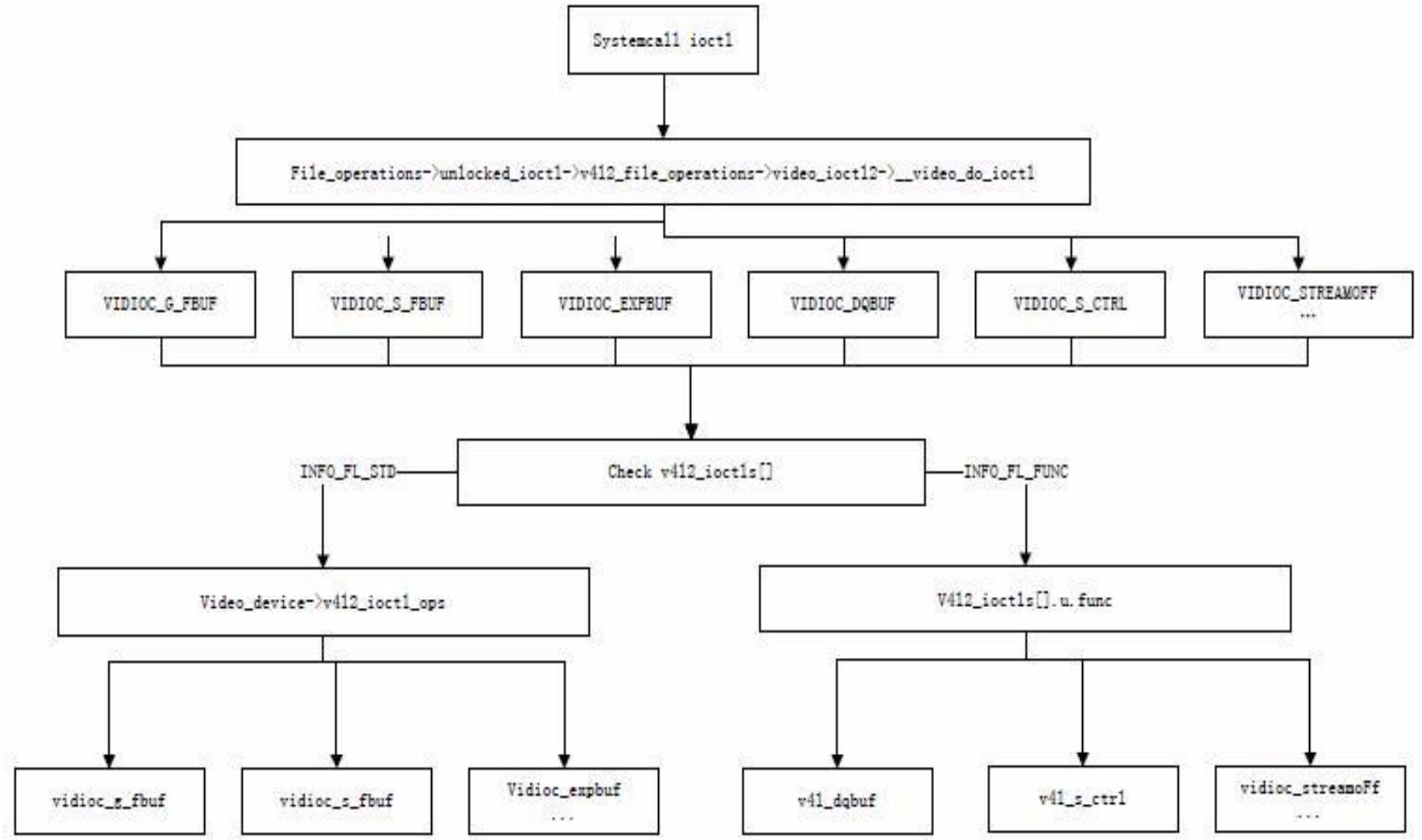
Ioctl 框架是由 v4l2_ioctl.c 文件实现，文件中定义结构体数组 v4l2_ioctls，可以看做是 ioctl 指令和回调函数的关系表。用户空间调用系统调用 ioctl，传递下来 ioctl 指令，然后通过查找此关系表找到对应回调函数。

以下是截取数组的两项：

```
IOCTL_INFO_FNC(VIDIOC_QUERYBUF, v4l_querybuf,v4l_print_buffer, INFO_FL_QUEUE | INFO_FL_CLEAR(v4l2_buffer, length)),
IOCTL_INFO_STD(VIDIOC_G_FBUF, vidioc_g_fbuf,v4l_print_framebuffer, 0),
```

内核提供两个宏(**IOCTL_INFO_FNC** 和 **IOCTL_INFO_STD**)来初始化结构体，参数依次是 ioctl 指令、回调函数或者 v4l2_ioctl_ops 结构体成员、debug 函数、flag。如果回调函数是 v4l2_ioctl_ops 结构体成员，则使用 **IOCTL_INFO_STD**；如果回调函数是 v4l2_ioctl.c 自己实现的，则使用 **IOCTL_INFO_FNC**。

IOCTL 调用的流程图如下：



用户空间通过打开/dev/目录下的设备节点，获取到文件的 file 结构体，通过系统调用 ioctl 把 cmd 和 arg 传入到内核。通过一系列的调用后最终会调用到 __video_do_ioctl 函数，然后通过 cmd 检索 v4l2_ioctls[]，判断是 INFO_FL_STD 还是 INFO_FL_FUNC。如果是 INFO_FL_STD 会直接调用到视频设备驱动中 video_device->v4l2_ioctl_ops 函数集。如果是 INFO_FL_FUNC 会先调用到 v4l2 自己实现的标准回调函数，然后根据 arg 再调用到 video_device->v4l2_ioctl_ops 或 v4l2_fh->v4l2_ctrl_handler 函数集。

4、IO 访问

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

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

内存映射缓冲区(V4L2_MEMORY_MMAP)，是在内核空间开辟缓冲区，应用通过 mmap()系统调用映射到用户地址空间。这些缓冲区可以是大而连续 DMA 缓冲区、通过 vmalloc()创建的虚拟缓冲区，或者直接在设备的 IO 内存中开辟的缓冲区(如果硬件支持)；

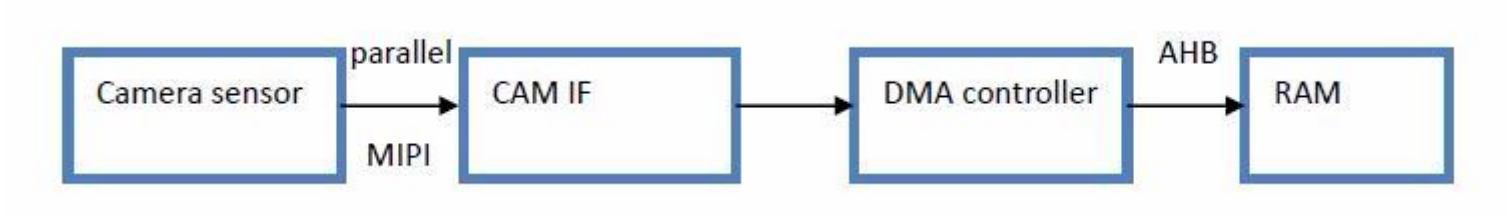
用户空间缓冲区(V4L2_MEMORY_USERPTR)，是用户空间的应用中开辟缓冲区，用户与内核空间之间交换缓冲区指针。很明显，在这种情况下是不需要 mmap()调用的，但驱动为有效的支持用户空间缓冲区，其工作将也会更困难。

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

内存映射缓存区方式

硬件层的数据流传输

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



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

vb2_queue

为了使设备支持流 IO 这种方式，驱动需要实现 struct vb2_queue，来看下这个结构体：

```
struct vb2_queue {
    enum v4l2_buf_type       type;  //buffer 类型
    unsigned int             io_modes;  //访问 IO 的方式:mmap、userptr etc
```

```
4.
5.      const struct vb2_ops         *ops;   //buffer 队列操作函数集合
6.      const struct vb2_mem_ops     *mem_ops;  //buffer memory 操作集合
7.
8.      struct vb2_buffer         *bufs[VIDEO_MAX_FRAME];  //代表每个 buffer
9.      unsigned int             num_buffers;    //分配的 buffer 个数
10.     ……
11. };
```

Vb2_queue 代表一个 video buffer 队列，vb2_buffer 是这个队列中的成员，vb2_mem_ops 是缓冲内存的操作函数集，vb2_ops 用来管理队列。

vb2_mem_ops

vb2_mem_ops 包含了内存映射缓冲区、用户空间缓冲区的内存操作方法：

```
1. struct vb2_mem_ops {
2.     void      *(*alloc)(void *alloc_ctx, unsigned long size);  //分配视频缓存
3.     void      (*put)(void *buf_priv);       //释放视频缓存
4.
5.     //获取用户空间视频缓冲区指针
6.     void      *(*get_userptr)(void *alloc_ctx,unsigned long vaddr, unsigned long size, int write);
7.     void      (*put_userptr)(void *buf_priv);       //释放用户空间视频缓冲区指针
8.
9.     //用于缓存同步
10.    void      (*prepare)(void *buf_priv);
11.    void      (*finish)(void *buf_priv);
12.
13.    void      *(*vaddr)(void *buf_priv);
14.    void      *(*cookie)(void *buf_priv);
15.    unsigned int     (*num_users)(void *buf_priv);    //返回当期在用户空间的 buffer 数
16.
17.    int         (*mmap)(void *buf_priv, struct vm_area_struct *vma);  //把缓冲区映射到用户空间
18. };
```

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

vb2_ops

vb2_ops 是用来管理 buffer 队列的函数集合，包括队列和缓冲区初始化

```
1. struct vb2_ops {
2.     //队列初始化
3.     int (*queue_setup)(struct vb2_queue *q, const struct v4l2_format *fmt,
4.                 unsigned int *num_buffers, unsigned int*num_planes,
5.                 unsigned int sizes[], void *alloc_ctxs[]);
6.     //释放和获取设备操作锁
7.     void (*wait_prepare)(struct vb2_queue *q);
8.     void (*wait_finish)(struct vb2_queue *q);
9.     //对 buffer 的操作
10.    int (*buf_init)(struct vb2_buffer *vb);
11.    int (*buf_prepare)(struct vb2_buffer *vb);
12.    int (*buf_finish)(struct vb2_buffer *vb);
13.    void (*buf_cleanup)(struct vb2_buffer *vb);
14.    //开始视频流
15.    int (*start_streaming)(struct vb2_queue *q, unsigned int count);
16.    //停止视频流
17.    int (*stop_streaming)(struct vb2_queue *q);
18.    //把 VB 传递给驱动
19.    void (*buf_queue)(struct vb2_buffer *vb);
20. };
```

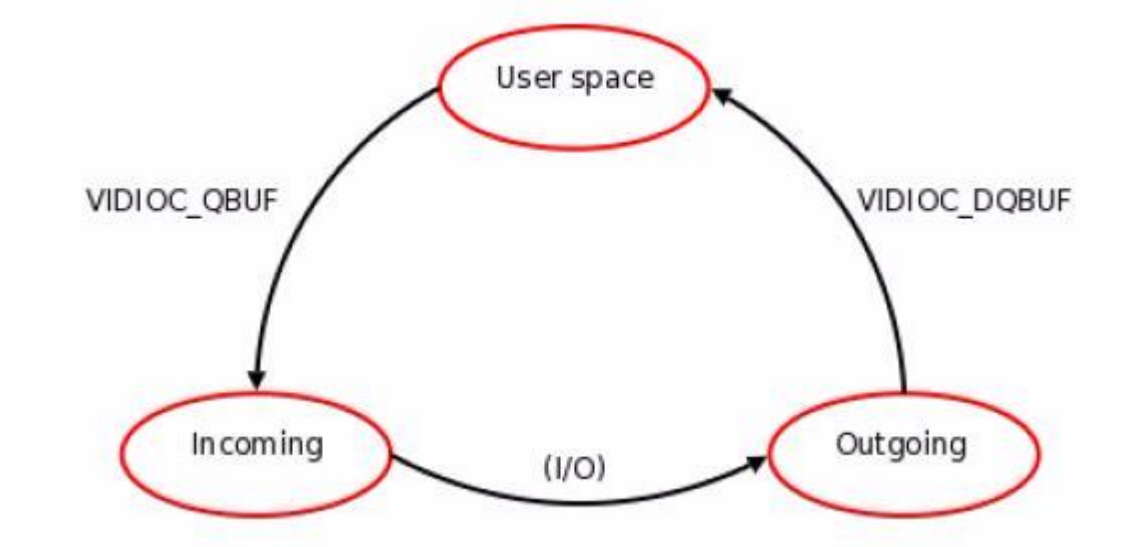
vb2_buffer 是缓存队列的基本单位，内嵌在其中的 v4l2_buffer 是核心成员。当开始流 IO 时，帧以 v4l2_buffer 的格式在应用和驱动之间传输。一个缓冲区可以有三种状态：

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

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

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

这三种状态的切换如下图所示：



v4l2_buffer 结构如下：

```
struct v4l2_buffer {
    __u32    index;  //buffer 序号
    __u32    type;   //buffer 类型
    __u32    bytesused;  缓冲区已使用 byte 数
    __u32    flags;
    __u32    field;
    struct timeval     timestamp;  //时间戳，代表帧捕获的时间
    struct v4l2_timecode      timecode;
    __u32    sequence;

    /*memory location */
    __u32    memory;  //表示缓冲区是内存映射缓冲区还是用户空间缓冲区
    union {
        __u32     offset;  //内核缓冲区的位置
        unsigned long   userptr;   //缓冲区的用户空间地址
        struct v4l2_plane *planes;
        __s32           fd;
    } m;
    __u32    length;   //缓冲区大小，单位 byte
};
```

当用户空间拿到 v4l2_buffer，可以获取到缓冲区的相关信息。Byteused 是图像数据所占的字节数，如果是 V4L2_MEMORY_MMAP 方式，m.offset 是内核空间图像数据存放的开始地址，会传递给 mmap 函数作为一个偏移，通过 mmap 映射返回一个缓冲区指针 p，p+byteused 是图像数据在进程的虚拟地址空间所占区域；如果是用户指针缓冲区的方式，可以获取的图像数据开始地址的指针 m.userptr，userptr 是一个用户空间的指针，userptr+byteused 便是所占的虚拟地址空间，应用可以直接访问。

5、用户空间访问设备

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

1> 打开设备文件

```
fd = open(dev_name, O_RDWR /* required */ | O_NONBLOCK, 0);
```

dev_name [/dev/videoX]

2> 查询设备支持的能力

```
Struct v4l2_capability  cap;
ioctl(fd, VIDIOC_QUERYCAP, &cap)
```

3> 设置视频捕获格式

```
fmt.type= V4L2_BUF_TYPE_VIDEO_CAPTURE;
fmt.fmt.pix.width       = 640;
fmt.fmt.pix.height      = 480;
fmt.fmt.pix.pixelformat = V4L2_PIX_FMT_YUYV;  //像素格式
fmt.fmt.pix.field       = V4L2_FIELD_INTERLACED;
ioctl(fd,VIDIOC_S_FMT, & fmt)
```

4> 向驱动申请缓冲区

```
Struct  v4l2_requestbuffers req;
req.count= 4;  //缓冲个数
req.type= V4L2_BUF_TYPE_VIDEO_CAPTURE;
req.memory= V4L2_MEMORY_MMAP;
if(-1 == xioctl(fd, VIDIOC_REQBUFS, &req))
```

5> 获取每个缓冲区的信息，映射到用户空间

```
struct buffer {
    void  *start;
    size_t length;
} *buffers;

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

for (n_buffers= 0; n_buffers < req.count; ++n_buffers) {
    struct  v4l2_buffer buf;

    buf.type    = V4L2_BUF_TYPE_VIDEO_CAPTURE;
    buf.memory  = V4L2_MEMORY_MMAP;
    buf.index   = n_buffers;

    if (-1 ==xioctl(fd, VIDIOC_QUERYBUF, & buf))
        errno_exit("VIDIOC_QUERYBUF");

    buffers[n_buffers].length = buf.length;
    buffers[n_buffers].start  = mmap(NULL /* start anywhere */,
                                buf.length,
                                PROT_READ | PROT_WRITE /* required */,
                                MAP_SHARED /* recommended */,
                                fd, buf.m.offset);
}
```

6> 把缓冲区放入到传入队列上，打开流 IO，开始视频采集

```
for (i =0; i < n_buffers; ++i) {
    struct v4l2_buffer buf;
    buf.type   = V4L2_BUF_TYPE_VIDEO_CAPTURE;
    buf.memory = V4L2_MEMORY_MMAP;
    buf.index  = i;

    if (-1 == xioctl(fd, VIDIOC_QBUF, &buf))
        errno_exit("VIDIOC_QBUF");
}

type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
if (-1 == xioctl(fd, VIDIOC_STREAMON, & type))
```

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

```
        for (;;) {
            fd_set fds;
            struct timeval tv;
            int r;

            FD_ZERO(&amp;fds);
            FD_SET(fd,&amp;fds);

            /* Timeout. */
            tv.tv_sec = 2;
            tv.tv_usec = 0;

            //监测文件描述是否变化
            r = select(fd + 1,& fds, NULL, NULL, & tv);

            if (-1 == r) {
                if (EINTR ==errno)
                    continue;

                errno_exit("select");
```

```
            }

             if (0 == r) {
                fprintf(stderr,"select timeout\n");
                exit(EXIT_FAILURE);
             }
             //对视频数据进行处理
             if (read_frame())
                break;
            /* EAGAIN - continueselect loop. */
        }
```

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

```
buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
buf.memory = V4L2_MEMORY_MMAP;

if (-1 ==ioctl(fd, VIDIOC_DQBUF, & buf))    //取出缓冲
    errno_exit("VIDIOC_QBUF");

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

if (-1 ==xioctl(fd, VIDIOC_QBUF, & buf))  //然后又放入到传入队列
    errno_exit("VIDIOC_QBUF");
```

9> 停止视频采集

```
type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
ioctl(fd,VIDIOC_STREAMOff, &type);
```

10> 关闭设备

```
Close(fd);
```

暂时分析到这里，后续在更新！