LINUX MEDIA INFRASTRUCTURE API

Table of Contents

List of Figures

List of Tables

List of Examples

Introduction

This document covers the Linux Kernel to Userspace API's used by video and radio straming devices, including video cameras, analog and digital TV receiver cards, AM/FM receiver cards, streaming capture devices.

It is divided into three parts.

The first part covers radio, capture, cameras and analog TV devices.

The second part covers the API used for digital TV and Internet reception via one of the several digital tv standards. While it is called as DVB API, in fact it covers several different video standards including DVB-T, DVB-S, DVB-C and ATSC. The API is currently being updated to doccument support also for DVB-S2, ISDB-T and ISDB-S.

The third part covers other API's used by all media infrastructure devices

The third part covers other API's used by all media infrastructure devices

For additional information and for the latest development code, see: http://linuxtv.org.

For discussing improvements, reporting troubles, sending new drivers, etc, please mail to: Linux Media Mailing List (LMML)..

Part I. Video for Linux Two API Specification

Revision 2.6.32

Michael H Schimek

<mschimek@gmx.at>

Bill Dirks

Original author of the V4L2 API and documentation.

Hans Verkuil

Designed and documented the VIDIOC_LOG_STATUS ioctl, the extended control ioctls and major parts of the sliced VBI API.

<hverkuil@xs4all.nl>

Martin Rubli

Designed and documented the VIDIOC_ENUM_FRAMESIZES and VIDIOC_ENUM_FRAMEINTERVALS ioctls.

Andy Walls

Documented the fielded V4L2_MPEG_STREAM_VBI_FMT_IVTV MPEG stream embedded, sliced VBI data format in this specification.

<awalls@radix.net>

Mauro Carvalho Chehab

Documented libv4l, designed and added v4l2grab example, Remote Controller chapter.

<mchehab@redhat.com>

Revision History		
Revision 2.6.32	2009-08-31	mcc
Now, revisions will match the kernel version where the V4L2 API changes will be used by the Linux Kernel. Also added Remote Controller chapter.		
Revision 0.29	2009-08-26	ev
Added documentation for string controls and for FM Transmitter controls.		
Revision 0.28	2009-08-26	gl

Revision 0.28	2009-08-26	gl
Added V4L2_CID_BAND_STOP_FILTER documentation.		
Revision 0.27	2009-08-15	mcc
Added libv4l and Remote Controller documentation; added v4l2grab and keytable application examples.		
Revision 0.26	2009-07-23	hv
Finalized the RDS capture API. Added modulator and RDS encoder capabilities. Added support for string controls.		
Revision 0.25	2009-01-18	hv
Added pixel formats VYUY, NV16 and NV61, and changed the debug ioctls VIDIOC_DBG_G/S_REGISTER and VIDIOC_DBG_G_CHIP_IDENT. Added camera controls V4L2_CID_ZOOM_ABSOLUTE, V4L2_CID_ZOOM_RELATIVE, V4L2_CID_ZOOM_CONTINUOUS and V4L2_CID_PRIVACY.		
Revision 0.24	2008-03-04	mhs
Added pixel formats Y16 and SBGGR16, new controls and a camera controls class. Removed VIDIOC_G/S_MPEGCOMP.		
Revision 0.23	2007-08-30	mhs
Fixed a typo in VIDIOC_DBG_G/S_REGISTER. Clarified the byte order of packed pixel formats.		
Revision 0.22	2007-08-29	mhs
Added the Video Output Overlay interface, new MPEG controls, V4L2_FIELD_INTERLACED_TB and V4L2_FIELD_INTERLACED_BT, VIDIOC_DBG_G/S_REGISTER, VIDIOC_(TRY_)ENCODER_CMD, VIDIOC_G_CHIP_IDENT, VIDIOC_G_ENC_INDEX, new pixel formats. Clarifications in the cropping chapter, about RGB pixel formats, the mmap(), poll(), select(), read() and write() functions. Typographical fixes.		
Revision 0.21	2006-12-19	mhs
Fixed a link in the VIDIOC_G_EXT_CTRLS section.		
Revision 0.20	2006-11-24	mhs
Clarified the purpose of the audioset field in struct v4l2_input and v4l2_output.		
Revision 0.19	2006-10-19	mhs
Documented V4L2_PIX_FMT_RGB444.		
Revision 0.18	2006-10-18	mhs
Added the description of extended controls by Hans Verkuil. Linked V4L2_PIX_FMT_MPEG to V4L2_CID_MPEG_STREAM_TYPE.		
Revision 0.17	2006-10-12	mhs
Corrected V4L2_PIX_FMT_HM12 description.		
Revision 0.16	2006-10-08	mhs
VIDIOC_ENUM_FRAMESIZES and VIDIOC_ENUM_FRAMEINTERVALS are now part of the API.		
Revision 0.15	2006-09-23	mhs
Cleaned up the bibliography, added BT.653 and BT.1119. capture.c/start_capturing() for user pointer I/O did not initialize the buffer index. Documented the V4L MPEG and MJPEG VID_TYPEs and V4L2_PIX_FMT_SBGGR8. Updated the list of reserved pixel formats. See the history chapter for API changes.		
Revision 0.14	2006-09-14	mr
Added VIDIOC_ENUM_FRAMESIZES and VIDIOC_ENUM_FRAMEINTERVALS proposal for frame format enumeration of digital devices.		
Revision 0.13	2006-04-07	mhs
Corrected the description of struct v4l2_window clips. New V4L2_STD_ and V4L2_TUNER_MODE_LANG1_LANG2 defines.		
Revision 0.12	2006-02-03	mhs
Corrected the description of struct v4l2_captureparm and v4l2_outputparm.		
Revision 0.11	2006-01-27	mhs
Improved the description of struct v4l2_tuner.		
Revision 0.10	2006-01-10	mhs
VIDIOC_G_INPUT and VIDIOC_S_PARM clarifications.		
Revision 0.9	2005-11-27	mhs
Improved the 525 line numbering diagram. Hans Verkuil and I rewrote the sliced VBI section. He also contributed a VIDIOC_LOG_STATUS page. Fixed VIDIOC_S_STD call in the video standard selection example. Various updates.		
Revision 0.8	2004-10-04	mhs

Revision 0.8	2004-10-04	mhs
Somehow a piece of junk slipped into the capture example, removed.		
Revision 0.7	2004-09-19	mhs
Fixed video standard selection, control enumeration, downscaling and aspect example. Added read and user pointer i/o to video capture example.		
Revision 0.6	2004-08-01	mhs
v4l2_buffer changes, added video capture example, various corrections.		
Revision 0.5	2003-11-05	mhs
Pixel format erratum.		
Revision 0.4	2003-09-17	mhs
Corrected source and Makefile to generate a PDF. SGML fixes. Added latest API changes. Closed gaps in the history chapter.		
Revision 0.3	2003-02-05	mhs
Another draft, more corrections.		
Revision 0.2	2003-01-15	mhs
Second draft, with corrections pointed out by Gerd Knorr.		
Revision 0.1	2002-12-01	mhs
First draft, based on documentation by Bill Dirks and discussions on the V4L mailing list.		

Table of Contents

Chapter 1. Common API Elements

Table of Contents

Programming a V4L2 device consists of these steps:

- Opening the device
- Changing device properties, selecting a video and audio input, video standard, picture brightness a. o.
- Negotiating a data format
- Negotiating an input/output method
- The actual input/output loop
- Closing the device

In practice most steps are optional and can be executed out of order. It depends on the V4L2 device type, you can read about the details in Chapter 4, *Interfaces*. In this chapter we will discuss the basic concepts applicable to all devices.

Opening and Closing Devices

Device Naming

V4L2 drivers are implemented as kernel modules, loaded manually by the system administrator or automatically when a device is first opened. The driver modules plug into the "videodev" kernel module. It provides helper functions and a common application interface specified in this document.

Each driver thus loaded registers one or more device nodes with major number 81 and a minor number between 0 and 255. Assigning minor numbers to V4L2 devices is entirely up to the system administrator, this is primarily intended to solve conflicts between devices.[1] The module options to select minor numbers are named after the device special file with a "_nr" suffix. For example "video_nr" for `/dev/video` video capture devices. The number is an offset to the base minor number associated with the device type. [2] When the driver supports multiple devices of the same type more than one minor number can be assigned, separated by commas:

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

In `/etc/modules.conf` this may be written as:

```
alias char-major-81-0 mydriver
alias char-major-81-1 mydriver
alias char-major-81-64 mydriver
options mydriver video_nr=0,1 radio_nr=0,1
```

When an application attempts to open a device special file with major number 81 and minor number 0, 1, or 64, load "mydriver" (and the "videodev" module it depends upon).

Register the first two video capture devices with minor number 0 and 1 (base number is 0), the first two radio device with minor number 64 and 65 (base 64).

When no minor number is given as module option the driver supplies a default. Chapter 4, *Interfaces* recommends the base minor numbers to be used for the various device types. Obviously minor numbers must be unique. When the number is already in use the *offending device* will not be registered.

By convention system administrators create various character device special files with these major and minor numbers in the `/dev` directory. The names recomended for the different V4L2 device types are listed in Chapter 4, *Interfaces*.

The creation of character special files (with mknod) is a privileged operation and devices cannot be opened by major and minor number. That means applications cannot *reliable* scan for loaded or installed drivers. The user must enter a device name, or the application can try the conventional device names.

Under the device filesystem (devfs) the minor number options are ignored. V4L2 drivers (or by proxy the "videodev" module) automatically create the required device files in the `/dev/v4l` directory using the conventional device names above.

Related Devices

Devices can support several related functions. For example video capturing, video overlay and VBI capturing are related because these functions share, amongst other, the same video input and tuner frequency. V4L and earlier versions of V4L2 used the same device name and minor number for video capturing and overlay, but different ones for VBI. Experience showed this approach has several problems[3], and to make things worse the V4L videodev module used to prohibit multiple opens of a device.

As a remedy the present version of the V4L2 API relaxed the concept of device types with specific names and minor numbers. For compatibility with old applications drivers must still register different minor numbers to assign a default function to the device. But if related functions are supported by the driver they must be available under all registered minor numbers. The desired function can be selected after opening the device as described in Chapter 4, *Interfaces*.

Imagine a driver supporting video capturing, video overlay, raw VBI capturing, and FM radio reception. It registers three devices with minor number 0, 64 and 224 (this numbering scheme is inherited from the V4L API). Regardless if `/dev/video` (81, 0) or `/dev/vbi` (81, 224) is opened the application can select any one of the video capturing, overlay or VBI capturing functions. Without programming (e. g. reading from the device with dd or cat) `/dev/video` captures video images, while `/dev/vbi` captures raw VBI data. `/dev/radio` (81, 64) is invariable a radio device, unrelated to the video functions. Being unrelated does not imply the devices can be used at the same time, however. The `open()` function may very well return an EBUSY error code.

Besides video input or output the hardware may also support audio sampling or playback. If so, these functions are implemented as OSS or ALSA PCM devices and eventually OSS or ALSA audio mixer. The V4L2 API makes no provisions yet to find these related devices. If you have an idea please write to the linux-media mailing list: http://www.linuxtv.org/lists.php.

Multiple Opens

In general, V4L2 devices can be opened more than once. When this is supported by the driver, users can for example start a "panel" application to change controls like brightness or audio volume, while another application captures video and audio. In other words, panel applications are comparable to an OSS or ALSA audio mixer application. When a device supports multiple functions like capturing and overlay *simultaneously*, multiple opens allow concurrent use of the device by forked processes or specialized applications.

Multiple opens are optional, although drivers should permit at least concurrent accesses without data exchange, i. e. panel applications. This implies `open()` can return an EBUSY error code when the device is already in use, as well as `ioctl()` functions initiating data exchange (namely the `VIDIOC_S_FMT` ioctl), and the `read()` and `write()` functions.

Mere opening a V4L2 device does not grant exclusive access.[4] Initiating data exchange however assigns the right to read or write the requested type of data, and to change related properties, to this file descriptor. Applications can request additional access privileges using the priority mechanism described in the section called “Application Priority”.

Shared Data Streams

V4L2 drivers should not support multiple applications reading or writing the same data stream on a device by copying buffers, time multiplexing or similar means. This is better handled by a proxy application in user space. When the driver supports stream sharing anyway it must be implemented transparently. The V4L2 API does not specify how conflicts are solved.

Functions

To open and close V4L2 devices applications use the `open()` and `close()` function, respectively. Devices are programmed using the `ioctl()` function as explained in the following sections.

Querying Capabilities

Because V4L2 covers a wide variety of devices not all aspects of the API are equally applicable to all types of devices. Furthermore devices of the same type have different capabilities and this specification permits the omission of a few complicated and less important parts of the API.

The `VIDIOC_QUERYCAP` ioctl is available to check if the kernel device is compatible with this specification, and to query the functions and I/O methods supported by the device. Other features can be queried by calling the respective ioctl, for example `VIDIOC_ENUMINPUT` to learn about the number, types and names of video connectors on the device. Although abstraction is a major objective of this API, the ioctl also allows driver specific applications to reliable identify the driver.

All V4L2 drivers must support `VIDIOC_QUERYCAP`. Applications should always call this ioctl after opening the device.

Application Priority

When multiple applications share a device it may be desirable to assign them different priorities. Contrary to the traditional "rm -rf /" school of thought a video recording application could for example block other applications from changing video controls or switching the current TV channel. Another objective is to permit low priority applications working in background, which can be preempted by user controlled applications and automatically regain control of the device at a later time.

Since these features cannot be implemented entirely in user space V4L2 defines the `VIDIOC_G_PRIORITY` and `VIDIOC_S_PRIORITY` ioctls to request and query the access priority associate with a file descriptor. Opening a device assigns a medium priority, compatible with earlier versions of V4L2 and drivers not supporting these ioctls. Applications requiring a different priority will usually call `VIDIOC_S_PRIORITY` after verifying the device with the `VIDIOC_QUERYCAP` ioctl.

Ioctls changing driver properties, such as `VIDIOC_S_INPUT`, return an EBUSY error code after another application obtained higher priority. An event mechanism to notify applications about asynchronous property changes has been proposed but not added yet.

Video Inputs and Outputs

Video inputs and outputs are physical connectors of a device. These can be for example RF connectors (antenna/cable), CVBS a.k.a. Composite Video, S-Video or RGB connectors. Only video and VBI capture devices have inputs, output devices have outputs, at least one each. Radio devices have no video inputs or outputs.

To learn about the number and attributes of the available inputs and outputs applications can enumerate them with the `VIDIOC_ENUMINPUT` and `VIDIOC_ENUMOUTPUT` ioctl, respectively. The struct v4l2_input returned by the `VIDIOC_ENUMINPUT` ioctl also contains signal status information applicable when the current video input is queried.

The `VIDIOC_G_INPUT` and `VIDIOC_G_OUTPUT` ioctl return the index of the current video input or output. To select a different input or output applications call the `VIDIOC_S_INPUT` and `VIDIOC_S_OUTPUT` ioctl. Drivers must implement all the input ioctls when the device has one or more inputs, all the output ioctls when the device has one or more outputs.

Example 1.1. Information about the current video input

```
struct v4l2_input input;
int index;

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

memset (&input, 0, sizeof (input));
input.index = index;

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

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

Example 1.2. Switching to the first video input

```
int index;

index = 0;

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

Audio Inputs and Outputs

Audio inputs and outputs are physical connectors of a device. Video capture devices have inputs, output devices have outputs, zero or more each. Radio devices have no audio inputs or outputs. They have exactly one tuner which in fact *is* an audio source, but this API associates tuners with video inputs or outputs only, and radio devices have none of these.[5] A connector on a TV card to loop back the received audio signal to a sound card is not considered an audio output.

Audio and video inputs and outputs are associated. Selecting a video source also selects an audio source. This is most evident when the video and audio source is a tuner. Further audio connectors can combine with more than one video input or output. Assumed two composite video inputs and two audio inputs exist, there may be up to four valid combinations. The relation of video and audio connectors is defined in the *audioset* field of the respective struct v4l2_input or struct v4l2_output, where each bit represents the index number, starting at zero, of one audio input or output.

To learn about the number and attributes of the available inputs and outputs applications can enumerate them with the `VIDIOC_ENUMAUDIO` and `VIDIOC_ENUMAUDOUT` ioctl, respectively. The struct v4l2_audio returned by the `VIDIOC_ENUMAUDIO` ioctl also contains signal status information applicable when the current audio input is queried.

The `VIDIOC_G_AUDIO` and `VIDIOC_G_AUDOUT` ioctl report the current audio input and output, respectively. Note that, unlike `VIDIOC_G_INPUT` and `VIDIOC_G_OUTPUT` these ioctls return a structure as `VIDIOC_ENUMAUDIO` and `VIDIOC_ENUMAUDOUT` do, not just an index.

To select an audio input and change its properties applications call the `VIDIOC_S_AUDIO` ioctl. To select an audio output (which presently has no changeable properties) applications call the `VIDIOC_S_AUDOUT` ioctl.

Drivers must implement all input ioctls when the device has one or more inputs, all output ioctls when the device has one or more outputs. When the device has any audio inputs or outputs the driver must set the `V4L2_CAP_AUDIO` flag in the struct v4l2_capability returned by the `VIDIOC_QUERYCAP` ioctl.

Example 1.3. Information about the current audio input

```
struct v4l2_audio audio;

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

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

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

Example 1.4. Switching to the first audio input

```
struct v4l2_audio audio;

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

audio.index = 0;

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

Tuners and Modulators

Tuners

Video input devices can have one or more tuners demodulating a RF signal. Each tuner is associated with one or more video inputs, depending on the number of RF connectors on the tuner. The *type* field of the respective struct v4l2_input returned by the

VIDIOC_ENUMINPUT ioctl is set to V4L2_INPUT_TYPE_TUNER and its *tuner* field contains the index number of the tuner.

Radio devices have exactly one tuner with index zero, no video inputs.

To query and change tuner properties applications use the VIDIOC_G_TUNER and VIDIOC_S_TUNER ioctl, respectively. The struct v4l2_tuner returned by VIDIOC_G_TUNER also contains signal status information applicable when the tuner of the current video input, or a radio tuner is queried. Note that VIDIOC_S_TUNER does not switch the current tuner, when there is more than one at all. The tuner is solely determined by the current video input. Drivers must support both ioctls and set the V4L2_CAP_TUNER flag in the struct v4l2_capability returned by the VIDIOC_QUERYCAP ioctl when the device has one or more tuners.

Modulators

Video output devices can have one or more modulators, uh, modulating a video signal for radiation or connection to the antenna input of a TV set or video recorder. Each modulator is associated with one or more video outputs, depending on the number of RF connectors on the modulator. The *type* field of the respective struct v4l2_output returned by the VIDIOC_ENUMOUTPUT ioctl is set to V4L2_OUTPUT_TYPE_MODULATOR and its *modulator* field contains the index number of the modulator. This specification does not define radio output devices.

To query and change modulator properties applications use the VIDIOC_G_MODULATOR and VIDIOC_S_MODULATOR ioctl. Note that VIDIOC_S_MODULATOR does not switch the current modulator, when there is more than one at all. The modulator is solely determined by the current video output. Drivers must support both ioctls and set the V4L2_CAP_MODULATOR flag in the struct v4l2_capability returned by the VIDIOC_QUERYCAP ioctl when the device has one or more modulators.

Radio Frequency

To get and set the tuner or modulator radio frequency applications use the VIDIOC_G_FREQUENCY and VIDIOC_S_FREQUENCY ioctl which both take a pointer to a struct v4l2_frequency. These ioctls are used for TV and radio devices alike. Drivers must support both ioctls when the tuner or modulator ioctls are supported, or when the device is a radio device.

Video Standards

Video devices typically support one or more different video standards or variations of standards. Each video input and output may support another set of standards. This set is reported by the *std* field of struct v4l2_input and struct v4l2_output returned by the VIDIOC_ENUMINPUT and VIDIOC_ENUMOUTPUT ioctl, respectively.

V4L2 defines one bit for each analog video standard currently in use worldwide, and sets aside bits for driver defined standards, e. g. hybrid standards to watch NTSC video tapes on PAL TVs and vice versa. Applications can use the predefined bits to select a particular standard, although presenting the user a menu of supported standards is preferred. To enumerate and query the attributes of the supported standards applications use the VIDIOC_ENUMSTD ioctl.

Many of the defined standards are actually just variations of a few major standards. The hardware may in fact not distinguish between them, or do so internal and switch automatically. Therefore enumerated standards also contain sets of one or more standard bits.

Assume a hypothetic tuner capable of demodulating B/PAL, G/PAL and I/PAL signals. The first enumerated standard is a set of B and G/PAL, switched automatically depending on the selected radio frequency in UHF or VHF band. Enumeration gives a "PAL-B/G" or "PAL-I" choice. Similar a Composite input may collapse standards, enumerating "PAL-B/G/H/I", "NTSC-M" and "SECAM-D/K".[6]

To query and select the standard used by the current video input or output applications call the VIDIOC_G_STD and VIDIOC_S_STD ioctl, respectively. The *received* standard can be sensed with the VIDIOC_QUERYSTD ioctl. Note parameter of all these ioctls is a pointer to a v4l2_std_id type (a standard set), *not* an index into the standard enumeration.[7] Drivers must implement all video standard ioctls when the device has one or more video inputs or outputs.

Special rules apply to USB cameras where the notion of video standards makes little sense. More generally any capture device, output devices accordingly, which is

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

Here the driver shall set the *std* field of struct v4l2_input and struct v4l2_output to zero, the VIDIOC_G_STD, VIDIOC_S_STD, Here the driver shall set the std field of struct v4l2_input and struct v4l2_output to zero, the VIDIOC_G_STD, VIDIOC_S_STD, VIDIOC_QUERYSTD and VIDIOC_ENUMSTD ioctls shall return the EINVAL error code.[8]

Example 1.5. Information about the current video standard

```
v4l2_std_id std_id;
struct v4l2_standard standard;

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

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

memset (&standard, 0, sizeof (standard));
standard.index = 0;

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

        standard.index++;
}

/* EINVAL indicates the end of the enumeration, which cannot be
   empty unless this device falls under the USB exception. */

if (errno == EINVAL || standard.index == 0) {
        perror ("VIDIOC_ENUMSTD");
        exit (EXIT_FAILURE);
}
```

Example 1.6. Listing the video standards supported by the current input

```
struct v4l2_input input;
struct v4l2_standard standard;

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

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

if (-1 == ioctl (fd, VIDIOC_ENUMINPUT, &input)) {
        perror ("VIDIOC_ENUM_INPUT");
        exit (EXIT_FAILURE);
}

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

memset (&standard, 0, sizeof (standard));
standard.index = 0;

while (0 == ioctl (fd, VIDIOC_ENUMSTD, &standard)) {
        if (standard.id & input.std)
                printf ("%s\n", standard.name);

        standard.index++;
}

/* EINVAL indicates the end of the enumeration, which cannot be
   empty unless this device falls under the USB exception. */

if (errno != EINVAL || standard.index == 0) {
        perror ("VIDIOC_ENUMSTD");
        exit (EXIT_FAILURE);
}
```

Example 1.7. Selecting a new video standard

```
struct v4l2_input input;
v4l2_std_id std_id;

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

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

if (-1 == ioctl (fd, VIDIOC_ENUMINPUT, &input)) {
        perror ("VIDIOC_ENUM_INPUT");
        exit (EXIT_FAILURE);
}

if (0 == (input.std & V4L2_STD_PAL_BG)) {
        fprintf (stderr, "Oops. B/G PAL is not supported.\n");
        exit (EXIT_FAILURE);
}

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

std_id = V4L2_STD_PAL_BG;

if (-1 == ioctl (fd, VIDIOC_S_STD, &std_id)) {
        perror ("VIDIOC_S_STD");
        exit (EXIT_FAILURE);
}
```

User Controls

Devices typically have a number of user-settable controls such as brightness, saturation and so on, which would be presented to the user on a graphical user interface. But, different devices will have different controls available, and furthermore, the range of possible values, and the default value will vary from device to device. The control ioctls provide the information and a mechanism to create a nice user interface for these controls that will work correctly with any device.

All controls are accessed using an ID value. V4L2 defines several IDs for specific purposes. Drivers can also implement their own custom controls using `V4L2_CID_PRIVATE_BASE` and higher values. The pre-defined control IDs have the prefix `V4L2_CID_`, and are listed in Table 1.1, "Control IDs". The ID is used when querying the attributes of a control, and when getting or setting the current value.

Generally applications should present controls to the user without assumptions about their purpose. Each control comes with a name string the user is supposed to understand. When the purpose is non-intuitive the driver writer should provide a user manual, a user interface plug-in or a driver specific panel application. Predefined IDs were introduced to change a few controls programmatically, for example to mute a device during a channel switch.

Drivers may enumerate different controls after switching the current video input or output, tuner or modulator, or audio input or output. Different in the sense of other bounds, another default and current value, step size or other menu items. A control with a certain *custom* ID can also change name and type.[9] Control values are stored globally, they do not change when switching except to stay within the reported bounds. They also do not change e. g. when the device is opened or closed, when the tuner radio frequency is changed or generally never without application request. Since V4L2 specifies no event mechanism, panel applications intended to cooperate with other panel applications (be they built into a larger application, as a TV viewer) may need to regularly poll control values to update their user interface.[10]

Table 1.1. Control IDs

ID	Type	Description
`V4L2_CID_BASE`		First predefined ID, equal to `V4L2_CID_BRIGHTNESS`.
`V4L2_CID_USER_BASE`		Synonym of `V4L2_CID_BASE`.

`V4L2_CID_BRIGHTNESS`	integer	Picture brightness, or more precisely, the black level.
`V4L2_CID_CONTRAST`	integer	Picture contrast or luma gain.
`V4L2_CID_SATURATION`	integer	Picture color saturation or chroma gain.
`V4L2_CID_HUE`	integer	Hue or color balance.
`V4L2_CID_AUDIO_VOLUME`	integer	Overall audio volume. Note some drivers also provide an OSS or ALSA mixer interface.
`V4L2_CID_AUDIO_BALANCE`	integer	Audio stereo balance. Minimum corresponds to all the way left, maximum to right.
`V4L2_CID_AUDIO_BASS`	integer	Audio bass adjustment.
`V4L2_CID_AUDIO_TREBLE`	integer	Audio treble adjustment.
`V4L2_CID_AUDIO_MUTE`	boolean	Mute audio, i. e. set the volume to zero, however without affecting `V4L2_CID_AUDIO_VOLUME`. Like ALSA drivers, V4L2 drivers must mute at load time to avoid excessive noise. Actually the entire device should be reset to a low power consumption state.
`V4L2_CID_AUDIO_LOUDNESS`	boolean	Loudness mode (bass boost).
`V4L2_CID_BLACK_LEVEL`	integer	Another name for brightness (not a synonym of `V4L2_CID_BRIGHTNESS`). This control is deprecated and should not be used in new drivers and applications.
`V4L2_CID_AUTO_WHITE_BALANCE`	boolean	Automatic white balance (cameras).
`V4L2_CID_DO_WHITE_BALANCE`	button	This is an action control. When set (the value is ignored), the device will do a white balance and then hold the current setting. Contrast this with the boolean `V4L2_CID_AUTO_WHITE_BALANCE`, which, when activated, keeps adjusting the white balance.
`V4L2_CID_RED_BALANCE`	integer	Red chroma balance.
`V4L2_CID_BLUE_BALANCE`	integer	Blue chroma balance.
`V4L2_CID_GAMMA`	integer	Gamma adjust.
`V4L2_CID_WHITENESS`	integer	Whiteness for grey-scale devices. This is a synonym for `V4L2_CID_GAMMA`. This control is deprecated and should not be used in new drivers and applications.
`V4L2_CID_EXPOSURE`	integer	Exposure (cameras). [Unit?]
`V4L2_CID_AUTOGAIN`	boolean	Automatic gain/exposure control.
`V4L2_CID_GAIN`	integer	Gain control.
`V4L2_CID_HFLIP`	boolean	Mirror the picture horizontally.
`V4L2_CID_VFLIP`	boolean	Mirror the picture vertically.
`V4L2_CID_HCENTER_DEPRECATED` (formerly `V4L2_CID_HCENTER`)	integer	Horizontal image centering. This control is deprecated. New drivers and applications should use the Camera class controls `V4L2_CID_PAN_ABSOLUTE`, `V4L2_CID_PAN_RELATIVE` and `V4L2_CID_PAN_RESET` instead.
`V4L2_CID_VCENTER_DEPRECATED` (formerly `V4L2_CID_VCENTER`)	integer	Vertical image centering. Centering is intended to *physically* adjust cameras. For image cropping see the section called "Image Cropping, Insertion and Scaling", for clipping the section called "Video Overlay Interface". This control is deprecated. New drivers and applications should use the Camera class controls `V4L2_CID_TILT_ABSOLUTE`, `V4L2_CID_TILT_RELATIVE` and `V4L2_CID_TILT_RESET` instead.
`V4L2_CID_POWER_LINE_FREQUENCY`	enum	Enables a power line frequency filter to avoid flicker. Possible values for `enum v4l2_power_line_frequency` are: `V4L2_CID_POWER_LINE_FREQUENCY_DISABLED` (0), `V4L2_CID_POWER_LINE_FREQUENCY_50HZ` (1) and `V4L2_CID_POWER_LINE_FREQUENCY_60HZ` (2).
`V4L2_CID_HUE_AUTO`	boolean	Enables automatic hue control by the device. The effect of setting `V4L2_CID_HUE` while automatic hue control is enabled is undefined, drivers should ignore such request.
`V4L2_CID_WHITE_BALANCE_TEMPERATURE`	integer	This control specifies the white balance settings as a color temperature in Kelvin. A driver should have a minimum of 2800 (incandescent) to 6500 (daylight). For more information about color temperature see Wikipedia.
`V4L2_CID_SHARPNESS`	integer	Adjusts the sharpness filters in a camera. The minimum value disables the filters, higher values give a sharper picture.
`V4L2_CID_BACKLIGHT_COMPENSATION`	integer	Adjusts the backlight compensation in a camera. The minimum value disables backlight compensation.
`V4L2_CID_CHROMA_AGC`	boolean	Chroma automatic gain control.

V4L2_CID_COLOR_KILLER	boolean	Enable the color killer (i. e. force a black & white image in case of a weak video signal).
V4L2_CID_COLORFX	enum	Selects a color effect. Possible values for enum v4l2_colorfx are: V4L2_COLORFX_NONE (0), V4L2_COLORFX_BW (1) and V4L2_COLORFX_SEPIA (2).
V4L2_CID_LASTP1		End of the predefined control IDs (currently V4L2_CID_COLORFX + 1).
V4L2_CID_PRIVATE_BASE		ID of the first custom (driver specific) control. Applications depending on particular custom controls should check the driver name and version, see the section called “Querying Capabilities”.

Applications can enumerate the available controls with the VIDIOC_QUERYCTRL and VIDIOC_QUERYMENU ioctls, get and set a control value with the VIDIOC_G_CTRL and VIDIOC_S_CTRL ioctls. Drivers must implement VIDIOC_QUERYCTRL, VIDIOC_G_CTRL and VIDIOC_S_CTRL when the device has one or more controls, VIDIOC_QUERYMENU when it has one or more menu type controls.

Example 1.8. Enumerating all controls

```
struct v4l2_queryctrl queryctrl;
struct v4l2_querymenu querymenu;

static void
enumerate_menu (void)
{
        printf ("  Menu items:\n");

        memset (&querymenu, 0, sizeof (querymenu));
        querymenu.id = queryctrl.id;

        for (querymenu.index = queryctrl.minimum;
             querymenu.index <= queryctrl.maximum;
              querymenu.index++) {
                if (0 == ioctl (fd, VIDIOC_QUERYMENU, &querymenu)) {
                        printf ("  %s\n", querymenu.name);
                } else {
                        perror ("VIDIOC_QUERYMENU");
                        exit (EXIT_FAILURE);
                }
        }
}

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

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

                printf ("Control %s\n", queryctrl.name);

                if (queryctrl.type == V4L2_CTRL_TYPE_MENU)
                        enumerate_menu ();
        } else {
                if (errno == EINVAL)
                        continue;

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

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

                printf ("Control %s\n", queryctrl.name);

                if (queryctrl.type == V4L2_CTRL_TYPE_MENU)
                        enumerate_menu ();
        } else {
                if (errno == EINVAL)
                        break;
```

```
                        break;

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

Example 1.9. Changing controls

```
struct v4l2_queryctrl queryctrl;
struct v4l2_control control;

memset (&queryctrl, 0, sizeof (queryctrl));
queryctrl.id = V4L2_CID_BRIGHTNESS;

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

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

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

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

        /* The driver may clamp the value or return ERANGE, ignored here */

        if (-1 == ioctl (fd, VIDIOC_S_CTRL, &control)
            && errno != ERANGE) {
                perror ("VIDIOC_S_CTRL");
                exit (EXIT_FAILURE);
        }
/* Ignore if V4L2_CID_CONTRAST is unsupported */
} else if (errno != EINVAL) {
        perror ("VIDIOC_G_CTRL");
        exit (EXIT_FAILURE);
}

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

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

Extended Controls

Introduction

The control mechanism as originally designed was meant to be used for user settings (brightness, saturation, etc). However, it turned out to be a very useful model for implementing more complicated driver APIs where each driver implements only a subset of a larger API.

The MPEG encoding API was the driving force behind designing and implementing this extended control mechanism: the MPEG standard is quite large and the currently supported hardware MPEG encoders each only implement a subset of this standard. Further standard is quite large and the currently supported hardware MPEG encoders each only implement a subset of this standard. Further more, many parameters relating to how the video is encoded into an MPEG stream are specific to the MPEG encoding chip since the MPEG standard only defines the format of the resulting MPEG stream, not how the video is actually encoded into that format.

Unfortunately, the original control API lacked some features needed for these new uses and so it was extended into the (not terribly originally named) extended control API.

Even though the MPEG encoding API was the first effort to use the Extended Control API, nowadays there are also other classes of Extended Controls, such as Camera Controls and FM Transmitter Controls. The Extended Controls API as well as all Extended Controls classes are described in the following text.

The Extended Control API

Three new ioctls are available: `VIDIOC_G_EXT_CTRLS`, `VIDIOC_S_EXT_CTRLS` and `VIDIOC_TRY_EXT_CTRLS`. These ioctls act on arrays of controls (as opposed to the `VIDIOC_G_CTRL` and `VIDIOC_S_CTRL` ioctls that act on a single control). This is needed since it is often required to atomically change several controls at once.

Each of the new ioctls expects a pointer to a struct v4l2_ext_controls. This structure contains a pointer to the control array, a count of the number of controls in that array and a control class. Control classes are used to group similar controls into a single class. For example, control class `V4L2_CTRL_CLASS_USER` contains all user controls (i. e. all controls that can also be set using the old `VIDIOC_S_CTRL` ioctl). Control class `V4L2_CTRL_CLASS_MPEG` contains all controls relating to MPEG encoding, etc.

All controls in the control array must belong to the specified control class. An error is returned if this is not the case.

It is also possible to use an empty control array (count == 0) to check whether the specified control class is supported.

The control array is a struct v4l2_ext_control array. The v4l2_ext_control structure is very similar to struct v4l2_control, except for the fact that it also allows for 64-bit values and pointers to be passed.

It is important to realize that due to the flexibility of controls it is necessary to check whether the control you want to set actually is supported in the driver and what the valid range of values is. So use the `VIDIOC_QUERYCTRL` and `VIDIOC_QUERYMENU` ioctls to check this. Also note that it is possible that some of the menu indices in a control of type `V4L2_CTRL_TYPE_MENU` may not be supported (`VIDIOC_QUERYMENU` will return an error). A good example is the list of supported MPEG audio bitrates. Some drivers only support one or two bitrates, others support a wider range.

Enumerating Extended Controls

The recommended way to enumerate over the extended controls is by using `VIDIOC_QUERYCTRL` in combination with the `V4L2_CTRL_FLAG_NEXT_CTRL` flag:

```
struct v4l2_queryctrl qctrl;

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

The initial control ID is set to 0 ORed with the `V4L2_CTRL_FLAG_NEXT_CTRL` flag. The `VIDIOC_QUERYCTRL` ioctl will return the first control with a higher ID than the specified one. When no such controls are found an error is returned.

If you want to get all controls within a specific control class, then you can set the initial *`qctrl.id`* value to the control class and add an extra check to break out of the loop when a control of another control class is found:

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

The 32-bit *`qctrl.id`* value is subdivided into three bit ranges: the top 4 bits are reserved for flags (e. g. `V4L2_CTRL_FLAG_NEXT_CTRL`) and are not actually part of the ID. The remaining 28 bits form the control ID, of which the most significant 12 bits define the control class and the least significant 16 bits identify the control within the control class. It is guaranteed that these last 16 bits are always non-zero for controls. The range of 0x1000 and up are reserved for driver-specific controls. The macro `V4L2_CTRL_ID2CLASS(id)` returns the control class ID based on a control ID.

If the driver does not support extended controls, then `VIDIOC_QUERYCTRL` will fail when used in combination with

If the driver does not support extended controls, then VIDIOC_QUERYCTRL will fail when used in combination with V4L2_CTRL_FLAG_NEXT_CTRL. In that case the old method of enumerating control should be used (see 1.8). But if it is supported, then it is guaranteed to enumerate over all controls, including driver-private controls.

Creating Control Panels

It is possible to create control panels for a graphical user interface where the user can select the various controls. Basically you will have to iterate over all controls using the method described above. Each control class starts with a control of type V4L2_CTRL_TYPE_CTRL_CLASS. VIDIOC_QUERYCTRL will return the name of this control class which can be used as the title of a tab page within a control panel.

The flags field of struct v4l2_queryctrl also contains hints on the behavior of the control. See the VIDIOC_QUERYCTRL documentation for more details.

MPEG Control Reference

Below all controls within the MPEG control class are described. First the generic controls, then controls specific for certain hardware.

Generic MPEG Controls

Table 1.2. MPEG Control IDs

ID	Type	Description
V4L2_CID_MPEG_CLASS	class	The MPEG class descriptor. Calling VIDIOC_QUERYCTRL for this control will return a description of this control class. This description can be used as the caption of a Tab page in a GUI, for example.
V4L2_CID_MPEG_STREAM_TYPE	enum v4l2_mpeg_stream_type	The MPEG-1, -2 or -4 output stream type. One cannot assume anything here. Each hardware MPEG encoder tends to support different subsets of the available MPEG stream types. The currently defined stream types are: V4L2_MPEG_STREAM_TYPE_MPEG2_PS — MPEG-2 program stream V4L2_MPEG_STREAM_TYPE_MPEG2_TS — MPEG-2 transport stream V4L2_MPEG_STREAM_TYPE_MPEG1_SS — MPEG-1 system stream V4L2_MPEG_STREAM_TYPE_MPEG2_DVD — MPEG-2 DVD-compatible stream V4L2_MPEG_STREAM_TYPE_MPEG1_VCD — MPEG-1 VCD-compatible stream V4L2_MPEG_STREAM_TYPE_MPEG2_SVCD — MPEG-2 SVCD-compatible stream
V4L2_CID_MPEG_STREAM_PID_PMT	integer	Program Map Table Packet ID for the MPEG transport stream (default 16)
V4L2_CID_MPEG_STREAM_PID_AUDIO	integer	Audio Packet ID for the MPEG transport stream (default 256)
V4L2_CID_MPEG_STREAM_PID_VIDEO	integer	Video Packet ID for the MPEG transport stream (default 260)
V4L2_CID_MPEG_STREAM_PID_PCR	integer	Packet ID for the MPEG transport stream carrying PCR fields (default 259)
V4L2_CID_MPEG_STREAM_PES_ID_AUDIO	integer	Audio ID for MPEG PES
V4L2_CID_MPEG_STREAM_PES_ID_VIDEO	integer	Video ID for MPEG PES

`V4L2_CID_MPEG_STREAM_VBI_FMT` enum v4l2_mpeg_stream_vbi_fmt

Some cards can embed VBI data (e. g. Closed Caption, Teletext) into the MPEG stream. This control selects whether VBI data should be embedded, and if so, what embedding method should be used. The list of possible VBI formats depends on the driver. The currently defined VBI format types are:

`V4L2_MPEG_STREAM_VBI_FMT_NONE`	No VBI in the MPEG stream
`V4L2_MPEG_STREAM_VBI_FMT_IVTV`	VBI in private packets, IVTV format (documented in the kernel sources in the file `Documentation/video4linux/cx2341x/README.vbi`)

`V4L2_CID_MPEG_AUDIO_SAMPLING_FREQ` enum v4l2_mpeg_audio_sampling_freq

MPEG Audio sampling frequency. Possible values are:

`V4L2_MPEG_AUDIO_SAMPLING_FREQ_44100`	44.1 kHz
`V4L2_MPEG_AUDIO_SAMPLING_FREQ_48000`	48 kHz
`V4L2_MPEG_AUDIO_SAMPLING_FREQ_32000`	32 kHz

`V4L2_CID_MPEG_AUDIO_ENCODING` enum v4l2_mpeg_audio_encoding

MPEG Audio encoding. Possible values are:

`V4L2_MPEG_AUDIO_ENCODING_LAYER_1`	MPEG-1/2 Layer I encoding
`V4L2_MPEG_AUDIO_ENCODING_LAYER_2`	MPEG-1/2 Layer II encoding
`V4L2_MPEG_AUDIO_ENCODING_LAYER_3`	MPEG-1/2 Layer III encoding
`V4L2_MPEG_AUDIO_ENCODING_AAC`	MPEG-2/4 AAC (Advanced Audio Coding)
`V4L2_MPEG_AUDIO_ENCODING_AC3`	AC-3 aka ATSC A/52 encoding

`V4L2_CID_MPEG_AUDIO_L1_BITRATE` enum v4l2_mpeg_audio_l1_bitrate

MPEG-1/2 Layer I bitrate. Possible values are:

`V4L2_MPEG_AUDIO_L1_BITRATE_32K`	32 kbit/s
`V4L2_MPEG_AUDIO_L1_BITRATE_64K`	64 kbit/s
`V4L2_MPEG_AUDIO_L1_BITRATE_96K`	96 kbit/s
`V4L2_MPEG_AUDIO_L1_BITRATE_128K`	128 kbit/s
`V4L2_MPEG_AUDIO_L1_BITRATE_160K`	160 kbit/s
`V4L2_MPEG_AUDIO_L1_BITRATE_192K`	192 kbit/s
`V4L2_MPEG_AUDIO_L1_BITRATE_224K`	224 kbit/s
`V4L2_MPEG_AUDIO_L1_BITRATE_256K`	256 kbit/s
`V4L2_MPEG_AUDIO_L1_BITRATE_288K`	288 kbit/s
`V4L2_MPEG_AUDIO_L1_BITRATE_320K`	320 kbit/s
`V4L2_MPEG_AUDIO_L1_BITRATE_352K`	352 kbit/s
`V4L2_MPEG_AUDIO_L1_BITRATE_384K`	384 kbit/s
`V4L2_MPEG_AUDIO_L1_BITRATE_416K`	416 kbit/s
`V4L2_MPEG_AUDIO_L1_BITRATE_448K`	448 kbit/s

`V4L2_CID_MPEG_AUDIO_L2_BITRATE` enum v4l2_mpeg_audio_l2_bitrate

MPEG-1/2 Layer II bitrate. Possible values are:

`V4L2_MPEG_AUDIO_L2_BITRATE_32K`	32 kbit/s
`V4L2_MPEG_AUDIO_L2_BITRATE_48K`	48 kbit/s
`V4L2_MPEG_AUDIO_L2_BITRATE_56K`	56 kbit/s
`V4L2_MPEG_AUDIO_L2_BITRATE_64K`	64 kbit/s
`V4L2_MPEG_AUDIO_L2_BITRATE_80K`	80 kbit/s
`V4L2_MPEG_AUDIO_L2_BITRATE_96K`	96 kbit/s
`V4L2_MPEG_AUDIO_L2_BITRATE_112K`	112 kbit/s
`V4L2_MPEG_AUDIO_L2_BITRATE_128K`	128 kbit/s
`V4L2_MPEG_AUDIO_L2_BITRATE_160K`	160 kbit/s

V4L2_MPEG_AUDIO_L2_BITRATE_192K	192 kbit/s
V4L2_MPEG_AUDIO_L2_BITRATE_224K	224 kbit/s
V4L2_MPEG_AUDIO_L2_BITRATE_256K	256 kbit/s
V4L2_MPEG_AUDIO_L2_BITRATE_320K	320 kbit/s
V4L2_MPEG_AUDIO_L2_BITRATE_384K	384 kbit/s

V4L2_CID_MPEG_AUDIO_L3_BITRATE enum v4l2_mpeg_audio_l3_bitrate

MPEG-1/2 Layer III bitrate. Possible values are:

V4L2_MPEG_AUDIO_L3_BITRATE_32K	32 kbit/s
V4L2_MPEG_AUDIO_L3_BITRATE_40K	40 kbit/s
V4L2_MPEG_AUDIO_L3_BITRATE_48K	48 kbit/s
V4L2_MPEG_AUDIO_L3_BITRATE_56K	56 kbit/s
V4L2_MPEG_AUDIO_L3_BITRATE_64K	64 kbit/s
V4L2_MPEG_AUDIO_L3_BITRATE_80K	80 kbit/s
V4L2_MPEG_AUDIO_L3_BITRATE_96K	96 kbit/s
V4L2_MPEG_AUDIO_L3_BITRATE_112K	112 kbit/s
V4L2_MPEG_AUDIO_L3_BITRATE_128K	128 kbit/s
V4L2_MPEG_AUDIO_L3_BITRATE_160K	160 kbit/s
V4L2_MPEG_AUDIO_L3_BITRATE_192K	192 kbit/s
V4L2_MPEG_AUDIO_L3_BITRATE_224K	224 kbit/s
V4L2_MPEG_AUDIO_L3_BITRATE_256K	256 kbit/s
V4L2_MPEG_AUDIO_L3_BITRATE_320K	320 kbit/s

V4L2_CID_MPEG_AUDIO_AAC_BITRATE integer

AAC bitrate in bits per second.

V4L2_CID_MPEG_AUDIO_AC3_BITRATE enum v4l2_mpeg_audio_ac3_bitrate

AC-3 bitrate. Possible values are:

V4L2_MPEG_AUDIO_AC3_BITRATE_32K	32 kbit/s
V4L2_MPEG_AUDIO_AC3_BITRATE_40K	40 kbit/s
V4L2_MPEG_AUDIO_AC3_BITRATE_48K	48 kbit/s
V4L2_MPEG_AUDIO_AC3_BITRATE_56K	56 kbit/s
V4L2_MPEG_AUDIO_AC3_BITRATE_64K	64 kbit/s
V4L2_MPEG_AUDIO_AC3_BITRATE_80K	80 kbit/s
V4L2_MPEG_AUDIO_AC3_BITRATE_96K	96 kbit/s
V4L2_MPEG_AUDIO_AC3_BITRATE_112K	112 kbit/s
V4L2_MPEG_AUDIO_AC3_BITRATE_128K	128 kbit/s
V4L2_MPEG_AUDIO_AC3_BITRATE_160K	160 kbit/s
V4L2_MPEG_AUDIO_AC3_BITRATE_192K	192 kbit/s
V4L2_MPEG_AUDIO_AC3_BITRATE_224K	224 kbit/s
V4L2_MPEG_AUDIO_AC3_BITRATE_256K	256 kbit/s
V4L2_MPEG_AUDIO_AC3_BITRATE_320K	320 kbit/s
V4L2_MPEG_AUDIO_AC3_BITRATE_384K	384 kbit/s
V4L2_MPEG_AUDIO_AC3_BITRATE_448K	448 kbit/s
V4L2_MPEG_AUDIO_AC3_BITRATE_512K	512 kbit/s
V4L2_MPEG_AUDIO_AC3_BITRATE_576K	576 kbit/s
V4L2_MPEG_AUDIO_AC3_BITRATE_640K	640 kbit/s

V4L2_CID_MPEG_AUDIO_MODE enum v4l2_mpeg_audio_mode

MPEG Audio mode. Possible values are:

V4L2_MPEG_AUDIO_MODE_STEREO	Stereo
V4L2_MPEG_AUDIO_MODE_JOINT_STEREO	Joint Stereo

`V4L2_MPEG_AUDIO_MODE_DUAL`	Bilingual
`V4L2_MPEG_AUDIO_MODE_MONO`	Mono

`V4L2_CID_MPEG_AUDIO_MODE_EXTENSION` enum v4l2_mpeg_audio_mode_extension

Joint Stereo audio mode extension. In Layer I and II they indicate which subbands are in intensity stereo. All other subbands are coded in stereo. Layer III is not (yet) supported. Possible values are:

`V4L2_MPEG_AUDIO_MODE_EXTENSION_BOUND_4`	Subbands 4-31 in intensity stereo
`V4L2_MPEG_AUDIO_MODE_EXTENSION_BOUND_8`	Subbands 8-31 in intensity stereo
`V4L2_MPEG_AUDIO_MODE_EXTENSION_BOUND_12`	Subbands 12-31 in intensity stereo
`V4L2_MPEG_AUDIO_MODE_EXTENSION_BOUND_16`	Subbands 16-31 in intensity stereo

`V4L2_CID_MPEG_AUDIO_EMPHASIS` enum v4l2_mpeg_audio_emphasis

Audio Emphasis. Possible values are:

`V4L2_MPEG_AUDIO_EMPHASIS_NONE`	None
`V4L2_MPEG_AUDIO_EMPHASIS_50_DIV_15_uS`	50/15 microsecond emphasis
`V4L2_MPEG_AUDIO_EMPHASIS_CCITT_J17`	CCITT J.17

`V4L2_CID_MPEG_AUDIO_CRC` enum v4l2_mpeg_audio_crc

CRC method. Possible values are:

`V4L2_MPEG_AUDIO_CRC_NONE`	None
`V4L2_MPEG_AUDIO_CRC_CRC16`	16 bit parity check

`V4L2_CID_MPEG_AUDIO_MUTE` boolean

Mutes the audio when capturing. This is not done by muting audio hardware, which can still produce a slight hiss, but in the encoder itself, guaranteeing a fixed and reproducable audio bitstream. 0 = unmuted, 1 = muted.

`V4L2_CID_MPEG_VIDEO_ENCODING` enum v4l2_mpeg_video_encoding

MPEG Video encoding method. Possible values are:

`V4L2_MPEG_VIDEO_ENCODING_MPEG_1`	MPEG-1 Video encoding
`V4L2_MPEG_VIDEO_ENCODING_MPEG_2`	MPEG-2 Video encoding
`V4L2_MPEG_VIDEO_ENCODING_MPEG_4_AVC`	MPEG-4 AVC (H.264) Video encoding

`V4L2_CID_MPEG_VIDEO_ASPECT` enum v4l2_mpeg_video_aspect

Video aspect. Possible values are:

`V4L2_MPEG_VIDEO_ASPECT_1x1`
`V4L2_MPEG_VIDEO_ASPECT_4x3`
`V4L2_MPEG_VIDEO_ASPECT_16x9`
`V4L2_MPEG_VIDEO_ASPECT_221x100`

`V4L2_CID_MPEG_VIDEO_B_FRAMES` integer

Number of B-Frames (default 2)

`V4L2_CID_MPEG_VIDEO_GOP_SIZE` integer

GOP size (default 12)

`V4L2_CID_MPEG_VIDEO_GOP_CLOSURE` boolean

GOP closure (default 1)

`V4L2_CID_MPEG_VIDEO_PULLDOWN` boolean

Enable 3:2 pulldown (default 0)

`V4L2_CID_MPEG_VIDEO_BITRATE_MODE` enum v4l2_mpeg_video_bitrate_mode

Video bitrate mode. Possible values are:

`V4L2_MPEG_VIDEO_BITRATE_MODE_VBR`	Variable bitrate
`V4L2_MPEG_VIDEO_BITRATE_MODE_CBR`	Constant bitrate

`V4L2_CID_MPEG_VIDEO_BITRATE` integer

Video bitrate in bits per second.

`V4L2_CID_MPEG_VIDEO_BITRATE_PEAK` integer

Peak video bitrate in bits per second. Must be larger or equal to the average video bitrate. It is ignored if the video bitrate mode is set to constant bitrate.

`V4L2_CID_MPEG_VIDEO_TEMPORAL_DECIMATION` integer

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

`V4L2_CID_MPEG_VIDEO_MUTE` boolean

"Mutes" the video to a fixed color when capturing. This is useful for testing, to produce a fixed video bitstream. 0 = unmuted, 1 = muted.

`V4L2_CID_MPEG_VIDEO_MUTE_YUV` integer

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

Bit 0:7	V chrominance information
Bit 8:15	U chrominance information
Bit 16:23	Y luminance information
Bit 24:31	Must be zero.

CX2341x MPEG Controls

The following MPEG class controls deal with MPEG encoding settings that are specific to the Conexant CX23415 and CX23416 MPEG encoding chips.

Table 1.3. CX2341x Control IDs

ID **Type**

Description

`V4L2_CID_MPEG_CX2341X_VIDEO_SPATIAL_FILTER_MODE` enum v4l2_mpeg_cx2341x_video_spatial_filter_mode

Sets the Spatial Filter mode (default `MANUAL`). Possible values are:

`V4L2_MPEG_CX2341X_VIDEO_SPATIAL_FILTER_MODE_MANUAL`	Choose the filter manually
`V4L2_MPEG_CX2341X_VIDEO_SPATIAL_FILTER_MODE_AUTO`	Choose the filter automatically

`V4L2_CID_MPEG_CX2341X_VIDEO_SPATIAL_FILTER` integer (0-15)

The setting for the Spatial Filter. 0 = off, 15 = maximum. (Default is 0.)

`V4L2_CID_MPEG_CX2341X_VIDEO_LUMA_SPATIAL_FILTER_TYPE` enum v4l2_mpeg_cx2341x_video_luma_spatial_filter_type

Select the algorithm to use for the Luma Spatial Filter (default `1D_HOR`). Possible values:

`V4L2_MPEG_CX2341X_VIDEO_LUMA_SPATIAL_FILTER_TYPE_OFF`	No filter
`V4L2_MPEG_CX2341X_VIDEO_LUMA_SPATIAL_FILTER_TYPE_1D_HOR`	One-dimensional horizontal
`V4L2_MPEG_CX2341X_VIDEO_LUMA_SPATIAL_FILTER_TYPE_1D_VERT`	One-dimensional

	vertical
`V4L2_MPEG_CX2341X_VIDEO_LUMA_SPATIAL_FILTER_TYPE_2D_HV_SEPARABLE`	Two-dimensional separable
`V4L2_MPEG_CX2341X_VIDEO_LUMA_SPATIAL_FILTER_TYPE_2D_SYM_NON_SEPARABLE`	Two-dimensional symmetrical non-separable

`V4L2_CID_MPEG_CX2341X_VIDEO_CHROMA_SPATIAL_FILTER_TYPE` enum v4l2_mpeg_cx2341x_video_chroma_spatial_filter_type

Select the algorithm for the Chroma Spatial Filter (default `1D_HOR`). Possible values are:

`V4L2_MPEG_CX2341X_VIDEO_CHROMA_SPATIAL_FILTER_TYPE_OFF`	No filter
`V4L2_MPEG_CX2341X_VIDEO_CHROMA_SPATIAL_FILTER_TYPE_1D_HOR`	One-dimensional horizontal

`V4L2_CID_MPEG_CX2341X_VIDEO_TEMPORAL_FILTER_MODE` enum v4l2_mpeg_cx2341x_video_temporal_filter_mode

Sets the Temporal Filter mode (default `MANUAL`). Possible values are:

`V4L2_MPEG_CX2341X_VIDEO_TEMPORAL_FILTER_MODE_MANUAL`	Choose the filter manually
`V4L2_MPEG_CX2341X_VIDEO_TEMPORAL_FILTER_MODE_AUTO`	Choose the filter automatically

`V4L2_CID_MPEG_CX2341X_VIDEO_TEMPORAL_FILTER` integer (0-31)

The setting for the Temporal Filter. 0 = off, 31 = maximum. (Default is 8 for full-scale capturing and 0 for scaled capturing.)

`V4L2_CID_MPEG_CX2341X_VIDEO_MEDIAN_FILTER_TYPE` enum v4l2_mpeg_cx2341x_video_median_filter_type

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

`V4L2_MPEG_CX2341X_VIDEO_MEDIAN_FILTER_TYPE_OFF`	No filter
`V4L2_MPEG_CX2341X_VIDEO_MEDIAN_FILTER_TYPE_HOR`	Horizontal filter
`V4L2_MPEG_CX2341X_VIDEO_MEDIAN_FILTER_TYPE_VERT`	Vertical filter
`V4L2_MPEG_CX2341X_VIDEO_MEDIAN_FILTER_TYPE_HOR_VERT`	Horizontal and vertical filter
`V4L2_MPEG_CX2341X_VIDEO_MEDIAN_FILTER_TYPE_DIAG`	Diagonal filter

`V4L2_CID_MPEG_CX2341X_VIDEO_LUMA_MEDIAN_FILTER_BOTTOM` integer (0-255)

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

`V4L2_CID_MPEG_CX2341X_VIDEO_LUMA_MEDIAN_FILTER_TOP` integer (0-255)

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

`V4L2_CID_MPEG_CX2341X_VIDEO_CHROMA_MEDIAN_FILTER_BOTTOM` integer (0-255)

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

`V4L2_CID_MPEG_CX2341X_VIDEO_CHROMA_MEDIAN_FILTER_TOP` integer (0-255)

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

`V4L2_CID_MPEG_CX2341X_STREAM_INSERT_NAV_PACKETS` boolean

The CX2341X MPEG encoder can insert one empty MPEG-2 PES packet into the stream between every four video frames. The packet size is 2048 bytes, including the packet_start_code_prefix and stream_id fields. The stream_id is 0xBF (private stream 2). The payload consists of 0x00 bytes, to be filled in by the application. 0 = do not insert, 1 = insert packets.

Camera Control Reference

The Camera class includes controls for mechanical (or equivalent digital) features of a device such as controllable lenses or sensors.

Table 1.4. Camera Control IDs

ID	Type	Description
`V4L2_CID_CAMERA_CLASS`	class	The Camera class descriptor. Calling `VIDIOC_QUERYCTRL` for this control will return a description of this control class.
`V4L2_CID_EXPOSURE_AUTO`	enum v4l2_exposure_auto_type	Enables automatic adjustments of the exposure time and/or iris aperture. The effect of manual changes of the exposure time or iris aperture while these features are enabled is undefined, drivers should ignore such requests. Possible values are: `V4L2_EXPOSURE_AUTO` — Automatic exposure time, automatic iris aperture. `V4L2_EXPOSURE_MANUAL` — Manual exposure time, manual iris. `V4L2_EXPOSURE_SHUTTER_PRIORITY` — Manual exposure time, auto iris. `V4L2_EXPOSURE_APERTURE_PRIORITY` — Auto exposure time, manual iris.
`V4L2_CID_EXPOSURE_ABSOLUTE`	integer	Determines the exposure time of the camera sensor. The exposure time is limited by the frame interval. Drivers should interpret the values as 100 μs units, where the value 1 stands for 1/10000th of a second, 10000 for 1 second and 100000 for 10 seconds.
`V4L2_CID_EXPOSURE_AUTO_PRIORITY`	boolean	When `V4L2_CID_EXPOSURE_AUTO` is set to `AUTO` or `APERTURE_PRIORITY`, this control determines if the device may dynamically vary the frame rate. By default this feature is disabled (0) and the frame rate must remain constant.
`V4L2_CID_PAN_RELATIVE`	integer	This control turns the camera horizontally by the specified amount. The unit is undefined. A positive value moves the camera to the right (clockwise when viewed from above), a negative value to the left. A value of zero does not cause motion. This is a write-only control.
`V4L2_CID_TILT_RELATIVE`	integer	This control turns the camera vertically by the specified amount. The unit is undefined. A positive value moves the camera up, a negative value down. A value of zero does not cause motion. This is a write-only control.
`V4L2_CID_PAN_RESET`	button	When this control is set, the camera moves horizontally to the default position.
`V4L2_CID_TILT_RESET`	button	When this control is set, the camera moves vertically to the default position.
`V4L2_CID_PAN_ABSOLUTE`	integer	This control turns the camera horizontally to the specified position. Positive values move the camera to the right (clockwise when viewed from above), negative values to the left. Drivers should interpret the values as arc seconds, with valid values between -180 * 3600 and +180 * 3600 inclusive.
`V4L2_CID_TILT_ABSOLUTE`	integer	This control turns the camera vertically to the specified position. Positive values move the camera up, negative values down. Drivers should interpret the values as arc seconds, with valid values between -180 * 3600 and +180 * 3600 inclusive.

`V4L2_CID_FOCUS_ABSOLUTE` integer
: This control sets the focal point of the camera to the specified position. The unit is undefined. Positive values set the focus closer to the camera, negative values towards infinity.

`V4L2_CID_FOCUS_RELATIVE` integer
: This control moves the focal point of the camera by the specified amount. The unit is undefined. Positive values move the focus closer to the camera, negative values towards infinity. This is a write-only control.

`V4L2_CID_FOCUS_AUTO` boolean
: Enables automatic focus adjustments. The effect of manual focus adjustments while this feature is enabled is undefined, drivers should ignore such requests.

`V4L2_CID_ZOOM_ABSOLUTE` integer
: Specify the objective lens focal length as an absolute value. The zoom unit is driver-specific and its value should be a positive integer.

`V4L2_CID_ZOOM_RELATIVE` integer
: Specify the objective lens focal length relatively to the current value. Positive values move the zoom lens group towards the telephoto direction, negative values towards the wide-angle direction. The zoom unit is driver-specific. This is a write-only control.

`V4L2_CID_ZOOM_CONTINUOUS` integer
: Move the objective lens group at the specified speed until it reaches physical device limits or until an explicit request to stop the movement. A positive value moves the zoom lens group towards the telephoto direction. A value of zero stops the zoom lens group movement. A negative value moves the zoom lens group towards the wide-angle direction. The zoom speed unit is driver-specific.

`V4L2_CID_PRIVACY` boolean
: Prevent video from being acquired by the camera. When this control is set to `TRUE` (1), no image can be captured by the camera. Common means to enforce privacy are mechanical obturation of the sensor and firmware image processing, but the device is not restricted to these methods. Devices that implement the privacy control must support read access and may support write access.

`V4L2_CID_BAND_STOP_FILTER` integer
: Switch the band-stop filter of a camera sensor on or off, or specify its strength. Such band-stop filters can be used, for example, to filter out the fluorescent light component.

FM Transmitter Control Reference

The FM Transmitter (FM_TX) class includes controls for common features of FM transmissions capable devices. Currently this class includes parameters for audio compression, pilot tone generation, audio deviation limiter, RDS transmission and tuning power features.

Table 1.5. FM_TX Control IDs

ID	Type
Description	
`V4L2_CID_FM_TX_CLASS`	class
The FM_TX class descriptor. Calling [`VIDIOC_QUERYCTRL`] for this control will return a description of this control class.	
`V4L2_CID_RDS_TX_DEVIATION`	integer
Configures RDS signal frequency deviation level in Hz. The range and step are driver-specific.	
`V4L2_CID_RDS_TX_PI`	integer
Sets the RDS Programme Identification field for transmission.	
`V4L2_CID_RDS_TX_PTY`	integer

V4L2_CID_RDS_TX_PTY integer

Sets the RDS Programme Type field for transmission. This encodes up to 31 pre-defined programme types.

V4L2_CID_RDS_TX_PS_NAME string

Sets the Programme Service name (PS_NAME) for transmission. It is intended for static display on a receiver. It is the primary aid to listeners in programme service identification and selection. In Annex E of [EN 50067], the RDS specification, there is a full description of the correct character encoding for Programme Service name strings. Also from RDS specification, PS is usually a single eight character text. However, it is also possible to find receivers which can scroll strings sized as 8 x N characters. So, this control must be configured with steps of 8 characters. The result is it must always contain a string with size multiple of 8.

V4L2_CID_RDS_TX_RADIO_TEXT string

Sets the Radio Text info for transmission. It is a textual description of what is being broadcasted. RDS Radio Text can be applied when broadcaster wishes to transmit longer PS names, programme-related information or any other text. In these cases, RadioText should be used in addition to V4L2_CID_RDS_TX_PS_NAME. The encoding for Radio Text strings is also fully described in Annex E of [EN 50067]. The length of Radio Text strings depends on which RDS Block is being used to transmit it, either 32 (2A block) or 64 (2B block). However, it is also possible to find receivers which can scroll strings sized as 32 x N or 64 x N characters. So, this control must be configured with steps of 32 or 64 characters. The result is it must always contain a string with size multiple of 32 or 64.

V4L2_CID_AUDIO_LIMITER_ENABLED boolean

Enables or disables the audio deviation limiter feature. The limiter is useful when trying to maximize the audio volume, minimize receiver-generated distortion and prevent overmodulation.

V4L2_CID_AUDIO_LIMITER_RELEASE_TIME integer

Sets the audio deviation limiter feature release time. Unit is in useconds. Step and range are driver-specific.

V4L2_CID_AUDIO_LIMITER_DEVIATION integer

Configures audio frequency deviation level in Hz. The range and step are driver-specific.

V4L2_CID_AUDIO_COMPRESSION_ENABLED boolean

Enables or disables the audio compression feature. This feature amplifies signals below the threshold by a fixed gain and compresses audio signals above the threshold by the ratio of Threshold/(Gain + Threshold).

V4L2_CID_AUDIO_COMPRESSION_GAIN integer

Sets the gain for audio compression feature. It is a dB value. The range and step are driver-specific.

V4L2_CID_AUDIO_COMPRESSION_THRESHOLD integer

Sets the threshold level for audio compression freature. It is a dB value. The range and step are driver-specific.

V4L2_CID_AUDIO_COMPRESSION_ATTACK_TIME integer

Sets the attack time for audio compression feature. It is a useconds value. The range and step are driver-specific.

V4L2_CID_AUDIO_COMPRESSION_RELEASE_TIME integer

Sets the release time for audio compression feature. It is a useconds value. The range and step are driver-specific.

V4L2_CID_PILOT_TONE_ENABLED boolean

Enables or disables the pilot tone generation feature.

V4L2_CID_PILOT_TONE_DEVIATION integer

Configures pilot tone frequency deviation level. Unit is in Hz. The range and step are driver-specific.

V4L2_CID_PILOT_TONE_FREQUENCY integer

Configures pilot tone frequency value. Unit is in Hz. The range and step are driver-specific.

V4L2_CID_TUNE_PREEMPHASIS integer

Configures the pre-emphasis value for broadcasting. A pre-emphasis filter is applied to the broadcast to accentuate the high audio frequencies. Depending on the region, a time constant of either 50 or 75 useconds is used. The enum v4l2_preemphasis defines possible values for pre-emphasis. Here they are:

V4L2_PREEMPHASIS_DISABLED	No pre-emphasis is applied.
V4L2_PREEMPHASIS_50_uS	A pre-emphasis of 50 uS is used.
V4L2_PREEMPHASIS_75_uS	A pre-emphasis of 75 uS is used.

V4L2_CID_TUNE_POWER_LEVEL integer

Sets the output power level for signal transmission. Unit is in dBuV. Range and step are driver-specific.

V4L2_CID_TUNE_ANTENNA_CAPACITOR integer

This selects the value of antenna tuning capacitor manually or automatically if set to zero. Unit, range and

This selects the value of antenna tuning capacitor manually or automatically if set to zero. Unit, range and step are driver-specific.

For more details about RDS specification, refer to [EN 50067] document, from CENELEC.

Data Formats

Data Format Negotiation

Different devices exchange different kinds of data with applications, for example video images, raw or sliced VBI data, RDS datagrams. Even within one kind many different formats are possible, in particular an abundance of image formats. Although drivers must provide a default and the selection persists across closing and reopening a device, applications should always negotiate a data format before engaging in data exchange. Negotiation means the application asks for a particular format and the driver selects and reports the best the hardware can do to satisfy the request. Of course applications can also just query the current selection.

A single mechanism exists to negotiate all data formats using the aggregate struct v4l2_format and the `VIDIOC_G_FMT` and `VIDIOC_S_FMT` ioctls. Additionally the `VIDIOC_TRY_FMT` ioctl can be used to examine what the hardware *could* do, without actually selecting a new data format. The data formats supported by the V4L2 API are covered in the respective device section in Chapter 4, *Interfaces*. For a closer look at image formats see Chapter 2, *Image Formats*.

The `VIDIOC_S_FMT` ioctl is a major turning-point in the initialization sequence. Prior to this point multiple panel applications can access the same device concurrently to select the current input, change controls or modify other properties. The first `VIDIOC_S_FMT` assigns a logical stream (video data, VBI data etc.) exclusively to one file descriptor.

Exclusive means no other application, more precisely no other file descriptor, can grab this stream or change device properties inconsistent with the negotiated parameters. A video standard change for example, when the new standard uses a different number of scan lines, can invalidate the selected image format. Therefore only the file descriptor owning the stream can make invalidating changes. Accordingly multiple file descriptors which grabbed different logical streams prevent each other from interfering with their settings. When for example video overlay is about to start or already in progress, simultaneous video capturing may be restricted to the same cropping and image size.

When applications omit the `VIDIOC_S_FMT` ioctl its locking side effects are implied by the next step, the selection of an I/O method with the `VIDIOC_REQBUFS` ioctl or implicit with the first `read()` or `write()` call.

Generally only one logical stream can be assigned to a file descriptor, the exception being drivers permitting simultaneous video capturing and overlay using the same file descriptor for compatibility with V4L and earlier versions of V4L2. Switching the logical stream or returning into "panel mode" is possible by closing and reopening the device. Drivers *may* support a switch using `VIDIOC_S_FMT`.

All drivers exchanging data with applications must support the `VIDIOC_G_FMT` and `VIDIOC_S_FMT` ioctl. Implementation of the `VIDIOC_TRY_FMT` is highly recommended but optional.

Image Format Enumeration

Apart of the generic format negotiation functions a special ioctl to enumerate all image formats supported by video capture, overlay or output devices is available.[11]

The `VIDIOC_ENUM_FMT` ioctl must be supported by all drivers exchanging image data with applications.

Important

Drivers are not supposed to convert image formats in kernel space. They must enumerate only formats directly supported by the hardware. If necessary driver writers should publish an example conversion routine or library for integration into applications.

Image Cropping, Insertion and Scaling

Some video capture devices can sample a subsection of the picture and shrink or enlarge it to an image of arbitrary size. We call these abilities cropping and scaling. Some video output devices can scale an image up or down and insert it at an arbitrary scan line and horizontal offset into a video signal.

Applications can use the following API to select an area in the video signal, query the default area and the hardware limits. *Despite their name, the `VIDIOC_CROPCAP`, `VIDIOC_G_CROP` and `VIDIOC_S_CROP` ioctls apply to input as well as output devices.*

Scaling requires a source and a target. On a video capture or overlay device the source is the video signal, and the cropping ioctls determine the area actually sampled. The target are images read by the application or overlaid onto the graphics screen. Their size (and position for an overlay) is negotiated with the `VIDIOC_G_FMT` and `VIDIOC_S_FMT` ioctls.

On a video output device the source are the images passed in by the application, and their size is again negotiated with the `VIDIOC_G/S_FMT` ioctls, or may be encoded in a compressed video stream. The target is the video signal, and the cropping ioctls determine the area where the images are inserted.

Source and target rectangles are defined even if the device does not support scaling or the `VIDIOC_G/S_CROP` ioctls. Their size (and position where applicable) will be fixed in this case. *All capture and output device must support the `VIDIOC_CROPCAP` ioctl such that applications can determine if scaling takes place.*

Cropping Structures

Figure 1.1. Image Cropping, Insertion and Scaling

For capture devices the coordinates of the top left corner, width and height of the area which can be sampled is given by the *`bounds`* substructure of the struct v4l2_cropcap returned by the `VIDIOC_CROPCAP` ioctl. To support a wide range of hardware this specification does not define an origin or units. However by convention drivers should horizontally count unscaled samples relative to 0H (the leading edge of the horizontal sync pulse, see Figure 4.1, “Line synchronization”). Vertically ITU-R line numbers of the first field (Figure 4.2, “ITU-R 525 line numbering (M/NTSC and M/PAL)”, Figure 4.3, “ITU-R 625 line numbering”), multiplied by two if the driver can capture both fields.

The top left corner, width and height of the source rectangle, that is the area actually sampled, is given by struct v4l2_crop using the same coordinate system as struct v4l2_cropcap. Applications can use the `VIDIOC_G_CROP` and `VIDIOC_S_CROP` ioctls to get and set this rectangle. It must lie completely within the capture boundaries and the driver may further adjust the requested size and/or position according to hardware limitations.

Each capture device has a default source rectangle, given by the *`defrect`* substructure of struct v4l2_cropcap. The center of this rectangle shall align with the center of the active picture area of the video signal, and cover what the driver writer considers the complete picture. Drivers shall reset the source rectangle to the default when the driver is first loaded, but not later.

For output devices these structures and ioctls are used accordingly, defining the *target* rectangle where the images will be inserted into the video signal.

Scaling Adjustments

Video hardware can have various cropping, insertion and scaling limitations. It may only scale up or down, support only discrete scaling factors, or have different scaling abilities in horizontal and vertical direction. Also it may not support scaling at all. At the same time the struct v4l2_crop rectangle may have to be aligned, and both the source and target rectangles may have arbitrary upper and lower size limits. In particular the maximum *`width`* and *`height`* in struct v4l2_crop may be smaller than the struct v4l2_cropcap.*`bounds`* area. Therefore, as usual, drivers are expected to adjust the requested parameters and return the actual values selected.

Applications can change the source or the target rectangle first, as they may prefer a particular image size or a certain area in the video signal. If the driver has to adjust both to satisfy hardware limitations, the last requested rectangle shall take priority, and the driver should preferably adjust the opposite one. The `VIDIOC_TRY_FMT` ioctl however shall not change the driver state and therefore only adjust the requested rectangle.

Suppose scaling on a video capture device is restricted to a factor 1:1 or 2:1 in either direction and the target image size must be a multiple of 16 × 16 pixels. The source cropping rectangle is set to defaults, which are also the upper limit in this example, of 640 × 400 pixels at offset 0, 0. An application requests an image size of 300 × 225 pixels, assuming video will be scaled down from the "full picture" accordingly. The driver sets the image size to the closest possible values 304 × 224, then chooses the cropping rectangle closest to the requested size, that is 608 × 224 (224 × 2:1 would exceed the limit 400). The offset 0, 0 is still valid, thus unmodified. Given the default cropping rectangle reported by `VIDIOC_CROPCAP` the application can easily propose another offset to center the cropping rectangle.

Now the application may insist on covering an area using a picture aspect ratio closer to the original request, so it asks for a cropping rectangle of 608 × 456 pixels. The present scaling factors limit cropping to 640 × 384, so the driver returns the cropping size 608 × 384 and adjusts the image size to closest possible 304 × 192.

Examples

Source and target rectangles shall remain unchanged across closing and reopening a device, such that piping data into or out of a device will work without special preparations. More advanced applications should ensure the parameters are suitable before starting I/O.

Example 1.10. Resetting the cropping parameters

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

```
struct v4l2_cropcap cropcap;
struct v4l2_crop crop;

memset (&cropcap, 0, sizeof (cropcap));
cropcap.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;

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

memset (&crop, 0, sizeof (crop));
crop.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
crop.c = cropcap.defrect;

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

if (-1 == ioctl (fd, VIDIOC_S_CROP, &crop)
    && errno != EINVAL) {
        perror ("VIDIOC_S_CROP");
        exit (EXIT_FAILURE);
}
```

Example 1.11. Simple downscaling

(A video capture device is assumed.)

```
struct v4l2_cropcap cropcap;
struct v4l2_format format;

reset_cropping_parameters ();

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

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

format.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;

format.fmt.pix.width = cropcap.defrect.width >> 1;
format.fmt.pix.height = cropcap.defrect.height >> 1;
format.fmt.pix.pixelformat = V4L2_PIX_FMT_YUYV;

if (-1 == ioctl (fd, VIDIOC_S_FMT, &format)) {
        perror ("VIDIOC_S_FORMAT");
        exit (EXIT_FAILURE);
}

/* We could check the actual image size now, the actual scaling factor
   or if the driver can scale at all. */
```

Example 1.12. Selecting an output area

```
struct v4l2_cropcap cropcap;
struct v4l2_crop crop;
```

```
struct v4l2_crop crop;

memset (&cropcap, 0, sizeof (cropcap));
cropcap.type = V4L2_BUF_TYPE_VIDEO_OUTPUT;

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

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

crop.type = V4L2_BUF_TYPE_VIDEO_OUTPUT;
crop.c = cropcap.defrect;

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

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

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

if (-1 == ioctl (fd, VIDIOC_S_CROP, &crop)
    && errno != EINVAL) {
        perror ("VIDIOC_S_CROP");
        exit (EXIT_FAILURE);
}
```

Example 1.13. Current scaling factor and pixel aspect

(A video capture device is assumed.)

```
struct v4l2_cropcap cropcap;
struct v4l2_crop crop;
struct v4l2_format format;
double hscale, vscale;
double aspect;
int dwidth, dheight;

memset (&cropcap, 0, sizeof (cropcap));
cropcap.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;

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

memset (&crop, 0, sizeof (crop));
crop.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;

if (-1 == ioctl (fd, VIDIOC_G_CROP, &crop)) {
        if (errno != EINVAL) {
                perror ("VIDIOC_G_CROP");
                exit (EXIT_FAILURE);
        }

        /* Cropping not supported. */
        crop.c = cropcap.defrect;
}

memset (&format, 0, sizeof (format));
format.fmt.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;

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

/* The scaling applied by the driver. */

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

```
aspect = cropcap.pixelaspect.numerator /
         (double) cropcap.pixelaspect.denominator;
aspect = aspect * hscale / vscale;

/* Devices following ITU-R BT.601 do not capture
   square pixels. For playback on a computer monitor
   we should scale the images to this size. */

dwidth = format.fmt.pix.width / aspect;
dheight = format.fmt.pix.height;
```

Streaming Parameters

Streaming parameters are intended to optimize the video capture process as well as I/O. Presently applications can request a high quality capture mode with the `VIDIOC_S_PARM` ioctl.

The current video standard determines a nominal number of frames per second. If less than this number of frames is to be captured or output, applications can request frame skipping or duplicating on the driver side. This is especially useful when using the `read()` or `write()`, which are not augmented by timestamps or sequence counters, and to avoid unneccessary data copying.

Finally these ioctls can be used to determine the number of buffers used internally by a driver in read/write mode. For implications see the section discussing the `read()` function.

To get and set the streaming parameters applications call the `VIDIOC_G_PARM` and `VIDIOC_S_PARM` ioctl, respectively. They take a pointer to a struct v4l2_streamparm, which contains a union holding separate parameters for input and output devices.

These ioctls are optional, drivers need not implement them. If so, they return the EINVAL error code.

---

[1] Access permissions are associated with character device special files, hence we must ensure device numbers cannot change with the module load order. To this end minor numbers are no longer automatically assigned by the "videodev" module as in V4L but requested by the driver. The defaults will suffice for most people unless two drivers compete for the same minor numbers.

[2] In earlier versions of the V4L2 API the module options where named after the device special file with a "unit_" prefix, expressing the minor number itself, not an offset. Rationale for this change is unknown. Lastly the naming and semantics are just a convention among driver writers, the point to note is that minor numbers are not supposed to be hardcoded into drivers.

[3] Given a device file name one cannot reliable find related devices. For once names are arbitrary and in a system with multiple devices, where only some support VBI capturing, a `/dev/video2` is not necessarily related to `/dev/vbi2`. The V4L `VIDIOCGUNIT` ioctl would require a search for a device file with a particular major and minor number.

[4] Drivers could recognize the `O_EXCL` open flag. Presently this is not required, so applications cannot know if it really works.

[5] Actually struct v4l2_audio ought to have a *`tuner`* field like struct v4l2_input, not only making the API more consistent but also permitting radio devices with multiple tuners.

[6] Some users are already confused by technical terms PAL, NTSC and SECAM. There is no point asking them to distinguish between B, G, D, or K when the software or hardware can do that automatically.

[7] An alternative to the current scheme is to use pointers to indices as arguments of `VIDIOC_G_STD` and `VIDIOC_S_STD`, the struct v4l2_input and struct v4l2_output *`std`* field would be a set of indices like *`audioset`*.

Indices are consistent with the rest of the API and identify the standard unambiguously. In the present scheme of things an enumerated standard is looked up by v4l2_std_id. Now the standards supported by the inputs of a device can overlap. Just assume the tuner and composite input in the example above both exist on a device. An enumeration of "PAL-B/G", "PAL-H/I" suggests a choice which does not exist. We cannot merge or omit sets, because applications would be unable to find the standards reported by `VIDIOC_G_STD`. That leaves separate enumerations for each input. Also selecting a standard by v4l2_std_id can be ambiguous. Advantage of this method is that applications need not identify the standard indirectly, after enumerating.

So in summary, the lookup itself is unavoidable. The difference is only whether the lookup is necessary to find an enumerated standard or to switch to a standard by v4l2_std_id.

[8] See the section called “Buffers” for a rationale. Probably even USB cameras follow some well known video standard. It might have been better to explicitly indicate elsewhere if a device cannot live up to normal expectations, instead of this exception.

[9] It will be more convenient for applications if drivers make use of the `V4L2_CTRL_FLAG_DISABLED` flag, but that was never required.

[10] Applications could call an ioctl to request events. After another process called `VIDIOC_S_CTRL` or another ioctl changing shared properties the `select()` function would indicate readability until any ioctl (querying the properties) is called.

[11] Enumerating formats an application has no a-priori knowledge of (otherwise it could explicitly ask for them and need not enumerate) seems useless, but there are applications serving as proxy between drivers and the actual video applications for which this is useful.

Chapter 2. Image Formats

Table of Contents

The V4L2 API was primarily designed for devices exchanging image data with applications. The v4l2_pix_format structure defines the format and layout of an image in memory. Image formats are negotiated with the `VIDIOC_S_FMT` ioctl. (The explanations here focus on video capturing and output, for overlay frame buffer formats see also `VIDIOC_G_FBUF`.)

Table 2.1. struct v4l2_pix_format

__u32	*width*	Image width in pixels.
__u32	*height*	Image height in pixels.
Applications set these fields to request an image size, drivers return the closest possible values. In case of planar formats the *width* and *height* applies to the largest plane. To avoid ambiguities drivers must return values rounded up to a multiple of the scale factor of any smaller planes. For example when the image format is YUV 4:2:0, *width* and *height* must be multiples of two.		
__u32	*pixelformat*	The pixel format or type of compression, set by the application. This is a little endian four character code. V4L2 defines standard RGB formats in Table 2.4, “Packed RGB Image Formats”, YUV formats in the section called “YUV Formats”, and reserved codes in Table 2.8, “Reserved Image Formats”
enum v4l2_field	*field*	Video images are typically interlaced. Applications can request to capture or output only the top or bottom field, or both fields interlaced or sequentially stored in one buffer or alternating in separate buffers. Drivers return the actual field order selected. For details see the section called “Field Order”.
__u32	*bytesperline*	Distance in bytes between the leftmost pixels in two adjacent lines.
Both applications and drivers can set this field to request padding bytes at the end of each line. Drivers however may ignore the value requested by the application, returning *width* times bytes per pixel or a larger value required by the hardware. That implies applications can just set this field to zero to get a reasonable default. Video hardware may access padding bytes, therefore they must reside in accessible memory. Consider cases where padding bytes after the last line of an image cross a system page boundary. Input devices may write padding bytes, the value is undefined. Output devices ignore the contents of padding bytes. When the image format is planar the *bytesperline* value applies to the largest plane and is divided by the same factor as the *width* field for any smaller planes. For example the Cb and Cr planes of a YUV 4:2:0 image have half as many padding bytes following each line as the Y plane. To avoid ambiguities drivers must return a *bytesperline* value rounded up to a multiple of the scale factor.		
__u32	*sizeimage*	Size in bytes of the buffer to hold a complete image, set by the driver. Usually this is *bytesperline* times *height*. When the image consists of variable length compressed data this is the maximum number of bytes required to hold an image.

		this is the maximum number of bytes required to hold an image.
enum v4l2_colorspace	*colorspace*	This information supplements the *pixelformat* and must be set by the driver, see the section called “Colorspaces”.
__u32	*priv*	Reserved for custom (driver defined) additional information about formats. When not used drivers and applications must set this field to zero.

Standard Image Formats

In order to exchange images between drivers and applications, it is necessary to have standard image data formats which both sides will interpret the same way. V4L2 includes several such formats, and this section is intended to be an unambiguous specification of the standard image data formats in V4L2.

V4L2 drivers are not limited to these formats, however. Driver-specific formats are possible. In that case the application may depend on a codec to convert images to one of the standard formats when needed. But the data can still be stored and retrieved in the proprietary format. For example, a device may support a proprietary compressed format. Applications can still capture and save the data in the compressed format, saving much disk space, and later use a codec to convert the images to the X Windows screen format when the video is to be displayed.

Even so, ultimately, some standard formats are needed, so the V4L2 specification would not be complete without well-defined standard formats.

The V4L2 standard formats are mainly uncompressed formats. The pixels are always arranged in memory from left to right, and from top to bottom. The first byte of data in the image buffer is always for the leftmost pixel of the topmost row. Following that is the pixel immediately to its right, and so on until the end of the top row of pixels. Following the rightmost pixel of the row there may be zero or more bytes of padding to guarantee that each row of pixel data has a certain alignment. Following the pad bytes, if any, is data for the leftmost pixel of the second row from the top, and so on. The last row has just as many pad bytes after it as the other rows.

In V4L2 each format has an identifier which looks like `PIX_FMT_XXX`, defined in the videodev.h header file. These identifiers represent four character codes which are also listed below, however they are not the same as those used in the Windows world.

Colorspaces

[intro]

Gamma Correction

[to do]

$E'_R = f(R)$

$E'_G = f(G)$

$E'_B = f(B)$

Construction of luminance and color-difference signals

[to do]

$E'_Y = Coeff_R\, E'_R + Coeff_G\, E'_G + Coeff_B\, E'_B$

$(E'_R - E'_Y) = E'_R - Coeff_R\, E'_R - Coeff_G\, E'_G - Coeff_B\, E'_B$

$(E'_B - E'_Y) = E'_B - Coeff_R\, E'_R - Coeff_G\, E'_G - Coeff_B\, E'_B$

Re-normalized color-difference signals

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

$K_B = 0.5 / (1 - Coeff_B)$

$K_R = 0.5 / (1 - Coeff_R)$

$P_B = K_B (E'_B - E'_Y) = 0.5 (Coeff_R / Coeff_B) E'_R + 0.5 (Coeff_G / Coeff_B) E'_G + 0.5 E'_B$

$P_R = K_R (E'_R - E'_Y) = 0.5 E'_R + 0.5 (Coeff_G / Coeff_R) E'_G + 0.5 (Coeff_B / Coeff_R) E'_B$

Quantization

[to do]

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

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

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

Rounding to the nearest integer and clamping to the range [0;255] finally yields the digital color components Y'CbCr stored in YUV images.

Example 2.1. ITU-R Rec. BT.601 color conversion

Forward Transformation

```
int ER, EG, EB;         /* gamma corrected RGB input [0;255] */
int Y1, Cb, Cr;         /* output [0;255] */

double r, g, b;         /* temporaries */
double y1, pb, pr;

int
clamp (double x)
{
        int r = x;      /* round to nearest */

        if (r < 0)         return 0;
        else if (r > 255)  return 255;
        else               return r;
}

r = ER / 255.0;
g = EG / 255.0;
b = EB / 255.0;

y1  =  0.299  * r + 0.587 * g + 0.114  * b;
pb  = -0.169  * r - 0.331 * g + 0.5    * b;
pr  =  0.5    * r - 0.419 * g - 0.081  * b;

Y1 = clamp (219 * y1 + 16);
Cb = clamp (224 * pb + 128);
Cr = clamp (224 * pr + 128);

/* or shorter */

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

Y1 = clamp ( (219 / 255.0)                    *       y1  + 16);
Cb = clamp (((224 / 255.0) / (2 - 2 * 0.114)) * (EB - y1) + 128);
Cr = clamp (((224 / 255.0) / (2 - 2 * 0.299)) * (ER - y1) + 128);
```

Inverse Transformation

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

double r, g, b;         /* temporaries */
double y1, pb, pr;

int
clamp (double x)
{
        int r = x;      /* round to nearest */

        if (r < 0)         return 0;
        else if (r > 255)  return 255;
```

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

y1 = (255 / 219.0) * (Y1 - 16);
pb = (255 / 224.0) * (Cb - 128);
pr = (255 / 224.0) * (Cr - 128);

r = 1.0 * y1 + 0     * pb + 1.402 * pr;
g = 1.0 * y1 - 0.344 * pb - 0.714 * pr;
b = 1.0 * y1 + 1.772 * pb + 0     * pr;

ER = clamp (r * 255); /* [ok? one should prob. limit y1,pb,pr] */
EG = clamp (g * 255);
EB = clamp (b * 255);
```

Table 2.2. enum v4l2_colorspace

Identifier	Value	Description	Chromaticities[a]			White Point	Gamma Correction	Lu
			Red	Green	Blue			
V4L2_COLORSPACE_SMPTE170M	1	NTSC/PAL according to [SMPTE 170M], [ITU BT.601]	x = 0.630, y = 0.340	x = 0.310, y = 0.595	x = 0.155, y = 0.070	x = 0.3127, y = 0.3290, Illuminant D_{65}	E' = 4.5 I for I ≤0.018, 1.099 $I^{0.45}$ - 0.099 for 0.018 < I	0. + 0 + 0
V4L2_COLORSPACE_SMPTE240M	2	1125-Line (US) HDTV, see [SMPTE 240M]	x = 0.630, y = 0.340	x = 0.310, y = 0.595	x = 0.155, y = 0.070	x = 0.3127, y = 0.3290, Illuminant D_{65}	E' = 4 I for I ≤0.0228, 1.1115 $I^{0.45}$ - 0.1115 for 0.0228 < I	0. + 0 + 0
V4L2_COLORSPACE_REC709	3	HDTV and modern devices, see [ITU BT.709]	x = 0.640, y = 0.330	x = 0.300, y = 0.600	x = 0.150, y = 0.060	x = 0.3127, y = 0.3290, Illuminant D_{65}	E' = 4.5 I for I ≤0.018, 1.099 $I^{0.45}$ - 0.099 for 0.018 < I	0.2 + 0. + 0.
V4L2_COLORSPACE_BT878	4	Broken Bt878 extents[b], [ITU BT.601]	?	?	?	?	?	0. + 0 + 0
V4L2_COLORSPACE_470_SYSTEM_M	5	M/NTSC[c] according to [ITU BT.470], [ITU BT.601]	x = 0.67, y = 0.33	x = 0.21, y = 0.71	x = 0.14, y = 0.08	x = 0.310, y = 0.316, Illuminant C	?	0. + 0 + 0
V4L2_COLORSPACE_470_SYSTEM_BG	6	625-line PAL and SECAM systems according to [ITU BT.470], [ITU BT.601]	x = 0.64, y = 0.33	x = 0.29, y = 0.60	x = 0.15, y = 0.06	x = 0.313, y = 0.329, Illuminant D_{65}	?	0. + 0 + 0
V4L2_COLORSPACE_JPEG	7	JPEG Y'CbCr, see [JFIF], [ITU BT.601]	?	?	?	?	?	0. + 0 + 0
V4L2_COLORSPACE_SRGB	8	[?]	x = 0.640, y = 0.330	x = 0.300, y = 0.600	x = 0.150, y = 0.060	x = 0.3127, y = 0.3290, Illuminant D_{65}	E' = 4.5 I for I ≤0.018, 1.099 $I^{0.45}$ - 0.099 for 0.018 < I	

[a] The coordinates of the color primaries are given in the CIE system (1931)

[b] The ubiquitous Bt878 video capture chip quantizes E'_Y to 238 levels, yielding a range of Y' = 16 … 253, unlike Rec. 601 Y' = 16 ..
documentation, it has been implemented in silicon. The chroma extents are unclear.

[c] No identifier exists for M/PAL which uses the chromaticities of M/NTSC, the remaining parameters are equal to B and G/PAL.

[d] Note JFIF quantizes $Y'P_BP_R$ in range [0;+1] and [-0.5;+0.5] to *257* levels, however Y'CbCr signals are still clamped to [0;255].

Indexed Format

In this format each pixel is represented by an 8 bit index into a 256 entry ARGB palette. It is intended for Video Output Overlays only. There are no ioctls to access the palette, this must be done with ioctls of the Linux framebuffer API.

Table 2.3. Indexed Image Format

Identifier	Code	Byte 0							
	Bit	7	6	5	4	3	2	1	0
`V4L2_PIX_FMT_PAL8`	'PAL8'	i_7	i_6	i_5	i_4	i_3	i_2	i_1	i_0

RGB Formats

Name

Packed RGB formats — Packed RGB formats

Description

These formats are designed to match the pixel formats of typical PC graphics frame buffers. They occupy 8, 16, 24 or 32 bits per pixel. These are all packed-pixel formats, meaning all the data for a pixel lie next to each other in memory.

When one of these formats is used, drivers shall report the colorspace `V4L2_COLORSPACE_SRGB`.

Table 2.4. Packed RGB Image Formats

Identifier	Code	Byte 0 in memory								Byte 1								Byte 2								Byte 3							
	Bit	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
`V4L2_PIX_FMT_RGB332`	'RGB1'	b_1	b_0	g_2	g_1	g_0	r_2	r_1	r_0																								
`V4L2_PIX_FMT_RGB444`	'R444'	g_3	g_2	g_1	g_0	b_3	b_2	b_1	b_0	a_3	a_2	a_1	a_0	r_3	r_2	r_1	r_0																
`V4L2_PIX_FMT_RGB555`	'RGBO'	g_2	g_1	g_0	r_4	r_3	r_2	r_1	r_0	a	b_4	b_3	b_2	b_1	b_0	g_4	g_3																
`V4L2_PIX_FMT_RGB565`	'RGBP'	g_2	g_1	g_0	r_4	r_3	r_2	r_1	r_0	b_4	b_3	b_2	b_1	b_0	g_5	g_4	g_3																
`V4L2_PIX_FMT_RGB555X`	'RGBQ'	a	b_4	b_3	b_2	b_1	b_0	g_4	g_3	g_2	g_1	g_0	r_4	r_3	r_2	r_1	r_0																
`V4L2_PIX_FMT_RGB565X`	'RGBR'	b_4	b_3	b_2	b_1	b_0	g_5	g_4	g_3	g_2	g_1	g_0	r_4	r_3	r_2	r_1	r_0																
`V4L2_PIX_FMT_BGR24`	'BGR3'	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0	g_7	g_6	g_5	g_4	g_3	g_2	g_1	g_0	r_7	r_6	r_5	r_4	r_3	r_2	r_1	r_0								
`V4L2_PIX_FMT_RGB24`	'RGB3'	r_7	r_6	r_5	r_4	r_3	r_2	r_1	r_0	g_7	g_6	g_5	g_4	g_3	g_2	g_1	g_0	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0								
`V4L2_PIX_FMT_BGR32`	'BGR4'	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0	g_7	g_6	g_5	g_4	g_3	g_2	g_1	g_0	r_7	r_6	r_5	r_4	r_3	r_2	r_1	r_0	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0
`V4L2_PIX_FMT_RGB32`	'RGB4'	r_7	r_6	r_5	r_4	r_3	r_2	r_1	r_0	g_7	g_6	g_5	g_4	g_3	g_2	g_1	g_0	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0

Bit 7 is the most significant bit. The value of a = alpha bits is undefined when reading from the driver, ignored when writing to the driver, except when alpha blending has been negotiated for a Video Overlay or Video Output Overlay.

Example 2.2. `V4L2_PIX_FMT_BGR24` 4 × 4 pixel image

Byte Order. Each cell is one byte.

start + 0:	B_{00}	G_{00}	R_{00}	B_{01}	G_{01}	R_{01}	B_{02}	G_{02}	R_{02}	B_{03}	G_{03}	R_{03}
start + 12:	B_{10}	G_{10}	R_{10}	B_{11}	G_{11}	R_{11}	B_{12}	G_{12}	R_{12}	B_{13}	G_{13}	R_{13}

start + 24: B_{20} G_{20} R_{20} B_{21} G_{21} R_{21} B_{22} G_{22} R_{22} B_{23} G_{23} R_{23}

start + 36: B_{30} G_{30} R_{30} B_{31} G_{31} R_{31} B_{32} G_{32} R_{32} B_{33} G_{33} R_{33}

Important

Drivers may interpret these formats differently.

Some RGB formats above are uncommon and were probably defined in error. Drivers may interpret them as in Table 2.5, "Packed RGB Image Formats (corrected)".

Table 2.5. Packed RGB Image Formats (corrected)

Identifier	Code	Byte 0 in memory								Byte 1								Byte 2								Byte 3							
	Bit	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
`V4L2_PIX_FMT_RGB332`	'RGB1'	r_2	r_1	r_0	g_2	g_1	g_0	b_1	b_0																								
`V4L2_PIX_FMT_RGB444`	'R444'	g_3	g_2	g_1	g_0	b_3	b_2	b_1	b_0	a_3	a_2	a_1	a_0	r_3	r_2	r_1	r_0																
`V4L2_PIX_FMT_RGB555`	'RGBO'	g_2	g_1	g_0	b_4	b_3	b_2	b_1	b_0	a	r_4	r_3	r_2	r_1	r_0	g_4	g_3																
`V4L2_PIX_FMT_RGB565`	'RGBP'	g_2	g_1	g_0	b_4	b_3	b_2	b_1	b_0	r_4	r_3	r_2	r_1	r_0	g_5	g_4	g_3																
`V4L2_PIX_FMT_RGB555X`	'RGBQ'	a	r_4	r_3	r_2	r_1	r_0	g_4	g_3	g_2	g_1	g_0	b_4	b_3	b_2	b_1	b_0																
`V4L2_PIX_FMT_RGB565X`	'RGBR'	r_4	r_3	r_2	r_1	r_0	g_5	g_4	g_3	g_2	g_1	g_0	b_4	b_3	b_2	b_1	b_0																
`V4L2_PIX_FMT_BGR24`	'BGR3'	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0	g_7	g_6	g_5	g_4	g_3	g_2	g_1	g_0	r_7	r_6	r_5	r_4	r_3	r_2	r_1	r_0								
`V4L2_PIX_FMT_RGB24`	'RGB3'	r_7	r_6	r_5	r_4	r_3	r_2	r_1	r_0	g_7	g_6	g_5	g_4	g_3	g_2	g_1	g_0	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0								
`V4L2_PIX_FMT_BGR32`	'BGR4'	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0	g_7	g_6	g_5	g_4	g_3	g_2	g_1	g_0	r_7	r_6	r_5	r_4	r_3	r_2	r_1	r_0	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0
`V4L2_PIX_FMT_RGB32`	'RGB4'	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0	r_7	r_6	r_5	r_4	r_3	r_2	r_1	r_0	g_7	g_6	g_5	g_4	g_3	g_2	g_1	g_0	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0

A test utility to determine which RGB formats a driver actually supports is available from the LinuxTV v4l-dvb repository. See http://linuxtv.org/repo/ for access instructions.

Name

`V4L2_PIX_FMT_SBGGR8` — Bayer RGB format

Description

This is commonly the native format of digital cameras, reflecting the arrangement of sensors on the CCD device. Only one red, green or blue value is given for each pixel. Missing components must be interpolated from neighbouring pixels. From left to right the first row consists of a blue and green value, the second row of a green and red value. This scheme repeats to the right and down for every two columns and rows.

Example 2.3. `V4L2_PIX_FMT_SBGGR8` 4 × 4 pixel image

Byte Order. Each cell is one byte.

start + 0: B_{00} G_{01} B_{02} G_{03}

start + 4: G_{10} R_{11} G_{12} R_{13}

start + 8: B_{20} G_{21} B_{22} G_{23}

start + 12: G_{30} R_{31} G_{32} R_{33}

Name

`V4L2_PIX_FMT_SGBRG8` — Bayer RGB format

Description

This is commonly the native format of digital cameras, reflecting the arrangement of sensors on the CCD device. Only one red, green or blue value is given for each pixel. Missing components must be interpolated from neighbouring pixels. From left to right the first row consists of a green and blue value, the second row of a red and green value. This scheme repeats to the right and down for every two columns and rows.

Example 2.4. `V4L2_PIX_FMT_SGBRG8` 4 × 4 pixel image

Byte Order. Each cell is one byte.