http://hybridego.net
번역한사람 : 전호철

by Vladimir Davydov (Mar. 13, 2009)

Foreword –V4L2(Video for Linux2)인터페이스에 대해 기술하고 있는 이 글은 인터페이스를 사용하여 디바이스 드라이버를 개발하는 첫번째 스텝부터 설명하고 있습니다. 이 글은 Linux 2.6.28 에 기반하고 있고 다른 커널버전에는 적용되지 않을 수 있습니다.

이 글은 Promwad의 엔지니어인 Vladimir Davydov가 기여했습니다. Promwad 는 Minsk, Belarus에 위치한 임베디드 서비스 개발 회사입니다. V4L2 인터페이스를 사용해서 몇몇 개발 프로젝트를 한 적이 있습니다.

Practical Application of V4L2 Drivers: Intro to V4L2 (Part 1)

by Vladimir Davydov

Video For Linux(V4L)은 TV튜너나 USB웹 카메라 같은 다양한 비디오 캡처 디바이스에 접근을 가능하게 해주는 하나의 API입니다. 이 인터페이스의 첫번째 버전은 Alan Cox에 의해서 Linux2.2에서 실행되었었습니다. 1999년에 두번쨰 버전 V4L2인터페이스 개발이 시작되었고 V4L에서 발견된 버그를 고치고 더 많은 디바이스를 지원하게 되었습니다.

General overview of Video4Linux2 interface (V4L2)

V4L2 드라이버들은 일반적인 커널 모듈들처럼 실행됩니다. 이것은 Administrator에 의해 로드 될 수 있거나 디바이스 파일을 열 때 자동으로 로드 될 수도 있습니다. 드라이버들은 기본 "videodev" 커널 모듈을 사용합니다. 이 커널 모듈은 비디오 디바이스로 작업하기 위한 커널 디바이스와 일치합니다.

V4L:2 API는 표준 Unix character Device들 처럼 접근하도록 지원해줍니다.
V4L2 는 다음의 인터페이스들을 지원합니다.

- Video capture interface (/dev/video)
- Video overlay interface (/dev/video)
- Video output interface (/dev/video)
- Video output interface (/dev/video)
- Codec interface (/dev/video)
- Effect devices interface (/dev/video)
- Raw vertical blancking interval (VBI) interface (/dev/vbi)
- Sliced VBI data interface (/dev/vbi)
- Teletext interface (/dev/vtx)
- Radio interface (/dev/radio)

- Radio data system (RDS) interface (/dev/radio)

이 글에서는 /dev/video character device가 지원하는 비디오 캡쳐 인터페이스의 사용법을 설명할 것입니다.

다음은 사용자 어플리케이션과 V4L2 드라이버간의 상호연동을 차트로 표현한 것입니다.

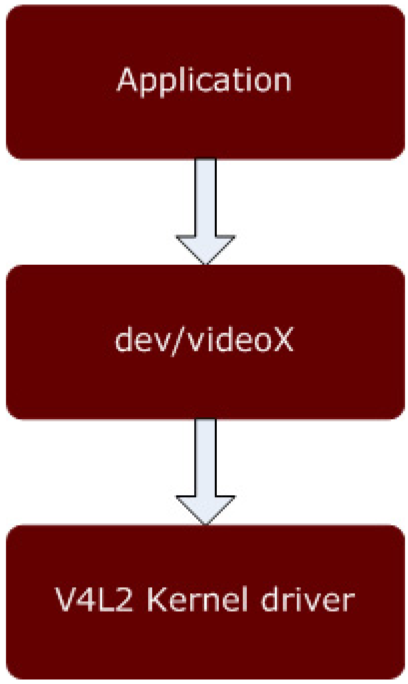

Chart 1: Interaction between a user application and the V4L2 driver

Linux V4L2 userspace programming

이 글에서는 사용자 프로세스의 항목들은 공식문서에 매우 잘 나와있기 때문에 다루지 않습니다.
공식문서 : http://www.linuxtv.org/downloads/video4linux/API/V4L2_API/
다음은 V4L2인터페이스로 레코딩하고 재생하기 위한 오픈소스로된 자유소프트웨어 프로그램들입니다.

- MEncoder -- free tool for coding, decoding and filtering video, contained in MPlayer package

- MPlayer -- free and open source media player

- FFmpeg -- robust solution for recording, converting, and streaming audio and video in different formats. FFmped contains a set of ready tools and libraries, which can be used for coding/decoding audio and video, and also for multiplexing and demultiplexing in different media containers.

- xawtv -- program for watching TV programs.

- tvtime -- a high-quality TV application for use with video capture cards

Linux V4L2 driver programming

위에서 언급한 바와 같이 V4L2 디바이스 드라이버는 "videodev" 기본 모듈을 사용합니다. 이 모듈은 V4L2인터페이스로 작업하기 위한 함수 세트를 지원합니다. 하지만 불행히도 지금은 이 함수들에 대한 Full description이 없습니다. "Video for Linux Two -- Driver Writer's Guide," 1999년 12월 24일에 작성된 것만 있습니다. 이 문서는 V4L2의 내부 인터페이스가 변했기 때문에 구식이 되어버렸습니다.

다행히도 Linux development의 오픈 모듈이 ready driver들의 great number를 지원합니다. 이것은 examining the V4L2 Interface 예제 같은 곳이나 비디오 캡쳐 디바이스를 위한 새로운 드라이버를 개발할 때 사용될 수 있습니다. V4L2 드라이버들을 위한 소스코드는 커널의 '/drivers/media/video" 디렉토리의 소스코드 트리에서 활용가능 합니다. 여기서 가장 흥미로운 것은 VIVI driver(drivers/media/video/vivi.c) 입니다.

VIVI – Virtual Video driver

VIVI 는 실제 Video device를 에뮬레이션하는 V4L2 드라이버 모듈입니다. 이 드라이버는 Video Technology Magazine 팀에 의해 개발되었고 Linux 2.6.17 커널에 추가되어 릴리즈되었습니다.

VIVI 개발의 주 목적은 V4L2 드라이버의 Working sample을 디자인 하고 stub driver는 새 드라이버의 개발을 간단히 하는 것 이었습니다. 또한 VIVI는 V4L2인터페이스를 사용하는 유저어플리케이션을 디버깅 하는 곳에 사용될 수 있습니다. VIVI code 는 V4L2 드라이버를 위한 리눅스 커널 인터페이스의 이상적 사용을 보여줍니다. 그리고 새 드라이버를 작성할 때 개발자들을 위해 예제를 추천해줍니다.

Loading the VIVI modul:

```
# modprobe vivi
# ls -l /dev/video*
lrwxrwxrwx 1 root root      6 Feb 14 00:07 /dev/video -> video0
crw-rw---- 1 root video 81, 0 Feb 14 00:07 /dev/video0
#
```

V4L2 Linux kernel interface

커널은 하나의 구조체 세트와 편리한 V4L2 드라이버 개발을 위한 도움 함수를 제공합니다.
이 함수를 사용하기 위해 다음 헤더파일들을 include 해야 합니다.

- media/v4l2-dev.h> 기본 구조체와 함수의 선언을 포함하고 있다. V4L2 driver 와 작업을 등록하는 것을 요구한다. 이 함수들의 실행의 대부분은 /drivers/media/video/v4l2-common.c 파일에 있다.
- ＜media/videobuf-core.h> 비디오 데이터 버퍼링, 메모리 관리, PCI DMA 실행을 위한 함수를 포함하고 있다.이 함수들의 실행은 drivers/media/video/videobuf-core.c 파일에 위치해 있다.
- ＜media/v4l2-ioctl.h> 디바이스 파일(/dev/video)의 ioctrl 시스템 콜을 처리하기 위한 구조체와 함수를 담고있습니다.

Initializing and registering a V4L2 driver

<u>몇가지 중요한 데이터 구조체들</u>

V4L2 디바이스를 등록하려면 Video_device, file_operations, v4l2_ioctl_ops 세개의 메인 구조체를 채우는 것이 필수적입니다.
각각의 세 구조체를 초기화 할때를 눈여겨 보시기 바랍니다. 다음은 VIVI driver에서 사용한 예제입니다.

```
static const struct file_operations vivi_fops = {
.owner               = THIS_MODULE,
.open                = vivi_open,
.release             = vivi_close,
.read                = vivi_read,
.poll                = vivi_poll,
.ioctl               = video_ioctl2, /* V4L2 ioctl handler */
.compat_ioctl        = v4l_compat_ioctl32,
.mmap                = vivi_mmap,
.llseek              = no_llseek,
};

static const struct v4l2_ioctl_ops vivi_ioctl_ops = {
.vidioc_querycap        = vidioc_querycap,
.vidioc_enum_fmt_vid_cap    = vidioc_enum_fmt_vid_cap,
.vidioc_g_fmt_vid_cap       = vidioc_g_fmt_vid_cap,
.vidioc_try_fmt_vid_cap     = vidioc_try_fmt_vid_cap,
.vidioc_s_fmt_vid_cap       = vidioc_s_fmt_vid_cap,
.vidioc_reqbufs             = vidioc_reqbufs,
.vidioc_querybuf            = vidioc_querybuf,
.vidioc_qbuf                = vidioc_qbuf,
.vidioc_dqbuf               = vidioc_dqbuf,
```

```
.vidioc_s_std              = vidioc_s_std,
.vidioc_enum_input       = vidioc_enum_input,
.vidioc_g_input           = vidioc_g_input,
.vidioc_s_input           = vidioc_s_input,
.vidioc_queryctrl        = vidioc_queryctrl,
.vidioc_g_ctrl             = vidioc_g_ctrl,
.vidioc_s_ctrl             = vidioc_s_ctrl,
.vidioc_streamon          = vidioc_streamon,
.vidioc_streamoff        = vidioc_streamoff,
#ifdef CONFIG_VIDEO_V4L1_COMPAT
.vidiocgmbuf               = vidiocgmbuf,
#endif
};

static struct video_device vivi_template = {
.name    = "vivi",
.fops    = &vivi_fops,
.ioctl_ops = &vivi_ioctl_ops,
.minor    = -1,
.release = video_device_release,
.tvnorms = V4L2_STD_525_60,
.current_norm = V4L2_STD_NTSC_M,
};
```

.이 구조체들의 몇가지 필드들의 기능을 좀더 자세히 알아볼까요?

Video_device 구조체는 <media/v4l2-dev.h>파일안에 선언되어 있습니다. 이것은 일반적인 비디오 정보에 관한 내용을 담고 있습니다. fops와 ioctrl_ops 필드들은 유사한 구조체들 file_operations 와 v4l2_ioctl_ops 들을 가리킵니다.

release 필드는 video device가 시스템에서 제거될 때 호출되는 함수를 가리킵니다. 이 경우에 이 필드는 기본video_device_release 함수를 가리킵니다. 이것은 video_device 구조체에 할당되었던 메모리를 해제해줍니다.

tvnorms 필드는 지원하는 비디오 표준들의 리스트를 정의합니다. 지원하는 표준들의 전체 리스트는 <linux/videodev2.h>헤더파일에 선언되어 있습니다. 우리경우는 V4L2_STD_525_60 비트마스크 하나를 나타냅니다.

```
#define V4L2_STD_525_60   (V4L2_STD_PAL_M   |₩
                           V4L2_STD_PAL_60 |₩
                           V4L2_STD_NTSC    |₩
                           V4L2_STD_NTSC_443)
```

current_norm 필드는 이름과 비슷한 역할을 하고 기본으로 사용될 비디오 표준을 정의합니다.

File_operations 구조체는 <linux/fs.h>파일에 정의되어 있습니다. 이것은 중요한 커널 구조체입니다. 함수를 가치키는 포인터를 갖고 있고 각각 일치하는 연산을 채우거나 연산이 지원 안되는 경우에는 NULL로 지정합니다. 이 구조체의 필드들은 Linux Device Drivers, 3rd Edition(LDD3)에 매우 잘 기술되어 있습니다. 여기에서 다운로드 가능합니다. download here

VIVI 드라이버는 /dev/video 디바이스 파일을 위한 메인 시스템콜 핸들러들을 정의합니다. open, release, read, poll, mmap, ioctl 같은것들을요. 이 기능들의 세부사항은 이 글 후반부에 다루겠습니다.

video_ioctl2 표준 함수는 ioctl 콜을 위한 핸들러 같이 사용한다. 이 함수는 ioctl콜의 초기 처리를 실행한다. command의 코드를 분석하고 v4l2_ioctl_ops 구조체에 있는 상응하는 핸들러가 사용하는 펑션 포인터를 호출한다. 여러분은 여러분이 갖고 있는 ioctl 처리를 위한 함수를 열거할수 있습니다. 하지만 이 경우에서 여러분은 독립적인 커맨드 분석을 위한 코드를 실행해야 합니다. V4L2의 개발자들은 드라이버 코드에서 있을법한 에러를 줄이기 위해서 video_iotrl2를 사용하길 권장합니다.

v4l2_ioctl_ops 구조체는 ioctl 시스템콜에 의해 변환된 V4L2 커맨드들의 핸들러를 가리키는 포인터를 저장합니다. 이 구조체는 <media/v4l2-ioctl.h> 파일에 정의도어 있습니다.

Registering a video device

미리 언급했듯이 커널은 video device를 기술하기 위해 video_device 구조체를 사용합니다. 이 구조체는 다음의 함수를 사용함으로써 정적으로도 동적으로도 생성할 수 있습니다.

```
struct video_device *video_device_alloc(void);
```

여러분의 코드에 video_device_alloc를 사용하려면 <media/v4l2-common.h>를 include 해야합니다.
구조체가 점유했던 메모리를 release 하려면 다음을 호출합니다.

```
void video_device_release(struct video_device *vfd);
```

비디오 디바이스는 다음 함수에 의해 등록됩니다.

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

위의 예제에 있는 Vfd 파라메터는 등록하길 원하는 비디오 디바이스 구조체를 가리키는 포인터입니다.

type parameter 는 등록된 디바이스의 타입을 정의합니다. 다음 값들은 디바이스 타입을 선언합니다.

VFL_TYPE_GRABBER -- video capturing device
VFL_TYPE_VTX -- teletext device
VFL_TYPE_VBI -- VBI device
VFL_TYPE_RADIO -- radio card

nr 파라메터는 등록된 디바이스의 숫자를 설정합니다. 0은 /dev/video1 뭐 이런식입니다.
여러분이 첫번째 가능한 숫자를 할당하고 싶다면 이 파라메터를 1로 하시면 됩니다.

시스템에서 비디오 디바이스를 제거하기 위해 여러분은 다음의 함수를 호출하셔야 합니다.

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

Registering video device in VIVI

대부분의 리눅스 디바이스 드라이버처럼 모듈 초기화와 해제를 위한 위임받은 함수가 있습니다. 이 프로세스를 이해하기 위해 LDD3 문서를 읽어보기를 권합니다.

VIVI드라이버는 많은 비디오 디바이스 드라이버를 등록할 수 있게 해줍니다. 현재 등록된 장치의 최대 개수는 videodev에 의해 32개의 디바이스로 제한됩니다.

VIVI 드라이버를 초기화하기 위한 함수를 수정한 코드를 보시겠습니다. 비디오 디바이스를 등록하는것을 보여줍니다.

```
int ret;
struct video_device *vfd;


vfd = video_device_alloc();
```

```
if (!vfd) {
	/*Allocation failed*/
}

*vfd = vivi_template;

ret = video_register_device(vfd, VFL_TYPE_GRABBER, -1);
if (ret < 0) {
	/*Registration failed*/
	video_device_release(vfd);
}
```

Conclusion

V4L2 API는 비디오 캡쳐하는 디바이스를 위한 현대적이고 강력한 인터페이스를 제공합니다. V4L2 API는 훌륭한 문서가 존재하고 매우 많은 수의 준비된 드라이버와 어플리케이션을 활용 가능하며 오픈소스는 개발시간을 많이 단축시켜 줄 것입니다. 다음 글에서는 구조와 버퍼링을 위한 V4L 인터페이스의 helper function을 검토해보겠습니다.

Useful links

V4L2 home

V4L2 API specification

FFmpeg tools and libraries

Xawtv TV application

Tvtime TV application

Video for Linux Two - Driver Writer's Guide

MPlayer and MEncoder home

The VIVI driver; a great starting point for V4L2 driver writers

[Linux Device Drivers, Third Edition]

About the author -- Vladimir Davydov works as an engineer at [Promwad Innovation Company]. He has more than seven years experience in network programming and Linux kernel development for multimedia applications. In 2002, he graduated from Belarusian State University of Informatics and Radioelectronics (Minsk, Belarus) with a Bachelor of Science in Computer Science, and main specialization in compute machines, complexes, systems, and networks.

by Vladimir Davydov (Apr. 15, 2009)

Foreword – 이 글은 V4L2(Video for Linux 2)인터페이스에 대한 두개의 설명 글 중 두번째 글입니다. 디바이스 드라이버를 개발하기 위한 첫번째 단계로 이 인터페이스를 사용합니다. 이것은 Linux 2.6.28에 기반합니다. 그리고 다른 커널버전에서는 적용되지 않을수 있습니다.

[the first article on V4L2] 처럼 이 글은 [Promwad]의 엔지니어인 Vladimir Davydov 에 의해 씌여졌습니다. [Promwad]는 Minsk, Belarus에 있는 임베디드 개발 서비스 회사입니다. Promwad는 몇몇 개발 프로젝트에서 V4L2 인터페이스를 사용한 경험이 있습니다. Davydov reports. Enjoy . . . !

Structures and helper functions in the V4L interface for buffering: Intro to V4L2 (Part 2)
by Vladimir Davydov

Buffering Management Functions
V4L 드라이버들의 중요한 태스크중에 하나는 캡쳐버퍼를 관리하는 것입니다. 이 부분은 실행하기 매우 복잡한 부분입니다. V4L2 인터페이스처럼 몇가지 방법으로 캡쳐된 비디오를 복구할 가능성을 제공해 줍니다.

다행히도 커널은 구조체 세트와 다음의 기능을 하는 도움말 함수를 제공해줍니다.

- 캡쳐보드를 위한 메모리 할당
- 캡쳐 버퍼와 큐 초기화
- mmap, read, poll 시스템 콜 처리
- ioctl 시스템콜의 메인 커멘트 처리

- DMA 사용

이런 기능들은 확실한 커널 모듈들을 가능하게 합니다

- *videobuf-core* ? 일반적인 버퍼링 함수들을 제공합니다.
- *video-dma-sg* ? Scatter-Gather DMA buffer 들을 관리하기 위한 저수준 함수들을 포함
- *video-dma-config* ?물리적으로 접근하는 버퍼들을 위한 helper 함수들을 포함.
- *videobuf-vmalloc* ? DMA 지원 없는 버퍼들을 위한 helper 함수 포함.

vieobuf-dma-sg, videobuf-dma-config, videobuf-vmalloc 모듈들은 다른타입의 버퍼들을 위한 같은 기능들을 수행합니다. 버퍼 타입들에 대해서는 다음에 자세히 설명하겠습니다.

Important Buffering Structures

Videobuf_buffer 구조는 <media/videobuf-core.h> 헤더파일에 선언되어 있고 캡쳐 버퍼들을 나타내는데 쓰입니다.

이 구조체의 필드들은 코멘트에 매우 잘 설명되어 있습니다. 보통 드라이버들은 하나의 버퍼와 구제적인 정보에 연결을 필요로 합니다. 이 경우에 드라이버는 버퍼를 위한 새로운 구조체를 선언하고 첫번째 필드처럼 videobuf_buffer를 포함합니다.

VIVI 드라이버에서 정확히 어떻게 해야하는지 보여줍니다.

```
/* buffer for one video frame */
struct vivi_buffer {
            /* common v4l buffer stuff -- must be first */
            struct videobuf_buffer vb;

            struct vivi_fmt            *fmt;
};
```

여기서 vivi_fmt 구조체배열을 가리키는 포인터는 기본 videobuf_buffer 구조체 필드들에 추가됩니다. 그리고 VIVI 드라이버가 지원하는 비디오 포맷 리스트를 포함합니다.

다음으로 중요한 구조체는 캡쳐버퍼의 큐를 구성하기 위해 디자인 된 videobuf_queue 입니다. 이 구조체의 선언은 <media/videobuf-core.h>파일에 되어 있습니다. 여러분들은 코멘트나 videobuf-core.c 코드를 공부함으로써 구조체 필드들의 선언에 대해 알 수 있을 것입니다.

Types of Capture Buffers

버퍼링을 사용하기 전에 여러분은 여러분이 필요한 캡쳐버퍼들의 타입을 정의해야만 합니다. 이

것은 대부분 개발된 디바이스 드라이버에 의존적입니다.

V4L에서 언급한것에는 세가지 타입의 캡쳐 버퍼들을 제공한다고 되어있습니다.

1. 버퍼들은 Scatter-Gather DMA (SG DMA)에서 비디오 데이터의 변환을 지원하는 디바이스에서 사용했습니다. 그 버퍼는 물리적인 메모리에서 선형적으로 위치해있지 않고 조각조각 쪼개어뎌있습니다. 이 조각들의 크기는 메모리 페이지의 크기(PAGE_SIZE)에 상응합니다. 이 버퍼의 타입은 cx88, bttv, saa7134 같은 디바이스 드라이버에서 사용합니다.

   SG DMA buffers 를 관리하기 위한 Low-level 함수들은 /drivers/media/video/videobuf-dma-sg.c 파일에 있습니다.

   이 버퍼 타입을 사용하려면 여러분은 videobuf-dma-sg 모듈을 로드해야 합니다.

2. 버퍼들은 물리적 메모리의 연속된 공간을 나타내고 디바이스에서 지원하는 Scatter-Gather 를 제외한 DMA 를 사용합니다.

   이 버퍼타입을 관리하는 Low-level 함수들은 driver/media/video/videobuf-dma-config.c 파일에 있습니다.

3. 버퍼는 Built-in DMA 지원이 없습니다. 이 버퍼타입은 보통 디바이스에서 사용됩니다. DMA 같에 데이터 전송을 지원하거나 DMA 처리를 위한 코드에서 사용할 수 있습니다.

   SG DMA 에서 이 버퍼는 물리메모리의 영역에서 선형적으로 배열되어 있지 않고 페이지 사이즈의 크기로 조각나있습니다.

   One of the drivers using this buffer type is VIVI. 이런 버퍼들로 작업하기 위한 함수들의 소스코드는 /drivers/media/video/videobuf-vmalloc.c 파일에 있습니다.

Initializing Capture Buffers
버퍼들을 초기화 하는 함수를 호출하기 전에 video_queue_ops 구초채를 채워야 합니다. 이 것은 <media/videobuf-core.h> 헤더파일에 선언되어 있습니다. 이 구초제는 버퍼로 작업하기 위한 함수 포인터를 포함하고 있고 반드시 드라이버에서 실행되어야 합니다.

```
int (*buf_setup)(struct videobuf_queue *q,
                        unsigned int *count, unsigned int *size);
```

Buf_setup 메소드는 버퍼들의 숫자(count)와 각 버퍼의 사이즈(size)를 세팅합니다.

```
int (*buf_prepare)(struct videobuf_queue *q,
```

```
                                    struct videobuf_buffer *vb,
                                    enum v4l2_field field);
```

buf_prepare 함수는 버퍼를 준비합니다. VIVI에서 이 함수는 일반적인 버퍼 파라메터들을 기술합니다. : size, width, height, field type

```
void (*buf_queue)(struct videobuf_queue *q,
                                    struct videobuf_buffer *vb);
```

buf_queue 함수는 캡쳐버퍼가 큐로 들어갈 때 호출 됩니다. 이 함수 실행의 전형적 변형은 VIVI 드라이버에서 주워진다.

```
void (*buf_release)(struct videobuf_queue *q,
                                    struct videobuf_buffer *vb);
```

buf_release 메소드는 버퍼를 릴리즈 하기위해 호출된다.

다음은 VIVI에서 videobuf_queue_ops 구조체의 초기화 예제입니다.

```
static struct videobuf_queue_ops vivi_video_qops = {
        .buf_setup      = buffer_setup,
        .buf_prepare    = buffer_prepare,
        .buf_queue      = buffer_queue,
        .buf_release    = buffer_release,
};
```

Videobuf_queue_ops 구조체를 채운 다음에 여러분은 캡쳐버퍼 타입에 따라 초기화 함수를 호출해야 합니다.

1. SG DMA 캡쳐버퍼를 위한 초기화 함수

```
void videobuf_queue_sg_init(struct videobuf_queue* q,
                            struct videobuf_queue_ops *ops,
                            struct device *dev,
                            spinlock_t *irqlock,
                            enum v4l2_buf_type type,
                            enum v4l2_field field,
                            unsigned int msize,
                            void *priv);
```

2. 물리적으로 연속된 캡쳐버퍼들을 위한 DMA

```
void videobuf_queue_dma_contig_init(struct videobuf_queue *q,
```

```
                                        struct videobuf_queue_ops *ops,
                                        struct device *dev,
                                        spinlock_t *irqlock,
                                        enum v4l2_buf_type type,
                                        enum v4l2_field field,
                                        unsigned int msize,
                                        void *priv);
```

3. Built-in DMA 지원이 없는 캡쳐버퍼를 위한 세팅

```
void videobuf_queue_vmalloc_init(struct videobuf_queue* q,
                                 struct videobuf_queue_ops *ops,
                                 void *dev,
                                 spinlock_t *irqlock,
                                 enum v4l2_buf_type type,
                                 enum v4l2_field field,
                                 unsigned int msize,
                                 void *priv);
```

이 함수들을 사용하기 위해 여러분은 해당 헤더파일을 연결해야 합니다.

```
<media/videobuf-dma-sg.h>, <media/videobuf-dma-contig.h>,
<media/videobuf-vmalloc.h>
```

이 함수들에서 파라메터들은 같습니다. 첫번째 파라메처 (q)는 캡쳐버퍼 큐를 가리키는 포인터입니다. ops 파라메터는 사전에 채워진 videobuf_queue_ops 구조체를 가리킵니다. Irqlock 파라메터는 싱크를 위해 필요한 spinlock을 가리킵니다.

파라메터 파입은 캡쳐버퍼 타입을 정의합니다. 가능한 타입의 리스트는 <media/videodev2.h>헤더에 들어있습니다. VIVI에서 이 파라메터는 V4L2_BUF_TYPE_VIDEO_CAPTURE로 설정합니다. 이것은 버퍼들을 비디오 캡쳐하는데 쓰겠다는 뜻입니다.

필드 파라메터는 비디오 이미지 필드들의 순서를 설정합니다. Enum v4l2_field의 정의는 <linux/videodev2.h>파일에 있습니다. 다음 값들은 허용됩니다.

- *V4L2_FIELD_ANY* ? 드라이버는 비디오 이미지 필드의 순서를 독립적으로 선택할 수 있습니다.
- *V4L2_FIELD_NONE* ? 디바이스는 필드들로 자른 이미지를 지원하지 않는다.
- *V4L2_FIELD_TOP* ? top field only

- *V4L2_FIELD_BOTTOM* ? bottom field only
- *V4L2_FIELD_INTERLACED* ? both fields are in interlaced condition
- *V4L2_FIELD_SEQ_TB* ?both fields are sequentially joined in the buffer; first the top field and then the bottom
- *V4L2_FIELD_SEQ_BT* ?the same as for *V4L2_FIELD_SEQ_TB*, but the fields order is the opposite
- *V4L2_FIELD_ALTERNATE* ?Sequential alternating of fields (top/bottom). The fields are saved to separate buffers
- *V4L2_FIELD_INTERLACED_TB* ?both fields are interlaced, the top field is the first in the image and the top field is passed first
- *V4L2_FIELD_INTERLACED_BT* ?both fields are interlaced, the top field is the first in the image, and the bottom field is passed first

필수적인 값들은 보통 디바이스 역량에 따라 선택됩니다.
msize 파라메터는 캡쳐버퍼 구조체의 사이즈를 기입합니다. VIVI에서 이것은 sizeof(struct vivi_buffer)입니다.
마지막 파라메터인 priv는 videobuf_queue 구조체에서 priv_data의 값을 기입합니다.

Releasing Capture Buffers

버퍼를 규칙대로 해제하는 함수는 메소드 코드 안에서 호출됩니다. Videobuf_queue_ops에 buf_release 포인터에 의해서 가능합니다.

다음 함수는 SG DMA에서 사용됩니다.

```
int videobuf_dma_free(struct videobuf_dmabuf *dma);
```

DMA를 지원하는 물리적으로 연속적인 버퍼에서는

```
void videobuf_dma_contig_free(struct videobuf_queue *q,
struct videobuf_buffer *buf);
```

DMA 지원 없는 캡쳐 버퍼에서는

```
void videobuf_vmalloc_free (struct videobuf_buffer *buf);
```

Processing the *mmap* System Call

V4L에서 mmap 시스템 콜은 버퍼에 메모리 할당을 매핑한 유저 어플리케이션에서 유저 프로세스 영역으로 캡쳐버퍼를 엑세스하는 메소드처럼 쓰인다.

버퍼링 함수들을 사용하는 것을 통해 mmap 시스템 콜 처리는 calling 함수로 분해된다.

```
int videobuf_mmap_mapper(struct videobuf_queue *q,
                                              struct vm_area_struct *vma);
```

다음은 VIVI 드라이버에서 mmap콜 처리가 어떻게 처리되는지 보여주는 예제입니다.

```
static int vivi_mmap(struct file *file, struct vm_area_struct *vma)
{
        struct vivi_fh   *fh = file->private_data;
        struct vivi_dev *dev = fh->dev;
        int ret;

        dprintk(dev, 1, "mmap called, vma=0x%08lx₩n", (unsigned long)vma);

        ret = videobuf_mmap_mapper(&fh->vb_vidq, vma);

        dprintk(dev, 1, "vma start=0x%08lx, size=%ld, ret=%d₩n",
                (unsigned long)vma->vm_start,
                (unsigned long)vma->vm_end-(unsigned long)vma->vm_start,
                ret);

        return ret;
}
```

Processing the *read* System Call

mmap 에서 버퍼링 함수들은 드라이버 개발자들의 코드를 줄여줍니다. 이것은 다음 함수를 호출하는데 충분합니다.

```
ssize_t videobuf_read_stream(struct videobuf_queue *q,
                                                  char __user *data, size_t count, loff_t *ppos,
                                                  int vbihack, int nonblocking);
```

다음은 VIVI 드라이버에서 read 콜이 처리되는 예제입니다.

```
static ssize_t
vivi_read(struct file *file, char __user *data, size_t count, loff_t *ppos)
{
        struct vivi_fh *fh = file->private_data;

        if (fh->type == V4L2_BUF_TYPE_VIDEO_CAPTURE) {
```

```
                return videobuf_read_stream(&fh->vb_vidq, data, count, ppos, 0,
                                        file->f_flags & O_NONBLOCK);
        }
        return 0;
}
```

Processing the *poll* System Call

poll을 호출하면 드라이버가 캡쳐된 데이터를 받을 때 까지 어플리케이션은 실행을 멈출 수 있습니다.

poll 호출을 어떻게 사용하느냐를 더 자세히 할고싶다면 V4L2의 공식 문서를 찾아보시면 됩니다.

helper buffering 함수들을 사용함으로써 poll콜의 처리는 별다른 어려움 없이 다음 함수를 사용함으로서 끝납니다.

```
unsigned int videobuf_poll_stream(struct file *file,
                                  struct videobuf_queue *q,
                                  poll_table *wait);
```

다음은 VIVI 드라이버에서 poll 콜이 처리되는 방법에 대한 예제입니다.

```
static unsigned int
vivi_poll(struct file *file, struct poll_table_struct *wait)
{
        struct vivi_fh          *fh = file->private_data;
        struct vivi_dev          *dev = fh->dev;
        struct videobuf_queue *q = &fh->vb_vidq;

        dprintk(dev, 1, "%s₩n", __func__);

        if (V4L2_BUF_TYPE_VIDEO_CAPTURE != fh->type)
                return POLLERR;

        return videobuf_poll_stream(file, q, wait);
}
```

Processing Commands of *ioctl* System Call

ioctl 버퍼링 커맨드들의 처리를 위해 다음 함수들을 사용할 수 있다.

VIDIOC_REQBUFS:
int videobuf_reqbufs(struct videobuf_queue *q, struct v4l2_requestbuffers *req);

VIDIOC_QUERYBUF:
int videobuf_querybuf(struct videobuf_queue *q, struct v4l2_buffer *b);

VIDIOC_QBUF:
int videobuf_qbuf(struct videobuf_queue *q,   struct v4l2_buffer *b);

VIDIOC_DQBUF:
int videobuf_dqbuf(struct videobuf_queue *q, struct v4l2_buffer *b, int nonblocking);

VIDIOC_STREAMON:
int videobuf_streamon(struct videobuf_queue *q);

VIDIOC_STREAMOFF:
int videobuf_streamoff(struct videobuf_queue *q);

이 함수들의 예제는 VIVI 드라이버 코드에서 찾을 수 있습니다.

Accessing Capture Buffer Video Data

규칙 처럼 V4L 드라이버는 비디오 디바이스로부터 받은 비디오 데이터에 직접 접근할 필요가 없었습니다. 드라이버는 디바이스로부터 그냥 데이터를 받습니다. 그리고 사용자 프로세스로 전송합니다. 때때로 드라이버 코드 안에서 데이터로 직접 접근의 허가가 필요할 때가 있을 수 있습니다. 예를 들면 VIVI드라이버는 독립적으로 비디오 데이터를 생성하고 실제 비디오 디바이스 같은 것은 비디오 데이터를 읽습니다.

타입에 따라 비디오에 접근하는 것은 다음 함수들로 할 수 있습니다.

1. SG DMA:

   struct videobuf_dmabuf *videobuf_to_dma (struct videobuf_buffer *buf);

2. DMA for physically contiguous capture buffers:

   dma_addr_t videobuf_to_dma_contig(struct videobuf_buffer *buf);

3. For capture buffers without built-in DMA support:

```
void *videobuf_to_vmalloc (struct videobuf_buffer *buf);
```

이 섹션은 구조체들과 함수들의 간단한 설명을 하고 있습니다.

```
#include <linux/fs.h>
```

이 헤더파일은 파일시스템과 디바이스 드라이버들을 개발하는데 가장 중요한 함수들과 구조체들을 선언합니다.

```
struct file_operations;

struct file;

struct inode;
```

디바이스 드라이버들의 majority 에 의해 사용되는 세가지 중요한 데이터 구조체가 있습니다. file_operations 구조체는 character device 를 위한 시스템 콜을 처리하기 위한 메소드들을 갖고있습니다. 파일 구조체는 시스템의 파일을 연 것을 기술합니다. inode 구조체는 디스크의 한 파일을 나타냅니다.

```
#include <media/v4l2-dev.h>
```

이 구조체는 helper 함수들과 모든 V4L 드라이버들에 의해 요구되는 구조체들을 포함합니다.

```
struct video_device;
```

이것은 한 비디오 디바이스를 기술하는 메인 구조체 입니다.

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

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

이 함수들은 시스템에 한 비디오 디바이스를 등록하고 해제합니다.

```
struct video_device * video_device_alloc(void);

void video_device_release(struct video_device *vfd);
```

이 helper 함수들은 video_deivice 구조체에 의해 메모리 점유를 할당하고 해제합니다.

```
#include <media/v4l2-ioctl.h>
```

이 구조는 ioctl 명령들의 처리를 위해 필요한 구조체들과 함수들의 선언을 포함합니다.

```
struct v4l2_ioctl_ops;
```

이 구조체는 V4L2 ioctl 커맨드들을 가리키는 포인터를 포함합니다.

```
int video_ioctl2(struct inode *inode, struct file *file,
                 unsigned int cmd, unsigned long arg);
```

이 메소드는 ioctl 시스템콜을 처리합니다. V2L2 커맨드 코드를 분석하고 v4l2_ioctl_ops 구조체로부터 상응하는 핸들러를 호출합니다.

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

V4L2 인터페이스의 메인 헤더파일 입니다. 이 파일은 모든 어플리케이션과 드라이버들에 의해 사용되는 모든 데이터 타입들의 메인 선언들을 포함합니다.

```
#include <media/videobuf-core.h>
```

이 구조는 캡쳐 버퍼들을 관리하는 메인 함수들을 포함합니다.

```
struct videobuf_buffer;
```

이 구조체는 캡쳐 버퍼의 기본 속성을 기술합니다.

```
struct videobuf_queue_ops;
```

이 구조체는 캡쳐버퍼 처리 메소드 들을 가리키는 포인터를 포함합니다.

```
struct videobuf_queue;
```

이구조체는 버퍼큐를 구성하기 위해 디자인 되었습니다.

```
int videobuf_reqbufs(struct videobuf_queue *q, struct v4l2_requestbuffers *req);

int videobuf_querybuf(struct videobuf_queue *q, struct v4l2_buffer *b);

int videobuf_qbuf(struct videobuf_queue *q,   struct v4l2_buffer *b);

int videobuf_dqbuf(struct videobuf_queue *q, struct v4l2_buffer *b, int nonblocking);

int videobuf_streamon(struct videobuf_queue *q);

int videobuf_streamoff(struct videobuf_queue *q);
```

이 함수들은 퍼버링과 연관된 ioctl 시스템콜의 메인 명령들을 처리합니다.

```
int videobuf_mmap_mapper(struct videobuf_queue *q,
                         struct vm_area_struct *vma);
```

이 함수는 mmap 시스템콜을 처리합니다.

```
ssize_t videobuf_read_stream(struct videobuf_queue *q,
                             char __user *data, size_t count, loff_t *ppos,
                             int vbihack, int nonblocking);
```

이 함수는 read 시스템콜을 처리합니다.

```
unsigned int videobuf_poll_stream(struct file *file,
                                  struct videobuf_queue *q,
                                  poll_table *wait);
```

이 함수는 poll 시스템콜을 처리합니다.

```
#include <media/videobuf-vmalloc.h>
```

이 헤더파일은 built-in DMA 지원 없는 캡쳐 버퍼로 작업하기 위한 helper 함수들을 담고 있습니다.

```
void videobuf_queue_vmalloc_init(struct videobuf_queue* q,
                        struct videobuf_queue_ops *ops,
                        void *dev,
                        spinlock_t *irqlock,
                        enum v4l2_buf_type type,
                        enum v4l2_field field,
                        unsigned int msize,
                        void *priv);
```

이 함수는 built-in DMA 지원 없는 캡쳐버퍼들을 위한 버퍼링 시스템을 초기화 합니다.

```
void *videobuf_to_vmalloc (struct videobuf_buffer *buf);
```

이 함수는 캡쳐버퍼의 비디오 데이터로 접근하는 것을 허가합니다.

```
void videobuf_vmalloc_free (struct videobuf_buffer *buf);
```

이 함수는 캡쳐 버퍼를 해제합니다.

```
#include <media/videobuf-dma-contig.h>
```

이 헤더 파일은 DMA 를 지원하는 캡쳐 버퍼로 작업하기 위한 helper 함수들과 구조체를 포함합니다.

```
void videobuf_queue_dma_contig_init(struct videobuf_queue *q,
                                    struct videobuf_queue_ops *ops,
                                    struct device *dev,
                                    spinlock_t *irqlock,
                                    enum v4l2_buf_type type,
                                    enum v4l2_field field,
```

```
unsigned int msize,
void *priv);
```

이 함수들은 built-in DMA 지원을 하는 캡쳐버퍼를 위한 버퍼링 시스템을 초기화 합니다.

```
#include <media/videobuf-dma-sg.h>
```

이 title 파일은 Scatter-Gather DMA support 캡쳐버퍼들로 작업하기 위한 helper 함수들과 구조체를 포함합니다.

```
void videobuf_queue_sg_init(struct videobuf_queue* q,
                            struct videobuf_queue_ops *ops,
                            struct device *dev,
                            spinlock_t *irqlock,
                            enum v4l2_buf_type type,
                            enum v4l2_field field,
                            unsigned int msize,
                            void *priv);
```

이 함수는 built-in SG DMA support 캡쳐버퍼들을 위한 버퍼링 시스템 초기화를 합니다.

Conclusion

V4L2 API는 비디오 캡쳐 디바이스를 쓰는데 현대적 인터페이스와 편리함을 제공합니다. V4L2 API에 대한 훌륭한 문서와 준비된 훌륭한 드라이버들이 많으며 오픈소스로 되어있는 어플리케이션도 많아 개발 시간을 많이 단축시켜 줄것입니다.

Useful links

V4L2 home

V4L2 API specification

FFmpeg tools and libraries

Xawtv TV application

Tvtime TV application

Video for Linux Two - Driver Writer's Guide

MPlayer and MEncoder home

The VIVI driver; a great starting point for V4L2 driver writers

Linux Device Drivers, Third Edition