UVC compliant webcamera interfacing using V4L2 API

Pranav Kant Gaur

December 3, 2013

Abstract

This document describes the preliminary study and work of the author for developing interface for accessing UVC-compliant webcamera. It gives introduction to the V4L2 API structure which is a standard for accessing UVC-compliant video output devices. However the API is not limited to video devices but supports audio, RF devices as well. It describes standard pipeline used by author to initiate, continue and terminate video feed output from webcamera. It then discusses the conversion between color spaces which becomes important when certain devices support different frame rates, resolution for different output formats. It then becomes neccessary to interconvert between formats to utilize the output capabilities of devices as was the case in author's work. It should be pointed out however, that this document is by no means an exhaustive introduction to V4L2 API. For exhaustive treatment please refer[**?**]

1 Objective of work

1. To develop camera capture interface for capturing images with UVC compatible cameras.
2. To allow interactive exposure control.

2 Motivation

This work was motivated by the requirement to interactively control exposure for cheap UVC compatible camera(specifically, webcameras) through indigineously developed 3D scanner application[**?**]. This control was available for digital cameras(using Gphoto2 library[**?**]) but for webcameras such high-level library support is unavailable except using OpenCV. Using OpenCV however allows only setting V4L2 *user controls* but not its *camera controls*. V4L2 API[**?**] provides customizable interface for accessing and controlling webcameras. Therefore, it was decided to develop a saparate module for capturing video feed from webcam and interactively controlling its exposure.

3 V4L2 API

V4L2 API is a video capture application programming interface for Linux. It abstracts the device driver details from video access applications. V4L2 provides interface for:

1. Opening and closing device
2. Querying its capabilites
3. Getting/setting value of any particular control
4. Exchange data formats
5. I/O methods used for mapping video data from device to application

Using this interface application interacts with device driver through set of *ioctl()* commands which driver is supposed to implement and register. For UVC-compliant webcams *uvcvideo* works like V4L2 driver. Following subsections will elaborate on standard stages of video acquisition using V4L2 API.

3.1 Opening and closing device

Since devices in Linux are abstracted as *files*, *device opening* and *device closing* commands are simply file *open* and *close* commands only. Listing **??** illustrate this:

Listing 1: Opening a device

```
int fd;

    fd = open("/dev/video0", O_RDWR);
    if (fd == -1)
    {
        perror("Opening video device");
        return 1;
    }

    ....
    Do stuff();
    ....

    close(fd);
```

3.2 Querying device information and capabilities

V4L2 API defines specific *ioctl()* commands for querying device information and capabilities. Listing **??** illustrates the use of **VIDIOC_QUERYCAP** ioctl command to query device controls.

Listing 2: Querying device information

```
struct v4l2_capability caps = {};
    if (-1 == ioctl(fd, VIDIOC_QUERYCAP, &caps))
    {
        perror("Querying Capabilities");
        return 1;
    }
```

Listing **??** shows how we can access device capabilities returned by **VIDIOC_QUERYCAP** command.

Listing 3: Accessing device capability information

```
    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);
```

3.3 Getting and setting value of any particular control

V4L2 API classifies device controls into classes depending upon their generality. Specifically, ***user class controls*** and ***extended controls***.

3.3.1 User class controls

These controls are generally availble to the user thorugh GUIs such as *Brightness,Saturation,Contrast* etc. In essesnce, this class belongs to the basic controls which are mostly supported by UVC-compliant devices. Current value of thses controls can be acquired using **VIDIOC_G_CTRL** command. For setting user class controls, **VIDIOC_S_CTRL** command is used. Listing **??** shows an example. Specifically, it shows first accessing the current value of contrast and then setting incrementing it by 1 and then again querying thr current value to check of value was set.

Listing 4: Getting and setting user class controls

```
control.id = V4L2_CID_CONTRAST;

if (0 == ioctl (fd, VIDIOC_G_CTRL, &control)) {
printf("Contrast level before modification:%d", control.value);
}

.....
some_code();
.....

control.value += 1; // incrementing contrast value by 1

if (-1 == ioctl (fd, VIDIOC_S_CTRL, &control)
            && errno != ERANGE) {
                perror ("VIDIOC_S_CTRL");
                exit (EXIT_FAILURE);
        }

if (0 == ioctl (fd, VIDIOC_G_CTRL, &control)) {
printf("Contrast level after modification:%d", control.value);
}
```

3.3.2 Extended controls

V4L2 API allows for accessing controls which are not assumed to be present in most video devices e.g., Pan,tilt capability etc. Extended controls are also further classified into: *Codec control class*, *Camera control class*, *FM Transmitter control class*, *Flash control class*, *JPEG control class*, *Image source control class*, *Image process control class*, *Digital video control class*, *FM receiver control class*. Since, we are interested only in camera control, therefore only camera control class will be discussed here. V4L2 API provides **VIDIOC_G_EXT_CTRLS** and **VIDIOC_S_EXT_CTRLS** for getting and setting multiple extended controls respectively. Listing **??** illustrates these controls with an example of setting absolute exposure from *Camera contol class*.

Listing 5: Extended controls

```
struct v4l2_queryctrl qctl;
struct v4l2_ext_controls ctrls= {0};
struct v4l2_ext_control ctrl_auto_exp[1];


qctl.id=V4L2_CID_EXPOSURE_ABSOLUTE;
```

```
    if(-1==ioctl(fd,VIDIOC_QUERYCTRL,&qctl))
    {
        perror("Not able to query exposure");
        exit(1);
    }

.........
some_code();
.........

// Setting absolute exposure for the video device

ctrl_auto_exp[0].id=V4L2_CID_EXPOSURE_AUTO;
ctrl_auto_exp[0].value=V4L2_EXPOSURE_MANUAL;

ctrls.ctrl_class=V4L2_CTRL_CLASS_CAMERA;
ctrls.count=1;
ctrls.controls=ctrl_auto_exp;

if(-1==ioctl(fd,VIDIOC_S_EXT_CTRLS,&ctrls))
perror("Failed to set exposure to manual mode");

ctrls.ctrl_class=V4L2_CTRL_CLASS_CAMERA;
ctrls.count=1;
ctrls.controls=ctrl_exp;

ctrl_exp[0].id=V4L2_CID_EXPOSURE_ABSOLUTE;
ctrl_exp[0].value=exposure;


if(-1==ioctl(fd,VIDIOC_S_EXT_CTRLS,&ctrls))
   perror("Failed to set exposure");
```

In Listing **??** only one control has been shown but in general there can be multiple extended controls all of which will be read or written to *atomically*. Their multiplicity is denoted by the member *count* shown in Listing **??**.

3.4 Data formats

Various devices support various data formats to exchange with applications. However, it is always recommended to enquire the data formats that the hardware can support before starting any data exchange. Therefore, V4L2 API provides **VIDIOC_ENUM_FMT** for enumerating the data formats supported by device, to get current data format **VIDIOC_G_FMT** is defined and to set a data format before data exchange **VIDIOC_S_FMT** is defined. Listing **??** shows how we can enumerate data formats supported by the device.

Listing 6: Data format enumeration and negotiation

```
struct v4l2_fmtdesc fmtdesc = {0};
fmtdesc.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
char fourcc[5] = {0};
char c, e;
printf("  FMT : CE Desc\n--------------------\n");
while (0 == xioctl(fd, VIDIOC_ENUM_FMT, &fmtdesc))
{
    strncpy(fourcc, (char *)&fmtdesc.pixelformat, 4);
    if (fmtdesc.pixelformat == V4L2_PIX_FMT_SGRBG10)
     support_grbg10 = 1;
    c = fmtdesc.flags & 1? 'C' : ' ';
    e = fmtdesc.flags & 2? 'E' : ' ';
    printf("  %s: %c%c %s\n", fourcc, c, e, fmtdesc.description);
    fmtdesc.index++;
}
```

Listing **??** shows how we can set a particular data format(in current case we are setting video data format for a video capture device). This code will set the image resolution and pixel format.

Listing 7: Data format negotiation

```
struct v4l2_format fmt = {0};
fmt.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
fmt.fmt.pix.width = cam_imagewidth;
fmt.fmt.pix.height = cam_imageheight;
fmt.fmt.pix.pixelformat = V4L2_PIX_FMT_YUYV ;
fmt.fmt.pix.field = V4L2_FIELD_NONE;

if (-1 == ioctl(fd, VIDIOC_S_FMT, &fmt))
 {
   perror("Failed 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);
```

3.5 Device I/O methods

V4L2 API defines multiple methods to read from or write to devices. Drivers must support atleast one of them. These controls are *Read/write*, Streaming I/O based on *Memory mapping*, Streaming I/O based on *User pointers*, Streaming I/O based on *DMA buffer importing*, *Asynchronous I/O*. Listing **??** shows an example of negotiating an I/O method with device driver. In this example, memory mapping I/O method is used. Application first requests for buffer using **VIDIOC_REQBUFS** and if this request succeeds then queries the information on allocated buffers using **VIDIOC_QUERYBUF**.

Listing 8: Using memory mapped I/O

```
struct v4l2_requestbuffers req = {0};
req.count = 1;
req.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
req.memory = V4L2_MEMORY_MMAP;

if (-1 == ioctl(fd, VIDIOC_REQBUFS, &req))
 {
  perror("Requesting Buffer");
  return 1;
 }

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

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

4 Conversion from YUYV format to RGB

Device used for this work supports its maximum resolution in YUYV color space but not in the conventional MJPEG format. This required first setting device data format to YUYV and internally converting the output data to RGB format so as to utilize the maximum image resolution supported by device.
Here, we will digress a little to first explore the concept of *Color space*. Then we will illustrate the above mentioned color space conversion with the help of a code example.

4.1 Color models

This section is intended to briefly describe existing color models to represent *human perceptible* colors. Color space is an *abstract* mathematical representation of colors in form of tuples. Each component of these tuple represents a specific dimension of color representation. Such *arbitrary* model becomes a *color space* if it also defines a mapping of these tuple values to an *absolute color space*.
First formal color space formulation was done in 1931 by CIE a.k.a *International commision on illumination*. Mainly there are following color models:

1. CIE XYZ color model
2. CIE RGB color model
3. CMYK color model

4.1.1 Trichromatic color space

Humans percieve colors through three types of *cone* cells which act like *color receptors*. These cells are classified on the basis of range of wavelength to which they are sensitive. These are *Short*, *Medium* and *Long* wavelength receptors. Range of human color perception is defined by the range of frequencies to which these cells are sensitive. Therefore, a 3D color space can be constructed with L,S,M as its three dimensions. It will be able to represent all colors perceptible by human vision system.

4.1.2 CIE XYZ color space

To formally define human color space, XYZ color model was proposed. In this model, Y represents luminance whereas other components are responsible for chromaticity. It can also be understood in terms of (luminance+chromaticity) model, by defining x,y coordinate space. (x,y) 2D space defines the *Chromaticity diagram* and Y defines the *luminance*.

4.1.3 CIE RGB color space

RGB color model represents color space as defined as a linear combination of Red, Blue and Green colors refered to as *Primary colors*. XYZ color model can

be transformed to RGB model and vice versa. Red, green and blue are prefered over other primary color combinations because they cover the largest part of human perceptible color space. Colors are represented as a *cartesian* cube.

4.1.4 HSV color representation

HSV and HSL are two most common cylindrical coordinate representations of RGB color model. HSV color space saparates luminance and chromaticity. *Hue* represents the color. *Saturation* represents the strength or *purity* of color. *Value* represents the luminance or brightness(hence it is also called *HSB* color model). Colors are represented along cyliderical coordinate system as opposed to RGB model's cartesian representation. In cylinder, central axis represents neutral(or gray) color with bottom being *black* and top as *white*.

4.1.5 HSL color representation

Here *Lightness* represents .

4.1.6 CMYK color space

5 Conclusions