Device Driver

- Video for Linux Ver. 2(V4L2) –

본 문서는 비트캠프 임베디드 과정 프로젝트 진행 내용 중 일부 입니다.
문서 내에 틀린 부분이 있을 수도 있으니 참고용만으로 사용하세요.

Author : 강 삼민 (thirdnsov@gmail.com)

목차

1. USB Device Driver 3

2. V4L2(Video For Linux Version 2) 4

2.1. V4L2 Driver 4
2.2. V4L2 동작 순서 5
2.3. QUERYCAP 6
2.4. S_FMT_VID_CAP 7
2.5. REQBUF 8
2.6. QUERYBUF 9
2.7. QBUF 10
2.8. DQBUF 10
2.9. STREAMON 11

3. 문제점 및 해결 13

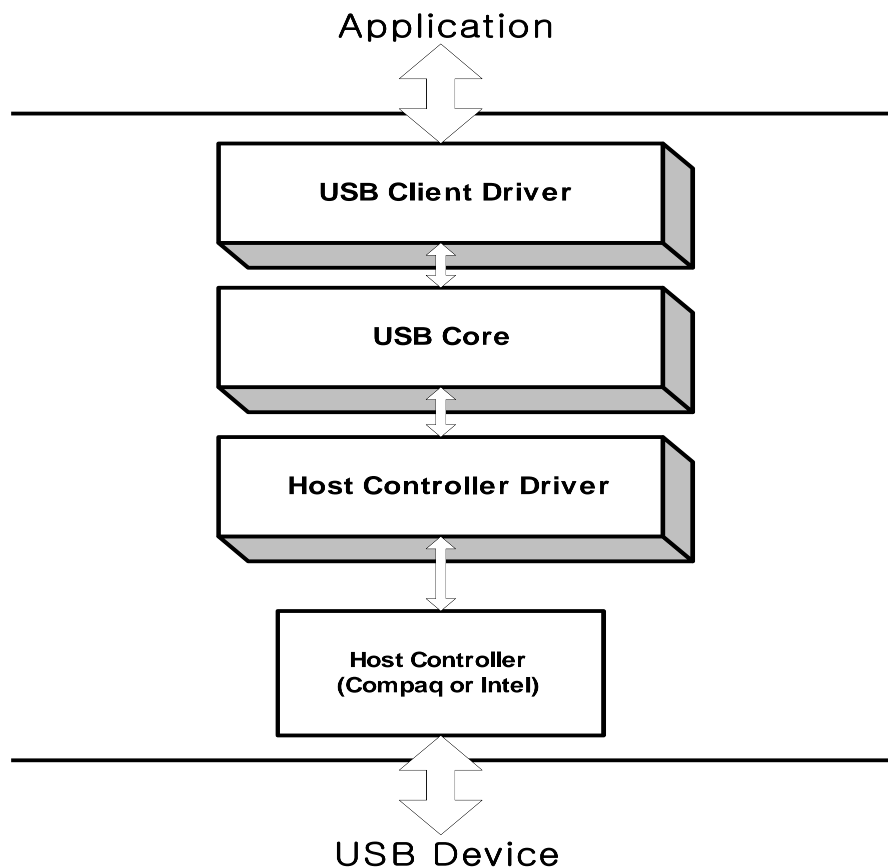
4. V4L2 Application Code 14

5. V4L2 Drver Code 18

1. USB Device Driver

본 문서에서는 USB가 주 내용이 아니기 때문에 USB는 간략하게 소개만 하였다

USB(Universal Serial Bus)는 컴퓨터와 주변 기기를 연결하는데 사용되는 입출력 표준 중 하나로 기존의 직렬, 병렬연결 방식을 대체하기 위해 개발되었다. 키보드, 마우스, 카메라, 저장장치 등 다양한 장치를 컴퓨터와 연결하는데 사용되고 있고 전원 공급 기능을 이용하여 휴대기기의 충전 용도로도 많이 사용되고 있다.



[그림 1] USB Device Driver Stack

[그림 1]은 일반적인 USB Device Driver에 관한 Stack 구조를 나타내는 그림이다. USB Device와 Application 사이에 Host Controller, Host Controller Driver, USB Core가 존재하는 것을 확인 할 수 있는데 이 세 부분은 현재 대부분의 컴퓨터에 내장되어 있고 안정적이기 때문에 일반적으로 드라이버 개발 시 신경 쓰는 부분이 아니다. 따라서 Driver 개발자는 USB Client Driver를 개발하면 된다고 볼 수 있다.

이 외의 자세한 내용은 O'Reilly에서 무료로 제공되는 '**Linux Device Driver**' 혹은 출판되는 서적이나 USB 관련 서적을 찾아보면 확인 할 수 있다.

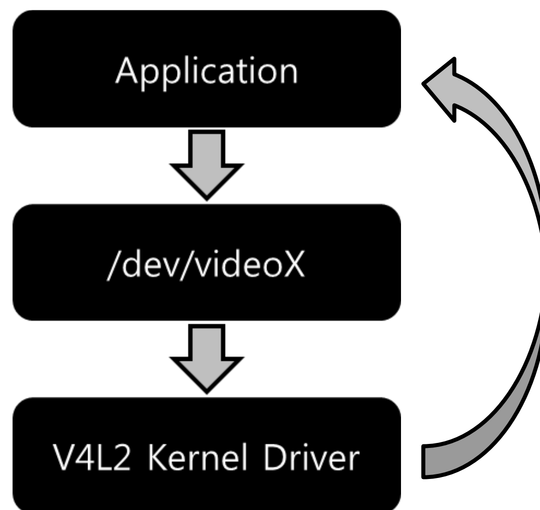
2. V4L2(Video For Linux Version 2)

V4L2는 TV나 USB 카메라, Audio 등 다양한 Audio/Video Device에 접근할 수 있도록 하는 일종의 Kernel API이다. 최초 버전(V4L)은 Alan Cox에 의해 Kernel 2.2에 탑재되었고 1999년에 두 번째 버전인 V4L2의 개발이 시작되었고 초기 버전인 V4L에서 발견된 버그들을 고치고 더 많은 Device를 지원할 수 있도록 수정된 뒤 Kernel에 완성된 V4L2가 탑재되었다.

본 문서에서는 USB Camera를 사용할 수 있도록 V4L2를 사용한 것을 기준으로 작성되었기 때문에 위에서 언급한 Audio 관련 내용은 누락되어있다.

2.1. V4L2 Driver

일반적으로 Video Device는 Character Device로 생성 된다. 이 Video Device는 일반 Character Device File과 달리 read()와 write()로 접근을 할 수 없고 ioctl()을 이용하여야만 접근이 가능하다. 이때 Device에 접근하기 편하도록 Kernel에서 제공되는 것이 바로 V4L2다.

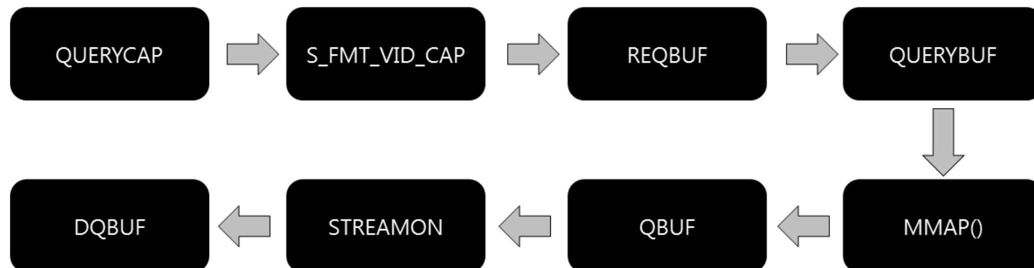


[그림 2] V4L2 Relations

[그림 2]는 사용자 수준 Application과 V4L2 Kernel Driver 사이의 관계를 나타낸 것이다. 크게 Application에서 Video Device에 접근하여 특정 명령을 요청했을 때 V4L2 Kernel Driver가 요청 받은 명령을 동작한 후 다시 Application에게 수행 결과를 돌려주는 형식이다. Application에서 보내는 명령의 순서가 중요한데 해당 내용은 다음절에서 설명하였다.

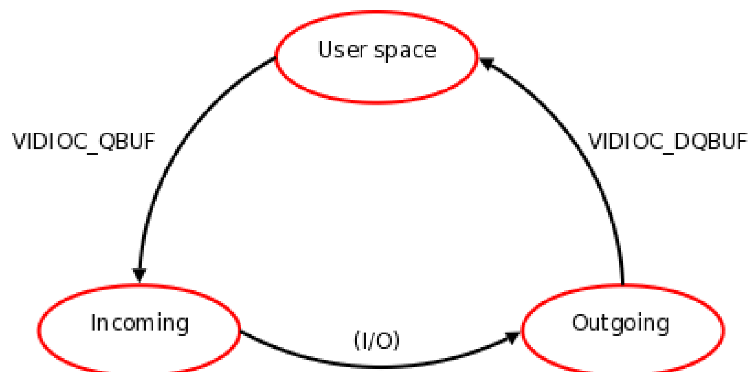
2.2. V4L2 동작 순서

Application에서는 ioctl()을 이용해 카메라 구동에 필요한 명령들을 **순차적**으로 전송해야 한다. 명령들의 순서가 바뀌거나 누락되는 경우 혹은 반복하면 안되는 명령을 반복할 경우에는 V4L2 Driver에서 명령 수행 도중 오류가 발생할 확률이 매우 높으니 주의해야 한다.



[그림 3] V4L2 - Sequence of Command

[그림 3]은 Application에서 V4L2 Driver로 보낼 명령의 순서도를 간략히 나타낸 것이다. 이 중 QBUF, DQBUF, STREAMON은 일반적으로 반복적으로 수행되어야 영상을 얻을 수 있고 반복하지 않는다면 스틸컷(1프레임)만 얻을 수 있다.



[그림 4] V4L2 - Relation of Qbuf and Dqbuf

[그림 4]는 QBUF와 DQBUF의 관계를 나타낸 것이다. 스틸컷을 이용하여 영상정보를 만들기 위해서는 1프레임 단위의 스틸컷을 지속적으로 Device에 요청하고 사용자 영역에 출력 시켜줘야 할 텐데, 앞서 언급했던 것 처럼 QBUF, DQBUF, STREAMON을 지속적으로 요청하고 수행시키면 된다. 실질적으로는 STREAMON 명령은 한번만 요청해도 된다. STREAMON을 요청 받으면 V4L2 Driver가 관련 동작들을 수행하는데 이 동작이 모두 수행되고 나면 Kernel 단에서는 지속적으로 동작 중인 상태가 된다. 하지만 QBUF와 DQBUF는 매번 요청을 해주고 [그림 4]에서 보여주는 동작을 지속적으로 수행되어야 한다. 그러기 위해서는 while() 혹은 for()와 같은 반복문을 이용해서 지속성을 유지시켜줘야 한다.

2.3. QUERYCAP

QUERYCAP(Query Capability)은 수행되어야 하는 명령 중 가장 간단한 것 중 하나로 연결된 Device의 이름, 수행 가능한 동작 등 장치의 정보를 사용자 영역에 알려주는 역할을 한다. 이 명령은 Device에 특별한 설정을 하거나 요청을 하는 것이 없기 때문에 코드를 작성 시에 크게 신경 쓰는 부분 없이 작성이 가능하다.

```
int thirdy_querycap(struct file *file, void *fh, struct v4l2_capability *cap)
{
	thirdy_t *t_struct;

	if((t_struct = kzalloc(sizeof *t_struct, GFP_KERNEL)) == NULL)
		return -ENOMEM;

	t_struct = fh;

	strlcpy(cap->driver, "Thirdy_Driver", sizeof(cap->driver));
	strlcpy(cap->card, t_struct->v_dev->name, sizeof(cap->card));
	usb_make_path(t_struct->u_dev, cap->bus_info, sizeof(cap->bus_info));
	cap->version = LINUX_VERSION_CODE;
	cap->capabilities = V4L2_CAP_DEVICE_CAPS | V4L2_CAP_STREAMING | t_struct->caps;
	cap->device_caps = V4L2_CAP_VIDEO_CAPTURE | V4L2_CAP_STREAMING;

	file->private_data = &t_struct->t_fh;

	printk("Thirdy_Driver : vidioc_querycap OK\n");
	return 0;
}
```

[그림 5] V4L2 - Query Capability Code

[그림 5]는 QUERYCAP 명령이 V4L2 Driver로 요청되었을 때 Driver가 수행되는 코드를 나타낸 것이다. 앞서 말한 것과 같이 특별하게 다른 함수를 호출하거나 복잡한 연산을 수행하는 부가 없는 것을 확인 할 수 있다. 수행하는 내용으로는 thirdy_t 구조체에 미리 저장되어 있던 현재 연결되어 있는 Device의 이름과 Bus 정보, 그리고 동작 할 수 있는 V4L2 모드 정보를 인자로 받아온 사용자 영역의 v4l2_capability 구조체에 담고 있다.

정상적으로 QUERYCAP 명령이 수행되고 난 후 사용자 영역에서 v4l2_capability 구조체에 담겨 있는 값을 적절하게 출력해보면 [그림 6]과 같은 결과를 얻을 수 있다.

```
Driver Caps:
  Driver: "uvcvideo"
  Card: "USB 2.0 Camera"
  Bus: "usb-xhci-hcd.1.auto-1.2"
  Version: 1.0
  Capabilities: 84200001
Selected Camera Mode:
  Width: 640
  Height: 480
```

[그림 6] V4L2 - QUERYCAP(Userspace)

2.4. S_FMT_VID_CAP

S_FMT_VID_CAP(Set Format Video Device Capability)는 Video Device를 실질적으로 동작 시키기 전에 Device의 몇몇 부분을 설정해주는 명령이다. Camera를 예로 간단하게 생각하면 Camera Device에서 촬영된 영상의 이미지 포맷을 어떤 걸로 할 것인지, 해상도는 어떻게 설정할 것인지 등을 설정 할 수 있다.

```
int thirdy_s_fmt_cap(struct file *file, void *fh, struct v4l2_format *f)
{
        int ret;
        int w, h;
        int maxw = 640, maxh = 480;
        __u32 fcc;
        thirdy_t *t_struct;
        __u8 *set_data;
        __u16 set_size = 26;


        if((t_struct = kzalloc(sizeof *t_struct, GFP_KERNEL)) == NULL)
                return -ENOMEM;
        t_struct = video_drvdata(file);

        w = f->fmt.pix.width;
        h = f->fmt.pix.height;
        if(t_fmts[0].fcc == f->fmt.pix.pixelformat)
                t_struct->t_fmt = &t_fmts[0];
        else
        {
                printk("Thirdy_Driver : thirdy_s_fmt_cap error 1\n");
                return -1;
        }

        fcc = t_struct->t_fmt->fcc;

        v4l_bound_align_image(&w, 48, maxw, 1, &h, 32, maxh, 1, 0);


//---------
        set_data = kzalloc(set_size, GFP_KERNEL);
        if(set_data == NULL)
                return -ENOMEM;
```

[그림 7] V4L2 - S_FMT_VID_CAP 일부

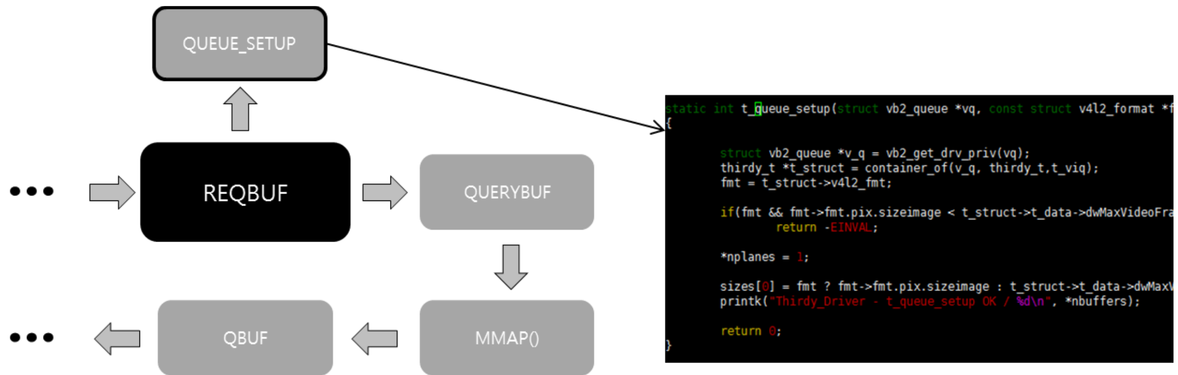
[그림 7]은 S_FMT_VID_CAP 명령이 V4L2 Driver에 요청 되었을 때 수행되는 코드의 일부를 나타낸 것이다. 본 문서를 작성하는 현재(2016. 07. 28.)까지 관련 내용을 제대로 이해하고 있지 못한 상태이다. 앞서 언급했던 Device 설정과 관련이 있고 실제로 설정이 되는 것은 확인 하였고, 위 코드 내용은 uvcvideo의 코드를 참고해서 간단하게 설정만 할 수 있도록 작성해 놓은 것이다. 실제로 이 함수에서는 많은 연산이 이루워지고, 많은 정보가 Device에 설정된다. 지금까지 이해한 내용으로는 Video Device에 대역폭 정보와 이미지의 크기 정보 등의 내용을 이용하여 실질적으로 받고자 하는 이미지에 대한 정보를 연산하고 v4l2_format의 구조체에 담겨있는 사용자 영역에서 설정한 값을 토대로 Device를 설정하는 것으로 보인다. 큰 내용으로는 방금 언급한 내용이 이 명령이 요청되었을 때 V4L2 Driver가 수행하는 내용이 맞을 것이다. 세부적인 내용까지 분석하고 코드로 작성한다면 보다 완벽한 V4L2 Driver가 될 수 있을 것이다.

```
Width: 640
Height: 480
PixFmt: MJPG
Field: 1
```

[그림 8] V4L2 - S_FMT_VID_CAP(Userspace)

2.5. REQBUF

REQBUF(Request Queue Buffer)는 Video Device에서 받아온 데이터를 저장하기 위한 Buffer를 할당하는 명령이다. V4L2 Driver는 이 명령을 받게 되면 내부적으로 Queue에 저장될 Buffer를 Set 하기 위해Queue_Setup()을 [그림 8]과 같이 호출한다.



[그림 9] V4L2 - REQBUF and QUEUE_SETUP

Queue_Setup()에서는 미리 얻어놓은 Device의 정보를 토대로 Buffer 사이즈를 결정하고 저장한다. 그 후 다시 REQBUF의 동작을 수행하게 되는데 이는 특별한 코드 작성 없이 V4L2에서 제공하는 자체 함수를 사용하면 편하게 작업을 할 수 있다. 하지만 제공되는 함수와 구조체나 변수 구조가 다르다면 새로운 함수를 작성하는 것이 더 편할 수 있다.

```
int vb2_ioctl_reqbufs(struct file *file, void *priv,
                          struct v4l2_requestbuffers *p)
{
        struct video_device *vdev = video_devdata(file);
        int res = __verify_memory_type(vdev->queue, p->memory, p->type);

        if (res)
                return res;
        if (vb2_queue_is_busy(vdev, file))
                return -EBUSY;
        res = __reqbufs(vdev->queue, p);
        /* If count == 0, then the owner has released all buffers and he
           is no longer owner of the queue. Otherwise we have a new owner. */
        if (res == 0)
                vdev->queue->owner = p->count ? file->private_data : NULL;
        return res;
}
EXPORT_SYMBOL_GPL(vb2_ioctl_reqbufs);
```

[그림 10] V4L2 - REQBUF(V4L2)

```
int thirdy_reqbufs(struct file *file, void *fh, struct v4l2_requestbuffers *b)
{
        thirdy_t *t_struct;
        int ret, chk = 0;

        if((t_struct = kzalloc(sizeof *t_struct, GFP_KERNEL)) == NULL)
                return -ENOMEM;

        t_struct = video_drvdata(file);

        mutex_lock(&t_struct->q_mutex);
        chk = vb2_reqbufs(&t_struct->t_viq, b);
        mutex_unlock(&t_struct->q_mutex);
        ret = chk ? chk : b->count;

        if(ret < 0)
                return ret;

        printk("Thirdy_Driver : thirdy_reqbufs OK\n");
        return 0;


}
```

[그림 11] V4L2 - REQBUF

[그림 10]은 V4L2 자체에서 제공되는 함수 코드이고, [그림 11]은 직접 작성한 코드를 나타낸 것이다. 두 함수가 매우 비슷한 구조로 작성되어 있는 것을 볼 수 있는데 실질적으로 동작에는 큰 차이가 없으나 [그림 10]의 경우에는 Kernel에서 제공되는 Kernel 코드라 볼 수 있고, [그림 11]은 직접 작성한 코드라는 차이가 있다. 이는 수정을 쉽게 하느냐 못하느냐라는 큰 차이를 보여준다. 아무래도 vb2_ioctl_reqbufs()를 사용하려면 제공되는 틀에 맞춰 개발에 제한이 생기는데 직접 작성을 하게 되면 조금 유연하게 코드를 작성할 수 있다.

마지막으로 해당 함수를 직접 작성할 때 주의해야 할 점은 G_FMT_VID_CAP 명령 관련 함수나 G_FMT_VID_CAP_MPLANE 명령과 관련된 함수가 존재해야 한다. 이는 V4L2 Kernel 코드 중 check_fmt()에 조건문으로 명시되어 있다.

2.6. QUERYBUF

QUERYBUF(Query Buffer)는 특정 Buffer의 정보를 요청할 때 사용하는 명령이다. 이 명령은사용자 영역에서 넘어온 Buffer의 Index 값을 이용하여 Buffer의 Offset 정보를 얻어온다. 이 정보는 mmap()의 인자로 사용된다. mmap()이 수행되고 나면 해당 Buffer는 사용자 영역의 Buffer와 Mapping된다.

```
struct v4l2_buffer buf = {0};
buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
buf.memory = V4L2_MEMORY_MMAP;
buf.index = 0;
if(-1 == xioctl(fd, VIDIOC_QUERYBUF, &buf))
{
    perror("Querying Buffer");
    printf("\nVIDIOC_QUERYBUF\n");
    return 1;
}

buffer = mmap (NULL, buf.length, PROT_READ | PROT_WRITE, MAP_SHARED, fd, buf.m.offset);
printf("Length: %d\nAddress: %p\n", buf.length, buffer);
printf("Image Length: %d\n", buf.bytesused);
```

[그림 12] V4L2 - Query Buffer and mmap()

[그림 12]는 Application 코드 중 QueryBuf와 mmap() 부분을 나타낸 것이다. QUERYBUF 명령을 요청하기 전에 v4l2_buffer 구조체의 index 변수에 0을 저장하는 것을 볼 수 있다. 이는 앞서 설명한 Buffer의 Index 값을 나타내는 것이다. 이 후 작업이 모두 수행되고 나면 v4l2_buffer 구조체에 저장되어 있는 데이터를 활용하여 mmap()을 호출한다. 이때 Mapping 되는 사용자 영역의 Buffer는 Video Device에서 얻어온 영상 정보를 담고 있게 된다.

```
int vb2_ioctl_querybuf(struct file *file, void *priv, struct v4l2_buffer *p)
{
        struct video_device *vdev = video_devdata(file);

        /* No need to call vb2_queue_is_busy(), anyone can query buffers. */
        return vb2_querybuf(vdev->queue, p);
}
EXPORT_SYMBOL_GPL(vb2_ioctl_querybuf);
```

[그림 13] V4L2 - QUERYBUF(V4L2)

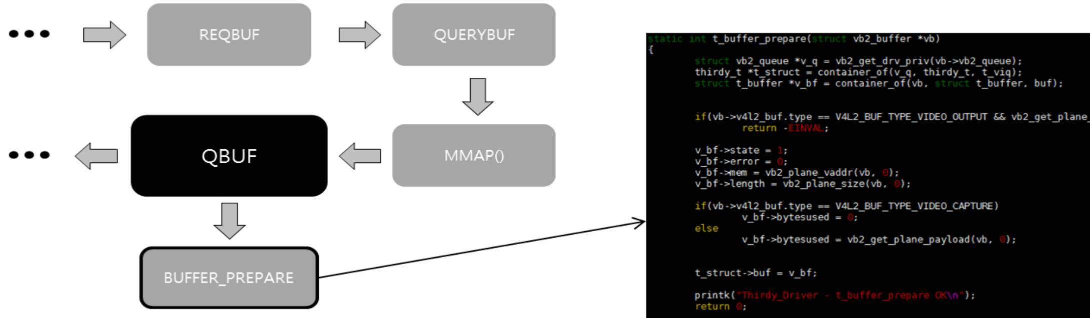
```
static int thirdy_querybuf(struct file *file, void *fh, struct v4l2_buffer *buf)
{
        thirdy_t *t_struct;
        int ret;

        if((t_struct = kzalloc(sizeof *t_struct, GFP_KERNEL)) == NULL)
                return -ENOMEM;

        t_struct = video_drvdata(file);

        mutex_lock(&t_struct->q_mutex);
        ret = vb2_querybuf(&t_struct->t_viq, buf);
        mutex_unlock(&t_struct->q_mutex);
        printk("Thirdy_Driver - thirdy_querybuf OK \n");

        return ret;
}
```

[그림 14] V4L2 – QUERYBUF

[그림 13]과 [그림 14]는 각각 V4L2에서 제공되는 함수와 직접 작성한 함수의 코드를 나타내는 것이다. 둘의 차이는 크게 없고 vb2_querybuf()의 정상적인 인자값을 넘겨주기 위해 직접 작성한 코드는 thirdy_t를 활용하였다.

2.7. QBUF

QBUF(Queue Buffer)는 Video Device에게 새로운 프레임을 요청하는 명령이다. 즉, 직접적으로 영상 정보를 요청하는 부분이라고 할 수 있다. 프레임을 요청하기 전에 Buffer_Prepare()를 호출하여 이미지 정보를 저장할 Buffer의 사이즈, 현재 상태 플래그, 메모리 주소 등의 정보를 설정한다.



[그림 15] V4L2 - QBUF and BUFFER_PREPARE

Buffer_Prepare()가 모두 수행되고 나면 이어서 프레임 요청을 마무리 한다. 이후에는 DQBUF 명령과 함께 반복하며 영상 정보를 요청한다.

※ **STREAMON**은 내용이 많아 DQBUF 명령에 대한 설명을 먼저 작성하였음 ※

2.8. DQBUF

DQBUF(DeQueue Buffer)는 QBUF 명령 수행으로 얻어진 프레임의 Index를 토대로 mmap()으로 Mapping 되어 있는 메모리 영역에서 실제 이미지 데이터를 가져오는 명령이다. 즉, Video Device에 요청하여 얻어진 프레임 정보를 이용하여 이미지 데이터를 사용자 영역으로 넘겨주는 일을 수행한다.

```
int vb2_ioctl_dqbuf(struct file *file, void *priv, struct v4l2_buffer *p)
{
        struct video_device *vdev = video_devdata(file);

        if (vb2_queue_is_busy(vdev, file))
                return -EBUSY;
        return vb2_dqbuf(vdev->queue, p, file->f_flags & O_NONBLOCK);
}
EXPORT_SYMBOL_GPL(vb2_ioctl_dqbuf);
```

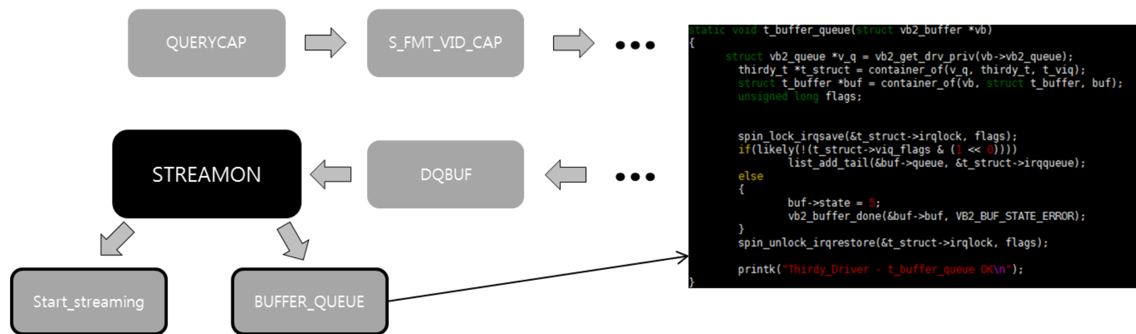
[그림 16] V4L2 - DQBUF(V4L2)

```
static int thirdy_dqbuf(struct file *file, void *fh, struct v4l2_buffer *buf)
{
        thirdy_t *t_struct;
        int ret;

        if((t_struct = kzalloc(sizeof *t_struct, GFP_KERNEL)) == NULL)
                return -ENOMEM;

        t_struct = video_drvdata(file);


        mutex_lock(&t_struct->q_mutex);
        ret = vb2_dqbuf(&t_struct->t_viq, buf, file->f_flags & O_NONBLOCK);
        mutex_unlock(&t_struct->q_mutex);
        printk("Thirdy_Driver - thirdy_dqbuf OK\n");
        return ret;

}
```

[그림 17] V4L2 - DQBUF

2.9. STREAMON

STREAMON(Streaming On)은 Video Device의 Stream 기능을 활성화 시키는 명령이다. Stream이란 일반적인 데이터나 패킷, 비트 등의 값이 연속성을 갖는 흐름을 말한다. 즉, Video Device의 Stream을 활성화 시키면 연속적인 이미지 정보를 얻을 수 있다는 것이다. 하지만 V4L2 Driver에서 STREAMON의 역할은 Device의 Stream을 활성화 시키는 것뿐만 아니라 얻어온 이미지 정보에 대한 해독(Decode)까지 한다. Decode가 끝나면 해당 이미지에 대한 정보를 DQBUF가 가져갈 수 있도록 Buffer에 저장한다. 이러한 일련의 작업들을 수행하기 전에 STREAMON 명령은 Buffer들을 Queue에 연결하는 동작도 수행하는데 이때 호출 되는 함수가 BUFFER_QUEUE()이다.



[그림 18] V4L2 - STREAMON and Functions

Queue에 Buffer를 연결하는 작업이 끝나고 나면 Start_Streaming()을 호출하여 실질적인 Video Device의 Stream을 활성화 시키는 동작이 수행된다. 이 함수가 동작하는 과정에서 앞서 언급했던 Decoding과 관련된 함수들이 호출되고 Buffer에 저장하는 동작이 반복된다. 이후 Application에서 STREAMON 명령을 요청하게 되더라도 이미 반복적으로 동작하고 있기 때문에 요청은 무시되고 Stop_Streaming()이 호출되기 전까지 동작하게 된다. 하지만 Stream이 활성화 되어있다고 해서 사용자 영역에 데이터가 전달되지는 않는다. STREAMON 명령 전후에 QBUF와 DQBUF 명령을 요청해야 Application은 이미지 정보를 얻어 올 수 있는 것이다.

```
IplImage * capture_image(int fd)
{
    struct v4l2_buffer buf = {0};
    buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
    buf.memory = V4L2_MEMORY_MMAP;
    buf.index = 0;
    if(-1 == xioctl(fd, VIDIOC_QBUF, &buf))
    {
        perror("Query Buffer");
        return NULL;
    }

    if(-1 == xioctl(fd, VIDIOC_STREAMON, &buf.type))
    {
        perror("Start Capture");
        return NULL;
    }

    fd_set fds;
    FD_ZERO(&fds);
    FD_SET(fd, &fds);
    struct timeval tv = {0};
    tv.tv_sec = 2;
    int r = select(fd+1, &fds, NULL, NULL, &tv);
    if(-1 == r)
    {
        perror("Waiting for Frame");
        return NULL;
    }

    if(-1 == xioctl(fd, VIDIOC_DQBUF, &buf))
    {
        perror("Retrieving Frame");
        return NULL;
    }

    IplImage* frame;
    CvMat cvmat = cvMat(240, 320, CV_8UC3, (void*)buffer);
    frame = cvDecodeImage(&cvmat, 1);

    return frame;
}
```

[그림 19] V4L2 - QBUF, STREAMON and DQBUF

앞서 언급한 내용들만 보면 STREAON 명령 자체가 다른 명령과 다를 것 없이 단순한 것 같지만 실제 Driver를 작성하는 과정에서는 결코 단순하지 않다. Start_Stream()이 동작되는 중간에 Decoding을 위한 함수들이 호출되는데 해당 함수들을 작성하기 위해선 Video Device와 USB Device의 동작과정, 그리고 이미지 Codec에 대한 전반적인 지식이 필요하다고 생각된다. 하지만 uvcvideo 코드를 적절히 참고한다면 최소한 V4L2 Driver가 정상적으로 동작할 수 있도록 작성 할 수 있을 것이다.

```c
static void uvc_video_decode_isoc(struct urb *urb, struct uvc_streaming *stream,
        struct uvc_buffer *buf)
{
        u8 *mem;
        int ret, i;

        for (i = 0; i < urb->number_of_packets; ++i) {
                if (urb->iso_frame_desc[i].status < 0) {
                        uvc_trace(UVC_TRACE_FRAME, "USB isochronous frame "
                                "lost (%d).\n", urb->iso_frame_desc[i].status);
                        /* Mark the buffer as faulty. */
                        if (buf != NULL)
                                buf->error = 1;
                        continue;
                }

                /* Decode the payload header. */
                mem = urb->transfer_buffer + urb->iso_frame_desc[i].offset;
                do {
                        ret = uvc_video_decode_start(stream, buf, mem,
                                urb->iso_frame_desc[i].actual_length);
                        if (ret == -EAGAIN) {
                                uvc_video_validate_buffer(stream, buf);
                                buf = uvc_queue_next_buffer(&stream->queue, buf);
                        }
                } while (ret == -EAGAIN);

                if (ret < 0)
                        continue;

                /* Decode the payload data. */
                uvc_video_decode_data(stream, buf, mem + ret, urb->iso_frame_desc[i].actual_length - ret);

                /* Process the header again. */
                uvc_video_decode_end(stream, buf, mem, urb->iso_frame_desc[i].actual_length);

                if (buf->state == UVC_BUF_STATE_READY) {
                        uvc_video_validate_buffer(stream, buf);
                        buf = uvc_queue_next_buffer(&stream->queue, buf);
                }
        }
}
```

[그림 20] V4L2 - Decode Code(uvcvideo) 일부

[그림 20]은 uvcvideo 코드 중 Decode와 관련된 함수 일부를 나타낸 것이다. 이 함수를 시작으로 Decoding 작업이 수행된다. 코드를 작성할 때 이 함수를 시작으로 Decode 부분을 작성해 나간다면 큰 문제 없이 진행 할 수 있을 것이다.

Decode 부분은 아직 분석을 다 하지 못했고 이해를 제대로 한 것이 아니기 때문에 더 기술하는 것이 의미가 없을 것으로 보인다.

3. 문제점 및 해결

문제점	분석 결과	해결방안
vidioc_reqbufs 함수 에러	특정 함수가 코드 내에 존재해야함	vidioc_g_fmt_vid_cap 함수 선언
VIDIOC_S_FMT_VID_CAP 함수 에러	V4L2 내부 함수에 필요한 데이터가 제대로 지정되지 않음	File->private_data 구조체에 필요 데이터 저장
Start_Streaming 함수 에러	Queue와 Buffer가 제대로 지정되지 않음	Queue_setup(), Buffer_queue(), Buffer_prepare()에서 사용되는 Queue와 Buffer 통일
VIDIOC_S_FMT_VID_CAP 함수 구현 문제	개발 초기에 함수 내용 자체를 제대로 분석을 하지 못함	미해결
Codec 부분 문제	코드 내용 분석을 제대로 못함	Uvcvideo Source에서 해당 부분 Copy
화면 출력 시 하단부 번짐현상	원인을 찾지 못함	임시방편으로 출력 해상도를 640x480으로 조정 (이후 번짐현상 없음)

4. 결과

아직 많은 부분들을 제대로 분석하지도 못했고 구현이 안된 부분도 많은 상태이지만 원하는 기능은 모두 구현이 되었다. 최초 개발 목표는 uvcvideo 소스에서 최대한 불필요한 부분을 지우고 필요한 부분만 구현해서 동작하는 Driver를 만드는 것이었다. 다소 부족한 부분이 있지만 첫 개발에 정상작동을 하는 드라이버를 개발했다는 것에 의의를 두었다.

동작 영상 : https://youtu.be/-VcN9z3zBMk

5. 참고

CAMERA DRIVER(V4L2) 에 관한 간략한 설명 - http://onecellboy.tistory.com/43

The Video4Linux2 API: an introduction - http://lwn.net/Articles/203924/

Video4Linux2 part 2: registration and open() - http://lwn.net/Articles/204545/

Video4Linux2 part 3: Basic ioctl() handling - http://lwn.net/Articles/206765/

Video4Linux2 part 6b: Streaming I/O - http://lwn.net/Articles/240667/

V4L2(Video For Linux 2) by Vladimir Davydov - http://hybridego.net/entry/V4L2Video-For-Linux-2-by-Vladimir-Davydov

UVCVIDEO Source - http://www.ideasonboard.org/uvc/

6. V4L2 Application Code

```
#include <errno.h>
#include <fcntl.h>
#include <linux/videodev2.h>
#include <stdint.h>
#include <stdio.h>
#include <string.h>
#include <sys/ioctl.h>
#include <sys/mman.h>
#include <unistd.h>
#include <opencv/cv.h>
#include <opencv/highgui.h>

uint8_t *buffer;

static int xioctl(int fd, int request, void *arg)
{
        int r;

        do r = ioctl (fd, request, arg);
        while (-1 == r && EINTR == errno);

        return r;
}

int print_caps(int fd)
{
        char fourcc[5] = {0};
        struct v4l2_capability caps = {};
        if (-1 == xioctl(fd, VIDIOC_QUERYCAP, &caps))
        {
                perror("Querying Capabilities");
                return 1;
        }

        printf( "Driver Caps:₩n"
                "  Driver: ₩"%s₩"₩n"
                "  Card: ₩"%s₩"₩n"
                "  Bus: ₩"%s₩"₩n"
```

```
                    "   Version: %d.%d₩n"
                    "   Capabilities: %08x₩n",
                    caps.driver,
                    caps.card,
                    caps.bus_info,
                    (caps.version>>16)&&0xff,
                    (caps.version>>24)&&0xff,
                    caps.capabilities);


        struct v4l2_format fmt = {0};
        fmt.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
        fmt.fmt.pix.width = 640;
        fmt.fmt.pix.height = 480;
        fmt.fmt.pix.pixelformat = V4L2_PIX_FMT_MJPEG;
        fmt.fmt.pix.field = V4L2_FIELD_NONE;

        if (-1 == xioctl(fd, VIDIOC_S_FMT, &fmt))
        {
                perror("Setting Pixel Format");
                return 1;
        }

        strncpy(fourcc, (char *)&fmt.fmt.pix.pixelformat, 4);
        printf( "Selected Camera Mode:₩n"
                    "   Width: %d₩n"
                    "   Height: %d₩n"
                    "   PixFmt: %s₩n"
                    "   Field: %d₩n",
                    fmt.fmt.pix.width,
                    fmt.fmt.pix.height,
                    fourcc,
                    fmt.fmt.pix.field);
        return 0;
}

int init_mmap(int fd)
{
    struct v4l2_requestbuffers req = {0};
```

```
	req.count = 1;
	req.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
	req.memory = V4L2_MEMORY_MMAP;

	if (-1 == xioctl(fd, VIDIOC_REQBUFS, &req))
	{
		perror("Requesting Buffer");
		printf("₩nVIDIOC_REQBUFS₩n");
		return 1;
	}

	struct v4l2_buffer buf = {0};
	buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
	buf.memory = V4L2_MEMORY_MMAP;
	buf.index = 0;
	if(-1 == xioctl(fd, VIDIOC_QUERYBUF, &buf))
	{
		perror("Querying Buffer");
		printf("₩nVIDIOC_QUERYBUF₩n");
		return 1;
	}

	buffer = (uint8_t *)mmap (NULL, buf.length, PROT_READ | PROT_WRITE, MAP_SHARED, fd,
buf.m.offset);
	printf("Length: %d₩nAddress: %p₩n", buf.length, buffer);
	printf("Image Length: %d₩n", buf.bytesused);

	return 0;
}

IplImage * capture_image(int fd)
{
		struct v4l2_buffer buf = {0};
		buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
		buf.memory = V4L2_MEMORY_MMAP;
		buf.index = 0;
		if(-1 == xioctl(fd, VIDIOC_QBUF, &buf))
		{
				perror("Query Buffer");
```

```
		return NULL;
	}

	if(-1 == xioctl(fd, VIDIOC_STREAMON, &buf.type))
	{
		perror("Start Capture");
		return NULL;
	}

	fd_set fds;
	FD_ZERO(&fds);
	FD_SET(fd, &fds);
	struct timeval tv = {0};
	tv.tv_sec = 2;
	int r = select(fd+1, &fds, NULL, NULL, &tv);
	if(-1 == r)
	{
		perror("Waiting for Frame");
		return NULL;
	}

	if(-1 == xioctl(fd, VIDIOC_DQBUF, &buf))
	{
		perror("Retrieving Frame");
		return NULL;
	}

	IplImage* frame;
	CvMat cvmat = cvMat(240, 320, CV_8UC3, (void*)buffer);
	frame = cvDecodeImage(&cvmat, 1);

	return frame;
}
```

7. V4L2 Drver Code

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/init.h>
#include <linux/usb.h>
#include <asm/atomic.h>
#include <asm/unaligned.h>
#include <media/v4l2-common.h>
#include <media/media-device.h>
#include <media/v4l2-device.h>
#include <media/v4l2-ioctl.h>
#include <media/videobuf2-core.h>
#include <media/videobuf2-dma-contig.h>
#include <media/videobuf2-vmalloc.h>
#include <uapi/linux/usb/video.h>
#include <trace/events/v4l2.h>

#define T_STREAM_EOH                    (1 << 7)
#define T_STREAM_ERR                    (1 << 6)
#define T_STREAM_STI                    (1 << 5)
#define T_STREAM_RES                    (1 << 4)
#define T_STREAM_SCR                    (1 << 3)
#define T_STREAM_PTS                    (1 << 2)
#define T_STREAM_EOF                    (1 << 1)
#define T_STREAM_FID                    (1 << 0)

#define DEVICE_VENDOR 0x05a3
#define DEVICE_PRODUCT_ID 0x9230
#define T_GUID_FORMAT_MJPEG { 'M',  'J',  'P',  'G', 0x00, 0x00, 0x10, 0x00, ₩
          0x80, 0x00, 0x00, 0xaa, 0x00, 0x38, 0x9b, 0x71}
#define T_URBS 5

extern long video_ioctl2(struct file *file, unsigned int cmd, unsigned long arg);
const struct v4l2_file_operations thirdy_fops;
static void thirdy_release(struct video_device *vdev);
static const struct v4l2_ioctl_ops thirdy_ioctl_ops;
static struct vb2_ops thirdy_qops;

__u8 int_in_addr_gl;
__u8 int_out_addr_gl;
__u8 *int_buf_gl;
size_t int_in_len_gl;
int ep_interval_gl;
```

```
int probe_chk;
struct usb_interface *intf_gl;
__u8 t_bep_gl;

unsigned int t_clock_param = CLOCK_MONOTONIC;
static void thirdy_status_complete(struct urb *urb);

static struct usb_device_id thirdy_ids[] = {
	{USB_DEVICE(DEVICE_VENDOR, DEVICE_PRODUCT_ID)},
	{}
};
MODULE_DEVICE_TABLE(usb, thirdy_ids);

struct thirdy_format
{
	char *name;
	__u8 guid[16];
	__u32 fcc;
	int depth;
};

static struct thirdy_format t_fmts[] = {
	{
		.name		= "MJPEG",
		.guid		= T_GUID_FORMAT_MJPEG,
		.fcc		= V4L2_PIX_FMT_MJPEG,
		.depth		= 16,
	},
	{}
};

struct ctrl_data
{
	__u16 bmHint;
	__u8   bFormatIndex;
	__u8   bFrameIndex;
	__u32 dwFrameInterval;
	__u16 wKeyFrameRate;
	__u16 wPFrameRate;
	__u16 wCompQuality;
	__u16 wCompWindowSize;
	__u16 wDelay;
	__u32 dwMaxVideoFrameSize;
```

```
	__u32 dwMaxPayloadTransferSize;
	__u32 dwClockFrequency;
	__u8	bmFramingInfo;
	__u8	bPreferedVersion;
	__u8	bMinVersion;
	__u8	bMaxVersion;
};

enum t_buffer_state {
	T_BUF_STATE_IDLE		= 0,
	T_BUF_STATE_QUEUED	= 1,
	T_BUF_STATE_ACTIVE	= 2,
	T_BUF_STATE_READY		= 3,
	T_BUF_STATE_DONE		= 4,
	T_BUF_STATE_ERROR		= 5,
};

struct t_buffer
{
	struct vb2_buffer buf;
	struct list_head queue;

	enum t_buffer_state state;
	unsigned int error;

	void *mem;
	unsigned int length;
	unsigned int bytesused;

	u32 pts;
};

struct t_stats_frame
{
	unsigned int size;		/* Number of bytes captured */
	unsigned int first_data;	/* Index of the first non-empty packet */

	unsigned int nb_packets;	/* Number of packets */
	unsigned int nb_empty;		/* Number of empty packets */
	unsigned int nb_invalid;	/* Number of packets with an invalid header */
	unsigned int nb_errors;	/* Number of packets with the error bit set */

	unsigned int nb_pts;		/* Number of packets with a PTS timestamp */
```

```
	unsigned int nb_pts_diffs;         /* Number of PTS differences inside a frame */
	unsigned int last_pts_diff;        /* Index of the last PTS difference */
	bool has_initial_pts;              /* Whether the first non-empty packet has a PTS */
	bool has_early_pts;                /* Whether a PTS is present before the first non-empty packet */
	u32 pts;                           /* PTS of the last packet */

	unsigned int nb_scr;               /* Number of packets with a SCR timestamp */
	unsigned int nb_scr_diffs;         /* Number of SCR.STC differences inside a frame */
	u16 scr_sof;                       /* SCR.SOF of the last packet */
	u32 scr_stc;                       /* SCR.STC of the last packet */
};

struct t_stats_stream
{
	struct timespec start_ts;          /* Stream start timestamp */
	struct timespec stop_ts;           /* Stream stop timestamp */

	unsigned int nb_frames;            /* Number of frames */

	unsigned int nb_packets;           /* Number of packets */
	unsigned int nb_empty;             /* Number of empty packets */
	unsigned int nb_invalid;           /* Number of packets with an invalid header */
	unsigned int nb_errors;            /* Number of packets with the error bit set */

	unsigned int nb_pts_constant;      /* Number of frames with constant PTS */
	unsigned int nb_pts_early;         /* Number of frames with early PTS */
	unsigned int nb_pts_initial;       /* Number of frames with initial PTS */

	unsigned int nb_scr_count_ok;      /* Number of frames with at least one SCR per non empty packet
*/
	unsigned int nb_scr_diffs_ok;      /* Number of frames with varying SCR.STC */
	unsigned int scr_sof_count;        /* STC.SOF counter accumulated since stream start */
	unsigned int scr_sof;              /* STC.SOF of the last packet */
	unsigned int min_sof;              /* Minimum STC.SOF value */
	unsigned int max_sof;              /* Maximum STC.SOF value */
};

struct t_clock
{
	struct t_clock_sample
	{
		u32 dev_stc;
		u16 dev_sof;
```

```
                    struct timespec host_ts;
                    u16 host_sof;
          } *samples;

          unsigned int head;
          unsigned int count;
          unsigned int size;

     u16 last_sof;
          u16 sof_offset;
          spinlock_t lock;
};

typedef struct
{
          struct usb_device *u_dev;
          struct usb_interface *intf;
          struct urb *ctrl_urb;
          struct usb_ctrlrequest ctrl_req;
          struct video_device *v_dev;
          struct v4l2_device v4l2_dev;
          struct media_device m_dev;
          struct thirdy_format *t_fmt;
          struct v4l2_format *v4l2_fmt;
          struct vb2_queue t_viq;
          struct v4l2_fh t_fh;
          struct v4l2_fh *t_fh_2;
          struct ctrl_data *t_data;
          struct vb2_buffer t_vb;
          struct vb2_alloc_ctx *alloc_ctx;
          struct t_buffer *buf;
          struct usb_host_endpoint *t_ep;
          struct mutex mutex;
          struct mutex q_mutex;
          size_t length;
          int intfnum;
          int t_width;
          int t_height;
          __u8 *status;
          __u8 int_in_addr;
          __u8 int_out_addr;
          __u8 *int_buf;
          size_t int_in_len;
```

```
	int ep_interval;
	int users;
	struct v4l2_prio_state prio;
	u32 caps;
	struct list_head irqqueue;
	struct list_head t_queue;
	spinlock_t irqlock;
	struct t_clock clock;
	__u8 bEndpointAddress;
	struct urb *urb[T_URBS];
	char *urb_buffer[T_URBS];
	dma_addr_t urb_dma[T_URBS];
	unsigned int urb_size;

	struct
	{
		struct t_stats_frame frame;
		struct t_stats_stream stream;
	} stats;

	unsigned int viq_flags;
	__u32 sequence;
	__u8 last_fid;
	u32 fourcc;
}thirdy_t;

static void t_video_decode_isoc(struct urb *urb, thirdy_t *t_struct, struct t_buffer *buf);
static int  thirdy_q_ctrl(thirdy_t *t_struct, __u8 query, __u8 unit, __u8 intfnum, __u8 cs, void *data, __u16 size,
int timeout);

static int thirdy_probe(struct usb_interface *interface, const struct usb_device_id *id)
{
	thirdy_t *t_struct;
	int ret;
	__u8 *data;
	__u16 size = 26;
	u32 interval;
	u32 bandwidth;
	struct usb_host_interface *alts; // = t_struct->intf->cur_altsetting;
	unsigned char *buffer; // alts->extra;
	__u8 *set_data;
	__u16 set_size = 26;
	__u8 *get_data;
```

```
	__u16 get_size = 26;
	unsigned int pipe;
	int intval = 0, i;
	struct usb_host_interface *iface_desc;
	struct usb_endpoint_descriptor *endpoint;
	struct usb_device *udev = interface_to_usbdev(interface);

	if((t_struct = kzalloc(sizeof *t_struct, GFP_KERNEL)) == NULL)
			return -ENOMEM;

	t_struct->u_dev = usb_get_dev(udev);
	t_struct->intf = usb_get_intf(interface);
	t_struct->intfnum = interface->cur_altsetting->desc.bInterfaceNumber;

	if(probe_chk != 0)
	{
		t_struct->int_in_addr = int_in_addr_gl;
		t_struct->int_out_addr = int_out_addr_gl;
		t_struct->int_buf = int_buf_gl;
		t_struct->int_in_len = int_in_len_gl;
		t_struct->ep_interval = ep_interval_gl;
		t_struct->bEndpointAddress = t_bep_gl;
	}

	iface_desc = interface->cur_altsetting;
	for(i = 0; i < iface_desc->desc.bNumEndpoints; ++i)
	{
			endpoint = &iface_desc->endpoint[i].desc;
			if(!t_struct->int_in_addr && usb_endpoint_is_int_in(endpoint))
			{
				t_struct->int_in_len		=		le16_to_cpu(endpoint-
>wMaxPacketSize);
				t_struct->int_in_addr = endpoint->bEndpointAddress;
				t_struct->int_buf = kmalloc(t_struct->int_in_len, GFP_KERNEL);
				t_struct->ep_interval = endpoint->bInterval;
			}

			if(!t_struct->int_out_addr && usb_endpoint_is_int_out(endpoint))
				t_struct->int_out_addr = endpoint->bEndpointAddress;
	}

	if(probe_chk == 0)
	{
```

```
		struct usb_host_interface *alts = t_struct->intf->cur_altsetting;
		unsigned char *buffer = alts->extra;
		int buflen = alts->extralen;

		struct usb_device *upsc_udev = t_struct->u_dev;
		struct usb_interface *upsc_intf;
		int i, n;

		n = buflen >= 12 ? buffer[11] : 0;

		for(i = 0; i < n; ++i)
		{
			struct usb_host_interface *for_alts;
			unsigned char *for_buffer;
			int for_buflen;

			upsc_intf = usb_ifnum_to_if(upsc_udev, buffer[12 + i]);
			if(upsc_intf == NULL)
				continue;

			for_alts = &upsc_intf->altsetting[0];
			for_buffer = for_alts->extra;
			for_buflen = for_alts->extralen;

			t_struct->bEndpointAddress = for_buffer[6];
		}
	}

	t_struct->m_dev.dev = &interface->dev;
	strcpy(t_struct->m_dev.bus_info, t_struct->u_dev->devpath);
	t_struct->m_dev.hw_revision = le16_to_cpu(t_struct->u_dev->descriptor.bcdDevice);
	t_struct->m_dev.driver_version = LINUX_VERSION_CODE;
	strlcpy(t_struct->m_dev.model, t_struct->u_dev->product, sizeof(t_struct->m_dev.model));

	if(media_device_register(&t_struct->m_dev) < 0)
	{
			printk("Thirdy_Driver : media_device_register error!₩n");
			media_device_unregister(&t_struct->m_dev);
			return -1;
	}
	t_struct->v4l2_dev.mdev = &t_struct->m_dev;

	if(v4l2_device_register(&interface->dev, &t_struct->v4l2_dev) < 0)
```

```
	{
			printk("Thirdy_Driver : v4l2_device_reister error! ₩n");
			v4l2_device_unregister(&t_struct->v4l2_dev);
			return -1;
	}

	t_struct->t_viq.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
	t_struct->t_viq.io_modes = VB2_MMAP | VB2_USERPTR | VB2_DMABUF;
	t_struct->t_viq.drv_priv = &t_struct->t_viq;
	t_struct->t_viq.buf_struct_size = sizeof(t_struct->t_vb);
	t_struct->t_viq.timestamp_flags		=		V4L2_BUF_FLAG_TIMESTAMP_MONOTONIC		|
V4L2_BUF_FLAG_TSTAMP_SRC_SOE;
	t_struct->t_viq.lock = &t_struct->mutex;
	t_struct->t_viq.mem_ops = &vb2_vmalloc_memops;
	t_struct->t_viq.ops = &thirdy_qops;
	t_struct->viq_flags = (1 << 1);

	ret = vb2_queue_init(&t_struct->t_viq);
	if(ret)
	{
			printk("Thirdy_Driver - vb2_queue_init error[%d]₩n", ret);
			return ret;
	}
	mutex_init(&t_struct->q_mutex);
	spin_lock_init(&t_struct->irqlock);
	INIT_LIST_HEAD(&t_struct->irqqueue);

	if((t_struct->t_data = kzalloc(sizeof(struct ctrl_data *), GFP_KERNEL)) == NULL)
			return -ENOMEM;

	data = kmalloc(size, GFP_KERNEL);
	if(data == NULL)
			return -ENOMEM;

	ret = thirdy_q_ctrl(t_struct, 0x87, 0, t_struct->intfnum, 0x01, data, size, 5000);
	if(ret < 0)
	{
		int_in_addr_gl = t_struct->int_in_addr;
		int_out_addr_gl = t_struct->int_out_addr;
		int_buf_gl = t_struct->int_buf;
		int_in_len_gl = t_struct->int_in_len;
		ep_interval_gl = t_struct->ep_interval;
		intf_gl = interface;
```

```
		t_bep_gl = t_struct->bEndpointAddress;

		probe_chk++;
		return -1;
	}

	t_struct->t_data->bmHint = le16_to_cpup((__le16 *)&data[0]);
	t_struct->t_data->bFormatIndex = data[2];
	t_struct->t_data->bFrameIndex = data[3];
	t_struct->t_data->dwFrameInterval = le32_to_cpup((__le32 *)&data[4]);
	t_struct->t_data->wKeyFrameRate = le16_to_cpup((__le16 *)&data[8]);
	t_struct->t_data->wPFrameRate = le16_to_cpup((__le16 *)&data[10]);
	t_struct->t_data->wCompQuality = le16_to_cpup((__le16 *)&data[12]);
	t_struct->t_data->wCompWindowSize = le16_to_cpup((__le16 *)&data[14]);
	t_struct->t_data->wDelay = le16_to_cpup((__le16 *)&data[16]);
	t_struct->t_data->dwMaxVideoFrameSize = get_unaligned_le32(&data[18]);
	t_struct->t_data->dwMaxPayloadTransferSize = get_unaligned_le32(&data[22]);

	alts = t_struct->intf->cur_altsetting;
	buffer = alts->extra;

	interval = t_struct->t_data->dwFrameInterval;
	bandwidth = 1024 * 768 / 8 * buffer[21];
	bandwidth *= 10000000 / interval + 1;
	bandwidth /= 1000;

	if(t_struct->u_dev->speed == USB_SPEED_HIGH)
		bandwidth /= 8;

	bandwidth += 12;
	bandwidth = max_t(u32, bandwidth, 1024);
	t_struct->t_data->dwMaxPayloadTransferSize = get_unaligned_le32(&bandwidth);

	set_data = kzalloc(set_size, GFP_KERNEL);
	if(set_data == NULL)
		return -ENOMEM;

	get_data = kzalloc(get_size, GFP_KERNEL);
	if(get_data == NULL)
		return -ENOMEM;

	*(__le16 *)&set_data[0] = cpu_to_le16(t_struct->t_data->bmHint);
	set_data[2] = t_struct->t_data->bFormatIndex;
```

```
set_data[3] = t_struct->t_data->bFrameIndex;
*(__le32 *)&set_data[4] = cpu_to_le32(t_struct->t_data->dwFrameInterval);
*(__le16 *)&set_data[8] = cpu_to_le16(t_struct->t_data->wKeyFrameRate);
*(__le16 *)&set_data[10] = cpu_to_le16(t_struct->t_data->wPFrameRate);
*(__le16 *)&set_data[12] = cpu_to_le16(t_struct->t_data->wCompQuality);
*(__le16 *)&set_data[14] = cpu_to_le16(t_struct->t_data->wCompWindowSize);
*(__le16 *)&set_data[16] = cpu_to_le16(t_struct->t_data->wDelay);
put_unaligned_le32(t_struct->t_data->dwMaxVideoFrameSize, &set_data[18]);
put_unaligned_le32(t_struct->t_data->dwMaxPayloadTransferSize, &set_data[22]);

ret = thirdy_q_ctrl(t_struct, 0x01, 0, t_struct->intfnum, 0x01, set_data, set_size, 5000);
ret = thirdy_q_ctrl(t_struct, 0x81, 0, t_struct->intfnum, 0x01, get_data, get_size, 5000);

t_struct->t_data->bmHint = le16_to_cpup((__le16 *)&get_data[0]);
t_struct->t_data->bFormatIndex = get_data[2];
t_struct->t_data->bFrameIndex = get_data[3];
t_struct->t_data->dwFrameInterval = le32_to_cpup((__le32 *)&get_data[4]);
t_struct->t_data->wKeyFrameRate = le16_to_cpup((__le16 *)&get_data[8]);
t_struct->t_data->wPFrameRate = le16_to_cpup((__le16 *)&get_data[10]);
t_struct->t_data->wCompQuality = le16_to_cpup((__le16 *)&get_data[12]);
t_struct->t_data->wCompWindowSize = le16_to_cpup((__le16 *)&get_data[14]);
t_struct->t_data->wDelay = le16_to_cpup((__le16 *)&get_data[16]);
t_struct->t_data->dwMaxVideoFrameSize = get_unaligned_le32(&get_data[18]);
t_struct->t_data->dwMaxPayloadTransferSize = get_unaligned_le32(&get_data[22]);

v4l2_prio_init(&t_struct->prio);

t_struct->v_dev = video_device_alloc();
if(t_struct->v_dev == NULL)
{
                printk("Thirdy_Driver : video_device_alloc error!₩n");
                kfree(t_struct->v_dev);
                return -ENOMEM;
}

t_struct->status = kzalloc(16, GFP_KERNEL);
t_struct->ctrl_urb = usb_alloc_urb(0, GFP_KERNEL);

pipe = usb_rcvintpipe(t_struct->u_dev, t_struct->int_in_addr);
interval = t_struct->ep_interval;

if(intval > 16 && t_struct->u_dev->speed == USB_SPEED_HIGH)
        intval = fls(intval) - 1;
```

```
	usb_fill_int_urb(t_struct->ctrl_urb, t_struct->u_dev, pipe, t_struct->status, 16, thirdy_status_complete, 
t_struct, intval);

	t_struct->v_dev->v4l2_dev = &t_struct->v4l2_dev;
	t_struct->v_dev->fops = &thirdy_fops;
	t_struct->v_dev->ioctl_ops = &thirdy_ioctl_ops;
	t_struct->v_dev->release = thirdy_release;
	t_struct->v_dev->prio = &t_struct->prio;

	strlcpy(t_struct->v_dev->name, t_struct->u_dev->product, sizeof t_struct->v_dev->name);

	if((ret = video_register_device(t_struct->v_dev, VFL_TYPE_GRABBER, -1)) < 0)
	{
			printk("Thirdy_Driver : video_register_device error!₩n");
			video_unregister_device(t_struct->v_dev);
			return ret;
	}

	t_struct->caps |= V4L2_CAP_VIDEO_CAPTURE;
	usb_set_intfdata(interface, t_struct);
	video_set_drvdata(t_struct->v_dev, t_struct);
	printk("Thirdy_Driver : Probe OK₩n");

	return 0;
}

static void thirdy_disconnect(struct usb_interface *interface)
{
	thirdy_t *t_struct;
	t_struct = usb_get_intfdata(interface);
	usb_set_intfdata(interface, NULL);
	media_device_unregister(&t_struct->m_dev);
	v4l2_device_unregister(&t_struct->v4l2_dev);
	video_unregister_device(t_struct->v_dev);
	kfree(t_struct);
	printk("Thirdy_Driver : thirdy_disconnect OK₩n");
}

static void thirdy_release(struct video_device *vdev)
{
	video_unregister_device(vdev);
	printk("Thirdy_Driver : thirdy_release OK₩n");
}
```

```
static void thirdy_status_complete(struct urb *urb)
{
	thirdy_t *t_struct = urb->context;
	int len, ret;

	switch (urb->status)
	{
		case 0:
				break;
		case -ENOENT:		/* usb_kill_urb() called. */
		case -ECONNRESET:	/* usb_unlink_urb() called. */
		case -ESHUTDOWN:	/* The endpoint is being disabled. */
		case -EPROTO:		/* Device is disconnected (reported by some host controller). */
				return;
		default:
			printk("thirdy_status_complete - 1₩n");
			return;
	}

	len = urb->actual_length;
	if (len > 0) {
		switch (t_struct->status[0] & 0x0f)
		{
			case 1:
			printk("thirdy_status_complete - 2₩n");
					break;
			case 2:
			printk("thirdy_status_complete - 3₩n");
					break;
			default:
			printk("thirdy_status_complete - 4₩n");
					break;
		}
	}
	/* Resubmit the URB. */
	urb->interval = t_struct->t_ep->desc.bInterval;
	if ((ret = usb_submit_urb(urb, GFP_ATOMIC)) < 0)
	{
		printk("thirdy_status_complete - 5₩n");
	}
}
```

```
static int t_queue_setup(struct vb2_queue *vq, const struct v4l2_format *fmt, unsigned int *nbuffers, unsigned int *nplanes, unsigned int sizes[], void *alloc_ctxs[])
{
	struct vb2_queue *v_q = vb2_get_drv_priv(vq);
	thirdy_t *t_struct = container_of(v_q, thirdy_t,t_viq);
	fmt = t_struct->v4l2_fmt;

	if(fmt && fmt->fmt.pix.sizeimage < t_struct->t_data->dwMaxVideoFrameSize)
		return -EINVAL;

	*nplanes = 1;

	sizes[0] = fmt ? fmt->fmt.pix.sizeimage : t_struct->t_data->dwMaxVideoFrameSize;
	printk("Thirdy_Driver - t_queue_setup OK / %d₩n", *nbuffers);

	return 0;
}

static void t_buffer_queue(struct vb2_buffer *vb)
{
	struct vb2_queue *v_q = vb2_get_drv_priv(vb->vb2_queue);
	thirdy_t *t_struct = container_of(v_q, thirdy_t, t_viq);
	struct t_buffer *buf = container_of(vb, struct t_buffer, buf);
	unsigned long flags;


	spin_lock_irqsave(&t_struct->irqlock, flags);
	if(likely(!(t_struct->viq_flags & (1 << 0))))
			list_add_tail(&buf->queue, &t_struct->irqqueue);
	else
	{
			buf->state = 5;
			vb2_buffer_done(&buf->buf, VB2_BUF_STATE_ERROR);
	}
	spin_unlock_irqrestore(&t_struct->irqlock, flags);

	printk("Thirdy_Driver - t_buffer_queue OK₩n");
}

static int t_buffer_prepare(struct vb2_buffer *vb)
{
	struct vb2_queue *v_q = vb2_get_drv_priv(vb->vb2_queue);
	thirdy_t *t_struct = container_of(v_q, thirdy_t, t_viq);
```

```
	struct t_buffer *v_bf = container_of(vb, struct t_buffer, buf);


	if(vb->v4l2_buf.type == V4L2_BUF_TYPE_VIDEO_OUTPUT && vb2_get_plane_payload(vb, 0) > 
vb2_plane_size(vb, 0))
		return -EINVAL;

	v_bf->state = 1;
	v_bf->error = 0;
	v_bf->mem = vb2_plane_vaddr(vb, 0);
	v_bf->length = vb2_plane_size(vb, 0);

	if(vb->v4l2_buf.type == V4L2_BUF_TYPE_VIDEO_CAPTURE)
			v_bf->bytesused = 0;
	else
			v_bf->bytesused = vb2_get_plane_payload(vb, 0);

	t_struct->buf = v_bf;

	printk("Thirdy_Driver - t_buffer_prepare OK₩n");
	return 0;

}

static void t_buffer_finish(struct vb2_buffer *vb)
{
	printk("thirdy_buf_finish Start₩n");
}

struct usb_host_endpoint *t_find_endpoint(struct usb_host_interface *alts, __u8 epaddr)
{
	struct usb_host_endpoint *ep;
	unsigned int i;

	for (i = 0; i < alts->desc.bNumEndpoints; ++i)
	{
		ep = &alts->endpoint[i];
		if (ep->desc.bEndpointAddress == epaddr)
				return ep;
	}
	return NULL;
}
```

```
static unsigned int t_endpoint_max_bpi(struct usb_device *dev, struct usb_host_endpoint *ep)
{
	u16 psize;

	switch (dev->speed)
	{
		case USB_SPEED_SUPER:
			return le16_to_cpu(ep->ss_ep_comp.wBytesPerInterval);
		case USB_SPEED_HIGH:
			psize = usb_endpoint_maxp(&ep->desc);
			return (psize & 0x07ff) * (1 + ((psize >> 11) & 3));
		case USB_SPEED_WIRELESS:
			psize = usb_endpoint_maxp(&ep->desc);
			return psize;
		default:
			psize = usb_endpoint_maxp(&ep->desc);
			return psize & 0x07ff;
	}
}

static int t_alloc_urb_buffers(thirdy_t *t_struct, unsigned int size, unsigned int psize, gfp_t gfp_flags)
{
	unsigned int npackets;
	unsigned int i;

	if (t_struct->urb_size)
			return t_struct->urb_size / psize;

	npackets = DIV_ROUND_UP(size, psize);
	if (npackets > 32)
			npackets = 32;

	for (; npackets > 1; npackets /= 2)
	{
		for (i = 0; i < T_URBS; ++i)
		{
			t_struct->urb_size = psize * npackets;
#ifndef CONFIG_DMA_NONCOHERENT
			t_struct->urb_buffer[i] = usb_alloc_coherent(t_struct->u_dev, t_struct->urb_size,
gfp_flags | __GFP_NOWARN, &t_struct->urb_dma[i]);
#else
			t_struct->urb_buffer[i] = kmalloc(t_struct->urb_size, gfp_flags | __GFP_NOWARN);
#endif
```

```
                        if (!t_struct->urb_buffer[i])
                        {
                                printk("urb_buffer error₩n");
                break;
            }
                }

                if (i == T_URBS)
                {
                        printk("taub OK₩n");
                        return npackets;
                }
        }

        printk("taub failed₩n");
        return 0;
}

static void t_video_stats_update(thirdy_t *t_struct)
{
        struct t_stats_frame *frame = &t_struct->stats.frame;

        t_struct->stats.stream.nb_frames++;
        t_struct->stats.stream.nb_packets += t_struct->stats.frame.nb_packets;
        t_struct->stats.stream.nb_empty += t_struct->stats.frame.nb_empty;
        t_struct->stats.stream.nb_errors += t_struct->stats.frame.nb_errors;
        t_struct->stats.stream.nb_invalid += t_struct->stats.frame.nb_invalid;

        if (frame->has_early_pts)
                        t_struct->stats.stream.nb_pts_early++;
        if (frame->has_initial_pts)
                        t_struct->stats.stream.nb_pts_initial++;
        if (frame->last_pts_diff <= frame->first_data)
                        t_struct->stats.stream.nb_pts_constant++;
        if (frame->nb_scr >= frame->nb_packets - frame->nb_empty)
                        t_struct->stats.stream.nb_scr_count_ok++;
        if (frame->nb_scr_diffs + 1 == frame->nb_scr)
                        t_struct->stats.stream.nb_scr_diffs_ok++;

        memset(&t_struct->stats.frame, 0, sizeof(t_struct->stats.frame));
}

static inline void t_video_get_ts(struct timespec *ts)
```

```
{
        if (t_clock_param == CLOCK_MONOTONIC)
                ktime_get_ts(ts);
        else
                ktime_get_real_ts(ts);
}

static void t_video_clock_decode(thirdy_t *t_struct, struct t_buffer *buf, const __u8 *data, int len)
{
        struct t_clock_sample *sample;
        unsigned int header_size;
        bool has_pts = false;
        bool has_scr = false;
        unsigned long flags;
        struct timespec ts;
        u16 host_sof;
        u16 dev_sof;

    switch (data[1] & (T_STREAM_PTS | T_STREAM_SCR))
        {
                case T_STREAM_PTS | T_STREAM_SCR:
                        header_size = 12;
                        has_pts = true;
                        has_scr = true;
                        break;
                case T_STREAM_PTS:
                        header_size = 6;
                        has_pts = true;
                        break;
                case T_STREAM_SCR:
                        header_size = 8;
                        has_scr = true;
                        break;
                default:
                        header_size = 2;
                        break;
        }

        if (len < header_size)
                return;

        if (has_pts && buf != NULL)
                buf->pts = get_unaligned_le32(&data[2]);
```

```
	if (!has_scr)
		return;

	dev_sof = get_unaligned_le16(&data[header_size - 2]);
	if (dev_sof == t_struct->clock.last_sof)
		return;

	t_struct->clock.last_sof = dev_sof;

	host_sof = usb_get_current_frame_number(t_struct->u_dev);
	t_video_get_ts(&ts);

	if (t_struct->clock.sof_offset == (u16)-1)
	{
		u16 delta_sof = (host_sof - dev_sof) & 255;
		if (delta_sof >= 10)
			t_struct->clock.sof_offset = delta_sof;
		else
			t_struct->clock.sof_offset = 0;
	}

	dev_sof = (dev_sof + t_struct->clock.sof_offset) & 2047;

	spin_lock_irqsave(&t_struct->clock.lock, flags);

	sample = &t_struct->clock.samples[t_struct->clock.head];
	sample->dev_stc = get_unaligned_le32(&data[header_size - 6]);
	sample->dev_sof = dev_sof;
	sample->host_sof = host_sof;
	sample->host_ts = ts;

	t_struct->clock.head = (t_struct->clock.head + 1) % t_struct->clock.size;

	if (t_struct->clock.count < t_struct->clock.size)
		t_struct->clock.count++;

	spin_unlock_irqrestore(&t_struct->clock.lock, flags);
}

static void t_video_stats_decode(thirdy_t *t_struct, const __u8 *data, int len)
{
	unsigned int header_size;
```

```
bool has_pts = false;
bool has_scr = false;
u16 uninitialized_var(scr_sof);
u32 uninitialized_var(scr_stc);
u32 uninitialized_var(pts);

if (t_struct->stats.stream.nb_frames == 0 && t_struct->stats.frame.nb_packets == 0)
        ktime_get_ts(&t_struct->stats.stream.start_ts);

switch (data[1] & (T_STREAM_PTS | T_STREAM_SCR))
{
        case T_STREAM_PTS | T_STREAM_SCR:
                header_size = 12;
                has_pts = true;
                has_scr = true;
                break;
        case T_STREAM_PTS:
                header_size = 6;
                has_pts = true;
                break;
        case T_STREAM_SCR:
                header_size = 8;
                has_scr = true;
                break;
        default:
                header_size = 2;
                break;
}

/* Check for invalid headers. */
if (len < header_size || data[0] < header_size)
{
        t_struct->stats.frame.nb_invalid++;
        return;
}

/* Extract the timestamps. */
if (has_pts)
        pts = get_unaligned_le32(&data[2]);

if (has_scr)
{
        scr_stc = get_unaligned_le32(&data[header_size - 6]);
```

```
		scr_sof = get_unaligned_le16(&data[header_size - 2]);
	}

	/* Is PTS constant through the whole frame ? */
	if (has_pts && t_struct->stats.frame.nb_pts)
	{
		if (t_struct->stats.frame.pts != pts)
		{
			t_struct->stats.frame.nb_pts_diffs++;
			t_struct->stats.frame.last_pts_diff =t_struct->stats.frame.nb_packets;
		}
	}

	if (has_pts)
	{
		t_struct->stats.frame.nb_pts++;
		t_struct->stats.frame.pts = pts;
	}

	if (t_struct->stats.frame.size == 0)
	{
		if (len > header_size)
			t_struct->stats.frame.has_initial_pts = has_pts;
		if (len == header_size && has_pts)
			t_struct->stats.frame.has_early_pts = true;
	}

	if (has_scr && t_struct->stats.frame.nb_scr)
	{
		if (t_struct->stats.frame.scr_stc != scr_stc)
			t_struct->stats.frame.nb_scr_diffs++;
	}

	if (has_scr)
	{
		if (t_struct->stats.stream.nb_frames > 0 || t_struct->stats.frame.nb_scr > 0)
			t_struct->stats.stream.scr_sof_count += (scr_sof - t_struct->stats.stream.scr_sof) %
2048;
		t_struct->stats.stream.scr_sof = scr_sof;

		t_struct->stats.frame.nb_scr++;
		t_struct->stats.frame.scr_stc = scr_stc;
		t_struct->stats.frame.scr_sof = scr_sof;
```

```
		if (scr_sof < t_struct->stats.stream.min_sof)
			t_struct->stats.stream.min_sof = scr_sof;
		if (scr_sof > t_struct->stats.stream.max_sof)
			t_struct->stats.stream.max_sof = scr_sof;
	}

	/* Record the first non-empty packet number. */
	if (t_struct->stats.frame.size == 0 && len > header_size)
		t_struct->stats.frame.first_data = t_struct->stats.frame.nb_packets;

	/* Update the frame size. */
	t_struct->stats.frame.size += len - header_size;

	/* Update the packets counters. */
	t_struct->stats.frame.nb_packets++;
	if (len > header_size)
		t_struct->stats.frame.nb_empty++;

	if (data[1] & T_STREAM_ERR)
		t_struct->stats.frame.nb_errors++;
}

static int t_video_decode_start(thirdy_t *t_struct, struct t_buffer *buf, const __u8 *data, int len)
{
	__u8 fid;


	if (len < 2 || data[0] < 2 || data[0] > len)
	{
		t_struct->stats.frame.nb_invalid++;
		return -EINVAL;
	}

	fid = data[1] & (1 << 0);

	if (t_struct->last_fid != fid)
	{
		t_struct->sequence++;
		if (t_struct->sequence)
			t_video_stats_update(t_struct);
	}
```

```
	t_video_clock_decode(t_struct, buf, data, len);
	t_video_stats_decode(t_struct, data, len);

	if(buf == NULL)
	{
		t_struct->last_fid = fid;
		return -ENODATA;
	}

	if (data[1] & (1 << 6))
	{
		printk("Marking buffer as bad.₩n");
		buf->error = 1;
	}

	if (buf->state != 2)
	{
		struct timespec ts;

		if (fid == t_struct->last_fid)
		{
			//printk("Dropping payload.₩n");
			return -ENODATA;
		}
		t_video_get_ts(&ts);

		buf->buf.v4l2_buf.field = V4L2_FIELD_NONE;
		buf->buf.v4l2_buf.sequence = t_struct->sequence;
		buf->buf.v4l2_buf.timestamp.tv_sec = ts.tv_sec;
		buf->buf.v4l2_buf.timestamp.tv_usec = ts.tv_nsec / NSEC_PER_USEC;

		buf->state = 2;
	}


	if (fid != t_struct->last_fid && buf->bytesused != 0)
	{
		printk("Frame complete (FID bit toggled).₩n");
		buf->state = 3;
		return -EAGAIN;
	}

	t_struct->last_fid = fid;
```

```
	return data[0];
}

struct t_buffer *t_queue_next_buffer(thirdy_t *t_struct, struct t_buffer *buf)
{
	struct t_buffer *nextbuf;
	unsigned long flags;

	if (buf->error && (t_struct->viq_flags & (1 << 1)))
	{
		buf->error = 0;
		buf->state = T_BUF_STATE_QUEUED;
		buf->bytesused = 0;
		vb2_set_plane_payload(&buf->buf, 0, 0);
		return buf;
	}

	spin_lock_irqsave(&t_struct->irqlock, flags);
	list_del(&buf->queue);
	if (!list_empty(&t_struct->irqqueue))
		nextbuf = list_first_entry(&t_struct->irqqueue, struct t_buffer, queue);
	else
		nextbuf = NULL;
	spin_unlock_irqrestore(&t_struct->irqlock, flags);

	buf->state = buf->error ? VB2_BUF_STATE_ERROR : 4;
	vb2_set_plane_payload(&buf->buf, 0, buf->bytesused);
	vb2_buffer_done(&buf->buf, VB2_BUF_STATE_DONE);

	return nextbuf;
}

static void t_video_validate_buffer(thirdy_t *t_struct, struct t_buffer *buf)
{
	if (t_struct->t_data->dwMaxVideoFrameSize != buf->bytesused && 0)
		buf->error = 1;
}

static void t_video_decode_data(struct t_buffer *buf, const __u8 *data, int len)
{
	unsigned int maxlen, nbytes;
	void *mem;
```

```
        if (len <= 0)
                return;
        /* Copy the video data to the buffer. */
        maxlen = buf->length - buf->bytesused;
        mem = buf->mem + buf->bytesused;
        nbytes = min((unsigned int)len, maxlen);
        memcpy(mem, data, nbytes);
        buf->bytesused += nbytes;
        /* Complete the current frame if the buffer size was exceeded. */
        if (len > maxlen)
        {
                printk("Frame complete (overflow).₩n");
                buf->state = T_BUF_STATE_READY;
        }
}

static void t_video_decode_end(thirdy_t *t_struct, struct t_buffer *buf, const __u8 *data, int len)
{
        /* Mark the buffer as done if the EOF marker is set. */
        if (data[1] & T_STREAM_EOF && buf->bytesused != 0) {
                printk("Frame complete (EOF found).₩n");
                if (data[0] == len)
                        printk("EOF in empty payload.₩n");
                buf->state = T_BUF_STATE_READY;
        }
}

static void t_video_decode_isoc(struct urb *urb, thirdy_t *t_struct, struct t_buffer *buf)
{
        u8 *mem;
        int ret, i;

        for(i = 0; i < urb->number_of_packets; ++i)
        {
                if (urb->iso_frame_desc[i].status < 0)
                {
                        printk("USB isochronous frame₩n");
                        /* Mark the buffer as faulty. */
                        if (buf != NULL)
                                buf->error = 1;
                        continue;
                }
              /* Decode the payload header. */
```

```
		mem = urb->transfer_buffer + urb->iso_frame_desc[i].offset;
		do
		{
			ret = t_video_decode_start(t_struct, buf, mem, urb->iso_frame_desc[i].actual_length);
			if (ret == -EAGAIN)
			{
				t_video_validate_buffer(t_struct, buf);
				buf = t_queue_next_buffer(t_struct, buf);
			}
		} while (ret == -EAGAIN);

		if (ret < 0)
			continue;

		t_video_decode_data(buf, mem + ret, urb->iso_frame_desc[i].actual_length - ret);
		t_video_decode_end(t_struct, buf, mem, urb->iso_frame_desc[i].actual_length);

		if (buf->state == T_BUF_STATE_READY)
		{
			t_video_validate_buffer(t_struct, buf);
			buf = t_queue_next_buffer(t_struct, buf);
		}
	}
}

static void t_video_complete(struct urb *urb)
{
	thirdy_t *t_struct = urb->context;
	struct t_buffer *buf = NULL;
	unsigned long flags;
	int ret;

	switch (urb->status)
	{
		case 0:
				break;
		default:
				printk("Non-zero status (%d) in video completion handler.₩n", urb->status);
		case -ENOENT:		/* usb_kill_urb() called. */
			printk("ENOENT₩n");
		case -ECONNRESET:	/* usb_unlink_urb() called. */
```

```
                case -ESHUTDOWN:            /* The endpoint is being disabled. */
                                        //t_queue_cancel(queue, urb->status == -ESHUTDOWN);
                        printk("endpoint is being disabled₩n");
                        return;
        }

        spin_lock_irqsave(&t_struct->irqlock, flags);
        if (!list_empty(&t_struct->irqqueue))
                buf = list_first_entry(&t_struct->irqqueue, struct t_buffer, queue);
        spin_unlock_irqrestore(&t_struct->irqlock, flags);

//      stream->decode(urb, stream, buf);
        t_video_decode_isoc(urb, t_struct, buf);

        if ((ret = usb_submit_urb(urb, GFP_ATOMIC)) < 0)
                printk("Failed to resubmit video URB (%d).₩n", ret);
}

static int t_init_video_isoc(thirdy_t *t_struct, struct usb_host_endpoint *ep, gfp_t gfp_flags)
{
        struct urb *urb;
        unsigned int npackets, i, j;
        u16 psize;
        u32 size;

        psize = t_endpoint_max_bpi(t_struct->u_dev, ep);
        size = t_struct->t_data->dwMaxVideoFrameSize;

        npackets = t_alloc_urb_buffers(t_struct, size, psize, gfp_flags);
        if (npackets == 0)
                return -ENOMEM;

        size = npackets * psize;

        for (i = 0; i < T_URBS; ++i)
        {
                urb = usb_alloc_urb(npackets, gfp_flags);
                printk("tivi %d₩n", i);
                if (urb == NULL)
                {
                                        //uvc_uninit_video(stream, 1);
                        printk("aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa₩n");
                        return -ENOMEM;
```

```
			}
			urb->dev = t_struct->u_dev;
			urb->context = t_struct;
			urb->pipe = usb_rcvisocpipe(t_struct->u_dev, ep->desc.bEndpointAddress);
#ifndef CONFIG_DMA_NONCOHERENT
			urb->transfer_flags = URB_ISO_ASAP | URB_NO_TRANSFER_DMA_MAP;
			urb->transfer_dma = t_struct->urb_dma[i];
#else
			urb->transfer_flags = URB_ISO_ASAP;
#endif
			urb->interval = ep->desc.bInterval;
			urb->transfer_buffer = t_struct->urb_buffer[i];
			urb->complete = t_video_complete; //----
			urb->number_of_packets = npackets;
			urb->transfer_buffer_length = size;

			for (j = 0; j < npackets; ++j)
			{
				urb->iso_frame_desc[j].offset = j * psize;
				urb->iso_frame_desc[j].length = psize;
			}
			t_struct->urb[i] = urb;
		}
		return 0;
}

static int t_video_clock_init(thirdy_t *t_struct)
{
		struct t_clock *clock = &t_struct->clock;

		spin_lock_init(&clock->lock);
		clock->size = 32;
		clock->samples = kmalloc(clock->size * sizeof(*clock->samples), GFP_KERNEL);

		if(clock->samples == NULL)
			return -ENOMEM;

		clock->head = 0;
		clock->count = 0;
		clock->last_sof = -1;
		clock->sof_offset = -1;

		return 0;
```

```
}

static int t_commit_video(thirdy_t *t_struct)
{
	__u8 *set_data;
	__u16 set_size = 26;
	int ret;

	set_data = kzalloc(set_size, GFP_KERNEL);
	if(set_data == NULL)
		return -ENOMEM;

	*(__le16 *)&set_data[0] = cpu_to_le16(t_struct->t_data->bmHint);
	set_data[2] = t_struct->t_data->bFormatIndex;
	set_data[3] = t_struct->t_data->bFrameIndex;
	*(__le32 *)&set_data[4] = cpu_to_le32(t_struct->t_data->dwFrameInterval);
	*(__le16 *)&set_data[8] = cpu_to_le16(t_struct->t_data->wKeyFrameRate);
	*(__le16 *)&set_data[10] = cpu_to_le16(t_struct->t_data->wPFrameRate);
	*(__le16 *)&set_data[12] = cpu_to_le16(t_struct->t_data->wCompQuality);
	*(__le16 *)&set_data[14] = cpu_to_le16(t_struct->t_data->wCompWindowSize);
	*(__le16 *)&set_data[16] = cpu_to_le16(t_struct->t_data->wDelay);
	put_unaligned_le32(t_struct->t_data->dwMaxVideoFrameSize, &set_data[18]);
	put_unaligned_le32(t_struct->t_data->dwMaxPayloadTransferSize, &set_data[22]);

	ret = thirdy_q_ctrl(t_struct, 0x01, 0, t_struct->intfnum, 0x02, set_data, set_size, 5000);
	kfree(set_data);

	if(ret != set_size)
		ret = -EIO;

	return ret;
}

static void t_video_stats_start(thirdy_t *t_struct)
{
	memset(&t_struct->stats, 0, sizeof(t_struct->stats));
	t_struct->stats.stream.min_sof = 2048;
}

static int t_init_video(thirdy_t *t_struct, gfp_t gfp_flags)
{
	struct usb_interface *intf = t_struct->intf;
	struct usb_host_endpoint *ep;
```

```
unsigned int i;
int ret = 0;

t_struct->sequence = -1;
t_struct->last_fid = -1;

t_video_stats_start(t_struct);


if(intf->num_altsetting > 1)
{
        struct usb_host_endpoint *best_ep = NULL;
        unsigned int best_psize = UINT_MAX;
        unsigned int bandwidth;
        unsigned int uninitialized_var(altsetting);
        int intfnum = t_struct->intfnum;

        bandwidth = t_struct->t_data->dwMaxPayloadTransferSize;

        if(bandwidth == 0)
                bandwidth = 1;
        else
                printk("bandwidth != 0₩n");

        for(i = 0; i < intf->num_altsetting; ++i)
        {
                struct usb_host_interface *alts;
                unsigned int psize;
                         //int j;
                alts = &intf->altsetting[i];
                ep = t_find_endpoint(alts, t_struct->bEndpointAddress);

                if(ep == NULL)
                        continue;

                psize = t_endpoint_max_bpi(t_struct->u_dev, ep);
                if(psize >= bandwidth && psize <= best_psize)
                {
                        altsetting = alts->desc.bAlternateSetting;
                        best_psize = psize;
                        best_ep = ep;
                }
        }
```

```
			if(best_ep == NULL)
					return -EIO;

			ret = usb_set_interface(t_struct->u_dev, intfnum, altsetting);
			if(ret < 0)
				return ret;

		ret = t_init_video_isoc(t_struct, best_ep, GFP_KERNEL);
		}
		else
			printk("Invalid Work.₩n");

		if (ret < 0)
			return ret;

			/* Submit the URBs. */
		for (i = 0; i < T_URBS; ++i)
		{
			ret = usb_submit_urb(t_struct->urb[i], GFP_KERNEL);
			if (ret < 0)
			{
			//uvc_uninit_video(stream, 1);
				return ret;
			}
		}

		return 0;
}

static void t_video_stats_stop(thirdy_t *t_struct)
{
	ktime_get_ts(&t_struct->stats.stream.stop_ts);
}

static void t_free_urb_buffers(thirdy_t *t_struct)
{
	unsigned int i;

		for (i = 0; i < T_URBS; ++i)
		{
			if (t_struct->urb_buffer[i])
			{
```

```
#ifndef CONFIG_DMA_NONCOHERENT
                        usb_free_coherent(t_struct->u_dev,   t_struct->urb_size,   t_struct->urb_buffer[i],
t_struct->urb_dma[i]);
#else
                        kfree(t_struct->urb_buffer[i]);
#endif
                        t_struct->urb_buffer[i] = NULL;
                }
        }
        t_struct->urb_size = 0;
}

static void t_uninit_video(thirdy_t *t_struct, int free_buffers)
{
        struct urb *urb;
        unsigned int i;

        t_video_stats_stop(t_struct);

        for (i = 0; i < T_URBS; ++i)
        {
                urb = t_struct->urb[i];
                if (urb == NULL)
                                continue;
                usb_kill_urb(urb);
                usb_free_urb(urb);
                t_struct->urb[i] = NULL;
        }

        if (free_buffers)
                t_free_urb_buffers(t_struct);
}

static void t_video_clock_cleanup(thirdy_t *t_struct)
{
        kfree(t_struct->clock.samples);
        t_struct->clock.samples = NULL;
}

int t_video_enable(thirdy_t *t_struct, int enable)
{
        int ret;
```

```
	if (!enable)
	{
		t_uninit_video(t_struct, 1);
		if (t_struct->intf->num_altsetting > 1)
			usb_set_interface(t_struct->u_dev, t_struct->intfnum, 0);

		t_video_clock_cleanup(t_struct);
		return 0;
	}

	ret = t_video_clock_init(t_struct);
	if (ret < 0)
		return ret;

	ret = t_commit_video(t_struct);
	if (ret < 0)
	{
		printk("t_commit_video error₩n");
		goto error;
	}

	ret = t_init_video(t_struct, GFP_KERNEL);
	if(ret < 0)
	{
		printk("t_init_video error₩n");
		goto error;
	}

	return 0;
error:
	return ret;
}

static void t_queue_return_buffers(thirdy_t *t_struct, enum t_buffer_state state)
{
	enum vb2_buffer_state vb2_state = state == T_BUF_STATE_ERROR ? VB2_BUF_STATE_ERROR : VB2_BUF_STATE_QUEUED;

	while (!list_empty(&t_struct->irqqueue))
	{
		struct t_buffer *buf = list_first_entry(&t_struct->irqqueue, struct t_buffer, queue);
		list_del(&buf->queue);
		buf->state = state;
```

```
			vb2_buffer_done(&buf->buf, vb2_state);
		}
}

static int t_start_streaming(struct vb2_queue *vq, unsigned int count)
{
		struct vb2_queue *v_q = vb2_get_drv_priv(vq);
		thirdy_t *t_struct = container_of(v_q, thirdy_t,t_viq);
		int ret;
		unsigned long flags;

		ret = t_video_enable(t_struct, 1);
		if(ret == 0)
			return 0;

		spin_lock_irqsave(&t_struct->irqlock, flags);
		t_queue_return_buffers(t_struct, T_BUF_STATE_QUEUED);
		spin_unlock_irqrestore(&t_struct->irqlock, flags);

		return ret;
}

static void t_stop_streaming(struct vb2_queue *vq)
{
		struct vb2_queue *v_q = vb2_get_drv_priv(vq);
		thirdy_t *t_struct = container_of(v_q, thirdy_t, t_viq);
		unsigned long flags;

		t_video_enable(t_struct, 0);

		spin_lock_irqsave(&t_struct->irqlock, flags);
		t_queue_return_buffers(t_struct, T_BUF_STATE_ERROR);
		spin_unlock_irqrestore(&t_struct->irqlock, flags);
}

static struct usb_driver thirdy_driver = {
		.name		= "thirdy_driver",
		.probe		= thirdy_probe,
		.disconnect	= thirdy_disconnect,
		.id_table	= thirdy_ids,
};

static struct vb2_ops thirdy_qops = {
```

```
        .queue_setup              = t_queue_setup,
        .buf_queue                = t_buffer_queue,
        .buf_prepare              = t_buffer_prepare,
        .buf_finish               = t_buffer_finish,
        .wait_prepare             = vb2_ops_wait_prepare,
        .wait_finish              = vb2_ops_wait_finish,
        .start_streaming          = t_start_streaming,
        .stop_streaming           = t_stop_streaming,
};

static int __init usb_thirdy_init(void)
{
        int result;
        result = usb_register(&thirdy_driver);
        printk("thirdy_driver init gogo₩n");

        return 0;
}

static void __exit usb_thirdy_exit(void)
{
        usb_deregister(&thirdy_driver);
        printk("thirdy_driver exit₩n");

        return ;
}

static int  thirdy_q_ctrl(thirdy_t *t_struct, __u8 query, __u8 unit, __u8 intfnum, __u8 cs, void *data, __u16 size,
int timeout)
{
        __u8 type = USB_TYPE_CLASS | USB_RECIP_INTERFACE;
        unsigned int t_pipe;

        t_pipe = (query & 0x80) ? usb_rcvctrlpipe(t_struct->u_dev, 0) : usb_sndctrlpipe(t_struct->u_dev, 0);
        type |= (query & 0x80) ? USB_DIR_IN : USB_DIR_OUT;

        return usb_control_msg(t_struct->u_dev, t_pipe, query, type, cs << 8, unit << 8 | intfnum, data, size,
timeout);
}

static int thirdy_open(struct file *file)
{
    thirdy_t *t_struct;
```

```
	int ret;

	if((t_struct = kzalloc(sizeof *t_struct, GFP_KERNEL)) == NULL)
		return -ENOMEM;

	t_struct = video_drvdata(file);

	if(t_struct == NULL)
		return -1;

	if(t_struct->users == 0)
	{
		if(t_struct->ctrl_urb == NULL)
			ret = 0;
		else
		{
			ret = usb_submit_urb(t_struct->ctrl_urb, GFP_KERNEL);
		}

		if(ret < 0)
			return ret;
	}

	t_struct->users++;

	v4l2_fh_init(&t_struct->t_fh, t_struct->v_dev);
	v4l2_fh_add(&t_struct->t_fh);
	file->private_data = t_struct;

	printk("Thirdy_Driver : Open Function₩n");
	return 0;
}

static int thirdy_close(struct file *file)
{
	printk("Thirdy_Driver : Close Function₩n");
	return 0;
}

static int thirdy_v4l2_mmap(struct file *file, struct vm_area_struct *vma)
{
	thirdy_t *t_struct = video_drvdata(file);
	printk("Thirdy_Driver : thirdy_v4l2_mmap OK₩n");
```

```
	return vb2_mmap(&t_struct->t_viq, vma);
}

const struct v4l2_file_operations thirdy_fops = {
	.owner		= THIS_MODULE,
	.open		= thirdy_open,
	.release	= thirdy_close,
	.unlocked_ioctl	= video_ioctl2,
	.mmap		= thirdy_v4l2_mmap,
};

int thirdy_querycap(struct file *file, void *fh, struct v4l2_capability *cap)
{
	thirdy_t *t_struct;

	if((t_struct = kzalloc(sizeof *t_struct, GFP_KERNEL)) == NULL)
		return -ENOMEM;

	t_struct = fh;

	strlcpy(cap->driver, "Thirdy_Driver", sizeof(cap->driver));
	strlcpy(cap->card, t_struct->v_dev->name, sizeof(cap->card));
	usb_make_path(t_struct->u_dev, cap->bus_info, sizeof(cap->bus_info));
	cap->version = LINUX_VERSION_CODE;
	cap->capabilities = V4L2_CAP_DEVICE_CAPS | V4L2_CAP_STREAMING | t_struct->caps;
	cap->device_caps = V4L2_CAP_VIDEO_CAPTURE | V4L2_CAP_STREAMING;

	file->private_data = &t_struct->t_fh;
	printk("Thirdy_Driver : vidioc_querycap OK₩n");
	return 0;
}

int thirdy_s_fmt_cap(struct file *file, void *fh, struct v4l2_format *f)
{
	int ret;
	int w, h;
	int maxw = 640, maxh = 480;
	__u32 fcc;
	thirdy_t *t_struct;
	__u8 *set_data;
	__u16 set_size = 26;
```

```
	if((t_struct = kzalloc(sizeof *t_struct, GFP_KERNEL)) == NULL)
		return -ENOMEM;
	t_struct = video_drvdata(file);

	w = f->fmt.pix.width;
	h = f->fmt.pix.height;
	if(t_fmts[0].fcc == f->fmt.pix.pixelformat)
		t_struct->t_fmt = &t_fmts[0];
	else
	{
		printk("Thirdy_Driver : thirdy_s_fmt_cap error 1₩n");
		return -1;
	}
	fcc = t_struct->t_fmt->fcc;
	v4l_bound_align_image(&w, 48, maxw, 1, &h, 32, maxh, 1, 0);
	//---------
	set_data = kzalloc(set_size, GFP_KERNEL);
	if(set_data == NULL)
			return -ENOMEM;

	t_struct->t_data->bFormatIndex = 1;
	t_struct->t_data->bFrameIndex = 4;
	t_struct->t_data->dwFrameInterval = 83263;

	*(__le16 *)&set_data[0] = cpu_to_le16(t_struct->t_data->bmHint);
	set_data[2] = t_struct->t_data->bFormatIndex;
	set_data[3] = t_struct->t_data->bFrameIndex;
	*(__le32 *)&set_data[4] = cpu_to_le32(t_struct->t_data->dwFrameInterval);
	*(__le16 *)&set_data[8] = cpu_to_le16(t_struct->t_data->wKeyFrameRate);
	*(__le16 *)&set_data[10] = cpu_to_le16(t_struct->t_data->wPFrameRate);
	*(__le16 *)&set_data[12] = cpu_to_le16(t_struct->t_data->wCompQuality);
	*(__le16 *)&set_data[14] = cpu_to_le16(t_struct->t_data->wCompWindowSize);
	*(__le16 *)&set_data[16] = cpu_to_le16(t_struct->t_data->wDelay);
	put_unaligned_le32(t_struct->t_data->dwMaxVideoFrameSize, &set_data[18]);
	put_unaligned_le32(t_struct->t_data->dwMaxPayloadTransferSize, &set_data[22]);

	ret = thirdy_q_ctrl(t_struct, 0x01, 0, t_struct->intfnum, 0x01, set_data, set_size, 5000);

	if(ret < 0)
		return ret;
//--------
	f->fmt.pix.width = w;
	f->fmt.pix.height = h;
```

```
	f->fmt.pix.pixelformat = fcc;
	f->fmt.pix.bytesperline = (w * t_struct->t_fmt->depth + 7) >> 3;
	f->fmt.pix.sizeimage = t_struct->t_data->dwMaxVideoFrameSize;
	f->fmt.pix.colorspace = V4L2_COLORSPACE_SMPTE170M;
	f->fmt.pix.field = V4L2_FIELD_INTERLACED;

	t_struct->t_width = w;
	t_struct->t_height = h;
	t_struct->v4l2_fmt = f;

	ret = vb2_is_busy(&t_struct->t_viq);
	if(ret < 0)
	{
		printk("s_fmt vb2_is_busy error : %d₩n", ret);
		return ret;
	}

	video_set_drvdata(video_devdata(file), t_struct);
	file->private_data = &t_struct->t_fh;

	printk("Thirdy_Driver : thirdy_s_fmt_cap OK₩n");
	return 0;
}

int thirdy_reqbufs(struct file *file, void *fh, struct v4l2_requestbuffers *b)
{
	thirdy_t *t_struct;
	int ret, chk = 0;

	if((t_struct = kzalloc(sizeof *t_struct, GFP_KERNEL)) == NULL)
		return -ENOMEM;

	t_struct = video_drvdata(file);
	mutex_lock(&t_struct->q_mutex);
	chk = vb2_reqbufs(&t_struct->t_viq, b);
	mutex_unlock(&t_struct->q_mutex);
	ret = chk ? chk : b->count;

	if(ret < 0)
		return ret;

	printk("Thirdy_Driver : thirdy_reqbufs OK₩n");
	return 0;
```

```
}

static int thirdy_g_fmt_cap(struct file *file, void *fh, struct v4l2_format *f)
{
	printk("Thirdy_Driver - thirdy_g_fmt_cap OK₩n");
	return 0;
}

static int thirdy_querybuf(struct file *file, void *fh, struct v4l2_buffer *buf)
{
	thirdy_t *t_struct;
	int ret;

	if((t_struct = kzalloc(sizeof *t_struct, GFP_KERNEL)) == NULL)
		return -ENOMEM;

	t_struct = video_drvdata(file);

	mutex_lock(&t_struct->q_mutex);
	ret = vb2_querybuf(&t_struct->t_viq, buf);
	mutex_unlock(&t_struct->q_mutex);
	printk("Thirdy_Driver - thirdy_querybuf OK ₩n");

	return ret;
}

static int thirdy_qbuf(struct file *file, void *fh, struct v4l2_buffer *buf)
{
	thirdy_t *t_struct;
	int ret;

	if((t_struct = kzalloc(sizeof *t_struct, GFP_KERNEL)) == NULL)
		return -ENOMEM;

	t_struct = video_drvdata(file);
	mutex_lock(&t_struct->q_mutex);
	ret = vb2_qbuf(&t_struct->t_viq, buf);
	mutex_unlock(&t_struct->q_mutex);

	printk("Thirdy_Driver - thirdy_qbuf OK₩n");
	return ret;
}
```

```
static int thirdy_streamon(struct file *file, void *fh, enum v4l2_buf_type type)
{
	int ret;
	thirdy_t *t_struct;

	if((t_struct = kzalloc(sizeof *t_struct, GFP_KERNEL)) == NULL)
		return -ENOMEM;

	t_struct = video_drvdata(file);
	mutex_lock(&t_struct->q_mutex);
	ret = vb2_streamon(&t_struct->t_viq, type);
	mutex_unlock(&t_struct->q_mutex);

	printk("Thirdy_Driver - thirdy_streamon OK₩n");
	return ret;
}

static int thirdy_streamoff(struct file *file, void *fh, enum v4l2_buf_type type)
{
	int ret;
	thirdy_t *t_struct;

	if((t_struct = kzalloc(sizeof *t_struct, GFP_KERNEL)) == NULL)
		return -ENOMEM;

	t_struct = video_drvdata(file);
	mutex_lock(&t_struct->q_mutex);
	ret = vb2_streamoff(&t_struct->t_viq, type);
	mutex_unlock(&t_struct->q_mutex);

	printk("Thirdy_Driver - thirdy_streamoff OK₩n");
	return 0;
}

static int thirdy_dqbuf(struct file *file, void *fh, struct v4l2_buffer *buf)
{
	thirdy_t *t_struct;
	int ret;

	if((t_struct = kzalloc(sizeof *t_struct, GFP_KERNEL)) == NULL)
		return -ENOMEM;

	t_struct = video_drvdata(file);
```

```
	mutex_lock(&t_struct->q_mutex);
	ret = vb2_dqbuf(&t_struct->t_viq, buf, file->f_flags & O_NONBLOCK);
	mutex_unlock(&t_struct->q_mutex);

	printk("Thirdy_Driver - thirdy_dqbuf OK₩n");
	return ret;
}

static long thirdy_default(struct file *file, void *fh, bool valid_prio, unsigned int cmd, void *arg)
{
	thirdy_t *t_struct;
	if((t_struct = kzalloc(sizeof *t_struct, GFP_KERNEL)) == NULL)
		return -ENOMEM;

	t_struct = video_drvdata(file);

	printk("Thirdy_Driver - default : %d₩n", cmd);
	printk("Thirdy_Driver - default : %x₩n", cmd);
	return -1;
}

static const struct v4l2_ioctl_ops thirdy_ioctl_ops = {
    .vidioc_querycap         = thirdy_querycap,
    .vidioc_s_fmt_vid_cap    = thirdy_s_fmt_cap,
    .vidioc_g_fmt_vid_cap    = thirdy_g_fmt_cap,
	.vidioc_reqbufs          = thirdy_reqbufs,
	.vidioc_querybuf    = thirdy_querybuf,
	.vidioc_qbuf             = thirdy_qbuf,
	.vidioc_dqbuf            = thirdy_dqbuf,
	.vidioc_streamon    = thirdy_streamon,
	.vidioc_streamoff   = thirdy_streamoff,
	.vidioc_default          = thirdy_default,
};

module_init(usb_thirdy_init);
module_exit(usb_thirdy_exit);
MODULE_LICENSE("GPL");
```