# LINUX MEDIA INFRASTRUCTURE API

# LINUX MEDIA INFRASTRUCTURE API

# Table of Contents

# List of Tables

# 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 documment support also for DVB-S2, ISDB-T and ISDB-S.

The third part covers Remote Controller API

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). (http://vger.kernel.org/vger-lists.html#linux-media).

# I. Video for Linux Two API Specification

## Revision 2.6.39

## Table of Contents

Michael H Schimek, Bill Dirks, Hans Verkuil, Martin Rubli, Andy Walls, Mauro Carvalho Chehab, Muralidharan Karicheri, and Pawel Osciak



## Revision History

Revision 2.6.39 2011-03-01 Revised by: mcc, po
Removed VIDIOC_*_OLD from videodev2.h header and update it to reflect latest changes. Added
Revision 2.6.37 2010-08-06 Revised by: hv
Removed obsolete vtx (videotext) API.
Revision 2.6.33 2009-12-03 Revised by: mk
Added documentation for the Digital Video timings API.
Revision 2.6.32 2009-08-31 Revised by: mcc
Now, revisions will match the kernel version where the V4L2 API changes will be used by the Linu
Revision 0.29 2009-08-26 Revised by: ev
Added documentation for string controls and for FM Transmitter controls.
Revision 0.28 2009-08-26 Revised by: gl
Added V4L2_CID_BAND_STOP_FILTER documentation.
Revision 0.27 2009-08-15 Revised by: mcc
Added libv4l and Remote Controller documentation; added v4l2grab and keytable application exan
Revision 0.26 2009-07-23 Revised by: hv
Finalized the RDS capture API. Added modulator and RDS encoder capabilities. Added support fo
Revision 0.25 2009-01-18 Revised by: hv
Added pixel formats VYUY, NV16 and NV61, and changed the debug ioctls VIDIOC_DBG_G/S_
Revision 0.24 2008-03-04 Revised by: mhs
Added pixel formats Y16 and SBGGR16, new controls and a camera controls class. Removed VID
Revision 0.23 2007-08-30 Revised by: mhs
Fixed a typo in VIDIOC_DBG_G/S_REGISTER. Clarified the byte order of packed pixel formats.
Revision 0.22 2007-08-29 Revised by: mhs
Added the Video Output Overlay interface, new MPEG controls, V4L2_FIELD_INTERLACED_T
Revision 0.21 2006-12-19 Revised by: mhs
Fixed a link in the VIDIOC_G_EXT_CTRLS section.
Revision 0.20 2006-11-24 Revised by: mhs
Clarified the purpose of the audioset field in struct v4l2_input and v4l2_output.
Revision 0.19 2006-10-19 Revised by: mhs
Documented V4L2_PIX_FMT_RGB444.
Revision 0.18 2006-10-18 Revised by: mhs
Added the description of extended controls by Hans Verkuil. Linked V4L2_PIX_FMT_MPEG to V
Revision 0.17 2006-10-12 Revised by: mhs
Corrected V4L2_PIX_FMT_HM12 description.
Revision 0.16 2006-10-08 Revised by: mhs

VIDIOC_ENUM_FRAMESIZES and VIDIOC_ENUM_FRAMEINTERVALS are now part of the
Revision 0.15 2006-09-23 Revised by: mhs
Cleaned up the bibliography, added BT.653 and BT.1119. capture.c/start_capturing() for user pointe
Revision 0.14 2006-09-14 Revised by: mr
Added VIDIOC_ENUM_FRAMESIZES and VIDIOC_ENUM_FRAMEINTERVALS proposal for
Revision 0.13 2006-04-07 Revised by: mhs
Corrected the description of struct v4l2_window clips. New V4L2_STD_ and V4L2_TUNER_MO
Revision 0.12 2006-02-03 Revised by: mhs
Corrected the description of struct v4l2_captureparm and v4l2_outputparm.
Revision 0.11 2006-01-27 Revised by: mhs
Improved the description of struct v4l2_tuner.
Revision 0.10 2006-01-10 Revised by: mhs
VIDIOC_G_INPUT and VIDIOC_S_PARM clarifications.
Revision 0.9 2005-11-27 Revised by: mhs
Improved the 525 line numbering diagram. Hans Verkuil and I rewrote the sliced VBI section. He a
Revision 0.8 2004-10-04 Revised by: mhs
Somehow a piece of junk slipped into the capture example, removed.
Revision 0.7 2004-09-19 Revised by: mhs
Fixed video standard selection, control enumeration, downscaling and aspect example. Added read
Revision 0.6 2004-08-01 Revised by: mhs
v4l2_buffer changes, added video capture example, various corrections.
Revision 0.5 2003-11-05 Revised by: mhs
Pixel format erratum.
Revision 0.4 2003-09-17 Revised by: mhs
Corrected source and Makefile to generate a PDF. SGML fixes. Added latest API changes. Closed g
Revision 0.3 2003-02-05 Revised by: mhs
Another draft, more corrections.
Revision 0.2 2003-01-15 Revised by: mhs
Second draft, with corrections pointed out by Gerd Knorr.
Revision 0.1 2002-12-01 Revised by: mhs
First draft, based on documentation by Bill Dirks and discussions on the V4L mailing list.

# Chapter 1. Common API Elements

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. In this chapter we will discuss the basic concepts applicable to all devices.

# 1.1. Opening and Closing Devices

## 1.1.1. 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                ❶
options mydriver video_nr=0,1 radio_nr=0,1     ❷
```

❶ 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).

❷ 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 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 recommended for the different V4L2 device types are listed in Chapter 4.

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.

## 1.1.2. 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.

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.

### 1.1.3. 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 Section 1.3.

### 1.1.4. 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.

### 1.1.5. 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.

## 1.2. 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.

## 1.3. 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.

## 1.4. 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);
}
```

# 1.5. 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);
}
```

# 1.6. Tuners and Modulators

## 1.6.1. 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.

### 1.6.2. 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.

### 1.6.3. 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.

## 1.7. 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, 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);
}
```

### 1.7.1. Digital Video (DV) Timings

The video standards discussed so far has been dealing with Analog TV and the corresponding video timings. Today there are many more different hardware interfaces such as High Definition TV interfaces (HDMI), VGA, DVI connectors etc., that carry video signals and there is a need to extend the API to select the video timings for these interfaces. Since it is not possible to extend the v4l2_std_id due to the limited bits available, a new set of IOCTLs is added to set/get video timings at the input and output:

- DV Presets: Digital Video (DV) presets. These are IDs representing a video timing at the input/output. Presets are pre-defined timings implemented by the hardware according to video standards. A __u32 data type is used to represent a preset unlike the bit mask that is used in v4l2_std_id allowing future extensions to support as many different presets as needed.
- Custom DV Timings: This will allow applications to define more detailed custom video timings for the interface. This includes parameters such as width, height, polarities, frontporch, backporch etc.

To enumerate and query the attributes of DV presets supported by a device, applications use the `VIDIOC_ENUM_DV_PRESETS` ioctl. To get the current DV preset, applications use the `VIDIOC_G_DV_PRESET` ioctl and to set a preset they use the `VIDIOC_S_DV_PRESET` ioctl.

To set custom DV timings for the device, applications use the `VIDIOC_S_DV_TIMINGS` ioctl and to get current custom DV timings they use the `VIDIOC_G_DV_TIMINGS` ioctl.

Applications can make use of the Table A-4 and Table A-3 flags to decide what ioctls are available to set the video timings for the device.

## 1.8. 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. 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. |

| ID | Type | Description |
|---|---|---|
| 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. |

| ID | Type | Description |
|---|---|---|
| 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 Section 1.12, for clipping Section 4.2. 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 (http://en.wikipedia.org/wiki/Color_temperature). |
| V4L2_CID_SHARPNESS | integer | Adjusts the sharpness filters in a camera. The minimum value disables the filters, higher values give a sharper picture. |

| ID | Type | Description |
|---|---|---|
| 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_CHROMA_GAIN | integer | Adjusts the Chroma gain control (for use when chroma AGC is disabled). |
| 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), V4L2_COLORFX_SEPIA (2), V4L2_COLORFX_NEGATIVE (3), V4L2_COLORFX_EMBOSS (4), V4L2_COLORFX_SKETCH (5), V4L2_COLORFX_SKY_BLUE (6), V4L2_COLORFX_GRASS_GREEN (7), V4L2_COLORFX_SKIN_WHITEN (8) and V4L2_COLORFX_VIVID (9). |
| V4L2_CID_ROTATE | integer | Rotates the image by specified angle. Common angles are 90, 270 and 180. Rotating the image to 90 and 270 will reverse the height and width of the display window. It is necessary to set the new height and width of the picture using the VIDIOC_S_FMT ioctl according to the rotation angle selected. |
| V4L2_CID_BG_COLOR | integer | Sets the background color on the current output device. Background color needs to be specified in the RGB24 format. The supplied 32 bit value is interpreted as bits 0-7 Red color information, bits 8-15 Green color information, bits 16-23 Blue color information and bits 24-31 must be zero. |
| V4L2_CID_ILLUMINATORS_1 V4L2_CID_ILLUMINATORS_2 | boolean | Switch on or off the illuminator 1 or 2 of the device (usually a microscope). |

| ID | Type | Description |
|---|---|---|
| V4L2_CID_LASTP1 | | End of the predefined control IDs (currently V4L2_CID_ILLUMINATORS_2 + 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 Section 1.2. |

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);
  }
 }
}

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;

  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);
```

# 1.9. Extended Controls

## 1.9.1. 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 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.

## 1.9.2. 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.

## 1.9.3. 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 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.

## 1.9.4. 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.

## 1.9.5. MPEG Control Reference

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

### 1.9.5.1. 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. | |

**ID** **Type**

**Description**

`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: **ENTRYTBL not supported.**

`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:

**ID** **Type**

**Description**

**ENTRYTBL not supported.**

V4L2_CID_MPEG_AUDIO_SAMPLING_FREQ enum v4l2_mpeg_audio_sampling_freq

MPEG Audio sampling frequency. Possible values are:

**ENTRYTBL not supported.**

V4L2_CID_MPEG_AUDIO_ENCODING enum v4l2_mpeg_audio_encoding

MPEG Audio encoding. Possible values are:

**ENTRYTBL not supported.**

V4L2_CID_MPEG_AUDIO_L1_BITRATE enum v4l2_mpeg_audio_l1_bitrate

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

**ENTRYTBL not supported.**

V4L2_CID_MPEG_AUDIO_L2_BITRATE enum v4l2_mpeg_audio_l2_bitrate

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

**ENTRYTBL not supported.**

V4L2_CID_MPEG_AUDIO_L3_BITRATE enum v4l2_mpeg_audio_l3_bitrate

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

**ENTRYTBL not supported.**

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:

**ENTRYTBL not supported.**

V4L2_CID_MPEG_AUDIO_MODE enum v4l2_mpeg_audio_mode

**ID** **Type**

**Description**

MPEG Audio mode. Possible values are:
**ENTRYTBL not supported.**

`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:
**ENTRYTBL not supported.**

`V4L2_CID_MPEG_AUDIO_EMPHASIS` enum v4l2_mpeg_audio_emphasis

Audio Emphasis. Possible values are:
**ENTRYTBL not supported.**

`V4L2_CID_MPEG_AUDIO_CRC` enum v4l2_mpeg_audio_crc

CRC method. Possible values are:
**ENTRYTBL not supported.**

`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 reproducible audio bitstream. 0 = unmuted, 1 = muted.

`V4L2_CID_MPEG_VIDEO_ENCODING` enum v4l2_mpeg_video_encoding

MPEG Video encoding method. Possible values are:
**ENTRYTBL not supported.**

`V4L2_CID_MPEG_VIDEO_ASPECT` enum v4l2_mpeg_video_aspect

Video aspect. Possible values are:
**ENTRYTBL not supported.**

`V4L2_CID_MPEG_VIDEO_B_FRAMES` integer

Number of B-Frames (default 2)

| ID | Type | Description |
|---|---|---|
| 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:<br>**ENTRYTBL not supported.** |
| 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): |

| ID | Type |
|---|---|
| Description | |
| **ENTRYTBL not supported.** | |

### 1.9.5.2. 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:<br>**ENTRYTBL not supported.** | |
| `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:<br>**ENTRYTBL not supported.** | |
| `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:<br>**ENTRYTBL not supported.** | |
| `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:<br>**ENTRYTBL not supported.** | |

| ID | Type |
|---|---|
| Description | |
| 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:<br>**ENTRYTBL not supported.** | |
| 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. | |

## 1.9.6. 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: **ENTRYTBL not supported.** |
| 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 textmus 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 | |

**ID** **Type**

**Description**

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.

| ID | Type |
|---|---|
| Description | |
| 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_IRIS_ABSOLUTE | integer |
| This control sets the camera's aperture to the specified value. The unit is undefined. Larger values open the iris wider, smaller values close it. | |
| V4L2_CID_IRIS_RELATIVE | integer |
| This control modifies the camera's aperture by the specified amount. The unit is undefined. Positive values open the iris one step further, negative values close it one step further. This is a write-only control. | |
| 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. | |

## 1.9.7. 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 |
| 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 |

**ID** **Type**

**Description**

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_ENABLE 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_ENABLE 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

**ID** **Type**

**Description**

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:

**ENTRYTBL not supported.**

`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 step are driver-specific.

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

# 1.10. Data Formats

## 1.10.1. 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. For a closer look at image formats see Chapter 2.

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.

### 1.10.2. 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.

## 1.11. Single- and multi-planar APIs

Some devices require data for each input or output video frame to be placed in discontiguous memory buffers. In such cases, one video frame has to be addressed using more than one memory address, i.e. one pointer per "plane". A plane is a sub-buffer of the current frame. For examples of such formats see Chapter 2.

Initially, V4L2 API did not support multi-planar buffers and a set of extensions has been introduced to handle them. Those extensions constitute what is being referred to as the "multi-planar API".

Some of the V4L2 API calls and structures are interpreted differently, depending on whether single- or multi-planar API is being used. An application can choose whether to use one or the other by passing a corresponding buffer type to its ioctl calls. Multi-planar versions of buffer types are suffixed with an '_MPLANE' string. For a list of available multi-planar buffer types see enum v4l2_buf_type.

### 1.11.1. Multi-planar formats

Multi-planar API introduces new multi-planar formats. Those formats use a separate set of FourCC codes. It is important to distinguish between the multi-planar API and a multi-planar format. Multi-planar API calls can handle all

single-planar formats as well (as long as they are passed in multi-planar API structures), while the single-planar API cannot handle multi-planar formats.

### 1.11.2. Calls that distinguish between single and multi-planar APIs

`VIDIOC_QUERYCAP`

Two additional multi-planar capabilities are added. They can be set together with non-multi-planar ones for devices that handle both single- and multi-planar formats.

`VIDIOC_G_FMT`, `VIDIOC_S_FMT`, `VIDIOC_TRY_FMT`

New structures for describing multi-planar formats are added: struct v4l2_pix_format_mplane and struct v4l2_plane_pix_format. Drivers may define new multi-planar formats, which have distinct FourCC codes from the existing single-planar ones.

`VIDIOC_QBUF`, `VIDIOC_DQBUF`, `VIDIOC_QUERYBUF`

A new struct v4l2_plane structure for describing planes is added. Arrays of this structure are passed in the new `m.planes` field of struct v4l2_buffer.

`VIDIOC_REQBUFS`

Will allocate multi-planar buffers as requested.

## 1.12. 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.*

## 1.12.1. 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). Vertically ITU-R line numbers of the first field (Figure 4-2, Figure 4-3), 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.

## 1.12.2. 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.

## 1.12.3. 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;

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;
```

## 1.13. 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 unnecessary 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.

## Notes

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 Section 3.5 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

The V4L2 API was primarily designed for devices exchanging image data with applications. The v4l2_pix_format and v4l2_pix_format_mplane structures define the format and layout of an image in memory. The former is used with the single-planar API, while the latter is used with the multi-planar version (see Section 1.11). 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.)

## 2.1. Single-planar format structure

**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 2-1, YUV formats in Section 2.7, and reserved codes in Table 2-10 |
| 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 Section 3.6. |
| __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. |
| enum v4l2_colorspace | *colorspace* | This information supplements the *pixelformat* and must be set by the driver, see Section 2.4. |
| __u32 | *priv* | Reserved for custom (driver defined) additional information about formats. When not used drivers and applications must set this field to zero. |

# 2.2. Multi-planar format structures

The v4l2_plane_pix_format structures define size and layout for each of the planes in a multi-planar format. The v4l2_pix_format_mplane structure contains information common to all planes (such as image width and height) and an array of v4l2_plane_pix_format structures, describing all planes of that format.

**Table 2-2. struct vl42_plane_pix_format**

| | | |
|---|---|---|
| __u32 | *sizeimage* | Maximum size in bytes required for image data in this plane. |
| __u16 | *bytesperline* | Distance in bytes between the leftmost pixels in two adjacent lines. |
| __u16 | *reserved[7]* | Reserved for future extensions. Should be zeroed by the application. |

**Table 2-3. struct v4l2_pix_format_mplane**

| | | |
|---|---|---|
| __u32 | *width* | Image width in pixels. |
| __u32 | *height* | Image height in pixels. |
| __u32 | *pixelformat* | The pixel format. Both single- and multi-planar four character codes can be used. |
| enum v4l2_field | *field* | See struct v4l2_pix_format. |
| enum v4l2_colorspace | *colorspace* | See struct v4l2_pix_format. |
| struct v4l2_plane_pix_format | *plane_fmt[VIDEO_MAX_PLANES]* | An array of structures describing format of each plane this pixel format consists of. The number of valid entries in this array has to be put in the *num_planes* field. |
| __u8 | *num_planes* | Number of planes (i.e. separate memory buffers) for this format and the number of valid entries in the *plane_fmt* array. |
| __u8 | *reserved[11]* | Reserved for future extensions. Should be zeroed by the application. |

# 2.3. 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 (FourCC) codes which are also listed below, however they are not the same as those used in the Windows world.

For some formats, data is stored in separate, discontiguous memory buffers. Those formats are identified by a separate set of FourCC codes and are referred to as "multi-planar formats". For example, a YUV422 frame is normally stored in one memory buffer, but it can also be placed in two or three separate buffers, with Y component in one buffer and CbCr components in another in the 2-planar version or with each component in its own buffer in the 3-planar case. Those sub-buffers are referred to as "planes".

# 2.4. 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]

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

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

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

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                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 2-4. enum v4l2_colorspace**

| Identifier | Value | Description | Chromaticities[a] | | | White Point | Gamma Correction | Luminance $E'_Y$ | Quantization | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | Red | Green | Blue | | | | $Y'$ | $C_b$, $C_r$ |
| 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 $I \leq 0.018$, $E' = 1.099 I^{0.45} - 0.099$ for $0.018 < I$ | $E'_Y = 0.299 E'_R + 0.587 E'_G + 0.114 E'_B$ | $Y' = 219 E'_Y + 16$ | $C_B = 224 E'_B + 128$, $C_R = 224 E'_R + 128$ |
| 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 $I \leq 0.0228$, $E' = 1.1115 I^{0.45} - 0.1115$ for $0.0228 < I$ | $E'_Y = 0.212 E'_R + 0.701 E'_G + 0.087 E'_B$ | $Y' = 219 E'_Y + 16$ | $C_B = 224 E'_B + 128$, $C_R = 224 E'_R + 128$ |
| 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 $I \leq 0.018$, $E' = 1.099 I^{0.45} - 0.099$ for $0.018 < I$ | $E'_Y = 0.2125 E'_R + 0.7154 E'_G + 0.0721 E'_B$ | $Y' = 219 E'_Y + 16$ | $C_B = 224 E'_B + 128$, $C_R = 224 E'_R + 128$ |
| V4L2_COLORSPACE_BT878 | 4 | Broken Bt878 extents[b], ITU BT.601 | ? | ? | ? | ? | ? | $E'_Y = 0.299 E'_R + 0.587 E'_G + 0.114 E'_B$ | $Y' = 237 E'_Y + 16$ | $C_B = 224 E'_B + 128$, $C_R = 224 E'_R + 128$ (probably) |

| Identifier | Value | Description | Chromaticities[a] | | | White Point | Gamma Correction | Luminance $E'_Y$ | Quantization | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | Red | Green | Blue | | | | $Y'$ | $C_b$, $C_r$ |
| V4L2_COLORSPACE_SMPTE170M | 5 | NTSC/M, ac-cording 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, Illu-mi-nant C | ? | $0.299\ E'_R + 0.587\ E'_G + 0.114\ E'_B$ | $219\ E'_Y + 16$ | $224\ E'_{B,R} + 128$ |
| V4L2_COLORSPACE_470_SYSTEM_BG | 6 | 625-line PAL and SE-CAM sys-tems ac-cord-ing 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, Illu-mi-nant $D_{65}$ | ? | $0.299\ E'_R + 0.587\ E'_G + 0.114\ E'_B$ | $219\ E'_Y + 16$ | $224\ E'_{B,R} + 128$ |
| V4L2_COLORSPACE_JPEG | 7 | JPEG Y'CbCr, see JFIF, ITU BT.601 | ? | ? | ? | ? | ? | $0.299\ E'_R + 0.587\ E'_G + 0.114\ E'_B$ | $256\ E'_Y + 0$ | $256\ E'_{B,R} + 128$ |
| V4L2_COLORSPACE_SRGB | 8 | [2] | x = 0.640, y = 0.330 | x = 0.300, y = 0.600 | x = 0.150, y = 0.060 | x = 0.3127, y = 0.3290, Illu-mi-nant $D_{65}$ | $E' = 4.5\ I$ for $I \leq 0.018$, $1.099\ I^{0.45} - 0.099$ for $0.018 < I$ | n/a | | |

| Identifier | Value | Description | Chromaticities[a] | | | White Point | Gamma Correction | Luminance E'$_Y$ | Quantization | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | Red | Green | Blue | | | | Y' | Cb, Cr |
| Notes:<br>a. The coordinates of the color primaries are given in the CIE system (1931)<br>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 ... 235. This is not a typo in the Bt878 documentation, it has been implemented in silicon. The chroma extents are unclear.<br>c. No identifier exists for M/PAL which uses the chromaticities of M/NTSC, the remaining parameters are equal to B and G/PAL.<br>d. Note JFIF quantizes Y'P$_B$P$_R$ in range [0;+1] and [-0.5;+0.5] to *257* levels, however Y'CbCr signals are still clamped to [0;255]. | | | | | | | | | | | |

# 2.5. 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 2-5. Indexed Image Format**

| Identifier | Byte 0 |
|---|---|
| | Bit 7 6 5 4 3 2 1 0 |
| V4L2_PIX_FMT_PAL8 | i$_7$ i$_6$ i$_5$ i$_4$ i$_3$ i$_2$ i$_1$ i$_0$ |
| 'PAL8' | |

# 2.6. RGB Formats

# Packed 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 2-1. Packed RGB Image Formats**

| Identifier | 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' | b1 b0 g2 g1 g0 r2 r1 r0 | | | |
| `V4L2_PIX_FMT_RGB444` | | | | |
| 'R444' | g3 g2 g1 g0 b3 b2 b1 b0 | a3 a2 a1 a0 r3 r2 r1 r0 | | |
| `V4L2_PIX_FMT_RGB555` | | | | |
| 'RGBO' | g2 g1 g0 r4 r3 r2 r1 r0 | a b4 b3 b2 b1 b0 g4 g3 | | |
| `V4L2_PIX_FMT_RGB565` | | | | |
| 'RGBP' | g2 g1 g0 r4 r3 r2 r1 r0 | b4 b3 b2 b1 b0 g5 g4 g3 | | |
| `V4L2_PIX_FMT_RGB555X` | | | | |
| 'RGBQ' | a b4 b3 b2 b1 b0 g4 g3 | g2 g1 g0 r4 r3 r2 r1 r0 | | |

| Identifier | 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_RGB565X | | | | |
| 'RGBR' | $r_4$ $r_3$ $r_2$ $r_1$ $r_0$ $g_5$ $g_4$ $g_3$ | $g_2$ $g_1$ $g_0$ $r_4$ $r_3$ $r_2$ $r_1$ $r_0$ | | |
| V4L2_PIX_FMT_BGR666 | | | | |
| 'BGRH' | $b_5$ $b_4$ $b_3$ $b_2$ $b_1$ $b_0$ $g_5$ $g_4$ | $g_3$ $g_2$ $g_1$ $g_0$ $r_5$ $r_4$ $r_3$ $r_2$ | $r_1$ $r_0$ | |
| 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 2-1. `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 2-2.

**Table 2-2. Packed RGB Image Formats (corrected)**

| Identifier | 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 | | | | |
| 'RGBQ' | $g_2$ $g_1$ $g_0$ $b_4$ $b_3$ $b_2$ $b_1$ $b_0$ | 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_BGR666 | | | | |
| 'BGRH' | $b_5$ $b_4$ $b_3$ $b_2$ $b_1$ $b_0$ $g_5$ $g_4$ | $g_3$ $g_2$ $g_1$ $g_0$ $r_5$ $r_4$ $r_3$ $r_2$ | $r_1$ $r_0$ | |
| 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' | $r_7$ $r_6$ $r_5$ $r_4$ $r_3$ $r_2$ $r_1$ $r_0$ | $g_7$ $g_6$ $g_5$ $g_4$ $g_3$ $g_2$ $g_1$ $g_0$ | $b_7$ $b_6$ $b_5$ $b_4$ $b_3$ $b_2$ $b_1$ $b_0$ | |

| Identifier | 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_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.

# V4L2_PIX_FMT_SBGGR8 (’BA81’)

## 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 2-1. 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: | $G_{30}$ | $R_{31}$ | $G_{32}$ | $R_{33}$ |

# V4L2_PIX_FMT_SGBRG8 ('GBRG')

## 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 2-1. `V4L2_PIX_FMT_SGBRG8` 4 × 4 pixel image**

**Byte Order.** Each cell is one byte.

| | | | | |
|---|---|---|---|---|
| start + 0: | $G_{00}$ | $B_{01}$ | $G_{02}$ | $B_{03}$ |
| start + 4: | $R_{10}$ | $G_{11}$ | $R_{12}$ | $G_{13}$ |
| start + 8: | $G_{20}$ | $B_{21}$ | $G_{22}$ | $B_{23}$ |
| start + 12: | $R_{30}$ | $G_{31}$ | $R_{32}$ | $G_{33}$ |

# V4L2_PIX_FMT_SGRBG8 ('GRBG')

## Name

V4L2_PIX_FMT_SGRBG8 — 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 2-1. V4L2_PIX_FMT_SGRBG8 4 × 4 pixel image**

**Byte Order.** Each cell is one byte.

| | | | | |
|---|---|---|---|---|
| start + 0: | $G_{00}$ | $R_{01}$ | $G_{02}$ | $R_{03}$ |
| start + 4: | $R_{10}$ | $B_{11}$ | $R_{12}$ | $B_{13}$ |
| start + 8: | $G_{20}$ | $R_{21}$ | $G_{22}$ | $R_{23}$ |
| start + 12: | $R_{30}$ | $B_{31}$ | $R_{32}$ | $B_{33}$ |

# V4L2_PIX_FMT_SRGGB8 ('RGGB')

## Name

V4L2_PIX_FMT_SRGGB8 — 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 red and green value, the second row of a green and blue value. This scheme repeats to the right and down for every two columns and rows.

**Example 2-1. `V4L2_PIX_FMT_SRGGB8` 4 × 4 pixel image**

**Byte Order.** Each cell is one byte.

| | | | | |
|---|---|---|---|---|
| start + 0: | $R_{00}$ | $G_{01}$ | $R_{02}$ | $G_{03}$ |
| start + 4: | $G_{10}$ | $B_{11}$ | $G_{12}$ | $B_{13}$ |
| start + 8: | $R_{20}$ | $G_{21}$ | $R_{22}$ | $G_{23}$ |
| start + 12: | $G_{30}$ | $B_{31}$ | $G_{32}$ | $B_{33}$ |

# V4L2_PIX_FMT_SBGGR16 ('BYR2')

## Name

`V4L2_PIX_FMT_SBGGR16` — Bayer RGB format

## Description

This format is similar to `V4L2_PIX_FMT_SBGGR8`, except each pixel has a depth of 16 bits. The least significant byte is stored at lower memory addresses (little-endian). Note the actual sampling precision may be lower than 16 bits, for example 10 bits per pixel with values in range 0 to 1023.

**Example 2-1. `V4L2_PIX_FMT_SBGGR16` 4 × 4 pixel image**

**Byte Order.** Each cell is one byte.

| | | | | | | |
|---|---|---|---|---|---|---|
| start + 0: | $B_{00low}$ | $B_{00high}$ | $G_{01low}$ | $G_{01high}$ | $B_{02low}$ | B |
| start + 8: | $G_{10low}$ | $G_{10high}$ | $R_{11low}$ | $R_{11high}$ | $G_{12low}$ | G |
| start + 16: | $B_{20low}$ | $B_{20high}$ | $G_{21low}$ | $G_{21high}$ | $B_{22low}$ | B |
| start + 24: | $G_{30low}$ | $G_{30high}$ | $R_{31low}$ | $R_{31high}$ | $G_{32low}$ | G |

# V4L2_PIX_FMT_SRGGB10 (’RG10’), V4L2_PIX_FMT_SGRBG10 (’BA10’), V4L2_PIX_FMT_SGBRG10 (’GB10’), V4L2_PIX_FMT_SBGGR10 (’BG10’),

## Name

`V4L2_PIX_FMT_SRGGB10`, `V4L2_PIX_FMT_SGRBG10`, `V4L2_PIX_FMT_SGBRG10`, `V4L2_PIX_FMT_SBGGR10` — 10-bit Bayer formats expanded to 16 bits

## Description

The following four pixel formats are raw sRGB / Bayer formats with 10 bits per colour. Each colour component is stored in a 16-bit word, with 6 unused high bits filled with zeros. Each n-pixel row contains n/2 green samples and n/2 blue or red samples, with alternating red and blue rows. Bytes are stored in memory in little endian order. They are conventionally described as GRGR... BGBG..., RGRG... GBGB..., etc. Below is an example of one of these formats

**Example 2-1. `V4L2_PIX_FMT_SBGGR10` 4 × 4 pixel image**

**Byte Order.** Each cell is one byte, high 6 bits in high bytes are 0.

| | | | | | | |
|---|---|---|---|---|---|---|
| start + 0: | $B_{00low}$ | $B_{00high}$ | $G_{01low}$ | $G_{01high}$ | $B_{02low}$ | B |
| start + 8: | $G_{10low}$ | $G_{10high}$ | $R_{11low}$ | $R_{11high}$ | $G_{12low}$ | G |

| | | | | | | |
|---|---|---|---|---|---|---|
| start + 16: | $B_{20low}$ | $B_{20high}$ | $G_{21low}$ | $G_{21high}$ | $B_{22low}$ | $B_{2}$ |
| start + 24: | $G_{30low}$ | $G_{30high}$ | $R_{31low}$ | $R_{31high}$ | $G_{32low}$ | $G_{3}$ |

# V4L2_PIX_FMT_SRGGB12 ('RG12'), V4L2_PIX_FMT_SGRBG12 ('BA12'), V4L2_PIX_FMT_SGBRG12 ('GB12'), V4L2_PIX_FMT_SBGGR12 ('BG12'),

## Name

`V4L2_PIX_FMT_SRGGB12`, `V4L2_PIX_FMT_SGRBG12`, `V4L2_PIX_FMT_SGBRG12`, `V4L2_PIX_FMT_SBGGR12` — 12-bit Bayer formats expanded to 16 bits

## Description

The following four pixel formats are raw sRGB / Bayer formats with 12 bits per colour. Each colour component is stored in a 16-bit word, with 6 unused high bits filled with zeros. Each n-pixel row contains n/2 green samples and n/2 blue or red samples, with alternating red and blue rows. Bytes are stored in memory in little endian order. They are conventionally described as GRGR... BGBG..., RGRG... GBGB..., etc. Below is an example of one of these formats

**Example 2-1. `V4L2_PIX_FMT_SBGGR12` 4 × 4 pixel image**

**Byte Order.** Each cell is one byte, high 6 bits in high bytes are 0.

| | | | | | | |
|---|---|---|---|---|---|---|
| start + 0: | $B_{00low}$ | $B_{00high}$ | $G_{01low}$ | $G_{01high}$ | $B_{02low}$ | $B$ |
| start + 8: | $G_{10low}$ | $G_{10high}$ | $R_{11low}$ | $R_{11high}$ | $G_{12low}$ | $G$ |
| start + 16: | $B_{20low}$ | $B_{20high}$ | $G_{21low}$ | $G_{21high}$ | $B_{22low}$ | $B_{2}$ |
| start + 24: | $G_{30low}$ | $G_{30high}$ | $R_{31low}$ | $R_{31high}$ | $G_{32low}$ | $G_{3}$ |

# 2.7. YUV Formats

YUV is the format native to TV broadcast and composite video signals. It separates the brightness information (Y) from the color information (U and V or Cb and Cr). The color information consists of red and blue *color difference* signals, this way the green component can be reconstructed by subtracting from the brightness component. See Section 2.4 for conversion examples. YUV was chosen because early television would only transmit brightness information. To add color in a way compatible with existing receivers a new signal carrier was added to transmit the color difference signals. Secondary in the YUV format the U and V components usually have lower resolution than the Y component. This is an analog video compression technique taking advantage of a property of the human visual system, being more sensitive to brightness information.

# Packed YUV formats

## Name

`Packed YUV formats` — Packed YUV formats

## Description

Similar to the packed RGB formats these formats store the Y, Cb and Cr component of each pixel in one 16 or 32 bit word.

**Table 2-1. Packed YUV Image Formats**

| Identifier | 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 |

| Identifier | 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_YUV444 | | | | |
| 'Y444 | Cb3 Cb2 Cb1 Cb0 Cr3 Cr2 Cr1 Cr0 | a3 a2 a1 a0 Y'3 Y'2 Y'1 Y'0 | | |
| V4L2_PIX_FMT_YUV555 | | | | |
| 'YUVO | Cb2 Cb1 Cb0 Cr4 Cr3 Cr2 Cr1 Cr0 | a Y'4 Y'3 Y'2 Y'1 Y'0 Cb4 Cb3 | | |
| V4L2_PIX_FMT_YUV565 | | | | |
| 'YUVP | Cb2 Cb1 Cb0 Cr4 Cr3 Cr2 Cr1 Cr0 | Y'4 Y'3 Y'2 Y'1 Y'0 Cb5 Cb4 Cb3 | | |
| V4L2_PIX_FMT_YUV32 | | | | |
| 'YUV4 | A7 a6 a5 a4 a3 a2 a1 a0 | Y'7 Y'6 Y'5 Y'4 Y'3 Y'2 Y'1 Y'0 | Cb7 Cb6 Cb5 Cb4 Cb3 Cb2 Cb1 Cb0 | Cr7 Cr6 Cr5 Cr4 Cr3 Cr2 Cr1 Cr0 |

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.

# V4L2_PIX_FMT_GREY ('GREY')

## Name

V4L2_PIX_FMT_GREY — Grey-scale image

## Description

This is a grey-scale image. It is really a degenerate Y'CbCr format which simply contains no Cb or Cr data.

**Example 2-1. V4L2_PIX_FMT_GREY 4 × 4 pixel image**

**Byte Order.** Each cell is one byte.

| | | | | |
|---|---|---|---|---|
| start + 0: | $Y'_{00}$ | $Y'_{01}$ | $Y'_{02}$ | $Y'_{03}$ |
| start + 4: | $Y'_{10}$ | $Y'_{11}$ | $Y'_{12}$ | $Y'_{13}$ |
| start + 8: | $Y'_{20}$ | $Y'_{21}$ | $Y'_{22}$ | $Y'_{23}$ |
| start + 12: | $Y'_{30}$ | $Y'_{31}$ | $Y'_{32}$ | $Y'_{33}$ |

# V4L2_PIX_FMT_Y10 ('Y10 ')

## Name

V4L2_PIX_FMT_Y10 — Grey-scale image

## Description

This is a grey-scale image with a depth of 10 bits per pixel. Pixels are stored in 16-bit words with unused high bits padded with 0. The least significant byte is stored at lower memory addresses (little-endian).

**Example 2-1. V4L2_PIX_FMT_Y10 4 × 4 pixel image**

**Byte Order.** Each cell is one byte.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| start + 0: | $Y'_{00low}$ | $Y'_{00high}$ | $Y'_{01low}$ | $Y'_{01high}$ | $Y'_{02low}$ | $Y'_{02high}$ | $Y'_{03low}$ | $Y'_{03high}$ |
| start + 8: | $Y'_{10low}$ | $Y'_{10high}$ | $Y'_{11low}$ | $Y'_{11high}$ | $Y'_{12low}$ | $Y'_{12high}$ | $Y'_{13low}$ | $Y'_{13high}$ |
| start + 16: | $Y'_{20low}$ | $Y'_{20high}$ | $Y'_{21low}$ | $Y'_{21high}$ | $Y'_{22low}$ | $Y'_{22high}$ | $Y'_{23low}$ | $Y'_{23high}$ |
| start + 24: | $Y'_{30low}$ | $Y'_{30high}$ | $Y'_{31low}$ | $Y'_{31high}$ | $Y'_{32low}$ | $Y'_{32high}$ | $Y'_{33low}$ | $Y'_{33high}$ |

# V4L2_PIX_FMT_Y12 ('Y12 ')

## Name

V4L2_PIX_FMT_Y12 — Grey-scale image

## Description

This is a grey-scale image with a depth of 12 bits per pixel. Pixels are stored in 16-bit words with unused high bits padded with 0. The least significant byte is stored at lower memory addresses (little-endian).

**Example 2-1. V4L2_PIX_FMT_Y12 4 × 4 pixel image**

**Byte Order.** Each cell is one byte.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| start + 0: | $Y'_{00low}$ | $Y'_{00high}$ | $Y'_{01low}$ | $Y'_{01high}$ | $Y'_{02low}$ | $Y'_{02high}$ | $Y'_{03low}$ | $Y'_{03high}$ |
| start + 8: | $Y'_{10low}$ | $Y'_{10high}$ | $Y'_{11low}$ | $Y'_{11high}$ | $Y'_{12low}$ | $Y'_{12high}$ | $Y'_{13low}$ | $Y'_{13high}$ |
| start + 16: | $Y'_{20low}$ | $Y'_{20high}$ | $Y'_{21low}$ | $Y'_{21high}$ | $Y'_{22low}$ | $Y'_{22high}$ | $Y'_{23low}$ | $Y'_{23high}$ |
| start + 24: | $Y'_{30low}$ | $Y'_{30high}$ | $Y'_{31low}$ | $Y'_{31high}$ | $Y'_{32low}$ | $Y'_{32high}$ | $Y'_{33low}$ | $Y'_{33high}$ |

# V4L2_PIX_FMT_Y10BPACK ('Y10B')

## Name

V4L2_PIX_FMT_Y10BPACK — Grey-scale image as a bit-packed array

## Description

This is a packed grey-scale image format with a depth of 10 bits per pixel. Pixels are stored in a bit-packed array of 10bit bits per pixel, with no padding between them and with the most significant bits coming first from the left.

**Example 2-1. V4L2_PIX_FMT_Y10BPACK 4 pixel data stream taking 5 bytes**

**Bit-packed representation.** pixels cross the byte boundary and have a ratio of 5 bytes for each 4 pixels.

| | | | | |
|---|---|---|---|---|
| $Y'_{00[9:2]}$ | $Y'_{00[1:0]}Y'_{01[9:4]}$ | $Y'_{01[3:0]}Y'_{02[9:6]}$ | $Y'_{02[5:0]}Y'_{03[9:8]}$ | $Y'_{03[7:0]}$ |

# V4L2_PIX_FMT_Y16 ('Y16 ')

## Name

V4L2_PIX_FMT_Y16 — Grey-scale image

## Description

This is a grey-scale image with a depth of 16 bits per pixel. The least significant byte is stored at lower memory addresses (little-endian). Note the actual sampling

precision may be lower than 16 bits, for example 10 bits per pixel with values in range 0 to 1023.

**Example 2-1. V4L2_PIX_FMT_Y16 4 × 4 pixel image**

**Byte Order.** Each cell is one byte.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| start + 0: | $Y'_{00low}$ | $Y'_{00high}$ | $Y'_{01low}$ | $Y'_{01high}$ | $Y'_{02low}$ | $Y'_{02high}$ | $Y'_{03low}$ | $Y'_{03high}$ |
| start + 8: | $Y'_{10low}$ | $Y'_{10high}$ | $Y'_{11low}$ | $Y'_{11high}$ | $Y'_{12low}$ | $Y'_{12high}$ | $Y'_{13low}$ | $Y'_{13high}$ |
| start + 16: | $Y'_{20low}$ | $Y'_{20high}$ | $Y'_{21low}$ | $Y'_{21high}$ | $Y'_{22low}$ | $Y'_{22high}$ | $Y'_{23low}$ | $Y'_{23high}$ |
| start + 24: | $Y'_{30low}$ | $Y'_{30high}$ | $Y'_{31low}$ | $Y'_{31high}$ | $Y'_{32low}$ | $Y'_{32high}$ | $Y'_{33low}$ | $Y'_{33high}$ |

# V4L2_PIX_FMT_YUYV ('YUYV')

## Name

V4L2_PIX_FMT_YUYV — Packed format with ½ horizontal chroma resolution, also known as YUV 4:2:2

## Description

In this format each four bytes is two pixels. Each four bytes is two Y's, a Cb and a Cr. Each Y goes to one of the pixels, and the Cb and Cr belong to both pixels. As you can see, the Cr and Cb components have half the horizontal resolution of the Y component. V4L2_PIX_FMT_YUYV is known in the Windows environment as YUY2.

**Example 2-1. `V4L2_PIX_FMT_YUYV` 4 × 4 pixel image**

**Byte Order.** Each cell is one byte.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| start + 0: | $Y'_{00}$ | $Cb_{00}$ | $Y'_{01}$ | $Cr_{00}$ | $Y'_{02}$ | $Cb_{01}$ | $Y'_{03}$ | $Cr_{01}$ |
| start + 8: | $Y'_{10}$ | $Cb_{10}$ | $Y'_{11}$ | $Cr_{10}$ | $Y'_{12}$ | $Cb_{11}$ | $Y'_{13}$ | $Cr_{11}$ |
| start + 16: | $Y'_{20}$ | $Cb_{20}$ | $Y'_{21}$ | $Cr_{20}$ | $Y'_{22}$ | $Cb_{21}$ | $Y'_{23}$ | $Cr_{21}$ |
| start + 24: | $Y'_{30}$ | $Cb_{30}$ | $Y'_{31}$ | $Cr_{30}$ | $Y'_{32}$ | $Cb_{31}$ | $Y'_{33}$ | $Cr_{31}$ |

**Color Sample Location.**

| | 0 | | 1 | | 2 | | 3 |
|---|---|---|---|---|---|---|---|
| 0 | Y | C | Y | | Y | C | Y |
| 1 | Y | C | Y | | Y | C | Y |
| 2 | Y | C | Y | | Y | C | Y |
| 3 | Y | C | Y | | Y | C | Y |

# V4L2_PIX_FMT_UYVY ('UYVY')

## Name

`V4L2_PIX_FMT_UYVY` — Variation of `V4L2_PIX_FMT_YUYV` with different order of samples in memory

## Description

In this format each four bytes is two pixels. Each four bytes is two Y's, a Cb and a Cr. Each Y goes to one of the pixels, and the Cb and Cr belong to both pixels. As you can see, the Cr and Cb components have half the horizontal resolution of the Y component.

**Example 2-1. V4L2_PIX_FMT_UYVY 4 × 4 pixel image**

**Byte Order.** Each cell is one byte.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| start + 0: | $Cb_{00}$ | $Y'_{00}$ | $Cr_{00}$ | $Y'_{01}$ | $Cb_{01}$ | $Y'_{02}$ | $Cr_{01}$ | $Y'_{03}$ |
| start + 8: | $Cb_{10}$ | $Y'_{10}$ | $Cr_{10}$ | $Y'_{11}$ | $Cb_{11}$ | $Y'_{12}$ | $Cr_{11}$ | $Y'_{13}$ |
| start + 16: | $Cb_{20}$ | $Y'_{20}$ | $Cr_{20}$ | $Y'_{21}$ | $Cb_{21}$ | $Y'_{22}$ | $Cr_{21}$ | $Y'_{23}$ |
| start + 24: | $Cb_{30}$ | $Y'_{30}$ | $Cr_{30}$ | $Y'_{31}$ | $Cb_{31}$ | $Y'_{32}$ | $Cr_{31}$ | $Y'_{33}$ |

**Color Sample Location.**

| | 0 | | 1 | 2 | | 3 |
|---|---|---|---|---|---|---|
| 0 | Y | C | Y | Y | C | Y |
| 1 | Y | C | Y | Y | C | Y |
| 2 | Y | C | Y | Y | C | Y |
| 3 | Y | C | Y | Y | C | Y |

# V4L2_PIX_FMT_YVYU ('YVYU')

## Name

V4L2_PIX_FMT_YVYU — Variation of V4L2_PIX_FMT_YUYV with different order of samples in memory

## Description

In this format each four bytes is two pixels. Each four bytes is two Y's, a Cb and a Cr. Each Y goes to one of the pixels, and the Cb and Cr belong to both pixels. As you can see, the Cr and Cb components have half the horizontal resolution of the Y component.

**Example 2-1. V4L2_PIX_FMT_YVYU 4 × 4 pixel image**

**Byte Order.** Each cell is one byte.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| start + 0: | $Y'_{00}$ | $Cr_{00}$ | $Y'_{01}$ | $Cb_{00}$ | $Y'_{02}$ | $Cr_{01}$ | $Y'_{03}$ | $Cb_{01}$ |
| start + 8: | $Y'_{10}$ | $Cr_{10}$ | $Y'_{11}$ | $Cb_{10}$ | $Y'_{12}$ | $Cr_{11}$ | $Y'_{13}$ | $Cb_{11}$ |
| start + 16: | $Y'_{20}$ | $Cr_{20}$ | $Y'_{21}$ | $Cb_{20}$ | $Y'_{22}$ | $Cr_{21}$ | $Y'_{23}$ | $Cb_{21}$ |
| start + 24: | $Y'_{30}$ | $Cr_{30}$ | $Y'_{31}$ | $Cb_{30}$ | $Y'_{32}$ | $Cr_{31}$ | $Y'_{33}$ | $Cb_{31}$ |

**Color Sample Location.**

| | 0 | | 1 | | 2 | | 3 |
|---|---|---|---|---|---|---|---|
| 0 | Y | C | Y | | Y | C | Y |
| 1 | Y | C | Y | | Y | C | Y |
| 2 | Y | C | Y | | Y | C | Y |
| 3 | Y | C | Y | | Y | C | Y |

# V4L2_PIX_FMT_VYUY ('VYUY')

## Name

V4L2_PIX_FMT_VYUY — Variation of V4L2_PIX_FMT_YUYV with different order of samples in memory

## Description

In this format each four bytes is two pixels. Each four bytes is two Y's, a Cb and a Cr. Each Y goes to one of the pixels, and the Cb and Cr belong to both pixels. As you can see, the Cr and Cb components have half the horizontal resolution of the Y component.

**Example 2-1. `V4L2_PIX_FMT_VYUY` 4 × 4 pixel image**

**Byte Order.** Each cell is one byte.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| start + 0: | $Cr_{00}$ | $Y'_{00}$ | $Cb_{00}$ | $Y'_{01}$ | $Cr_{01}$ | $Y'_{02}$ | $Cb_{01}$ | $Y'_{03}$ |
| start + 8: | $Cr_{10}$ | $Y'_{10}$ | $Cb_{10}$ | $Y'_{11}$ | $Cr_{11}$ | $Y'_{12}$ | $Cb_{11}$ | $Y'_{13}$ |
| start + 16: | $Cr_{20}$ | $Y'_{20}$ | $Cb_{20}$ | $Y'_{21}$ | $Cr_{21}$ | $Y'_{22}$ | $Cb_{21}$ | $Y'_{23}$ |
| start + 24: | $Cr_{30}$ | $Y'_{30}$ | $Cb_{30}$ | $Y'_{31}$ | $Cr_{31}$ | $Y'_{32}$ | $Cb_{31}$ | $Y'_{33}$ |

**Color Sample Location.**

| | 0 | | 1 | 2 | | 3 |
|---|---|---|---|---|---|---|
| 0 | Y | C | Y | Y | C | Y |
| 1 | Y | C | Y | Y | C | Y |
| 2 | Y | C | Y | Y | C | Y |
| 3 | Y | C | Y | Y | C | Y |

# V4L2_PIX_FMT_Y41P ('Y41P')

## Name

`V4L2_PIX_FMT_Y41P` — Format with ¼ horizontal chroma resolution, also known as YUV 4:1:1

## Description

In this format each 12 bytes is eight pixels. In the twelve bytes are two CbCr pairs and eight Y's. The first CbCr pair goes with the first four Y's, and the second CbCr pair goes with the other four Y's. The Cb and Cr components have one fourth the horizontal resolution of the Y component.

Do not confuse this format with `V4L2_PIX_FMT_YUV411P`. Y41P is derived from "YUV 4:1:1 *packed*", while YUV411P stands for "YUV 4:1:1 *planar*".

**Example 2-1. `V4L2_PIX_FMT_Y41P` 8 × 4 pixel image**

**Byte Order.** Each cell is one byte.

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| start + 0: | $Cb_{00}$ | $Y'_{00}$ | $Cr_{00}$ | $Y'_{01}$ | $Cb_{01}$ | $Y'_{02}$ | $Cr_{01}$ | $Y'_{03}$ | $Y'_{04}$ | $Y'_{05}$ | $Y'_{06}$ | $Y'_{07}$ |
| start + 12: | $Cb_{10}$ | $Y'_{10}$ | $Cr_{10}$ | $Y'_{11}$ | $Cb_{11}$ | $Y'_{12}$ | $Cr_{11}$ | $Y'_{13}$ | $Y'_{14}$ | $Y'_{15}$ | $Y'_{16}$ | $Y'_{17}$ |
| start + 24: | $Cb_{20}$ | $Y'_{20}$ | $Cr_{20}$ | $Y'_{21}$ | $Cb_{21}$ | $Y'_{22}$ | $Cr_{21}$ | $Y'_{23}$ | $Y'_{24}$ | $Y'_{25}$ | $Y'_{26}$ | $Y'_{27}$ |
| start + 36: | $Cb_{30}$ | $Y'_{30}$ | $Cr_{30}$ | $Y'_{31}$ | $Cb_{31}$ | $Y'_{32}$ | $Cr_{31}$ | $Y'_{33}$ | $Y'_{34}$ | $Y'_{35}$ | $Y'_{36}$ | $Y'_{37}$ |

**Color Sample Location.**

| | 0 | 1 | | 2 | 3 | 4 | 5 | | 6 | 7 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | Y | Y | C | Y | Y | Y | Y | C | Y | Y |
| 1 | Y | Y | C | Y | Y | Y | Y | C | Y | Y |
| 2 | Y | Y | C | Y | Y | Y | Y | C | Y | Y |
| 3 | Y | Y | C | Y | Y | Y | Y | C | Y | Y |

# V4L2_PIX_FMT_YVU420 ('YV12'), V4L2_PIX_FMT_YUV420 ('YU12')

## Name

`V4L2_PIX_FMT_YVU420, V4L2_PIX_FMT_YUV420` — Planar formats with ½ horizontal and vertical chroma resolution, also known as YUV 4:2:0

## Description

These are planar formats, as opposed to a packed format. The three components are separated into three sub- images or planes. The Y plane is first. The Y plane has one

byte per pixel. For V4L2_PIX_FMT_YVU420, the Cr plane immediately follows the Y plane in memory. The Cr plane is half the width and half the height of the Y plane (and of the image). Each Cr belongs to four pixels, a two-by-two square of the image. For example, $Cr_0$ belongs to $Y'_{00}$, $Y'_{01}$, $Y'_{10}$, and $Y'_{11}$. Following the Cr plane is the Cb plane, just like the Cr plane. V4L2_PIX_FMT_YUV420 is the same except the Cb plane comes first, then the Cr plane.

If the Y plane has pad bytes after each row, then the Cr and Cb planes have half as many pad bytes after their rows. In other words, two Cx rows (including padding) is exactly as long as one Y row (including padding).

**Example 2-1. V4L2_PIX_FMT_YVU420 4 × 4 pixel image**

**Byte Order.** Each cell is one byte.

| | | | | |
|---|---|---|---|---|
| start + 0: | $Y'_{00}$ | $Y'_{01}$ | $Y'_{02}$ | $Y'_{03}$ |
| start + 4: | $Y'_{10}$ | $Y'_{11}$ | $Y'_{12}$ | $Y'_{13}$ |
| start + 8: | $Y'_{20}$ | $Y'_{21}$ | $Y'_{22}$ | $Y'_{23}$ |
| start + 12: | $Y'_{30}$ | $Y'_{31}$ | $Y'_{32}$ | $Y'_{33}$ |
| start + 16: | $Cr_{00}$ | $Cr_{01}$ | | |
| start + 18: | $Cr_{10}$ | $Cr_{11}$ | | |
| start + 20: | $Cb_{00}$ | $Cb_{01}$ | | |
| start + 22: | $Cb_{10}$ | $Cb_{11}$ | | |

**Color Sample Location.**

| | 0 | | 1 | | 2 | | 3 |
|---|---|---|---|---|---|---|---|
| 0 | Y | | Y | | Y | | Y |
| | | C | | | | C | |
| 1 | Y | | Y | | Y | | Y |
| 2 | Y | | Y | | Y | | Y |
| | | C | | | | C | |
| 3 | Y | | Y | | Y | | Y |

# V4L2_PIX_FMT_YUV420M ('YU12M')

## Name

V4L2_PIX_FMT_YUV420M — Variation of V4L2_PIX_FMT_YUV420 with planes non contiguous in memory.

## Description

This is a multi-planar format, as opposed to a packed format. The three components are separated into three sub- images or planes. The Y plane is first. The Y plane has one byte per pixel. The Cb data constitutes the second plane which is half the width and half the height of the Y plane (and of the image). Each Cb belongs to four pixels, a two-by-two square of the image. For example, $Cb_0$ belongs to $Y'_{00}$, $Y'_{01}$, $Y'_{10}$, and $Y'_{11}$. The Cr data, just like the Cb plane, is in the third plane.

If the Y plane has pad bytes after each row, then the Cb and Cr planes have half as many pad bytes after their rows. In other words, two Cx rows (including padding) is exactly as long as one Y row (including padding).

V4L2_PIX_FMT_NV12M is intended to be used only in drivers and applications that support the multi-planar API, described in Section 1.11.

**Example 2-1. V4L2_PIX_FMT_YVU420M 4 × 4 pixel image**

**Byte Order.** Each cell is one byte.

| | | | | |
|---|---|---|---|---|
| start0 + 0: | $Y'_{00}$ | $Y'_{01}$ | $Y'_{02}$ | $Y'_{03}$ |
| start0 + 4: | $Y'_{10}$ | $Y'_{11}$ | $Y'_{12}$ | $Y'_{13}$ |
| start0 + 8: | $Y'_{20}$ | $Y'_{21}$ | $Y'_{22}$ | $Y'_{23}$ |
| start0 + 12: | $Y'_{30}$ | $Y'_{31}$ | $Y'_{32}$ | $Y'_{33}$ |
| | | | | |
| start1 + 0: | $Cb_{00}$ | $Cb_{01}$ | | |
| start1 + 2: | $Cb_{10}$ | $Cb_{11}$ | | |
| | | | | |
| start2 + 0: | $Cr_{00}$ | $Cr_{01}$ | | |
| start2 + 2: | $Cr_{10}$ | $Cr_{11}$ | | |

**Color Sample Location.**

| 0 | 1 | 2 | 3 |
|---|---|---|---|

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | Y | | Y | | Y | | Y |
| | | C | | | | C | |
| 1 | Y | | Y | | Y | | Y |
| 2 | Y | | Y | | Y | | Y |
| | | C | | | | C | |
| 3 | Y | | Y | | Y | | Y |

# V4L2_PIX_FMT_YVU410 ('YVU9'), V4L2_PIX_FMT_YUV410 ('YUV9')

## Name

`V4L2_PIX_FMT_YVU410, V4L2_PIX_FMT_YUV410` — Planar formats with ¼ horizontal and vertical chroma resolution, also known as YUV 4:1:0

## Description

These are planar formats, as opposed to a packed format. The three components are separated into three sub-images or planes. The Y plane is first. The Y plane has one byte per pixel. For `V4L2_PIX_FMT_YVU410`, the Cr plane immediately follows the Y plane in memory. The Cr plane is ¼ the width and ¼ the height of the Y plane (and of the image). Each Cr belongs to 16 pixels, a four-by-four square of the image. Following the Cr plane is the Cb plane, just like the Cr plane. `V4L2_PIX_FMT_YUV410` is the same, except the Cb plane comes first, then the Cr plane.

If the Y plane has pad bytes after each row, then the Cr and Cb planes have ¼ as many pad bytes after their rows. In other words, four Cx rows (including padding) are exactly as long as one Y row (including padding).

**Example 2-1. V4L2_PIX_FMT_YVU410 4 × 4 pixel image**

**Byte Order.** Each cell is one byte.

| | | | | |
|---|---|---|---|---|
| start + 0: | $Y'_{00}$ | $Y'_{01}$ | $Y'_{02}$ | $Y'_{03}$ |
| start + 4: | $Y'_{10}$ | $Y'_{11}$ | $Y'_{12}$ | $Y'_{13}$ |
| start + 8: | $Y'_{20}$ | $Y'_{21}$ | $Y'_{22}$ | $Y'_{23}$ |
| start + 12: | $Y'_{30}$ | $Y'_{31}$ | $Y'_{32}$ | $Y'_{33}$ |
| start + 16: | $Cr_{00}$ | | | |
| start + 17: | $Cb_{00}$ | | | |

**Color Sample Location.**

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | Y | Y | Y | Y |
| 1 | Y | Y | Y | Y |
| | | C | | |
| 2 | Y | Y | Y | Y |
| 3 | Y | Y | Y | Y |

# V4L2_PIX_FMT_YUV422P ('422P')

## Name

V4L2_PIX_FMT_YUV422P — Format with ½ horizontal chroma resolution, also known as YUV 4:2:2. Planar layout as opposed to V4L2_PIX_FMT_YUYV

## Description

This format is not commonly used. This is a planar version of the YUYV format. The three components are separated into three sub-images or planes. The Y plane is

first. The Y plane has one byte per pixel. The Cb plane immediately follows the Y plane in memory. The Cb plane is half the width of the Y plane (and of the image). Each Cb belongs to two pixels. For example, $Cb_0$ belongs to $Y'_{00}$, $Y'_{01}$. Following the Cb plane is the Cr plane, just like the Cb plane.

If the Y plane has pad bytes after each row, then the Cr and Cb planes have half as many pad bytes after their rows. In other words, two Cx rows (including padding) is exactly as long as one Y row (including padding).

**Example 2-1. `V4L2_PIX_FMT_YUV422P` 4 × 4 pixel image**

**Byte Order.** Each cell is one byte.

| | | | | |
|---|---|---|---|---|
| start + 0: | $Y'_{00}$ | $Y'_{01}$ | $Y'_{02}$ | $Y'_{03}$ |
| start + 4: | $Y'_{10}$ | $Y'_{11}$ | $Y'_{12}$ | $Y'_{13}$ |
| start + 8: | $Y'_{20}$ | $Y'_{21}$ | $Y'_{22}$ | $Y'_{23}$ |
| start + 12: | $Y'_{30}$ | $Y'_{31}$ | $Y'_{32}$ | $Y'_{33}$ |
| start + 16: | $Cb_{00}$ | $Cb_{01}$ | | |
| start + 18: | $Cb_{10}$ | $Cb_{11}$ | | |
| start + 20: | $Cb_{20}$ | $Cb_{21}$ | | |
| start + 22: | $Cb_{30}$ | $Cb_{31}$ | | |
| start + 24: | $Cr_{00}$ | $Cr_{01}$ | | |
| start + 26: | $Cr_{10}$ | $Cr_{11}$ | | |
| start + 28: | $Cr_{20}$ | $Cr_{21}$ | | |
| start + 30: | $Cr_{30}$ | $Cr_{31}$ | | |

**Color Sample Location.**

| | 0 | | 1 | 2 | | 3 |
|---|---|---|---|---|---|---|
| 0 | Y | C | Y | Y | C | Y |
| 1 | Y | C | Y | Y | C | Y |
| 2 | Y | C | Y | Y | C | Y |
| 3 | Y | C | Y | Y | C | Y |

# V4L2_PIX_FMT_YUV411P ('411P')

## Name

V4L2_PIX_FMT_YUV411P — Format with ¼ horizontal chroma resolution, also known as YUV 4:1:1. Planar layout as opposed to V4L2_PIX_FMT_Y41P

## Description

This format is not commonly used. This is a planar format similar to the 4:2:2 planar format except with half as many chroma. The three components are separated into three sub-images or planes. The Y plane is first. The Y plane has one byte per pixel. The Cb plane immediately follows the Y plane in memory. The Cb plane is ¼ the width of the Y plane (and of the image). Each Cb belongs to 4 pixels all on the same row. For example, $Cb_0$ belongs to $Y'_{00}$, $Y'_{01}$, $Y'_{02}$ and $Y'_{03}$. Following the Cb plane is the Cr plane, just like the Cb plane.

If the Y plane has pad bytes after each row, then the Cr and Cb planes have ¼ as many pad bytes after their rows. In other words, four C x rows (including padding) is exactly as long as one Y row (including padding).

**Example 2-1. V4L2_PIX_FMT_YUV411P 4 × 4 pixel image**

**Byte Order.** Each cell is one byte.

| | | | | |
|---|---|---|---|---|
| start + 0: | $Y'_{00}$ | $Y'_{01}$ | $Y'_{02}$ | $Y'_{03}$ |
| start + 4: | $Y'_{10}$ | $Y'_{11}$ | $Y'_{12}$ | $Y'_{13}$ |
| start + 8: | $Y'_{20}$ | $Y'_{21}$ | $Y'_{22}$ | $Y'_{23}$ |
| start + 12: | $Y'_{30}$ | $Y'_{31}$ | $Y'_{32}$ | $Y'_{33}$ |
| start + 16: | $Cb_{00}$ | | | |
| start + 17: | $Cb_{10}$ | | | |
| start + 18: | $Cb_{20}$ | | | |
| start + 19: | $Cb_{30}$ | | | |
| start + 20: | $Cr_{00}$ | | | |
| start + 21: | $Cr_{10}$ | | | |
| start + 22: | $Cr_{20}$ | | | |
| start + 23: | $Cr_{30}$ | | | |

**Color Sample Location.**

| | 0 | 1 | | 2 | 3 |
|---|---|---|---|---|---|
| 0 | Y | Y | C | Y | Y |
| 1 | Y | Y | C | Y | Y |
| 2 | Y | Y | C | Y | Y |
| 3 | Y | Y | C | Y | Y |

# V4L2_PIX_FMT_NV12 (’NV12’), V4L2_PIX_FMT_NV21 (’NV21’)

## Name

`V4L2_PIX_FMT_NV12, V4L2_PIX_FMT_NV21` — Formats with ½ horizontal and vertical chroma resolution, also known as YUV 4:2:0. One luminance and one chrominance plane with alternating chroma samples as opposed to `V4L2_PIX_FMT_YVU420`

## Description

These are two-plane versions of the YUV 4:2:0 format. The three components are separated into two sub-images or planes. The Y plane is first. The Y plane has one byte per pixel. For `V4L2_PIX_FMT_NV12`, a combined CbCr plane immediately follows the Y plane in memory. The CbCr plane is the same width, in bytes, as the Y plane (and of the image), but is half as tall in pixels. Each CbCr pair belongs to four pixels. For example, $Cb_0/Cr_0$ belongs to $Y'_{00}$, $Y'_{01}$, $Y'_{10}$, $Y'_{11}$. `V4L2_PIX_FMT_NV21` is the same except the Cb and Cr bytes are swapped, the CrCb plane starts with a Cr byte.

If the Y plane has pad bytes after each row, then the CbCr plane has as many pad bytes after its rows.

**Example 2-1. `V4L2_PIX_FMT_NV12` 4 × 4 pixel image**

**Byte Order.** Each cell is one byte.

| | | | | |
|---|---|---|---|---|
| start + 0: | $Y'_{00}$ | $Y'_{01}$ | $Y'_{02}$ | $Y'_{03}$ |
| start + 4: | $Y'_{10}$ | $Y'_{11}$ | $Y'_{12}$ | $Y'_{13}$ |
| start + 8: | $Y'_{20}$ | $Y'_{21}$ | $Y'_{22}$ | $Y'_{23}$ |
| start + 12: | $Y'_{30}$ | $Y'_{31}$ | $Y'_{32}$ | $Y'_{33}$ |
| start + 16: | $Cb_{00}$ | $Cr_{00}$ | $Cb_{01}$ | $Cr_{01}$ |
| start + 20: | $Cb_{10}$ | $Cr_{10}$ | $Cb_{11}$ | $Cr_{11}$ |

**Color Sample Location.**

| | 0 | | 1 | | 2 | | 3 |
|---|---|---|---|---|---|---|---|
| 0 | Y | | Y | | Y | | Y |
| | | C | | | | C | |
| 1 | Y | | Y | | Y | | Y |
| 2 | Y | | Y | | Y | | Y |
| | | C | | | | C | |
| 3 | Y | | Y | | Y | | Y |

# V4L2_PIX_FMT_NV12M ('NV12M')

## Name

`V4L2_PIX_FMT_NV12M` — Variation of `V4L2_PIX_FMT_NV12` with planes non contiguous in memory.

## Description

This is a multi-planar, two-plane version of the YUV 4:2:0 format. The three components are separated into two sub-images or planes. `V4L2_PIX_FMT_NV12M`

differs from V4L2_PIX_FMT_NV12 in that the two planes are non-contiguous in memory, i.e. the chroma plane do not necessarily immediately follows the luma plane. The luminance data occupies the first plane. The Y plane has one byte per pixel. In the second plane there is a chrominance data with alternating chroma samples. The CbCr plane is the same width, in bytes, as the Y plane (and of the image), but is half as tall in pixels. Each CbCr pair belongs to four pixels. For example, $Cb_0/Cr_0$ belongs to $Y'_{00}$, $Y'_{01}$, $Y'_{10}$, $Y'_{11}$.

V4L2_PIX_FMT_NV12M is intended to be used only in drivers and applications that support the multi-planar API, described in Section 1.11.

If the Y plane has pad bytes after each row, then the CbCr plane has as many pad bytes after its rows.

**Example 2-1. V4L2_PIX_FMT_NV12M 4 × 4 pixel image**

**Byte Order.** Each cell is one byte.

| | | | | |
|---|---|---|---|---|
| start0 + 0: | $Y'_{00}$ | $Y'_{01}$ | $Y'_{02}$ | $Y'_{03}$ |
| start0 + 4: | $Y'_{10}$ | $Y'_{11}$ | $Y'_{12}$ | $Y'_{13}$ |
| start0 + 8: | $Y'_{20}$ | $Y'_{21}$ | $Y'_{22}$ | $Y'_{23}$ |
| start0 + 12: | $Y'_{30}$ | $Y'_{31}$ | $Y'_{32}$ | $Y'_{33}$ |
| | | | | |
| start1 + 0: | $Cb_{00}$ | $Cr_{00}$ | $Cb_{01}$ | $Cr_{01}$ |
| start1 + 4: | $Cb_{10}$ | $Cr_{10}$ | $Cb_{11}$ | $Cr_{11}$ |

**Color Sample Location.**

| | 0 | | 1 | | 2 | | 3 |
|---|---|---|---|---|---|---|---|
| 0 | Y | | Y | | Y | | Y |
| | | C | | | | C | |
| 1 | Y | | Y | | Y | | Y |
| 2 | Y | | Y | | Y | | Y |
| | | C | | | | C | |
| 3 | Y | | Y | | Y | | Y |

# V4L2_PIX_FMT_NV12MT ('TM12')

## Name

V4L2_PIX_FMT_NV12MT — Formats with ½ horizontal and vertical chroma resolution. This format has two planes - one for luminance and one for chrominance. Chroma samples are interleaved. The difference to V4L2_PIX_FMT_NV12 is the memory layout. Pixels are grouped in macroblocks of 64x32 size. The order of macroblocks in memory is also not standard.

## Description

This is the two-plane versions of the YUV 4:2:0 format where data is grouped into 64x32 macroblocks. The three components are separated into two sub-images or planes. The Y plane has one byte per pixel and pixels are grouped into 64x32 macroblocks. The CbCr plane has the same width, in bytes, as the Y plane (and the image), but is half as tall in pixels. The chroma plane is also grouped into 64x32 macroblocks.

Width of the buffer has to be aligned to the multiple of 128, and height alignment is 32. Every four adjactent buffers - two horizontally and two vertically are grouped together and are located in memory in Z or flipped Z order.

Layout of macroblocks in memory is presented in the following figure.

**Figure 2-1. V4L2_PIX_FMT_NV12MT macroblock Z shape memory layout**

The requirement that width is multiple of 128 is implemented because, the Z shape cannot be cut in half horizontally. In case the vertical resolution of macroblocks is odd then the last row of macroblocks is arranged in a linear order.

In case of chroma the layout is identical. Cb and Cr samples are interleaved. Height of the buffer is aligned to 32.

**Example 2-1. Memory layout of macroblocks in V4L2_PIX_FMT_NV12 format pixel image - extreme case**

**Figure 2-2. Example V4L2_PIX_FMT_NV12MT memory layout of macroblocks**

Memory layout of macroblocks of V4L2_PIX_FMT_NV12MT format in most extreme case.

# V4L2_PIX_FMT_NV16 ('NV16'), V4L2_PIX_FMT_NV61 ('NV61')

## Name

V4L2_PIX_FMT_NV16, V4L2_PIX_FMT_NV61 — Formats with ½ horizontal chroma resolution, also known as YUV 4:2:2. One luminance and one chrominance plane with alternating chroma samples as opposed to V4L2_PIX_FMT_YVU420

## Description

These are two-plane versions of the YUV 4:2:2 format. The three components are separated into two sub-images or planes. The Y plane is first. The Y plane has one byte per pixel. For V4L2_PIX_FMT_NV16, a combined CbCr plane immediately follows the Y plane in memory. The CbCr plane is the same width and height, in bytes, as the Y plane (and of the image). Each CbCr pair belongs to two pixels. For example, $Cb_0/Cr_0$ belongs to $Y'_{00}$, $Y'_{01}$. V4L2_PIX_FMT_NV61 is the same except the Cb and Cr bytes are swapped, the CrCb plane starts with a Cr byte.

If the Y plane has pad bytes after each row, then the CbCr plane has as many pad bytes after its rows.

**Example 2-1. V4L2_PIX_FMT_NV16 4 × 4 pixel image**

**Byte Order.** Each cell is one byte.

| | | | | |
|---|---|---|---|---|
| start + 0: | $Y'_{00}$ | $Y'_{01}$ | $Y'_{02}$ | $Y'_{03}$ |
| start + 4: | $Y'_{10}$ | $Y'_{11}$ | $Y'_{12}$ | $Y'_{13}$ |
| start + 8: | $Y'_{20}$ | $Y'_{21}$ | $Y'_{22}$ | $Y'_{23}$ |
| start + 12: | $Y'_{30}$ | $Y'_{31}$ | $Y'_{32}$ | $Y'_{33}$ |
| start + 16: | $Cb_{00}$ | $Cr_{00}$ | $Cb_{01}$ | $Cr_{01}$ |
| start + 20: | $Cb_{10}$ | $Cr_{10}$ | $Cb_{11}$ | $Cr_{11}$ |

| | | | | |
|---|---|---|---|---|
| start + 24: | $Cb_{20}$ | $Cr_{20}$ | $Cb_{21}$ | $Cr_{21}$ |
| start + 28: | $Cb_{30}$ | $Cr_{30}$ | $Cb_{31}$ | $Cr_{31}$ |

**Color Sample Location.**

| | 0 | | 1 | | 2 | | 3 |
|---|---|---|---|---|---|---|---|
| 0 | Y | | Y | | Y | | Y |
| | | C | | | | C | |
| 1 | Y | | Y | | Y | | Y |
| | | C | | | | C | |
| 2 | Y | | Y | | Y | | Y |
| | | C | | | | C | |
| 3 | Y | | Y | | Y | | Y |
| | | C | | | | C | |

# V4L2_PIX_FMT_M420 ('M420')

## Name

`V4L2_PIX_FMT_M420` — Format with ½ horizontal and vertical chroma resolution, also known as YUV 4:2:0. Hybrid plane line-interleaved layout.

## Description

M420 is a YUV format with ½ horizontal and vertical chroma subsampling (YUV 4:2:0). Pixels are organized as interleaved luma and chroma planes. Two lines of luma data are followed by one line of chroma data.

The luma plane has one byte per pixel. The chroma plane contains interleaved CbCr pixels subsampled by ½ in the horizontal and vertical directions. Each CbCr pair belongs to four pixels. For example, $Cb_0/Cr_0$ belongs to $Y'_{00}$, $Y'_{01}$, $Y'_{10}$, $Y'_{11}$.

All line lengths are identical: if the Y lines include pad bytes so do the CbCr lines.

**Example 2-1. V4L2_PIX_FMT_M420 4 × 4 pixel image**

**Byte Order.** Each cell is one byte.

| | | | | |
|---|---|---|---|---|
| start + 0: | $Y'_{00}$ | $Y'_{01}$ | $Y'_{02}$ | $Y'_{03}$ |
| start + 4: | $Y'_{10}$ | $Y'_{11}$ | $Y'_{12}$ | $Y'_{13}$ |
| start + 8: | $Cb_{00}$ | $Cr_{00}$ | $Cb_{01}$ | $Cr_{01}$ |
| start + 16: | $Y'_{20}$ | $Y'_{21}$ | $Y'_{22}$ | $Y'_{23}$ |
| start + 20: | $Y'_{30}$ | $Y'_{31}$ | $Y'_{32}$ | $Y'_{33}$ |
| start + 24: | $Cb_{10}$ | $Cr_{10}$ | $Cb_{11}$ | $Cr_{11}$ |

**Color Sample Location.**

| | 0 | | 1 | 2 | | 3 |
|---|---|---|---|---|---|---|
| 0 | Y | | Y | Y | | Y |
| | | C | | | C | |
| 1 | Y | | Y | Y | | Y |
| 2 | Y | | Y | Y | | Y |
| | | C | | | C | |
| 3 | Y | | Y | Y | | Y |

# 2.8. Compressed Formats

**Table 2-9. Compressed Image Formats**

| Identifier | Code | Details |
|---|---|---|
| V4L2_PIX_FMT_JPEG | 'JPEG' | TBD. See also VIDIOC_G_JPEGCOMP, VIDIOC_S_JPEGCOMP. |
| V4L2_PIX_FMT_MPEG | 'MPEG' | MPEG stream. The actual format is determined by extended control V4L2_CID_MPEG_STREAM_TYPE, see Table 1-2. |

# 2.9. Reserved Format Identifiers

These formats are not defined by this specification, they are just listed for reference and to avoid naming conflicts. If you want to register your own format, send an e-mail to the linux-media mailing list http://www.linuxtv.org/lists.php for inclusion in the `videodev2.h` file. If you want to share your format with other developers add a link to your documentation and send a copy to the linux-media mailing list for inclusion in this section. If you think your format should be listed in a standard format section please make a proposal on the linux-media mailing list.

**Table 2-10. Reserved Image Formats**

| Identifier | Code | Details |
|---|---|---|
| V4L2_PIX_FMT_DV | 'dvsd' | unknown |
| V4L2_PIX_FMT_ET61X251 | 'E625' | Compressed format of the ET61X251 driver. |
| V4L2_PIX_FMT_HI240 | 'HI24' | 8 bit RGB format used by the BTTV driver. |
| V4L2_PIX_FMT_HM12 | 'HM12' | YUV 4:2:0 format used by the IVTV driver, http://www.ivtvdriver.org/ (http://www.ivtvdriver.org/) The format is documented in the kernel sources in the file Documentation/video4linux/cx2341x/README.h |
| V4L2_PIX_FMT_CPIA1 | 'CPIA' | YUV format used by the gspca cpia1 driver. |
| V4L2_PIX_FMT_SPCA501 | 'S501' | YUYV per line used by the gspca driver. |
| V4L2_PIX_FMT_SPCA505 | 'S505' | YYUV per line used by the gspca driver. |
| V4L2_PIX_FMT_SPCA508 | 'S508' | YUVY per line used by the gspca driver. |
| V4L2_PIX_FMT_SPCA561 | 'S561' | Compressed GBRG Bayer format used by the gspca driver. |
| V4L2_PIX_FMT_SGRBG10DPCM8 | 'DB10' | 10 bit raw Bayer DPCM compressed to 8 bits. |
| V4L2_PIX_FMT_PAC207 | 'P207' | Compressed BGGR Bayer format used by the gspca driver. |
| V4L2_PIX_FMT_MR97310A | 'M310' | Compressed BGGR Bayer format used by the gspca driver. |

| Identifier | Code | Details |
|---|---|---|
| V4L2_PIX_FMT_OV511 | 'O511' | OV511 JPEG format used by the gspca driver. |
| V4L2_PIX_FMT_OV518 | 'O518' | OV518 JPEG format used by the gspca driver. |
| V4L2_PIX_FMT_PJPG | 'PJPG' | Pixart 73xx JPEG format used by the gspca driver. |
| V4L2_PIX_FMT_SQ905C | '905C' | Compressed RGGB bayer format used by the gspca driver. |
| V4L2_PIX_FMT_MJPEG | 'MJPG' | Compressed format used by the Zoran driver |
| V4L2_PIX_FMT_PWC1 | 'PWC1' | Compressed format of the PWC driver. |
| V4L2_PIX_FMT_PWC2 | 'PWC2' | Compressed format of the PWC driver. |
| V4L2_PIX_FMT_SN9C10X | 'S910' | Compressed format of the SN9C102 driver. |
| V4L2_PIX_FMT_SN9C20X_I420 | 'S920' | YUV 4:2:0 format of the gspca sn9c20x driver. |
| V4L2_PIX_FMT_SN9C2028 | 'SONX' | Compressed GBRG bayer format of the gspca sn9c2028 driver. |
| V4L2_PIX_FMT_STV0680 | 'S680' | Bayer format of the gspca stv0680 driver. |
| V4L2_PIX_FMT_WNVA | 'WNVA' | Used by the Winnov Videum driver, http://www.thedirks.org/winnov/ (http://www.thedirks.org/winnov/) |
| V4L2_PIX_FMT_TM6000 | 'TM60' | Used by Trident tm6000 |
| V4L2_PIX_FMT_CIT_YYVYUY | 'CITV' | Used by xirlink CIT, found at IBM webcams.<br>Uses one line of Y then 1 line of VYUY |
| V4L2_PIX_FMT_KONICA420 | 'KONI' | Used by Konica webcams.<br>YUV420 planar in blocks of 256 pixels. |
| V4L2_PIX_FMT_YYUV | 'YYUV' | unknown |
| V4L2_PIX_FMT_Y4 | 'Y04 ' | Old 4-bit greyscale format. Only the least significant 4 bits of each byte are used, the other bits are set to 0. |
| V4L2_PIX_FMT_Y6 | 'Y06 ' | Old 6-bit greyscale format. Only the least significant 6 bits of each byte are used, the other bits are set to 0. |

# Chapter 3. Input/Output

The V4L2 API defines several different methods to read from or write to a device. All drivers exchanging data with applications must support at least one of them.

The classic I/O method using the `read()` and `write()` function is automatically selected after opening a V4L2 device. When the driver does not support this method attempts to read or write will fail at any time.

Other methods must be negotiated. To select the streaming I/O method with memory mapped or user buffers applications call the `VIDIOC_REQBUFS` ioctl. The asynchronous I/O method is not defined yet.

Video overlay can be considered another I/O method, although the application does not directly receive the image data. It is selected by initiating video overlay with the `VIDIOC_S_FMT` ioctl. For more information see Section 4.2.

Generally exactly one I/O method, including overlay, is associated with each file descriptor. The only exceptions are applications not exchanging data with a driver ("panel applications", see Section 1.1) and drivers permitting simultaneous video capturing and overlay using the same file descriptor, for compatibility with V4L and earlier versions of V4L2.

`VIDIOC_S_FMT` and `VIDIOC_REQBUFS` would permit this to some degree, but for simplicity drivers need not support switching the I/O method (after first switching away from read/write) other than by closing and reopening the device.

The following sections describe the various I/O methods in more detail.

## 3.1. Read/Write

Input and output devices support the `read()` and `write()` function, respectively, when the `V4L2_CAP_READWRITE` flag in the *`capabilities`* field of struct v4l2_capability returned by the `VIDIOC_QUERYCAP` ioctl is set.

Drivers may need the CPU to copy the data, but they may also support DMA to or from user memory, so this I/O method is not necessarily less efficient than other methods merely exchanging buffer pointers. It is considered inferior though because no meta-information like frame counters or timestamps are passed. This information is necessary to recognize frame dropping and to synchronize with other data streams. However this is also the simplest I/O method, requiring little or no setup to exchange data. It permits command line stunts like this (the vidctrl tool is fictitious):

```
> vidctrl /dev/video --input=0 --format=YUYV --size=352x288
> dd if=/dev/video of=myimage.422 bs=202752 count=1
```

To read from the device applications use the `read()` function, to write the `write()` function. Drivers must implement one I/O method if they exchange data with applications, but it need not be this.[1] When reading or writing is supported, the driver must also support the `select()` and `poll()` function.[2]

## 3.2. Streaming I/O (Memory Mapping)

Input and output devices support this I/O method when the `V4L2_CAP_STREAMING` flag in the *capabilities* field of struct v4l2_capability returned by the `VIDIOC_QUERYCAP` ioctl is set. There are two streaming methods, to determine if the memory mapping flavor is supported applications must call the `VIDIOC_REQBUFS` ioctl.

Streaming is an I/O method where only pointers to buffers are exchanged between application and driver, the data itself is not copied. Memory mapping is primarily intended to map buffers in device memory into the application's address space. Device memory can be for example the video memory on a graphics card with a video capture add-on. However, being the most efficient I/O method available for a long time, many other drivers support streaming as well, allocating buffers in DMA-able main memory.

A driver can support many sets of buffers. Each set is identified by a unique buffer type value. The sets are independent and each set can hold a different type of data. To access different sets at the same time different file descriptors must be used.[3]

To allocate device buffers applications call the `VIDIOC_REQBUFS` ioctl with the desired number of buffers and buffer type, for example `V4L2_BUF_TYPE_VIDEO_CAPTURE`. This ioctl can also be used to change the number of buffers or to free the allocated memory, provided none of the buffers are still mapped.

Before applications can access the buffers they must map them into their address space with the `mmap()` function. The location of the buffers in device memory can be determined with the `VIDIOC_QUERYBUF` ioctl. In the single-planar API case, the *m.offset* and *length* returned in a struct v4l2_buffer are passed as sixth and second parameter to the `mmap()` function. When using the multi-planar API, struct struct v4l2_buffer contains an array of struct v4l2_plane structures, each containing its own *m.offset* and *length*. When using the multi-planar API, every plane of every buffer has to be mapped separately, so the number of calls to `mmap()` should be equal to number of buffers times number of planes in each buffer. The offset and length values must not be modified. Remember, the buffers are allocated in physical

memory, as opposed to virtual memory, which can be swapped out to disk. Applications should free the buffers as soon as possible with the `munmap()` function.

**Example 3-1. Mapping buffers in the single-planar API**

```
struct v4l2_requestbuffers reqbuf;
struct {
 void *start;
 size_t length;
} *buffers;
unsigned int i;

memset(&reqbuf, 0, sizeof(reqbuf));
reqbuf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
reqbuf.memory = V4L2_MEMORY_MMAP;
reqbuf.count = 20;

if (-1 == ioctl (fd, VIDIOC_REQBUFS, &reqbuf)) {
 if (errno == EINVAL)
  printf("Video capturing or mmap-streaming is not supported\n");
 else
  perror("VIDIOC_REQBUFS");

 exit(EXIT_FAILURE);
}

/* We want at least five buffers. */

if (reqbuf.count < 5) {
 /* You may need to free the buffers here. */
 printf("Not enough buffer memory\n");
 exit(EXIT_FAILURE);
}

buffers = calloc(reqbuf.count, sizeof(*buffers));
assert(buffers != NULL);

for (i = 0; i < reqbuf.count; i++) {
 struct v4l2_buffer buffer;

 memset(&buffer, 0, sizeof(buffer));
 buffer.type = reqbuf.type;
 buffer.memory = V4L2_MEMORY_MMAP;
 buffer.index = i;

 if (-1 == ioctl (fd, VIDIOC_QUERYBUF, &buffer)) {
```

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

 buffers[i].length = buffer.length; /* remember for munmap() */

 buffers[i].start = mmap(NULL, buffer.length,
    PROT_READ | PROT_WRITE, /* recommended */
    MAP_SHARED,             /* recommended */
    fd, buffer.m.offset);

 if (MAP_FAILED == buffers[i].start) {
  /* If you do not exit here you should unmap() and free()
     the buffers mapped so far. */
  perror("mmap");
  exit(EXIT_FAILURE);
 }
}

/* Cleanup. */

for (i = 0; i < reqbuf.count; i++)
 munmap(buffers[i].start, buffers[i].length);
```

**Example 3-2. Mapping buffers in the multi-planar API**

```
struct v4l2_requestbuffers reqbuf;
/* Our current format uses 3 planes per buffer */
#define FMT_NUM_PLANES = 3;

struct {
 void *start[FMT_NUM_PLANES];
 size_t length[FMT_NUM_PLANES];
} *buffers;
unsigned int i, j;

memset(&reqbuf, 0, sizeof(reqbuf));
reqbuf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE_MPLANE;
reqbuf.memory = V4L2_MEMORY_MMAP;
reqbuf.count = 20;

if (ioctl(fd, VIDIOC_REQBUFS, &reqbuf) < 0) {
 if (errno == EINVAL)
  printf("Video capturing or mmap-streaming is not supported\n");
 else
  perror("VIDIOC_REQBUFS");
```

```
 exit(EXIT_FAILURE);
}

/* We want at least five buffers. */

if (reqbuf.count < 5) {
 /* You may need to free the buffers here. */
 printf("Not enough buffer memory\n");
 exit(EXIT_FAILURE);
}

buffers = calloc(reqbuf.count, sizeof(*buffers));
assert(buffers != NULL);

for (i = 0; i < reqbuf.count; i++) {
 struct v4l2_buffer buffer;
 struct v4l2_plane planes[FMT_NUM_PLANES];

 memset(&buffer, 0, sizeof(buffer));
 buffer.type = reqbuf.type;
 buffer.memory = V4L2_MEMORY_MMAP;
 buffer.index = i;
 /* length in struct v4l2_buffer in multi-planar API stores the size
  * of planes array. */
 buffer.length = FMT_NUM_PLANES;
 buffer.m.planes = planes;

 if (ioctl(fd, VIDIOC_QUERYBUF, &buffer) < 0) {
  perror("VIDIOC_QUERYBUF");
  exit(EXIT_FAILURE);
 }

 /* Every plane has to be mapped separately */
 for (j = 0; j < FMT_NUM_PLANES; j++) {
  buffers[i].length[j] = buffer.m.planes[j].length; /* remember for munmap

  buffers[i].start[j] = mmap(NULL, buffer.m.planes[j].length,
     PROT_READ | PROT_WRITE, /* recommended */
     MAP_SHARED,             /* recommended */
     fd, buffer.m.planes[j].m.offset);

  if (MAP_FAILED == buffers[i].start[j]) {
   /* If you do not exit here you should unmap() and free()
      the buffers and planes mapped so far. */
   perror("mmap");
   exit(EXIT_FAILURE);
  }
```

```
  }
}

/* Cleanup. */

for (i = 0; i < reqbuf.count; i++)
 for (j = 0; j < FMT_NUM_PLANES; j++)
  munmap(buffers[i].start[j], buffers[i].length[j]);
```

Conceptually streaming drivers maintain two buffer queues, an incoming and an outgoing queue. They separate the synchronous capture or output operation locked to a video clock from the application which is subject to random disk or network delays and preemption by other processes, thereby reducing the probability of data loss. The queues are organized as FIFOs, buffers will be output in the order enqueued in the incoming FIFO, and were captured in the order dequeued from the outgoing FIFO.

The driver may require a minimum number of buffers enqueued at all times to function, apart of this no limit exists on the number of buffers applications can enqueue in advance, or dequeue and process. They can also enqueue in a different order than buffers have been dequeued, and the driver can *fill* enqueued *empty* buffers in any order. [4] The index number of a buffer (struct v4l2_buffer *index*) plays no role here, it only identifies the buffer.

Initially all mapped buffers are in dequeued state, inaccessible by the driver. For capturing applications it is customary to first enqueue all mapped buffers, then to start capturing and enter the read loop. Here the application waits until a filled buffer can be dequeued, and re-enqueues the buffer when the data is no longer needed. Output applications fill and enqueue buffers, when enough buffers are stacked up the output is started with VIDIOC_STREAMON. In the write loop, when the application runs out of free buffers, it must wait until an empty buffer can be dequeued and reused.

To enqueue and dequeue a buffer applications use the VIDIOC_QBUF and VIDIOC_DQBUF ioctl. The status of a buffer being mapped, enqueued, full or empty can be determined at any time using the VIDIOC_QUERYBUF ioctl. Two methods exist to suspend execution of the application until one or more buffers can be dequeued. By default VIDIOC_DQBUF blocks when no buffer is in the outgoing queue. When the O_NONBLOCK flag was given to the open() function, VIDIOC_DQBUF returns immediately with an EAGAIN error code when no buffer is available. The select() or poll() function are always available.

To start and stop capturing or output applications call the VIDIOC_STREAMON and VIDIOC_STREAMOFF ioctl. Note VIDIOC_STREAMOFF removes all buffers from both queues as a side effect. Since there is no notion of doing anything "now" on a multitasking system, if an application needs to synchronize with another event it

should examine the struct v4l2_buffer *timestamp* of captured buffers, or set the field before enqueuing buffers for output.

Drivers implementing memory mapping I/O must support the VIDIOC_REQBUFS, VIDIOC_QUERYBUF, VIDIOC_QBUF, VIDIOC_DQBUF, VIDIOC_STREAMON and VIDIOC_STREAMOFF ioctl, the mmap(), munmap(), select() and poll() function.[5]

[capture example]

# 3.3. Streaming I/O (User Pointers)

Input and output devices support this I/O method when the V4L2_CAP_STREAMING flag in the *capabilities* field of struct v4l2_capability returned by the VIDIOC_QUERYCAP ioctl is set. If the particular user pointer method (not only memory mapping) is supported must be determined by calling the VIDIOC_REQBUFS ioctl.

This I/O method combines advantages of the read/write and memory mapping methods. Buffers (planes) are allocated by the application itself, and can reside for example in virtual or shared memory. Only pointers to data are exchanged, these pointers and meta-information are passed in struct v4l2_buffer (or in struct v4l2_plane in the multi-planar API case). The driver must be switched into user pointer I/O mode by calling the VIDIOC_REQBUFS with the desired buffer type. No buffers (planes) are allocated beforehand, consequently they are not indexed and cannot be queried like mapped buffers with the VIDIOC_QUERYBUF ioctl.

**Example 3-3. Initiating streaming I/O with user pointers**

```
struct v4l2_requestbuffers reqbuf;

memset (&reqbuf, 0, sizeof (reqbuf));
reqbuf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
reqbuf.memory = V4L2_MEMORY_USERPTR;

if (ioctl (fd, VIDIOC_REQBUFS, &reqbuf) == -1) {
 if (errno == EINVAL)
  printf ("Video capturing or user pointer streaming is not supported\n");
 else
  perror ("VIDIOC_REQBUFS");

 exit (EXIT_FAILURE);
}
```

Buffer (plane) addresses and sizes are passed on the fly with the VIDIOC_QBUF ioctl. Although buffers are commonly cycled, applications can pass different addresses and sizes at each VIDIOC_QBUF call. If required by the hardware the driver swaps memory pages within physical memory to create a continuous area of memory. This happens transparently to the application in the virtual memory subsystem of the kernel. When buffer pages have been swapped out to disk they are brought back and finally locked in physical memory for DMA.[6]

Filled or displayed buffers are dequeued with the VIDIOC_DQBUF ioctl. The driver can unlock the memory pages at any time between the completion of the DMA and this ioctl. The memory is also unlocked when VIDIOC_STREAMOFF is called, VIDIOC_REQBUFS, or when the device is closed. Applications must take care not to free buffers without dequeuing. For once, the buffers remain locked until further, wasting physical memory. Second the driver will not be notified when the memory is returned to the application's free list and subsequently reused for other purposes, possibly completing the requested DMA and overwriting valuable data.

For capturing applications it is customary to enqueue a number of empty buffers, to start capturing and enter the read loop. Here the application waits until a filled buffer can be dequeued, and re-enqueues the buffer when the data is no longer needed. Output applications fill and enqueue buffers, when enough buffers are stacked up output is started. In the write loop, when the application runs out of free buffers it must wait until an empty buffer can be dequeued and reused. Two methods exist to suspend execution of the application until one or more buffers can be dequeued. By default VIDIOC_DQBUF blocks when no buffer is in the outgoing queue. When the O_NONBLOCK flag was given to the open() function, VIDIOC_DQBUF returns immediately with an EAGAIN error code when no buffer is available. The select() or poll() function are always available.

To start and stop capturing or output applications call the VIDIOC_STREAMON and VIDIOC_STREAMOFF ioctl. Note VIDIOC_STREAMOFF removes all buffers from both queues and unlocks all buffers as a side effect. Since there is no notion of doing anything "now" on a multitasking system, if an application needs to synchronize with another event it should examine the struct v4l2_buffer *timestamp* of captured buffers, or set the field before enqueuing buffers for output.

Drivers implementing user pointer I/O must support the VIDIOC_REQBUFS, VIDIOC_QBUF, VIDIOC_DQBUF, VIDIOC_STREAMON and VIDIOC_STREAMOFF ioctl, the select() and poll() function.[7]

# 3.4. Asynchronous I/O

This method is not defined yet.

# 3.5. Buffers

A buffer contains data exchanged by application and driver using one of the Streaming I/O methods. In the multi-planar API, the data is held in planes, while the buffer structure acts as a container for the planes. Only pointers to buffers (planes) are exchanged, the data itself is not copied. These pointers, together with meta-information like timestamps or field parity, are stored in a struct v4l2_buffer, argument to the VIDIOC_QUERYBUF, VIDIOC_QBUF and VIDIOC_DQBUF ioctl. In the multi-planar API, some plane-specific members of struct v4l2_buffer, such as pointers and sizes for each plane, are stored in struct v4l2_plane instead. In that case, struct v4l2_buffer contains an array of plane structures.

Nominally timestamps refer to the first data byte transmitted. In practice however the wide range of hardware covered by the V4L2 API limits timestamp accuracy. Often an interrupt routine will sample the system clock shortly after the field or frame was stored completely in memory. So applications must expect a constant difference up to one field or frame period plus a small (few scan lines) random error. The delay and error can be much larger due to compression or transmission over an external bus when the frames are not properly stamped by the sender. This is frequently the case with USB cameras. Here timestamps refer to the instant the field or frame was received by the driver, not the capture time. These devices identify by not enumerating any video standards, see Section 1.7.

Similar limitations apply to output timestamps. Typically the video hardware locks to a clock controlling the video timing, the horizontal and vertical synchronization pulses. At some point in the line sequence, possibly the vertical blanking, an interrupt routine samples the system clock, compares against the timestamp and programs the hardware to repeat the previous field or frame, or to display the buffer contents.

Apart of limitations of the video device and natural inaccuracies of all clocks, it should be noted system time itself is not perfectly stable. It can be affected by power saving cycles, warped to insert leap seconds, or even turned back or forth by the system administrator affecting long term measurements. [8]

**Table 3-1. struct v4l2_buffer**

| | | |
|---|---|---|
| __u32 | *index* | Number of the buffer, set by the application. This field is only used for memory mapping I/O and can range from zero to the number of buffers allocated with the VIDIOC_REQBUFS ioctl (struct v4l2_requestbuffers *count*) minus one. |

| | | |
|---|---|---|
| enum v4l2_buf_type | *type* | Type of the buffer, same as struct v4l2_format *type* or struct v4l2_requestbuffers *type*, set by the application. |
| __u32 | *bytesused* | The number of bytes occupied by the data in the buffer. It depends on the negotiated data format and may change with each buffer for compressed variable size data like JPEG images. Drivers must set this field when *type* refers to an input stream, applications when an output stream. |
| __u32 | *flags* | Flags set by the application or driver, see Table 3-4. |
| enum v4l2_field | *field* | Indicates the field order of the image in the buffer, see Table 3-9. This field is not used when the buffer contains VBI data. Drivers must set it when *type* refers to an input stream, applications when an output stream. |

| | | |
|---|---|---|
| struct timeval | *timestamp* | For input streams this is the system time (as returned by the `gettimeofday()` function) when the first data byte was captured. For output streams the data will not be displayed before this time, secondary to the nominal frame rate determined by the current video standard in enqueued order. Applications can for example zero this field to display frames as soon as possible. The driver stores the time at which the first data byte was actually sent out in the *timestamp* field. This permits applications to monitor the drift between the video and system clock. |
| struct v4l2_timecode | *timecode* | When *type* is `V4L2_BUF_TYPE_VIDEO_CAPTURE` and the `V4L2_BUF_FLAG_TIMECODE` flag is set in *flags*, this structure contains a frame timecode. In V4L2_FIELD_ALTERNATE mode the top and bottom field contain the same timecode. Timecodes are intended to help video editing and are typically recorded on video tapes, but also embedded in compressed formats like MPEG. This field is independent of the *timestamp* and *sequence* fields. |
| __u32 | *sequence* | Set by the driver, counting the frames in the sequence. |

In V4L2_FIELD_ALTERNATE mode the top and bottom field have the same sequence number. The count starts at zero and includes dropped or repeated frames. A dropped frame was received by an input device but could not be stored due to lack of free buffer space. A repeated frame was displayed again by an output device because the application did not pass new data in time.
Note this may count the frames received e.g. over USB, without taking into account the frames dropped by the remote hardware due to limited compression throughput or bus bandwidth. These devices identify by not enumerating any video standards, see Section 1.7.

| | | | |
|---|---|---|---|
| enum v4l2_memory | *memory* | | This field must be set by applications and/or drivers in accordance with the selected I/O method. |
| union | *m* | | |
| | __u32 | *offset* | For the single-planar API and when *memory* is V4L2_MEMORY_MMAP this is the offset of the buffer from the start of the device memory. The value is returned by the driver and apart of serving as parameter to the mmap() function not useful for applications. See Section 3.2 for details |
| | unsigned long | *userptr* | For the single-planar API and when *memory* is V4L2_MEMORY_USERPTR this is a pointer to the buffer (casted to unsigned long type) in virtual memory, set by the application. See Section 3.3 for details. |
| | struct v4l2_plane | **planes* | When using the multi-planar API, contains a userspace pointer to an array of struct v4l2_plane. The size of the array should be put in the *length* field of this v4l2_buffer structure. |

| | | | |
|---|---|---|---|
| __u32 | *length* | | Size of the buffer (not the payload) in bytes for the single-planar API. For the multi-planar API should contain the number of elements in the *planes* array. |
| __u32 | *input* | | Some video capture drivers support rapid and synchronous video input changes, a function useful for example in video surveillance applications. For this purpose applications set the V4L2_BUF_FLAG_INPUT flag, and this field to the number of a video input as in struct v4l2_input field *index*. |
| __u32 | *reserved* | | A place holder for future extensions and custom (driver defined) buffer types V4L2_BUF_TYPE_PRIVATE and higher. Applications should set this to 0. |

**Table 3-2. struct v4l2_plane**

| | | | |
|---|---|---|---|
| __u32 | *bytesused* | | The number of bytes occupied by data in the plane (its payload). |
| __u32 | *length* | | Size in bytes of the plane (not its payload). |
| union | *m* | | |
| | __u32 | *mem_offset* | When the memory type in the containing struct v4l2_buffer is V4L2_MEMORY_MMAP, this is the value that should be passed to mmap(), similar to the *offset* field in struct v4l2_buffer. |

| | | | |
|---|---|---|---|
| | __unsigned long | *userptr* | When the memory type in the containing struct v4l2_buffer is V4L2_MEMORY_USERPTR, this is a userspace pointer to the memory allocated for this plane by an application. |
| __u32 | *data_offset* | | Offset in bytes to video data in the plane, if applicable. |
| __u32 | *reserved[11]* | | Reserved for future use. Should be zeroed by an application. |

**Table 3-3. enum v4l2_buf_type**

| | | |
|---|---|---|
| V4L2_BUF_TYPE_VIDEO_CAPTURE | 1 | Buffer of a single-planar video capture stream, see Section 4.1. |
| V4L2_BUF_TYPE_VIDEO_CAPTURE_MPLANE | 9 | Buffer of a multi-planar video capture stream, see Section 4.1. |
| V4L2_BUF_TYPE_VIDEO_OUTPUT | 2 | Buffer of a single-planar video output stream, see Section 4.3. |
| V4L2_BUF_TYPE_VIDEO_OUTPUT_MPLANE | 10 | Buffer of a multi-planar video output stream, see Section 4.3. |
| V4L2_BUF_TYPE_VIDEO_OVERLAY | 3 | Buffer for video overlay, see Section 4.2. |
| V4L2_BUF_TYPE_VBI_CAPTURE | 4 | Buffer of a raw VBI capture stream, see Section 4.7. |
| V4L2_BUF_TYPE_VBI_OUTPUT | 5 | Buffer of a raw VBI output stream, see Section 4.7. |
| V4L2_BUF_TYPE_SLICED_VBI_CAPTURE | 6 | Buffer of a sliced VBI capture stream, see Section 4.8. |
| V4L2_BUF_TYPE_SLICED_VBI_OUTPUT | 7 | Buffer of a sliced VBI output stream, see Section 4.8. |
| V4L2_BUF_TYPE_VIDEO_OUTPUT_OVERLAY | 8 | Buffer for video output overlay (OSD), see Section 4.4. Status: Experimental. |
| V4L2_BUF_TYPE_PRIVATE | 0x80 | This and higher values are reserved for custom (driver defined) buffer types. |

**Table 3-4. Buffer Flags**

| | | |
|---|---|---|
| V4L2_BUF_FLAG_MAPPED | 0x0001 | The buffer resides in device memory and has been mapped into the application's address space, see Section 3.2 for details. Drivers set or clear this flag when the VIDIOC_QUERYBUF, VIDIOC_QBUF or VIDIOC_DQBUF ioctl is called. Set by the driver. |
| V4L2_BUF_FLAG_QUEUED | 0x0002 | Internally drivers maintain two buffer queues, an incoming and outgoing queue. When this flag is set, the buffer is currently on the incoming queue. It automatically moves to the outgoing queue after the buffer has been filled (capture devices) or displayed (output devices). Drivers set or clear this flag when the VIDIOC_QUERYBUF ioctl is called. After (successful) calling the VIDIOC_QBUF ioctl it is always set and after VIDIOC_DQBUF always cleared. |
| V4L2_BUF_FLAG_DONE | 0x0004 | When this flag is set, the buffer is currently on the outgoing queue, ready to be dequeued from the driver. Drivers set or clear this flag when the VIDIOC_QUERYBUF ioctl is called. After calling the VIDIOC_QBUF or VIDIOC_DQBUF it is always cleared. Of course a buffer cannot be on both queues at the same time, the V4L2_BUF_FLAG_QUEUED and V4L2_BUF_FLAG_DONE flag are mutually exclusive. They can be both cleared however, then the buffer is in "dequeued" state, in the application domain to say so. |
| V4L2_BUF_FLAG_ERROR | 0x0040 | When this flag is set, the buffer has been dequeued successfully, although the data might have been corrupted. This is recoverable, streaming may continue as normal and the buffer may be reused normally. Drivers set this flag when the VIDIOC_DQBUF ioctl is called. |

| | | |
|---|---|---|
| V4L2_BUF_FLAG_KEYFRAME | 0x0008 | Drivers set or clear this flag when calling the VIDIOC_DQBUF ioctl. It may be set by video capture devices when the buffer contains a compressed image which is a key frame (or field), i. e. can be decompressed on its own. |
| V4L2_BUF_FLAG_PFRAME | 0x0010 | Similar to V4L2_BUF_FLAG_KEYFRAME this flags predicted frames or fields which contain only differences to a previous key frame. |
| V4L2_BUF_FLAG_BFRAME | 0x0020 | Similar to V4L2_BUF_FLAG_PFRAME this is a bidirectional predicted frame or field. [ooc tbd] |
| V4L2_BUF_FLAG_TIMECODE | 0x0100 | The *timecode* field is valid. Drivers set or clear this flag when the VIDIOC_DQBUF ioctl is called. |
| V4L2_BUF_FLAG_INPUT | 0x0200 | The *input* field is valid. Applications set or clear this flag before calling the VIDIOC_QBUF ioctl. |

**Table 3-5. enum v4l2_memory**

| | | |
|---|---|---|
| V4L2_MEMORY_MMAP | 1 | The buffer is used for memory mapping I/O. |
| V4L2_MEMORY_USERPTR | 2 | The buffer is used for user pointer I/O. |
| V4L2_MEMORY_OVERLAY | 3 | [to do] |

## 3.5.1. Timecodes

The v4l2_timecode structure is designed to hold a SMPTE 12M or similar timecode. (struct timeval timestamps are stored in struct v4l2_buffer field *timestamp*.)

**Table 3-6. struct v4l2_timecode**

| | | |
|---|---|---|
| __u32 | *type* | Frame rate the timecodes are based on, see Table 3-7. |
| __u32 | *flags* | Timecode flags, see Table 3-8. |
| __u8 | *frames* | Frame count, 0 ... 23/24/29/49/59, depending on the type of timecode. |

| | | |
|---|---|---|
| __u8 | *seconds* | Seconds count, 0 ... 59. This is a binary, not BCD number. |
| __u8 | *minutes* | Minutes count, 0 ... 59. This is a binary, not BCD number. |
| __u8 | *hours* | Hours count, 0 ... 29. This is a binary, not BCD number. |
| __u8 | *userbits*[4] | The "user group" bits from the timecode. |

**Table 3-7. Timecode Types**

| | | |
|---|---|---|
| V4L2_TC_TYPE_24FPS | 1 | 24 frames per second, i. e. film. |
| V4L2_TC_TYPE_25FPS | 2 | 25 frames per second, i. e. PAL or SECAM video. |
| V4L2_TC_TYPE_30FPS | 3 | 30 frames per second, i. e. NTSC video. |
| V4L2_TC_TYPE_50FPS | 4 | |
| V4L2_TC_TYPE_60FPS | 5 | |

**Table 3-8. Timecode Flags**

| | | |
|---|---|---|
| V4L2_TC_FLAG_DROPFRAME | 0x0001 | Indicates "drop frame" semantics for counting frames in 29.97 fps material. When set, frame numbers 0 and 1 at the start of each minute, except minutes 0, 10, 20, 30, 40, 50 are omitted from the count. |
| V4L2_TC_FLAG_COLORFRAME | 0x0002 | The "color frame" flag. |
| V4L2_TC_USERBITS_field | 0x000C | Field mask for the "binary group flags". |
| V4L2_TC_USERBITS_USERDEFINED | 0x0000 | Unspecified format. |
| V4L2_TC_USERBITS_8BITCHARS | 0x0008 | 8-bit ISO characters. |

# 3.6. Field Order

We have to distinguish between progressive and interlaced video. Progressive video transmits all lines of a video image sequentially. Interlaced video divides an image into two fields, containing only the odd and even lines of the image, respectively. Alternating the so called odd and even field are transmitted, and due to a small delay between fields a cathode ray TV displays the lines interleaved, yielding the original frame. This curious technique was invented because at refresh rates similar to film the image would fade out too quickly. Transmitting fields reduces the flicker without the necessity of doubling the frame rate and with it the bandwidth required for each channel.

It is important to understand a video camera does not expose one frame at a time, merely transmitting the frames separated into fields. The fields are in fact captured at two different instances in time. An object on screen may well move between one field and the next. For applications analysing motion it is of paramount importance to recognize which field of a frame is older, the *temporal order*.

When the driver provides or accepts images field by field rather than interleaved, it is also important applications understand how the fields combine to frames. We distinguish between top (aka odd) and bottom (aka even) fields, the *spatial order*: The first line of the top field is the first line of an interlaced frame, the first line of the bottom field is the second line of that frame.

However because fields were captured one after the other, arguing whether a frame commences with the top or bottom field is pointless. Any two successive top and bottom, or bottom and top fields yield a valid frame. Only when the source was progressive to begin with, e. g. when transferring film to video, two fields may come from the same frame, creating a natural order.

Counter to intuition the top field is not necessarily the older field. Whether the older field contains the top or bottom lines is a convention determined by the video standard. Hence the distinction between temporal and spatial order of fields. The diagrams below should make this clearer.

All video capture and output devices must report the current field order. Some drivers may permit the selection of a different order, to this end applications initialize the `field` field of struct v4l2_pix_format before calling the `VIDIOC_S_FMT` ioctl. If this is not desired it should have the value `V4L2_FIELD_ANY` (0).

**Table 3-9. enum v4l2_field**

| | | |
|---|---|---|
| V4L2_FIELD_ANY | 0 | Applications request this field order when any one of the V4L2_FIELD_NONE, V4L2_FIELD_TOP, V4L2_FIELD_BOTTOM, or V4L2_FIELD_INTERLACED formats is acceptable. Drivers choose depending on hardware capabilities or e. g. the requested image size, and return the actual field order. struct v4l2_buffer *field* can never be V4L2_FIELD_ANY. |
| V4L2_FIELD_NONE | 1 | Images are in progressive format, not interlaced. The driver may also indicate this order when it cannot distinguish between V4L2_FIELD_TOP and V4L2_FIELD_BOTTOM. |
| V4L2_FIELD_TOP | 2 | Images consist of the top (aka odd) field only. |
| V4L2_FIELD_BOTTOM | 3 | Images consist of the bottom (aka even) field only. Applications may wish to prevent a device from capturing interlaced images because they will have "comb" or "feathering" artefacts around moving objects. |
| V4L2_FIELD_INTERLACED | 4 | Images contain both fields, interleaved line by line. The temporal order of the fields (whether the top or bottom field is first transmitted) depends on the current video standard. M/NTSC transmits the bottom field first, all other standards the top field first. |
| V4L2_FIELD_SEQ_TB | 5 | Images contain both fields, the top field lines are stored first in memory, immediately followed by the bottom field lines. Fields are always stored in temporal order, the older one first in memory. Image sizes refer to the frame, not fields. |

| | | |
|---|---|---|
| V4L2_FIELD_SEQ_BT | 6 | Images contain both fields, the bottom field lines are stored first in memory, immediately followed by the top field lines. Fields are always stored in temporal order, the older one first in memory. Image sizes refer to the frame, not fields. |
| V4L2_FIELD_ALTERNATE | 7 | The two fields of a frame are passed in separate buffers, in temporal order, i. e. the older one first. To indicate the field parity (whether the current field is a top or bottom field) the driver or application, depending on data direction, must set struct v4l2_buffer *field* to V4L2_FIELD_TOP or V4L2_FIELD_BOTTOM. Any two successive fields pair to build a frame. If fields are successive, without any dropped fields between them (fields can drop individually), can be determined from the struct v4l2_buffer *sequence* field. Image sizes refer to the frame, not fields. This format cannot be selected when using the read/write I/O method. |
| V4L2_FIELD_INTERLACED_TB | 8 | Images contain both fields, interleaved line by line, top field first. The top field is transmitted first. |
| V4L2_FIELD_INTERLACED_BT | 9 | Images contain both fields, interleaved line by line, top field first. The bottom field is transmitted first. |

**Figure 3-1. Field Order, Top Field First Transmitted**

**Figure 3-2. Field Order, Bottom Field First Transmitted**


# Notes

1. It would be desirable if applications could depend on drivers supporting all I/O interfaces, but as much as the complex memory mapping I/O can be inadequate for some devices we have no reason to require this interface, which is most useful for simple applications capturing still images.
2. At the driver level `select()` and `poll()` are the same, and `select()` is too important to be optional.
3. One could use one file descriptor and set the buffer type field accordingly when calling `VIDIOC_QBUF` etc., but it makes the `select()` function ambiguous. We also like the clean approach of one file descriptor per logical stream. Video overlay for example is also a logical stream, although the CPU is not needed for continuous operation.
4. Random enqueue order permits applications processing images out of order (such as video codecs) to return buffers earlier, reducing the probability of data loss. Random fill order allows drivers to reuse buffers on a LIFO-basis, taking advantage of caches holding scatter-gather lists and the like.
5. At the driver level `select()` and `poll()` are the same, and `select()` is too important to be optional. The rest should be evident.
6. We expect that frequently used buffers are typically not swapped out. Anyway, the process of swapping, locking or generating scatter-gather lists may be time consuming. The delay can be masked by the depth of the incoming buffer queue, and perhaps by maintaining caches assuming a buffer will be soon enqueued again. On the other hand, to optimize memory usage drivers can limit the number of buffers locked in advance and recycle the most recently used buffers first. Of course, the pages of empty buffers in the incoming queue need not be saved to disk. Output buffers must be saved on the incoming and outgoing queue because an application may share them with other processes.
7. At the driver level `select()` and `poll()` are the same, and `select()` is too important to be optional. The rest should be evident.

8. Since no other Linux multimedia API supports unadjusted time it would be foolish to introduce here. We must use a universally supported clock to synchronize different media, hence time of day.

# Chapter 4. Interfaces

## 4.1. Video Capture Interface

Video capture devices sample an analog video signal and store the digitized images in memory. Today nearly all devices can capture at full 25 or 30 frames/second. With this interface applications can control the capture process and move images from the driver into user space.

Conventionally V4L2 video capture devices are accessed through character device special files named `/dev/video` and `/dev/video0` to `/dev/video63` with major number 81 and minor numbers 0 to 63. `/dev/video` is typically a symbolic link to the preferred video device. Note the same device files are used for video output devices.

### 4.1.1. Querying Capabilities

Devices supporting the video capture interface set the `V4L2_CAP_VIDEO_CAPTURE` or `V4L2_CAP_VIDEO_CAPTURE_MPLANE` flag in the *`capabilities`* field of struct v4l2_capability returned by the `VIDIOC_QUERYCAP` ioctl. As secondary device functions they may also support the video overlay (`V4L2_CAP_VIDEO_OVERLAY`) and the raw VBI capture (`V4L2_CAP_VBI_CAPTURE`) interface. At least one of the read/write or streaming I/O methods must be supported. Tuners and audio inputs are optional.

### 4.1.2. Supplemental Functions

Video capture devices shall support audio input, tuner, controls, cropping and scaling and streaming parameter ioctls as needed. The video input and video standard ioctls must be supported by all video capture devices.

### 4.1.3. Image Format Negotiation

The result of a capture operation is determined by cropping and image format parameters. The former select an area of the video picture to capture, the latter how images are stored in memory, i. e. in RGB or YUV format, the number of bits per pixel or width and height. Together they also define how images are scaled in the process.

As usual these parameters are *not* reset at `open()` time to permit Unix tool chains, programming a device and then reading from it as if it was a plain file. Well written V4L2 applications ensure they really get what they want, including cropping and scaling.

Cropping initialization at minimum requires to reset the parameters to defaults. An example is given in Section 1.12.

To query the current image format applications set the *type* field of a struct v4l2_format to `V4L2_BUF_TYPE_VIDEO_CAPTURE` or `V4L2_BUF_TYPE_VIDEO_CAPTURE_MPLANE` and call the `VIDIOC_G_FMT` ioctl with a pointer to this structure. Drivers fill the struct v4l2_pix_format *pix* or the struct v4l2_pix_format_mplane *pix_mp* member of the *fmt* union.

To request different parameters applications set the *type* field of a struct v4l2_format as above and initialize all fields of the struct v4l2_pix_format *vbi* member of the *fmt* union, or better just modify the results of `VIDIOC_G_FMT`, and call the `VIDIOC_S_FMT` ioctl with a pointer to this structure. Drivers may adjust the parameters and finally return the actual parameters as `VIDIOC_G_FMT` does.

Like `VIDIOC_S_FMT` the `VIDIOC_TRY_FMT` ioctl can be used to learn about hardware limitations without disabling I/O or possibly time consuming hardware preparations.

The contents of struct v4l2_pix_format and struct v4l2_pix_format_mplane are discussed in Chapter 2. See also the specification of the `VIDIOC_G_FMT`, `VIDIOC_S_FMT` and `VIDIOC_TRY_FMT` ioctls for details. Video capture devices must implement both the `VIDIOC_G_FMT` and `VIDIOC_S_FMT` ioctl, even if `VIDIOC_S_FMT` ignores all requests and always returns default parameters as `VIDIOC_G_FMT` does. `VIDIOC_TRY_FMT` is optional.

### 4.1.4. Reading Images

A video capture device may support the read() function and/or streaming (memory mapping or user pointer) I/O. See Chapter 3 for details.

## 4.2. Video Overlay Interface

Video overlay devices have the ability to genlock (TV-)video into the (VGA-)video signal of a graphics card, or to store captured images directly in video memory of a graphics card, typically with clipping. This can be considerable more efficient than capturing images and displaying them by other means. In the old days when only

nuclear power plants needed cooling towers this used to be the only way to put live video into a window.

Video overlay devices are accessed through the same character special files as video capture devices. Note the default function of a `/dev/video` device is video capturing. The overlay function is only available after calling the `VIDIOC_S_FMT` ioctl.

The driver may support simultaneous overlay and capturing using the read/write and streaming I/O methods. If so, operation at the nominal frame rate of the video standard is not guaranteed. Frames may be directed away from overlay to capture, or one field may be used for overlay and the other for capture if the capture parameters permit this.

Applications should use different file descriptors for capturing and overlay. This must be supported by all drivers capable of simultaneous capturing and overlay. Optionally these drivers may also permit capturing and overlay with a single file descriptor for compatibility with V4L and earlier versions of V4L2.[1]

## 4.2.1. Querying Capabilities

Devices supporting the video overlay interface set the `V4L2_CAP_VIDEO_OVERLAY` flag in the *`capabilities`* field of struct v4l2_capability returned by the `VIDIOC_QUERYCAP` ioctl. The overlay I/O method specified below must be supported. Tuners and audio inputs are optional.

## 4.2.2. Supplemental Functions

Video overlay devices shall support audio input, tuner, controls, cropping and scaling and streaming parameter ioctls as needed. The video input and video standard ioctls must be supported by all video overlay devices.

## 4.2.3. Setup

Before overlay can commence applications must program the driver with frame buffer parameters, namely the address and size of the frame buffer and the image format, for example RGB 5:6:5. The `VIDIOC_G_FBUF` and `VIDIOC_S_FBUF` ioctls are available to get and set these parameters, respectively. The `VIDIOC_S_FBUF` ioctl is privileged because it allows to set up DMA into physical memory, bypassing the memory protection mechanisms of the kernel. Only the superuser can change the frame buffer address and size. Users are not supposed to run TV applications as

root or with SUID bit set. A small helper application with suitable privileges should query the graphics system and program the V4L2 driver at the appropriate time.

Some devices add the video overlay to the output signal of the graphics card. In this case the frame buffer is not modified by the video device, and the frame buffer address and pixel format are not needed by the driver. The VIDIOC_S_FBUF ioctl is not privileged. An application can check for this type of device by calling the VIDIOC_G_FBUF ioctl.

A driver may support any (or none) of five clipping/blending methods:

1. Chroma-keying displays the overlaid image only where pixels in the primary graphics surface assume a certain color.
2. A bitmap can be specified where each bit corresponds to a pixel in the overlaid image. When the bit is set, the corresponding video pixel is displayed, otherwise a pixel of the graphics surface.
3. A list of clipping rectangles can be specified. In these regions *no* video is displayed, so the graphics surface can be seen here.
4. The framebuffer has an alpha channel that can be used to clip or blend the framebuffer with the video.
5. A global alpha value can be specified to blend the framebuffer contents with video images.

When simultaneous capturing and overlay is supported and the hardware prohibits different image and frame buffer formats, the format requested first takes precedence. The attempt to capture (VIDIOC_S_FMT) or overlay (VIDIOC_S_FBUF) may fail with an EBUSY error code or return accordingly modified parameters..

## 4.2.4. Overlay Window

The overlaid image is determined by cropping and overlay window parameters. The former select an area of the video picture to capture, the latter how images are overlaid and clipped. Cropping initialization at minimum requires to reset the parameters to defaults. An example is given in Section 1.12.

The overlay window is described by a struct v4l2_window. It defines the size of the image, its position over the graphics surface and the clipping to be applied. To get the current parameters applications set the *type* field of a struct v4l2_format to V4L2_BUF_TYPE_VIDEO_OVERLAY and call the VIDIOC_G_FMT ioctl. The driver fills the v4l2_window substructure named *win*. It is not possible to retrieve a previously programmed clipping list or bitmap.

To program the overlay window applications set the *type* field of a struct v4l2_format to V4L2_BUF_TYPE_VIDEO_OVERLAY, initialize the *win* substructure and call the VIDIOC_S_FMT ioctl. The driver adjusts the parameters against hardware limits and returns the actual parameters as VIDIOC_G_FMT does. Like VIDIOC_S_FMT, the VIDIOC_TRY_FMT ioctl can be used to learn about driver capabilities without actually changing driver state. Unlike VIDIOC_S_FMT this also works after the overlay has been enabled.

The scaling factor of the overlaid image is implied by the width and height given in struct v4l2_window and the size of the cropping rectangle. For more information see Section 1.12.

When simultaneous capturing and overlay is supported and the hardware prohibits different image and window sizes, the size requested first takes precedence. The attempt to capture or overlay as well (VIDIOC_S_FMT) may fail with an EBUSY error code or return accordingly modified parameters.

**Table 4-1. struct v4l2_window**

| | | |
|---|---|---|
| struct v4l2_rect | *w* | Size and position of the window relative to the top, left corner of the frame buffer defined with VIDIOC_S_FBUF. The window can extend the frame buffer width and height, the *x* and *y* coordinates can be negative, and it can lie completely outside the frame buffer. The driver clips the window accordingly, or if that is not possible, modifies its size and/or position. |
| enum v4l2_field | *field* | Applications set this field to determine which video field shall be overlaid, typically one of V4L2_FIELD_ANY (0), V4L2_FIELD_TOP, V4L2_FIELD_BOTTOM or V4L2_FIELD_INTERLACED. Drivers may have to choose a different field order and return the actual setting here. |

| | | |
|---|---|---|
| __u32 | *chromakey* | When chroma-keying has been negotiated with VIDIOC_S_FBUF applications set this field to the desired pixel value for the chroma key. The format is the same as the pixel format of the framebuffer (struct v4l2_framebuffer *fmt.pixelformat* field), with bytes in host order. E. g. for V4L2_PIX_FMT_BGR24 the value should be 0xRRGGBB on a little endian, 0xBBGGRR on a big endian host. |
| struct v4l2_clip * | *clips* | When chroma-keying has *not* been negotiated and VIDIOC_G_FBUF indicated this capability, applications can set this field to point to an array of clipping rectangles.<br>Like the window coordinates *w*, clipping rectangles are defined relative to the top, left corner of the frame buffer. However clipping rectangles must not extend the frame buffer width and height, and they must not overlap. If possible applications should merge adjacent rectangles. Whether this must create x-y or y-x bands, or the order of rectangles, is not defined. When clip lists are not supported the driver ignores this field. Its contents after calling VIDIOC_S_FMT are undefined. |
| __u32 | *clipcount* | When the application set the *clips* field, this field must contain the number of clipping rectangles in the list. When clip lists are not supported the driver ignores this field, its contents after calling VIDIOC_S_FMT are undefined. When clip lists are supported but no clipping is desired this field must be set to zero. |

| | | |
|---|---|---|
| void * | *bitmap* | When chroma-keying has *not* been negotiated and VIDIOC_G_FBUF indicated this capability, applications can set this field to point to a clipping bit mask. |

It must be of the same size as the window, *w.width* and *w.height*. Each bit corresponds to a pixel in the overlaid image, which is displayed only when the bit is *set*. Pixel coordinates translate to bits like:

```
((__u8 *) bitmap)[w.width * y + x / 8] & (1 << (x & 7))
```

where $0 \leq x < w.width$ and $0 \leq y < w.height$.[a]

When a clipping bit mask is not supported the driver ignores this field, its contents after calling VIDIOC_S_FMT are undefined. When a bit mask is supported but no clipping is desired this field must be set to NULL.

Applications need not create a clip list or bit mask. When they pass both, or despite negotiating chroma-keying, the results are undefined. Regardless of the chosen method, the clipping abilities of the hardware may be limited in quantity or quality. The results when these limits are exceeded are undefined.[b]

| | | |
|---|---|---|
| __u8 | *global_alpha* | The global alpha value used to blend the framebuffer with video images, if global alpha blending has been negotiated (V4L2_FBUF_FLAG_GLOBAL_ALPHA, see VIDIOC_S_FBUF, Table A-3). Note this field was added in Linux 2.6.23, extending the structure. However the VIDIOC_G/S/TRY_FMT ioctls, which take a pointer to a v4l2_format parent structure with padding bytes at the end, are not affected. |

Notes:

a. Should we require *w.width* to be a multiple of eight?

b. When the image is written into frame buffer memory it will be undesirable if the driver clips out less pixels than expected, because the application and graphics system are not aware these regions need to be refreshed. The driver should clip out more pixels or not write the image at all.

**Table 4-2. struct v4l2_clip[2]**

| | | |
|---|---|---|
| struct v4l2_rect | *c* | Coordinates of the clipping rectangle, relative to the top, left corner of the frame buffer. Only window pixels *outside* all clipping rectangles are displayed. |
| struct v4l2_clip * | *next* | Pointer to the next clipping rectangle, NULL when this is the last rectangle. Drivers ignore this field, it cannot be used to pass a linked list of clipping rectangles. |

**Table 4-3. struct v4l2_rect**

| | | |
|---|---|---|
| __s32 | *left* | Horizontal offset of the top, left corner of the rectangle, in pixels. |
| __s32 | *top* | Vertical offset of the top, left corner of the rectangle, in pixels. Offsets increase to the right and down. |
| __s32 | *width* | Width of the rectangle, in pixels. |
| __s32 | *height* | Height of the rectangle, in pixels. Width and height cannot be negative, the fields are signed for hysterical reasons. |

### 4.2.5. Enabling Overlay

To start or stop the frame buffer overlay applications call the VIDIOC_OVERLAY ioctl.

## 4.3. Video Output Interface

Video output devices encode stills or image sequences as analog video signal. With this interface applications can control the encoding process and move images from user space to the driver.

Conventionally V4L2 video output devices are accessed through character device special files named /dev/video and /dev/video0 to /dev/video63 with major number 81 and minor numbers 0 to 63. /dev/video is typically a symbolic link to

the preferred video device. Note the same device files are used for video capture devices.

### 4.3.1. Querying Capabilities

Devices supporting the video output interface set the V4L2_CAP_VIDEO_OUTPUT or V4L2_CAP_VIDEO_OUTPUT_MPLANE flag in the *capabilities* field of struct v4l2_capability returned by the VIDIOC_QUERYCAP ioctl. As secondary device functions they may also support the raw VBI output (V4L2_CAP_VBI_OUTPUT) interface. At least one of the read/write or streaming I/O methods must be supported. Modulators and audio outputs are optional.

### 4.3.2. Supplemental Functions

Video output devices shall support audio output, modulator, controls, cropping and scaling and streaming parameter ioctls as needed. The video output and video standard ioctls must be supported by all video output devices.

### 4.3.3. Image Format Negotiation

The output is determined by cropping and image format parameters. The former select an area of the video picture where the image will appear, the latter how images are stored in memory, i. e. in RGB or YUV format, the number of bits per pixel or width and height. Together they also define how images are scaled in the process.

As usual these parameters are *not* reset at open() time to permit Unix tool chains, programming a device and then writing to it as if it was a plain file. Well written V4L2 applications ensure they really get what they want, including cropping and scaling.

Cropping initialization at minimum requires to reset the parameters to defaults. An example is given in Section 1.12.

To query the current image format applications set the *type* field of a struct v4l2_format to V4L2_BUF_TYPE_VIDEO_OUTPUT or V4L2_BUF_TYPE_VIDEO_OUTPUT_MPLANE and call the VIDIOC_G_FMT ioctl with a pointer to this structure. Drivers fill the struct v4l2_pix_format *pix* or the struct v4l2_pix_format_mplane *pix_mp* member of the *fmt* union.

To request different parameters applications set the *type* field of a struct v4l2_format as above and initialize all fields of the struct v4l2_pix_format *vbi* member of the *fmt* union, or better just modify the results of VIDIOC_G_FMT,

and call the VIDIOC_S_FMT ioctl with a pointer to this structure. Drivers may adjust the parameters and finally return the actual parameters as VIDIOC_G_FMT does.

Like VIDIOC_S_FMT the VIDIOC_TRY_FMT ioctl can be used to learn about hardware limitations without disabling I/O or possibly time consuming hardware preparations.

The contents of struct v4l2_pix_format and struct v4l2_pix_format_mplane are discussed in Chapter 2. See also the specification of the VIDIOC_G_FMT, VIDIOC_S_FMT and VIDIOC_TRY_FMT ioctls for details. Video output devices must implement both the VIDIOC_G_FMT and VIDIOC_S_FMT ioctl, even if VIDIOC_S_FMT ignores all requests and always returns default parameters as VIDIOC_G_FMT does. VIDIOC_TRY_FMT is optional.

### 4.3.4. Writing Images

A video output device may support the write() function and/or streaming (memory mapping or user pointer) I/O. See Chapter 3 for details.

## 4.4. Video Output Overlay Interface

**Experimental:** This is an experimental interface and may change in the future.

Some video output devices can overlay a framebuffer image onto the outgoing video signal. Applications can set up such an overlay using this interface, which borrows structures and ioctls of the Video Overlay interface.

The OSD function is accessible through the same character special file as the Video Output function. Note the default function of such a /dev/video device is video capturing or output. The OSD function is only available after calling the VIDIOC_S_FMT ioctl.

### 4.4.1. Querying Capabilities

Devices supporting the *Video Output Overlay* interface set the V4L2_CAP_VIDEO_OUTPUT_OVERLAY flag in the *capabilities* field of struct v4l2_capability returned by the VIDIOC_QUERYCAP ioctl.

## 4.4.2. Framebuffer

Contrary to the *Video Overlay* interface the framebuffer is normally implemented on the TV card and not the graphics card. On Linux it is accessible as a framebuffer device (`/dev/fbN`). Given a V4L2 device, applications can find the corresponding framebuffer device by calling the `VIDIOC_G_FBUF` ioctl. It returns, amongst other information, the physical address of the framebuffer in the *`base`* field of struct v4l2_framebuffer. The framebuffer device ioctl `FBIOGET_FSCREENINFO` returns the same address in the *`smem_start`* field of struct fb_fix_screeninfo. The `FBIOGET_FSCREENINFO` ioctl and struct fb_fix_screeninfo are defined in the `linux/fb.h` header file.

The width and height of the framebuffer depends on the current video standard. A V4L2 driver may reject attempts to change the video standard (or any other ioctl which would imply a framebuffer size change) with an EBUSY error code until all applications closed the framebuffer device.

**Example 4-1. Finding a framebuffer device for OSD**

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

struct v4l2_framebuffer fbuf;
unsigned int i;
int fb_fd;

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

for (i = 0; i > 30; ++i) {
 char dev_name[16];
 struct fb_fix_screeninfo si;

 snprintf (dev_name, sizeof (dev_name), "/dev/fb%u", i);

 fb_fd = open (dev_name, O_RDWR);
 if (-1 == fb_fd) {
  switch (errno) {
  case ENOENT: /* no such file */
  case ENXIO:  /* no driver */
   continue;

  default:
   perror ("open");
   exit (EXIT_FAILURE);
  }
```

```
  }

  if (0 == ioctl (fb_fd, FBIOGET_FSCREENINFO, &si)) {
   if (si.smem_start == (unsigned long) fbuf.base)
    break;
  } else {
   /* Apparently not a framebuffer device. */
  }

  close (fb_fd);
  fb_fd = -1;
}

/* fb_fd is the file descriptor of the framebuffer device
   for the video output overlay, or -1 if no device was found. */
```

### 4.4.3. Overlay Window and Scaling

The overlay is controlled by source and target rectangles. The source rectangle selects a subsection of the framebuffer image to be overlaid, the target rectangle an area in the outgoing video signal where the image will appear. Drivers may or may not support scaling, and arbitrary sizes and positions of these rectangles. Further drivers may support any (or none) of the clipping/blending methods defined for the Video Overlay interface.

A struct v4l2_window defines the size of the source rectangle, its position in the framebuffer and the clipping/blending method to be used for the overlay. To get the current parameters applications set the *type* field of a struct v4l2_format to V4L2_BUF_TYPE_VIDEO_OUTPUT_OVERLAY and call the VIDIOC_G_FMT ioctl. The driver fills the v4l2_window substructure named *win*. It is not possible to retrieve a previously programmed clipping list or bitmap.

To program the source rectangle applications set the *type* field of a struct v4l2_format to V4L2_BUF_TYPE_VIDEO_OUTPUT_OVERLAY, initialize the *win* substructure and call the VIDIOC_S_FMT ioctl. The driver adjusts the parameters against hardware limits and returns the actual parameters as VIDIOC_G_FMT does. Like VIDIOC_S_FMT, the VIDIOC_TRY_FMT ioctl can be used to learn about driver capabilities without actually changing driver state. Unlike VIDIOC_S_FMT this also works after the overlay has been enabled.

A struct v4l2_crop defines the size and position of the target rectangle. The scaling factor of the overlay is implied by the width and height given in struct v4l2_window and struct v4l2_crop. The cropping API applies to *Video Output* and *Video Output Overlay* devices in the same way as to *Video Capture* and *Video Overlay* devices,

merely reversing the direction of the data flow. For more information see Section 1.12.

### 4.4.4. Enabling Overlay

There is no V4L2 ioctl to enable or disable the overlay, however the framebuffer interface of the driver may support the `FBIOBLANK` ioctl.

## 4.5. Codec Interface

**Suspended:** This interface has been be suspended from the V4L2 API implemented in Linux 2.6 until we have more experience with codec device interfaces.

A V4L2 codec can compress, decompress, transform, or otherwise convert video data from one format into another format, in memory. Applications send data to be converted to the driver through a `write()` call, and receive the converted data through a `read()` call. For efficiency a driver may also support streaming I/O.

[to do]

## 4.6. Effect Devices Interface

**Suspended:** This interface has been be suspended from the V4L2 API implemented in Linux 2.6 until we have more experience with effect device interfaces.

A V4L2 video effect device can do image effects, filtering, or combine two or more images or image streams. For example video transitions or wipes. Applications send data to be processed and receive the result data either with `read()` and `write()` functions, or through the streaming I/O mechanism.

[to do]

# 4.7. Raw VBI Data Interface

VBI is an abbreviation of Vertical Blanking Interval, a gap in the sequence of lines of an analog video signal. During VBI no picture information is transmitted, allowing some time while the electron beam of a cathode ray tube TV returns to the top of the screen. Using an oscilloscope you will find here the vertical synchronization pulses and short data packages ASK modulated[3] onto the video signal. These are transmissions of services such as Teletext or Closed Caption.

Subject of this interface type is raw VBI data, as sampled off a video signal, or to be added to a signal for output. The data format is similar to uncompressed video images, a number of lines times a number of samples per line, we call this a VBI image.

Conventionally V4L2 VBI devices are accessed through character device special files named `/dev/vbi` and `/dev/vbi0` to `/dev/vbi31` with major number 81 and minor numbers 224 to 255. `/dev/vbi` is typically a symbolic link to the preferred VBI device. This convention applies to both input and output devices.

To address the problems of finding related video and VBI devices VBI capturing and output is also available as device function under `/dev/video`. To capture or output raw VBI data with these devices applications must call the `VIDIOC_S_FMT` ioctl. Accessed as `/dev/vbi`, raw VBI capturing or output is the default device function.

## 4.7.1. Querying Capabilities

Devices supporting the raw VBI capturing or output API set the `V4L2_CAP_VBI_CAPTURE` or `V4L2_CAP_VBI_OUTPUT` flags, respectively, in the *`capabilities`* field of struct v4l2_capability returned by the `VIDIOC_QUERYCAP` ioctl. At least one of the read/write, streaming or asynchronous I/O methods must be supported. VBI devices may or may not have a tuner or modulator.

## 4.7.2. Supplemental Functions

VBI devices shall support video input or output, tuner or modulator, and controls ioctls as needed. The video standard ioctls provide information vital to program a VBI device, therefore must be supported.

## 4.7.3. Raw VBI Format Negotiation

Raw VBI sampling abilities can vary, in particular the sampling frequency. To

properly interpret the data V4L2 specifies an ioctl to query the sampling parameters. Moreover, to allow for some flexibility applications can also suggest different parameters.

As usual these parameters are *not* reset at `open()` time to permit Unix tool chains, programming a device and then reading from it as if it was a plain file. Well written V4L2 applications should always ensure they really get what they want, requesting reasonable parameters and then checking if the actual parameters are suitable.

To query the current raw VBI capture parameters applications set the *type* field of a struct v4l2_format to `V4L2_BUF_TYPE_VBI_CAPTURE` or `V4L2_BUF_TYPE_VBI_OUTPUT`, and call the `VIDIOC_G_FMT` ioctl with a pointer to this structure. Drivers fill the struct v4l2_vbi_format *vbi* member of the *fmt* union.

To request different parameters applications set the *type* field of a struct v4l2_format as above and initialize all fields of the struct v4l2_vbi_format *vbi* member of the *fmt* union, or better just modify the results of `VIDIOC_G_FMT`, and call the `VIDIOC_S_FMT` ioctl with a pointer to this structure. Drivers return an EINVAL error code only when the given parameters are ambiguous, otherwise they modify the parameters according to the hardware capabilites and return the actual parameters. When the driver allocates resources at this point, it may return an EBUSY error code to indicate the returned parameters are valid but the required resources are currently not available. That may happen for instance when the video and VBI areas to capture would overlap, or when the driver supports multiple opens and another process already requested VBI capturing or output. Anyway, applications must expect other resource allocation points which may return EBUSY, at the `VIDIOC_STREAMON` ioctl and the first read(), write() and select() call.

VBI devices must implement both the `VIDIOC_G_FMT` and `VIDIOC_S_FMT` ioctl, even if `VIDIOC_S_FMT` ignores all requests and always returns default parameters as `VIDIOC_G_FMT` does. `VIDIOC_TRY_FMT` is optional.

**Table 4-4. struct v4l2_vbi_format**

| | | |
|---|---|---|
| __u32 | *sampling_rate* | Samples per second, i. e. unit 1 Hz. |
| __u32 | *offset* | Horizontal offset of the VBI image, relative to the leading edge of the line synchronization pulse and counted in samples: The first sample in the VBI image will be located *offset* / *sampling_rate* seconds following the leading edge. See also Figure 4-1. |
| __u32 | *samples_per_line* | |

| | | |
|---|---|---|
| __u32 | *sample_format* | Defines the sample format as in Chapter 2, a four-character-code.[a] Usually this is V4L2_PIX_FMT_GREY, i. e. each sample consists of 8 bits with lower values oriented towards the black level. Do not assume any other correlation of values with the signal level. For example, the MSB does not necessarily indicate if the signal is 'high' or 'low' because 128 may not be the mean value of the signal. Drivers shall not convert the sample format by software. |
| __u32 | *start*[2] | This is the scanning system line number associated with the first line of the VBI image, of the first and the second field respectively. See Figure 4-2 and Figure 4-3 for valid values. VBI input drivers can return start values 0 if the hardware cannot reliable identify scanning lines, VBI acquisition may not require this information. |
| __u32 | *count*[2] | The number of lines in the first and second field image, respectively. |

Drivers should be as flexibility as possible. For example, it may be possible to extend or move the VBI capture window down to the picture area, implementing a 'full field mode' to capture data service transmissions embedded in the picture. An application can set the first or second *count* value to zero if no data is required from the respective field; *count*[1] if the scanning system is progressive, i. e. not interlaced. The corresponding start value shall be ignored by the application and driver. Anyway, drivers may not support single field capturing and return both count values non-zero.

Both *count* values set to zero, or line numbers outside the bounds depicted in Figure 4-2 and Figure 4-3, or a field image covering lines of two fields, are invalid and shall not be returned by the driver.

To initialize the *start* and *count* fields, applications must first determine the current video standard selection. The v4l2_std_id or the *framelines* field of struct v4l2_standard can be evaluated for this purpose.

| | | |
|---|---|---|
| __u32 | *flags* | See Table 4-5 below. Currently only drivers set flags, applications must set this field to zero. |
| __u32 | *reserved*[2] | This array is reserved for future extensions. Drivers and applications must set it to zero. |

Notes:
a. A few devices may be unable to sample VBI data at all but can extend the video capture window to the VBI region.

**Table 4-5. Raw VBI Format Flags**

| | | |
|---|---|---|
| V4L2_VBI_UNSYNC | 0x0001 | This flag indicates hardware which does not properly distinguish between fields. Normally the VBI image stores the first field (lower scanning line numbers) first in memory. This may be a top or bottom field depending on the video standard. When this flag is set the first or second field may be stored first, however the fields are still in correct temporal order with the older field first in memory.[a] |

| | | |
|---|---|---|
| V4L2_VBI_INTERLACED | 0x0002 | By default the two field images will be passed sequentially; all lines of the first field followed by all lines of the second field (compare Section 3.6 V4L2_FIELD_SEQ_TB and V4L2_FIELD_SEQ_BT, whether the top or bottom field is first in memory depends on the video standard). When this flag is set, the two fields are interlaced (cf. V4L2_FIELD_INTERLACED). The first line of the first field followed by the first line of the second field, then the two second lines, and so on. Such a layout may be necessary when the hardware has been programmed to capture or output interlaced video images and is unable to separate the fields for VBI capturing at the same time. For simplicity setting this flag implies that both *count* values are equal and non-zero. |

Notes:
a. Most VBI services transmit on both fields, but some have different semantics depending on the field number. These cannot be reliable decoded or encoded when V4L2_VBI_UNSYNC is set.

**Figure 4-1. Line synchronization**

**Figure 4-2. ITU-R 525 line numbering (M/NTSC and M/PAL)**


(1) For the purpose of this specification field 2 starts in line 264 and not 263.5 because half line capturing is not supported.

**Figure 4-3. ITU-R 625 line numbering**


(1) For the purpose of this specification field 2 starts in line 314 and not 313.5 because half line capturing is not supported.

Remember the VBI image format depends on the selected video standard, therefore the application must choose a new standard or query the current standard first. Attempts to read or write data ahead of format negotiation, or after switching the video standard which may invalidate the negotiated VBI parameters, should be refused by the driver. A format change during active I/O is not permitted.

## 4.7.4. Reading and writing VBI images

To assure synchronization with the field number and easier implementation, the smallest unit of data passed at a time is one frame, consisting of two fields of VBI images immediately following in memory.

The total size of a frame computes as follows:

```
(count[0] + count[1]) *
samples_per_line * sample size in bytes
```

The sample size is most likely always one byte, applications must check the *sample_format* field though, to function properly with other drivers.

A VBI device may support read/write and/or streaming (memory mapping or user pointer) I/O. The latter bears the possibility of synchronizing video and VBI data by

using buffer timestamps.

Remember the VIDIOC_STREAMON ioctl and the first read(), write() and select() call can be resource allocation points returning an EBUSY error code if the required hardware resources are temporarily unavailable, for example the device is already in use by another process.

# 4.8. Sliced VBI Data Interface

VBI stands for Vertical Blanking Interval, a gap in the sequence of lines of an analog video signal. During VBI no picture information is transmitted, allowing some time while the electron beam of a cathode ray tube TV returns to the top of the screen.

Sliced VBI devices use hardware to demodulate data transmitted in the VBI. V4L2 drivers shall *not* do this by software, see also the raw VBI interface. The data is passed as short packets of fixed size, covering one scan line each. The number of packets per video frame is variable.

Sliced VBI capture and output devices are accessed through the same character special files as raw VBI devices. When a driver supports both interfaces, the default function of a /dev/vbi device is *raw* VBI capturing or output, and the sliced VBI function is only available after calling the VIDIOC_S_FMT ioctl as defined below. Likewise a /dev/video device may support the sliced VBI API, however the default function here is video capturing or output. Different file descriptors must be used to pass raw and sliced VBI data simultaneously, if this is supported by the driver.

## 4.8.1. Querying Capabilities

Devices supporting the sliced VBI capturing or output API set the V4L2_CAP_SLICED_VBI_CAPTURE or V4L2_CAP_SLICED_VBI_OUTPUT flag respectively, in the *capabilities* field of struct v4l2_capability returned by the VIDIOC_QUERYCAP ioctl. At least one of the read/write, streaming or asynchronous I/O methods must be supported. Sliced VBI devices may have a tuner or modulator.

## 4.8.2. Supplemental Functions

Sliced VBI devices shall support video input or output and tuner or modulator ioctls if they have these capabilities, and they may support control ioctls. The video

standard ioctls provide information vital to program a sliced VBI device, therefore must be supported.

## 4.8.3. Sliced VBI Format Negotiation

To find out which data services are supported by the hardware applications can call the VIDIOC_G_SLICED_VBI_CAP ioctl. All drivers implementing the sliced VBI interface must support this ioctl. The results may differ from those of the VIDIOC_S_FMT ioctl when the number of VBI lines the hardware can capture or output per frame, or the number of services it can identify on a given line are limited. For example on PAL line 16 the hardware may be able to look for a VPS or Teletext signal, but not both at the same time.

To determine the currently selected services applications set the *type* field of struct v4l2_format to V4L2_BUF_TYPE_SLICED_VBI_CAPTURE or V4L2_BUF_TYPE_SLICED_VBI_OUTPUT, and the VIDIOC_G_FMT ioctl fills the *fmt.sliced* member, a struct v4l2_sliced_vbi_format.

Applications can request different parameters by initializing or modifying the *fmt.sliced* member and calling the VIDIOC_S_FMT ioctl with a pointer to the v4l2_format structure.

The sliced VBI API is more complicated than the raw VBI API because the hardware must be told which VBI service to expect on each scan line. Not all services may be supported by the hardware on all lines (this is especially true for VBI output where Teletext is often unsupported and other services can only be inserted in one specific line). In many cases, however, it is sufficient to just set the *service_set* field to the required services and let the driver fill the *service_lines* array according to hardware capabilities. Only if more precise control is needed should the programmer set the *service_lines* array explicitly.

The VIDIOC_S_FMT ioctl modifies the parameters according to hardware capabilities. When the driver allocates resources at this point, it may return an EBUSY error code if the required resources are temporarily unavailable. Other resource allocation points which may return EBUSY can be the VIDIOC_STREAMON ioctl and the first read(), write() and select() call.

**Table 4-6. struct v4l2_sliced_vbi_format**

| | | |
|---|---|---|
| __u32 | *service_set* | If *service_set* is non-zero when passed with VIDIOC_S_FMT or VIDIOC_TRY_FMT, the *service_lines* array will be filled by the driver according to the services specified in this field. For example, if *service_set* is initialized with V4L2_SLICED_TELETEXT_B \| V4L2_SLICED_WSS_625, a driver for the cx25840 video decoder sets lines 7-22 of both fields[a] to V4L2_SLICED_TELETEXT_B and line 23 of the first field to V4L2_SLICED_WSS_625. If *service_set* is set to zero, then the values of *service_lines* will be used instead.<br>On return the driver sets this field to the union of all elements of the returned *service_lines* array. It may contain less services than requested, perhaps just one, if the hardware cannot handle more services simultaneously. It may be empty (zero) if none of the requested services are supported by the hardware. |

__u16 | *service_lines*[2][24] | Applications initialize this array with sets of data services the driver shall look for or insert on the respective scan line. Subject to hardware capabilities drivers return the requested set, a subset, which may be just a single service, or an empty set. When the hardware cannot handle multiple services on the same line the driver shall choose one. No assumptions can be made on which service the driver chooses.
Data services are defined in Table 4-7. Array indices map to ITU-R line numbers (see also Figure 4-2 and Figure 4-3) as follows:

| Element | 525 line systems | 625 line systems |
|---|---|---|
| *service_lines*[0][1] | 1 | 1 |
| *service_lines*[0][23] | 23 | 23 |
| *service_lines*[1][1] | 264 | 314 |
| *service_lines*[1][23] | 286 | 336 |

Drivers must set *service_lines*[0][0] and *service_lines*[1][0] to zero.

__u32 | *io_size* | Maximum number of bytes passed by one `read()` or `write()` call, and the buffer size in bytes for the `VIDIOC_QBUF` and `VIDIOC_DQBUF` ioctl. Drivers set this field to the size of struct v4l2_sliced_vbi_data times the number of non-zero elements in the returned *service_lines* array (that is the number of lines potentially carrying data).

| | | |
|---|---|---|
| __u32 | *reserved*[2] | This array is reserved for future extensions. Applications and drivers must set it to zero. |

Notes:
a. According to ETS 300 706 lines 6-22 of the first field and lines 5-22 of the second field may carry Teletext data.

**Table 4-7. Sliced VBI services**

| Symbol | Value | Reference | Lines, usually | Payload |
|---|---|---|---|---|
| V4L2_SLICED_TELETEXT_B (Teletext System B) | 0x0001 | ETS 300 706, ITU BT.653 | PAL/SECAM line 7-22, 320-335 (second field 7-22) | Last 42 of the 45 byte Teletext packet, that is without clock run-in and framing code, lsb first transmitted. |
| V4L2_SLICED_VPS | 0x0400 | ETS 300 231 | PAL line 16 | Byte number 3 to 15 according to Figure 9 of ETS 300 231, lsb first transmitted. |
| V4L2_SLICED_CAPTION_525 | 0x1000 | EIA 608-B | NTSC line 21, 284 (second field 21) | Two bytes in transmission order, including parity bit, lsb first transmitted. |
| V4L2_SLICED_WSS_625 | 0x4000 | ITU BT.1119, EN 300 294 | PAL/SECAM line 23 | Byte 0<br>msb lsb msb<br>Bit 7 6 5 4 3 2 1 0 x x |
| V4L2_SLICED_VBI_525 | 0x1000 | | Set of services applicable to 525 line systems. | |
| V4L2_SLICED_VBI_625 | 0x4401 | | Set of services applicable to 625 line systems. | |

Drivers may return an EINVAL error code when applications attempt to read or write data without prior format negotiation, after switching the video standard (which may invalidate the negotiated VBI parameters) and after switching the video input (which may change the video standard as a side effect). The VIDIOC_S_FMT

ioctl may return an EBUSY error code when applications attempt to change the format while i/o is in progress (between a VIDIOC_STREAMON and VIDIOC_STREAMOFF call, and after the first read() or write() call).

## 4.8.4. Reading and writing sliced VBI data

A single read() or write() call must pass all data belonging to one video frame. That is an array of v4l2_sliced_vbi_data structures with one or more elements and a total size not exceeding *io_size* bytes. Likewise in streaming I/O mode one buffer of *io_size* bytes must contain data of one video frame. The *id* of unused v4l2_sliced_vbi_data elements must be zero.

**Table 4-8. struct v4l2_sliced_vbi_data**

| | | |
|---|---|---|
| __u32 | *id* | A flag from Table A-2 identifying the type of data in this packet. Only a single bit must be set. When the *id* of a captured packet is zero, the packet is empty and the contents of other fields are undefined. Applications shall ignore empty packets. When the *id* of a packet for output is zero the contents of the *data* field are undefined and the driver must no longer insert data on the requested *field* and *line*. |
| __u32 | *field* | The video field number this data has been captured from, or shall be inserted at. 0 for the first field, 1 for the second field. |
| __u32 | *line* | The field (as opposed to frame) line number this data has been captured from, or shall be inserted at. See Figure 4-2 and Figure 4-3 for valid values. Sliced VBI capture devices can set the line number of all packets to 0 if the hardware cannot reliably identify scan lines. The field number must always be valid. |
| __u32 | *reserved* | This field is reserved for future extensions. Applications and drivers must set it to zero. |

| | | |
|---|---|---|
| __u8 | *data*[48] | The packet payload. See Table A-2 for the contents and number of bytes passed for each data type. The contents of padding bytes at the end of this array are undefined, drivers and applications shall ignore them. |

Packets are always passed in ascending line number order, without duplicate line numbers. The `write()` function and the `VIDIOC_QBUF` ioctl must return an EINVAL error code when applications violate this rule. They must also return an EINVAL error code when applications pass an incorrect field or line number, or a combination of *field*, *line* and *id* which has not been negotiated with the `VIDIOC_G_FMT` or `VIDIOC_S_FMT` ioctl. When the line numbers are unknown the driver must pass the packets in transmitted order. The driver can insert empty packets with *id* set to zero anywhere in the packet array.

To assure synchronization and to distinguish from frame dropping, when a captured frame does not carry any of the requested data services drivers must pass one or more empty packets. When an application fails to pass VBI data in time for output, the driver must output the last VPS and WSS packet again, and disable the output of Closed Caption and Teletext data, or output data which is ignored by Closed Caption and Teletext decoders.

A sliced VBI device may support read/write and/or streaming (memory mapping and/or user pointer) I/O. The latter bears the possibility of synchronizing video and VBI data by using buffer timestamps.

## 4.8.5. Sliced VBI Data in MPEG Streams

If a device can produce an MPEG output stream, it may be capable of providing negotiated sliced VBI services as data embedded in the MPEG stream. Users or applications control this sliced VBI data insertion with the V4L2_CID_MPEG_STREAM_VBI_FMT control.

If the driver does not provide the V4L2_CID_MPEG_STREAM_VBI_FMT control, or only allows that control to be set to `V4L2_MPEG_STREAM_VBI_FMT_NONE`, then the device cannot embed sliced VBI data in the MPEG stream.

The V4L2_CID_MPEG_STREAM_VBI_FMT control does not implicitly set the device driver to capture nor cease capturing sliced VBI data. The control only indicates to embed sliced VBI data in the MPEG stream, if an application has negotiated sliced VBI service be captured.

It may also be the case that a device can embed sliced VBI data in only certain types of MPEG streams: for example in an MPEG-2 PS but not an MPEG-2 TS. In this situation, if sliced VBI data insertion is requested, the sliced VBI data will be embedded in MPEG stream types when supported, and silently omitted from MPEG stream types where sliced VBI data insertion is not supported by the device.

The following subsections specify the format of the embedded sliced VBI data.

#### 4.8.5.1. MPEG Stream Embedded, Sliced VBI Data Format: NONE

The `V4L2_MPEG_STREAM_VBI_FMT_NONE` embedded sliced VBI format shall be interpreted by drivers as a control to cease embedding sliced VBI data in MPEG streams. Neither the device nor driver shall insert "empty" embedded sliced VBI data packets in the MPEG stream when this format is set. No MPEG stream data structures are specified for this format.

#### 4.8.5.2. MPEG Stream Embedded, Sliced VBI Data Format: IVTV

The `V4L2_MPEG_STREAM_VBI_FMT_IVTV` embedded sliced VBI format, when supported, indicates to the driver to embed up to 36 lines of sliced VBI data per frame in an MPEG-2 *Private Stream 1 PES* packet encapsulated in an MPEG-2 *Program Pack* in the MPEG stream.

*Historical context*: This format specification originates from a custom, embedded, sliced VBI data format used by the `ivtv` driver. This format has already been informally specified in the kernel sources in the file `Documentation/video4linux/cx2341x/README.vbi` . The maximum size of the payload and other aspects of this format are driven by the CX23415 MPEG decoder's capabilities and limitations with respect to extracting, decoding, and displaying sliced VBI data embedded within an MPEG stream.

This format's use is *not* exclusive to the `ivtv` driver *nor* exclusive to CX2341x devices, as the sliced VBI data packet insertion into the MPEG stream is implemented in driver software. At least the `cx18` driver provides sliced VBI data insertion into an MPEG-2 PS in this format as well.

The following definitions specify the payload of the MPEG-2 *Private Stream 1 PES* packets that contain sliced VBI data when `V4L2_MPEG_STREAM_VBI_FMT_IVTV` is set. (The MPEG-2 *Private Stream 1 PES* packet header and encapsulating MPEG-2 *Program Pack* header are not detailed here. Please refer to the MPEG-2 specifications for details on those packet headers.)

The payload of the MPEG-2 *Private Stream 1 PES* packets that contain sliced VBI data is specified by struct v4l2_mpeg_vbi_fmt_ivtv. The payload is variable length, depending on the actual number of lines of sliced VBI data present in a video frame. The payload may be padded at the end with unspecified fill bytes to align the end of the payload to a 4-byte boundary. The payload shall never exceed 1552 bytes (2 fields with 18 lines/field with 43 bytes of data/line and a 4 byte magic number).

**Table 4-9. struct v4l2_mpeg_vbi_fmt_ivtv**

| | | |
|---|---|---|
| __u8 | *magic*[4] | A "magic" constant from Table 4-10 that indicates this is a valid sliced VBI data payload and also indicates which member of the anonymous union, *itv0* or *ITV0*, to use for the payload data. |
| union | (anonymous) | |
| | struct *itv0* v4l2_mpeg_vbi_itv0 | The primary form of the sliced VBI data payload that contains anywhere from 1 to 35 lines of sliced VBI data. Line masks are provided in this form of the payload indicating which VBI lines are provided. |
| | struct *ITV0* v4l2_mpeg_vbi_ITV0 | An alternate form of the sliced VBI data payload used when 36 lines of sliced VBI data are present. No line masks are provided in this form of the payload; all valid line mask bits are implcitly set. |

**Table 4-10. Magic Constants for struct v4l2_mpeg_vbi_fmt_ivtv *magic* field**

| Defined Symbol | Value | Description |
|---|---|---|
| V4L2_MPEG_VBI_IVTV_MAGIC0 | "itv0" | Indicates the *itv0* member of the union in struct v4l2_mpeg_vbi_fmt_ivtv is valid. |
| V4L2_MPEG_VBI_IVTV_MAGIC1 | "ITV0" | Indicates the *ITV0* member of the union in struct v4l2_mpeg_vbi_fmt_ivtv is valid and that 36 lines of sliced VBI data are present. |

**Table 4-11. struct v4l2_mpeg_vbi_itv0**

| | | |
|---|---|---|
| __le32 | *linemask*[2] | Bitmasks indicating the VBI service lines present. These *linemask* values are stored in little endian byte order in the MPEG stream. Some reference *linemask* bit positions with their corresponding VBI line number and video field are given below. $b_0$ indicates the least significant bit of a *linemask* value:<br>*linemask*[0] $b_0$: line 6 first field<br>*linemask*[0] $b_{17}$: line 23 first field<br>*linemask*[0] $b_{18}$: line 6 second field<br>*linemask*[0] $b_{31}$: line 19 second field<br>*linemask*[1] $b_0$: line 20 second field<br>*linemask*[1] $b_3$: line 23 second field<br>*linemask*[1] $b_4$–$b_{31}$: unused and set to 0 |
| struct v4l2_mpeg_vbi_itv0_line | *line*[35] | This is a variable length array that holds from 1 to 35 lines of sliced VBI data. The sliced VBI data lines present correspond to the bits set in the *linemask* array, starting from $b_0$ of *linemask*[0] up through $b_{31}$ of *linemask*[0], and from $b_0$ of *linemask*[1] up through $b_3$ of *linemask*[1]. *line*[0] corresponds to the first bit found set in the *linemask* array, *line*[1] corresponds to the second bit found set in the *linemask* array, etc. If no *linemask* array bits are set, then *line*[0] may contain one line of unspecified data that should be ignored by applications. |

**Table 4-12. struct v4l2_mpeg_vbi_ITV0**

| | | |
|---|---|---|
| struct v4l2_mpeg_vbi_itv0_line | *line*[36] | A fixed length array of 36 lines of sliced VBI data. *line*[0] through *line* [17] correspond to lines 6 through 23 of the first field. *line*[18] through *line*[35] corresponds to lines 6 through 23 of the second field. |

**Table 4-13. struct v4l2_mpeg_vbi_itv0_line**

| | | |
|---|---|---|
| __u8 | *id* | A line identifier value from Table 4-14 that indicates the type of sliced VBI data stored on this line. |
| __u8 | *data*[42] | The sliced VBI data for the line. |

**Table 4-14. Line Identifiers for struct v4l2_mpeg_vbi_itv0_line *id* field**

| Defined Symbol | Value | Description |
|---|---|---|
| V4L2_MPEG_VBI_IVTV_TELETEXT_B | 1 | Refer to Sliced VBI services for a description of the line payload. |
| V4L2_MPEG_VBI_IVTV_CAPTION_525 | 4 | Refer to Sliced VBI services for a description of the line payload. |
| V4L2_MPEG_VBI_IVTV_WSS_625 | 5 | Refer to Sliced VBI services for a description of the line payload. |
| V4L2_MPEG_VBI_IVTV_VPS | 7 | Refer to Sliced VBI services for a description of the line payload. |

# 4.9. Teletext Interface

This interface was aimed at devices receiving and demodulating Teletext data [ETS 300 706, ITU BT.653], evaluating the Teletext packages and storing formatted pages in cache memory. Such devices are usually implemented as microcontrollers with serial interface ($I^2C$) and could be found on old TV cards, dedicated Teletext decoding cards and home-brew devices connected to the PC parallel port.

The Teletext API was designed by Martin Buck. It was defined in the kernel header file `linux/videotext.h`, the specification is available from ftp://ftp.gwdg.de/pub/linux/misc/videotext/ (ftp://ftp.gwdg.de/pub/linux/misc/videotext/). (Videotext is the name of the German

public television Teletext service.)

Eventually the Teletext API was integrated into the V4L API with character device file names `/dev/vtx0` to `/dev/vtx31`, device major number 81, minor numbers 192 to 223.

However, teletext decoders were quickly replaced by more generic VBI demodulators and those dedicated teletext decoders no longer exist. For many years the vtx devices were still around, even though nobody used them. So the decision was made to finally remove support for the Teletext API in kernel 2.6.37.

Modern devices all use the raw or sliced VBI API.

# 4.10. Radio Interface

This interface is intended for AM and FM (analog) radio receivers and transmitters.

Conventionally V4L2 radio devices are accessed through character device special files named `/dev/radio` and `/dev/radio0` to `/dev/radio63` with major number 81 and minor numbers 64 to 127.

## 4.10.1. Querying Capabilities

Devices supporting the radio interface set the `V4L2_CAP_RADIO` and `V4L2_CAP_TUNER` or `V4L2_CAP_MODULATOR` flag in the *`capabilities`* field of struct v4l2_capability returned by the `VIDIOC_QUERYCAP` ioctl. Other combinations of capability flags are reserved for future extensions.

## 4.10.2. Supplemental Functions

Radio devices can support controls, and must support the tuner or modulator ioctls.

They do not support the video input or output, audio input or output, video standard, cropping and scaling, compression and streaming parameter, or overlay ioctls. All other ioctls and I/O methods are reserved for future extensions.

## 4.10.3. Programming

Radio devices may have a couple audio controls (as discussed in Section 1.8) such as a volume control, possibly custom controls. Further all radio devices have one tuner or modulator (these are discussed in Section 1.6) with index number zero to select the radio frequency and to determine if a monaural or FM stereo program is

received/emitted. Drivers switch automatically between AM and FM depending on the selected frequency. The VIDIOC_G_TUNER or VIDIOC_G_MODULATOR ioctl reports the supported frequency range.

# 4.11. RDS Interface

The Radio Data System transmits supplementary information in binary format, for example the station name or travel information, on an inaudible audio subcarrier of a radio program. This interface is aimed at devices capable of receiving and/or transmitting RDS information.

For more information see the core RDS standard EN 50067 and the RBDS standard NRSC-4.

Note that the RBDS standard as is used in the USA is almost identical to the RDS standard. Any RDS decoder/encoder can also handle RBDS. Only some of the fields have slightly different meanings. See the RBDS standard for more information.

The RBDS standard also specifies support for MMBS (Modified Mobile Search). This is a proprietary format which seems to be discontinued. The RDS interface does not support this format. Should support for MMBS (or the so-called 'E blocks' in general) be needed, then please contact the linux-media mailing list: http://www.linuxtv.org/lists.php.

## 4.11.1. Querying Capabilities

Devices supporting the RDS capturing API set the V4L2_CAP_RDS_CAPTURE flag in the *capabilities* field of struct v4l2_capability returned by the VIDIOC_QUERYCAP ioctl. Any tuner that supports RDS will set the V4L2_TUNER_CAP_RDS flag in the *capability* field of struct v4l2_tuner. If the driver only passes RDS blocks without interpreting the data the V4L2_TUNER_SUB_RDS_BLOCK_IO flag has to be set, see Reading RDS data. For future use the flag V4L2_TUNER_SUB_RDS_CONTROLS has also been defined. However, a driver for a radio tuner with this capability does not yet exist, so if you are planning to write such a driver you should discuss this on the linux-media mailing list: http://www.linuxtv.org/lists.php.

Whether an RDS signal is present can be detected by looking at the *rxsubchans* field of struct v4l2_tuner: the V4L2_TUNER_SUB_RDS will be set if RDS data was detected.

Devices supporting the RDS output API set the V4L2_CAP_RDS_OUTPUT flag in the *capabilities* field of struct v4l2_capability returned by the VIDIOC_QUERYCAP

ioctl. Any modulator that supports RDS will set the V4L2_TUNER_CAP_RDS flag in the *capability* field of struct v4l2_modulator. In order to enable the RDS transmission one must set the V4L2_TUNER_SUB_RDS bit in the *txsubchans* field of struct v4l2_modulator. If the driver only passes RDS blocks without interpreting the data the V4L2_TUNER_SUB_RDS_BLOCK_IO flag has to be set. If the tuner is capable of handling RDS entities like program identification codes and radio text, the flag V4L2_TUNER_SUB_RDS_CONTROLS should be set, see Writing RDS data and FM Transmitter Control Reference.

## 4.11.2. Reading RDS data

RDS data can be read from the radio device with the read() function. The data is packed in groups of three bytes.

## 4.11.3. Writing RDS data

RDS data can be written to the radio device with the write() function. The data is packed in groups of three bytes, as follows:

## 4.11.4. RDS datastructures

**Table 4-15. struct v4l2_rds_data**

| | | |
|---|---|---|
| __u8 | *lsb* | Least Significant Byte of RDS Block |
| __u8 | *msb* | Most Significant Byte of RDS Block |
| __u8 | *block* | Block description |

**Table 4-16. Block description**

| | |
|---|---|
| Bits 0-2 | Block (aka offset) of the received data. |
| Bits 3-5 | Deprecated. Currently identical to bits 0-2. Do not use these bits. |
| Bit 6 | Corrected bit. Indicates that an error was corrected for this data block. |
| Bit 7 | Error bit. Indicates that an uncorrectable error occurred during reception of this block. |

**Table 4-17. Block defines**

| | | |
|---|---|---|
| V4L2_RDS_BLOCK_MSK | 7 | Mask for bits 0-2 to get the block ID. |
| V4L2_RDS_BLOCK_A | 0 | Block A. |
| V4L2_RDS_BLOCK_B | 1 | Block B. |
| V4L2_RDS_BLOCK_C | 2 | Block C. |
| V4L2_RDS_BLOCK_D | 3 | Block D. |
| V4L2_RDS_BLOCK_C_ALT | 4 | Block C’. |
| V4L2_RDS_BLOCK_INVALID | 7 | An invalid block. Read-only |
| V4L2_RDS_BLOCK_CORRECTED | 0x40 | A bit error was detected but corrected. Read-only |
| V4L2_RDS_BLOCK_ERROR | 0x80 | An uncorrectable error occurred. Read-only |

# 4.12. Event Interface

The V4L2 event interface provides means for user to get immediately notified on certain conditions taking place on a device. This might include start of frame or loss of signal events, for example.

To receive events, the events the user is interested in first must be subscribed using the `VIDIOC_SUBSCRIBE_EVENT` ioctl. Once an event is subscribed, the events of subscribed types are dequeueable using the `VIDIOC_DQEVENT` ioctl. Events may be unsubscribed using VIDIOC_UNSUBSCRIBE_EVENT ioctl. The special event type V4L2_EVENT_ALL may be used to unsubscribe all the events the driver supports.

The event subscriptions and event queues are specific to file handles. Subscribing an event on one file handle does not affect other file handles.

The information on dequeueable events is obtained by using select or poll system calls on video devices. The V4L2 events use POLLPRI events on poll system call

and exceptions on select system call.

# 4.13. Sub-device Interface

**Experimental:** This is an experimental interface and may change in the future.

The complex nature of V4L2 devices, where hardware is often made of several integrated circuits that need to interact with each other in a controlled way, leads to complex V4L2 drivers. The drivers usually reflect the hardware model in software, and model the different hardware components as software blocks called sub-devices.

V4L2 sub-devices are usually kernel-only objects. If the V4L2 driver implements the media device API, they will automatically inherit from media entities. Applications will be able to enumerate the sub-devices and discover the hardware topology using the media entities, pads and links enumeration API.

In addition to make sub-devices discoverable, drivers can also choose to make them directly configurable by applications. When both the sub-device driver and the V4L2 device driver support this, sub-devices will feature a character device node on which ioctls can be called to

- query, read and write sub-devices controls
- subscribe and unsubscribe to events and retrieve them
- negotiate image formats on individual pads

Sub-device character device nodes, conventionally named `/dev/v4l-subdev*`, use major number 81.

## 4.13.1. Controls

Most V4L2 controls are implemented by sub-device hardware. Drivers usually merge all controls and expose them through video device nodes. Applications can control all sub-devices through a single interface.

Complex devices sometimes implement the same control in different pieces of hardware. This situation is common in embedded platforms, where both sensors and image processing hardware implement identical functions, such as contrast adjustment, white balance or faulty pixels correction. As the V4L2 controls API

doesn't support several identical controls in a single device, all but one of the identical controls are hidden.

Applications can access those hidden controls through the sub-device node with the V4L2 control API described in Section 1.8. The ioctls behave identically as when issued on V4L2 device nodes, with the exception that they deal only with controls implemented in the sub-device.

Depending on the driver, those controls might also be exposed through one (or several) V4L2 device nodes.

## 4.13.2. Events

V4L2 sub-devices can notify applications of events as described in Section 4.12. The API behaves identically as when used on V4L2 device nodes, with the exception that it only deals with events generated by the sub-device. Depending on the driver, those events might also be reported on one (or several) V4L2 device nodes.

## 4.13.3. Pad-level Formats

**Warning**

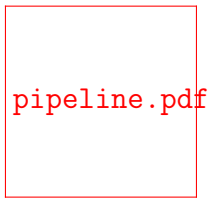
Pad-level formats are only applicable to very complex device that need to expose low-level format configuration to user space. Generic V4L2 applications do *not* need to use the API described in this section.

**Note:** For the purpose of this section, the term *format* means the combination of media bus data format, frame width and frame height.

Image formats are typically negotiated on video capture and output devices using the cropping and scaling ioctls. The driver is responsible for configuring every block in the video pipeline according to the requested format at the pipeline input and/or output.

For complex devices, such as often found in embedded systems, identical image sizes at the output of a pipeline can be achieved using different hardware configurations. One such example is shown on Figure 4-4, where image scaling can be performed on both the video sensor and the host image processing hardware.

**Figure 4-4. Image Format Negotiation on Pipelines**




The sensor scaler is usually of less quality than the host scaler, but scaling on the sensor is required to achieve higher frame rates. Depending on the use case (quality vs. speed), the pipeline must be configured differently. Applications need to configure the formats at every point in the pipeline explicitly.

Drivers that implement the media API can expose pad-level image format configuration to applications. When they do, applications can use the VIDIOC_SUBDEV_G_FMT and VIDIOC_SUBDEV_S_FMT ioctls. to negotiate formats on a per-pad basis.

Applications are responsible for configuring coherent parameters on the whole pipeline and making sure that connected pads have compatible formats. The pipeline is checked for formats mismatch at VIDIOC_STREAMON time, and an EPIPE error code is then returned if the configuration is invalid.

Pad-level image format configuration support can be tested by calling the VIDIOC_SUBDEV_G_FMT ioctl on pad 0. If the driver returns an EINVAL error code pad-level format configuration is not supported by the sub-device.

### 4.13.3.1. Format Negotiation

Acceptable formats on pads can (and usually do) depend on a number of external parameters, such as formats on other pads, active links, or even controls. Finding a combination of formats on all pads in a video pipeline, acceptable to both application and driver, can't rely on formats enumeration only. A format negotiation mechanism is required.

Central to the format negotiation mechanism are the get/set format operations. When called with the *which* argument set to V4L2_SUBDEV_FORMAT_TRY, the VIDIOC_SUBDEV_G_FMT and VIDIOC_SUBDEV_S_FMT ioctls operate on a set of formats parameters that are not connected to the hardware configuration. Modifying those 'try' formats leaves the device state untouched (this applies to both the software state stored in the driver and the hardware state stored in the device itself).

While not kept as part of the device state, try formats are stored in the sub-device file handles. A VIDIOC_SUBDEV_G_FMT call will return the last try format set *on the same sub-device file handle*. Several applications querying the same sub-device at the same time will thus not interact with each other.

To find out whether a particular format is supported by the device, applications use the VIDIOC_SUBDEV_S_FMT ioctl. Drivers verify and, if needed, change the requested *format* based on device requirements and return the possibly modified value. Applications can then choose to try a different format or accept the returned value and continue.

Formats returned by the driver during a negotiation iteration are guaranteed to be supported by the device. In particular, drivers guarantee that a returned format will not be further changed if passed to an VIDIOC_SUBDEV_S_FMT call as-is (as long as external parameters, such as formats on other pads or links' configuration are not changed).

Drivers automatically propagate formats inside sub-devices. When a try or active format is set on a pad, corresponding formats on other pads of the same sub-device can be modified by the driver. Drivers are free to modify formats as required by the device. However, they should comply with the following rules when possible:

- Formats should be propagated from sink pads to source pads. Modifying a format on a source pad should not modify the format on any sink pad.
- Sub-devices that scale frames using variable scaling factors should reset the scale factors to default values when sink pads formats are modified. If the 1:1 scaling ratio is supported, this means that source pads formats should be reset to the sink pads formats.

Formats are not propagated across links, as that would involve propagating them from one sub-device file handle to another. Applications must then take care to configure both ends of every link explicitly with compatible formats. Identical formats on the two ends of a link are guaranteed to be compatible. Drivers are free to accept different formats matching device requirements as being compatible.

Table 4-18 shows a sample configuration sequence for the pipeline described in Figure 4-4 (table columns list entity names and pad numbers).

**Table 4-18. Sample Pipeline Configuration**

| | Sensor/0 | Frontend/0 | Frontend/1 | Scaler/0 | Scaler/1 |
|---|---|---|---|---|---|
| Initial state | | 2048x1536 | | - | - |
| Configure frontend input | | 2048x1536 | | *2048x1536* | *2046x1534* |
| Configure scaler input | | 2048x1536 | | 2048x1536 | 2046x1534 |

| | Sensor/0 | Frontend/0 | Frontend/1 | Scaler/0 | Scaler/1 |
|---|---|---|---|---|---|
| Configure scaler output | | 2048x1536 | | 2048x1536 | 2046x1534 |

1. Initial state. The sensor output is set to its native 3MP resolution. Resolutions on the host frontend and scaler input and output pads are undefined.
2. The application configures the frontend input pad resolution to 2048x1536. The driver propagates the format to the frontend output pad. Note that the propagated output format can be different, as in this case, than the input format, as the hardware might need to crop pixels (for instance when converting a Bayer filter pattern to RGB or YUV).
3. The application configures the scaler input pad resolution to 2046x1534 to match the frontend output resolution. The driver propagates the format to the scaler output pad.
4. The application configures the scaler output pad resolution to 1280x960.

When satisfied with the try results, applications can set the active formats by setting the *which* argument to V4L2_SUBDEV_FORMAT_TRY. Active formats are changed exactly as try formats by drivers. To avoid modifying the hardware state during format negotiation, applications should negotiate try formats first and then modify the active settings using the try formats returned during the last negotiation iteration. This guarantees that the active format will be applied as-is by the driver without being modified.

### 4.13.3.2. Cropping and scaling

Many sub-devices support cropping frames on their input or output pads (or possible even on both). Cropping is used to select the area of interest in an image, typically on a video sensor or video decoder. It can also be used as part of digital zoom implementations to select the area of the image that will be scaled up.

Crop settings are defined by a crop rectangle and represented in a struct v4l2_rect by the coordinates of the top left corner and the rectangle size. Both the coordinates and sizes are expressed in pixels.

The crop rectangle is retrieved and set using the VIDIOC_SUBDEV_G_CROP and VIDIOC_SUBDEV_S_CROP ioctls. Like for pad formats, drivers store try and active crop rectangles. The format negotiation mechanism applies to crop settings as well.

On input pads, cropping is applied relatively to the current pad format. The pad format represents the image size as received by the sub-device from the previous block in the pipeline, and the crop rectangle represents the sub-image that will be transmitted further inside the sub-device for processing. The crop rectangle be entirely containted inside the input image size.

Input crop rectangle are reset to their default value when the input image format is modified. Drivers should use the input image size as the crop rectangle default value, but hardware requirements may prevent this.

Cropping behaviour on output pads is not defined.

## 4.13.4. Media Bus Formats

**Table 4-19. struct v4l2_mbus_framefmt**

| | | |
|---|---|---|
| __u32 | *width* | Image width, in pixels. |
| __u32 | *height* | Image height, in pixels. |
| __u32 | *code* | Format code, from enum v4l2_mbus_pixelcode. |
| __u32 | *field* | Field order, from enum v4l2_field. See Section 3.6 for details. |
| __u32 | *colorspace* | Image colorspace, from enum v4l2_colorspace. See Section 2.4 for details. |
| __u32 | *reserved*[7] | Reserved for future extensions. Applications and drivers must set the array to zero. |

### 4.13.4.1. Media Bus Pixel Codes

The media bus pixel codes describe image formats as flowing over physical busses (both between separate physical components and inside SoC devices). This should not be confused with the V4L2 pixel formats that describe, using four character codes, image formats as stored in memory.

While there is a relationship between image formats on busses and image formats in memory (a raw Bayer image won't be magically converted to JPEG just by storing it to memory), there is no one-to-one correspondance between them.

#### 4.13.4.1.1. Packed RGB Formats

Those formats transfer pixel data as red, green and blue components. The format code is made of the following information.

- The red, green and blue components order code, as encoded in a pixel sample. Possible values are RGB and BGR.
- The number of bits per component, for each component. The values can be different for all components. Common values are 555 and 565.
- The number of bus samples per pixel. Pixels that are wider than the bus width must be transferred in multiple samples. Common values are 1 and 2.
- The bus width.
- For formats where the total number of bits per pixel is smaller than the number of bus samples per pixel times the bus width, a padding value stating if the bytes are padded in their most high order bits (PADHI) or low order bits (PADLO).
- For formats where the number of bus samples per pixel is larger than 1, an endianness value stating if the pixel is transferred MSB first (BE) or LSB first (LE).

For instance, a format where pixels are encoded as 5-bits red, 5-bits green and 5-bit blue values padded on the high bit, transferred as 2 8-bit samples per pixel with the most significant bits (padding, red and half of the green value) transferred first will be named `V4L2_MBUS_FMT_RGB555_2X8_PADHI_BE`.

The following tables list existing packet RGB formats.

**Table 4-20. RGB formats**

| Identifier | Code | Data organization | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| V4L2_MBUS_FMT_RGB444_2X8_PADHI_BE | 0x1001 | 0 | 0 | 0 | 0 | $r_3$ | $r_2$ | $r_1$ | $r_0$ |
| | | $g_3$ | $g_2$ | $g_1$ | $g_0$ | $b_3$ | $b_2$ | $b_1$ | $b_0$ |
| V4L2_MBUS_FMT_RGB444_2X8_PADHI_LE | 0x1002 | 0 | 0 | 0 | 0 | $b_3$ | $b_2$ | $b_1$ | $b_0$ |
| | | 0 | 0 | 0 | 0 | $r_3$ | $r_2$ | $r_1$ | $r_0$ |

| Identifier | Code | Data organization | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| V4L2_MBUS_FMT_RGB555_2X8_PADHI_BE | 0x1003 | g2 | g1 | g0 | b4 | b3 | b2 | b1 | b0 |
| V4L2_MBUS_FMT_RGB555_2X8_PADHI_LE | 0x1004 | g2 | g1 | g0 | b4 | b3 | b2 | b1 | b0 |
| V4L2_MBUS_FMT_BGR565_2X8_BE | 0x1005 | 0 | r4 | r3 | r2 | r1 | r0 | g4 | g3 |
| V4L2_MBUS_FMT_BGR565_2X8_LE | 0x1006 | g2 | g1 | g0 | r4 | r3 | r2 | r1 | r0 |
| V4L2_MBUS_FMT_RGB565_2X8_BE | 0x1007 | b4 | b3 | b2 | b1 | b0 | g5 | g4 | g3 |
| V4L2_MBUS_FMT_RGB565_2X8_LE | 0x1008 | g2 | g1 | g0 | b4 | b3 | b2 | b1 | b0 |
| | | r4 | r3 | r2 | r1 | r0 | g5 | g4 | g3 |

#### 4.13.4.1.2. Bayer Formats

Those formats transfer pixel data as red, green and blue components. The format code is made of the following information.

- The red, green and blue components order code, as encoded in a pixel sample. The possible values are shown in Figure 4-5.
- The number of bits per pixel component. All components are transferred on the same number of bits. Common values are 8, 10 and 12.
- If the pixel components are DPCM-compressed, a mention of the DPCM compression and the number of bits per compressed pixel component.

- The number of bus samples per pixel. Pixels that are wider than the bus width must be transferred in multiple samples. Common values are 1 and 2.
- The bus width.
- For formats where the total number of bits per pixel is smaller than the number of bus samples per pixel times the bus width, a padding value stating if the bytes are padded in their most high order bits (PADHI) or low order bits (PADLO).
- For formats where the number of bus samples per pixel is larger than 1, an endianness value stating if the pixel is transferred MSB first (BE) or LSB first (LE).

For instance, a format with uncompressed 10-bit Bayer components arranged in a red, green, green, blue pattern transferred as 2 8-bit samples per pixel with the least significant bits transferred first will be named `V4L2_MBUS_FMT_SRGGB10_2X8_PADHI_LE`.

**Figure 4-5. Bayer Patterns**


The following table lists existing packet Bayer formats. The data organization is given as an example for the first pixel only.

**Table 4-21. Bayer Formats**

| Identifier Code | Data organization | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Bit | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| V4L2_MBUS_FMT_SBGGR8_1X8 0x3001 | | | | | $b_7$ | $b_6$ | $b_5$ | $b_4$ | $b_3$ | $b_2$ | $b_1$ | $b_0$ |
| V4L2_MBUS_FMT_SGBRG8_1X8 0x3013 | | | | | $g_7$ | $g_6$ | $g_5$ | $g_4$ | $g_3$ | $g_2$ | $g_1$ | $g_0$ |
| V4L2_MBUS_FMT_SGRBG8_1X8 0x3002 | | | | | $g_7$ | $g_6$ | $g_5$ | $g_4$ | $g_3$ | $g_2$ | $g_1$ | $g_0$ |

| Identifier Code | Data organization | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Bit | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| V4L2_MBUS_FMT_SRGGB8_1X8 0x3014 | | | | | $r_7$ | $r_6$ | $r_5$ | $r_4$ | $r_3$ | $r_2$ | $r_1$ | $r_0$ |
| V4L2_MBUS_FMT_SBGGR10_DPCM8_1X8 0x300b | | | | | $b_7$ | $b_6$ | $b_5$ | $b_4$ | $b_3$ | $b_2$ | $b_1$ | $b_0$ |
| V4L2_MBUS_FMT_SGBRG10_DPCM8_1X8 0x300c | | | | | $g_7$ | $g_6$ | $g_5$ | $g_4$ | $g_3$ | $g_2$ | $g_1$ | $g_0$ |
| V4L2_MBUS_FMT_SGRBG10_DPCM8_1X8 0x3009 | | | | | $g_7$ | $g_6$ | $g_5$ | $g_4$ | $g_3$ | $g_2$ | $g_1$ | $g_0$ |
| V4L2_MBUS_FMT_SRGGB10_DPCM8_1X8 0x300d | | | | | $r_7$ | $r_6$ | $r_5$ | $r_4$ | $r_3$ | $r_2$ | $r_1$ | $r_0$ |
| V4L2_MBUS_FMT_SBGGR10_2X8_PADHI_BE 0x3003 | | | | | $b_9$ | $b_8$ | 0 | 0 | 0 | 0 | $b_9$ | $b_8$ |
| | - | - | - | - | $b_7$ | $b_6$ | $b_5$ | $b_4$ | $b_3$ | $b_2$ | $b_1$ | $b_0$ |
| V4L2_MBUS_FMT_SBGGR10_2X8_PADHI_LE 0x3004 | | | | | $b_9$ | $b_8$ | $b_7$ | $b_6$ | $b_5$ | $b_4$ | $b_3$ | $b_2$ |
| | - | - | - | - | 0 | 0 | 0 | 0 | 0 | 0 | $b_9$ | $b_8$ |
| V4L2_MBUS_FMT_SBGGR10_2X8_PADLO_BE 0x3005 | | | | | $b_1$ | $b_0$ | 0 | 0 | 0 | 0 | 0 | 0 |
| | - | - | - | - | $b_7$ | $b_6$ | $b_5$ | $b_4$ | $b_3$ | $b_2$ | $b_1$ | $b_0$ |
| V4L2_MBUS_FMT_SBGGR10_2X8_PADLO_LE 0x3006 | | | | | $b_9$ | $b_8$ | $b_7$ | $b_6$ | $b_5$ | $b_4$ | $b_3$ | $b_2$ |
| V4L2_MBUS_FMT_SBGGR10_1X10 0x3007 | | | | | $b_7$ | $b_6$ | $b_5$ | $b_4$ | $b_3$ | $b_2$ | $b_1$ | $b_0$ |
| V4L2_MBUS_FMT_SGBRG10_1X10 0x300e | | | | | $g_7$ | $g_6$ | $g_5$ | $g_4$ | $g_3$ | $g_2$ | $g_1$ | $g_0$ |

| Identifier Code | Bit | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| V4L2_MBUS_FMT_SGRBG10_1X10 0x300a | | | | | | $g_7$ | $g_6$ | $g_5$ | $g_4$ | $g_3$ | $g_2$ | $g_1$ | $g_0$ |
| V4L2_MBUS_FMT_SRGGB10_1X10 0x300f | | | | | | $r_7$ | $r_6$ | $r_5$ | $r_4$ | $r_3$ | $r_2$ | $r_1$ | $r_0$ |
| V4L2_MBUS_FMT_SBGGR12_1X12 0x3008 | | | | | | $b_7$ | $b_6$ | $b_5$ | $b_4$ | $b_3$ | $b_2$ | $b_1$ | $b_0$ |
| V4L2_MBUS_FMT_SGBRG12_1X12 0x3010 | | | | | | $g_7$ | $g_6$ | $g_5$ | $g_4$ | $g_3$ | $g_2$ | $g_1$ | $g_0$ |
| V4L2_MBUS_FMT_SGRBG12_1X12 0x3011 | | | | | | $g_7$ | $g_6$ | $g_5$ | $g_4$ | $g_3$ | $g_2$ | $g_1$ | $g_0$ |
| V4L2_MBUS_FMT_SRGGB12_1X12 0x3012 | | | | | | $r_7$ | $r_6$ | $r_5$ | $r_4$ | $r_3$ | $r_2$ | $r_1$ | $r_0$ |

#### 4.13.4.1.3. Packed YUV Formats

Those data formats transfer pixel data as (possibly downsampled) Y, U and V components. The format code is made of the following information.

- The Y, U and V components order code, as transferred on the bus. Possible values are YUYV, UYVY, YVYU and VYUY.
- The number of bits per pixel component. All components are transferred on the same number of bits. Common values are 8, 10 and 12.
- The number of bus samples per pixel. Pixels that are wider than the bus width must be transferred in multiple samples. Common values are 1, 1.5 (encoded as 1_5) and 2.
- The bus width. When the bus width is larger than the number of bits per pixel component, several components are packed in a single bus sample. The components are ordered as specified by the order code, with components on the left of the code transferred in the high order bits. Common values are 8 and 16.

For instance, a format where pixels are encoded as 8-bit YUV values downsampled to 4:2:2 and transferred as 2 8-bit bus samples per pixel in the U, Y, V, Y order will be named `V4L2_MBUS_FMT_UYVY8_2X8`.

The following table lisst existing packet YUV formats.

**Table 4-22. YUV Formats**

| Identifier Code | Data organization | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Bit | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| V4L2_MBUS_FMT_Y8_1X8 0x2001 | - | - | - | - | - | - | - | - | - | - | - | - | $y_7$ | $y_6$ | $y_5$ | $y_4$ | $y_3$ | $y_2$ | $y_1$ | $y_0$ |
| V4L2_MBUS_FMT_UYVY8_1_5X8 0x2002 | - | - | - | - | - | - | - | - | - | - | - | - | $u_7$ | $u_6$ | $u_5$ | $u_4$ | $u_3$ | $u_2$ | $u_1$ | $u_0$ |
| | - | - | - | - | - | - | - | - | - | - | - | - | $y_7$ | $y_6$ | $y_5$ | $y_4$ | $y_3$ | $y_2$ | $y_1$ | $y_0$ |
| | - | - | - | - | - | - | - | - | - | - | - | - | $y_7$ | $y_6$ | $y_5$ | $y_4$ | $y_3$ | $y_2$ | $y_1$ | $y_0$ |
| | - | - | - | - | - | - | - | - | - | - | - | - | $v_7$ | $v_6$ | $v_5$ | $v_4$ | $v_3$ | $v_2$ | $v_1$ | $v_0$ |
| | - | - | - | - | - | - | - | - | - | - | - | - | $y_7$ | $y_6$ | $y_5$ | $y_4$ | $y_3$ | $y_2$ | $y_1$ | $y_0$ |
| | - | - | - | - | - | - | - | - | - | - | - | - | $y_7$ | $y_6$ | $y_5$ | $y_4$ | $y_3$ | $y_2$ | $y_1$ | $y_0$ |
| V4L2_MBUS_FMT_VYUY8_1_5X8 0x2003 | - | - | - | - | - | - | - | - | - | - | - | - | $v_7$ | $v_6$ | $v_5$ | $v_4$ | $v_3$ | $v_2$ | $v_1$ | $v_0$ |
| | - | - | - | - | - | - | - | - | - | - | - | - | $y_7$ | $y_6$ | $y_5$ | $y_4$ | $y_3$ | $y_2$ | $y_1$ | $y_0$ |
| | - | - | - | - | - | - | - | - | - | - | - | - | $y_7$ | $y_6$ | $y_5$ | $y_4$ | $y_3$ | $y_2$ | $y_1$ | $y_0$ |
| | - | - | - | - | - | - | - | - | - | - | - | - | $u_7$ | $u_6$ | $u_5$ | $u_4$ | $u_3$ | $u_2$ | $u_1$ | $u_0$ |
| | - | - | - | - | - | - | - | - | - | - | - | - | $y_7$ | $y_6$ | $y_5$ | $y_4$ | $y_3$ | $y_2$ | $y_1$ | $y_0$ |
| | - | - | - | - | - | - | - | - | - | - | - | - | $y_7$ | $y_6$ | $y_5$ | $y_4$ | $y_3$ | $y_2$ | $y_1$ | $y_0$ |

| Identifier Code | Data organization | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Bit | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| V4L2_MBUS_FMT_YUYV8_1_5X8 0x2004 | - | - | - | - | - | - | - | - | - | - | - | - | $y_7$ | $y_6$ | $y_5$ | $y_4$ | $y_3$ | $y_2$ | $y_1$ | $y_0$ |
| | - | - | - | - | - | - | - | - | - | - | - | - | $y_7$ | $y_6$ | $y_5$ | $y_4$ | $y_3$ | $y_2$ | $y_1$ | $y_0$ |
| | - | - | - | - | - | - | - | - | - | - | - | - | $u_7$ | $u_6$ | $u_5$ | $u_4$ | $u_3$ | $u_2$ | $u_1$ | $u_0$ |
| | - | - | - | - | - | - | - | - | - | - | - | - | $y_7$ | $y_6$ | $y_5$ | $y_4$ | $y_3$ | $y_2$ | $y_1$ | $y_0$ |
| | - | - | - | - | - | - | - | - | - | - | - | - | $y_7$ | $y_6$ | $y_5$ | $y_4$ | $y_3$ | $y_2$ | $y_1$ | $y_0$ |
| | - | - | - | - | - | - | - | - | - | - | - | - | $v_7$ | $v_6$ | $v_5$ | $v_4$ | $v_3$ | $v_2$ | $v_1$ | $v_0$ |
| V4L2_MBUS_FMT_YVYU8_1_5X8 0x2005 | - | - | - | - | - | - | - | - | - | - | - | - | $y_7$ | $y_6$ | $y_5$ | $y_4$ | $y_3$ | $y_2$ | $y_1$ | $y_0$ |
| | - | - | - | - | - | - | - | - | - | - | - | - | $y_7$ | $y_6$ | $y_5$ | $y_4$ | $y_3$ | $y_2$ | $y_1$ | $y_0$ |
| | - | - | - | - | - | - | - | - | - | - | - | - | $v_7$ | $v_6$ | $v_5$ | $v_4$ | $v_3$ | $v_2$ | $v_1$ | $v_0$ |
| | - | - | - | - | - | - | - | - | - | - | - | - | $y_7$ | $y_6$ | $y_5$ | $y_4$ | $y_3$ | $y_2$ | $y_1$ | $y_0$ |
| | - | - | - | - | - | - | - | - | - | - | - | - | $y_7$ | $y_6$ | $y_5$ | $y_4$ | $y_3$ | $y_2$ | $y_1$ | $y_0$ |
| | - | - | - | - | - | - | - | - | - | - | - | - | $u_7$ | $u_6$ | $u_5$ | $u_4$ | $u_3$ | $u_2$ | $u_1$ | $u_0$ |
| V4L2_MBUS_FMT_UYVY8_2X8 0x2006 | - | - | - | - | - | - | - | - | - | - | - | - | $u_7$ | $u_6$ | $u_5$ | $u_4$ | $u_3$ | $u_2$ | $u_1$ | $u_0$ |
| | - | - | - | - | - | - | - | - | - | - | - | - | $y_7$ | $y_6$ | $y_5$ | $y_4$ | $y_3$ | $y_2$ | $y_1$ | $y_0$ |
| | - | - | - | - | - | - | - | - | - | - | - | - | $v_7$ | $v_6$ | $v_5$ | $v_4$ | $v_3$ | $v_2$ | $v_1$ | $v_0$ |
| | - | - | - | - | - | - | - | - | - | - | - | - | $y_7$ | $y_6$ | $y_5$ | $y_4$ | $y_3$ | $y_2$ | $y_1$ | $y_0$ |

| Identifier Code | Data organization | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Bit | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| V4L2_MBUS_FMT_VYUY8_2X8 0x2007 | - | - | - | - | - | - | - | - | - | - | - | - | $V_{7}$ | $V_{6}$ | $V_{5}$ | $V_{4}$ | $V_{3}$ | $V_{2}$ | $V_{1}$ | $V_{0}$ |
| | - | - | - | - | - | - | - | - | - | - | - | - | $Y_{7}$ | $Y_{6}$ | $Y_{5}$ | $Y_{4}$ | $Y_{3}$ | $Y_{2}$ | $Y_{1}$ | $Y_{0}$ |
| | - | - | - | - | - | - | - | - | - | - | - | - | $U_{7}$ | $U_{6}$ | $U_{5}$ | $U_{4}$ | $U_{3}$ | $U_{2}$ | $U_{1}$ | $U_{0}$ |
| | - | - | - | - | - | - | - | - | - | - | - | - | $Y_{7}$ | $Y_{6}$ | $Y_{5}$ | $Y_{4}$ | $Y_{3}$ | $Y_{2}$ | $Y_{1}$ | $Y_{0}$ |
| V4L2_MBUS_FMT_YUYV8_2X8 0x2008 | - | - | - | - | - | - | - | - | - | - | - | - | $Y_{7}$ | $Y_{6}$ | $Y_{5}$ | $Y_{4}$ | $Y_{3}$ | $Y_{2}$ | $Y_{1}$ | $Y_{0}$ |
| | - | - | - | - | - | - | - | - | - | - | - | - | $U_{7}$ | $U_{6}$ | $U_{5}$ | $U_{4}$ | $U_{3}$ | $U_{2}$ | $U_{1}$ | $U_{0}$ |
| | - | - | - | - | - | - | - | - | - | - | - | - | $Y_{7}$ | $Y_{6}$ | $Y_{5}$ | $Y_{4}$ | $Y_{3}$ | $Y_{2}$ | $Y_{1}$ | $Y_{0}$ |
| | - | - | - | - | - | - | - | - | - | - | - | - | $V_{7}$ | $V_{6}$ | $V_{5}$ | $V_{4}$ | $V_{3}$ | $V_{2}$ | $V_{1}$ | $V_{0}$ |
| V4L2_MBUS_FMT_YVYU8_2X8 0x2009 | - | - | - | - | - | - | - | - | - | - | - | - | $Y_{7}$ | $Y_{6}$ | $Y_{5}$ | $Y_{4}$ | $Y_{3}$ | $Y_{2}$ | $Y_{1}$ | $Y_{0}$ |
| | - | - | - | - | - | - | - | - | - | - | - | - | $V_{7}$ | $V_{6}$ | $V_{5}$ | $V_{4}$ | $V_{3}$ | $V_{2}$ | $V_{1}$ | $V_{0}$ |
| | - | - | - | - | - | - | - | - | - | - | - | - | $Y_{7}$ | $Y_{6}$ | $Y_{5}$ | $Y_{4}$ | $Y_{3}$ | $Y_{2}$ | $Y_{1}$ | $Y_{0}$ |
| | - | - | - | - | - | - | - | - | - | - | - | - | $U_{7}$ | $U_{6}$ | $U_{5}$ | $U_{4}$ | $U_{3}$ | $U_{2}$ | $U_{1}$ | $U_{0}$ |
| V4L2_MBUS_FMT_Y10_1X10 0x200a | - | - | - | - | - | - | - | - | - | - | $Y_{9}$ | $Y_{8}$ | $Y_{7}$ | $Y_{6}$ | $Y_{5}$ | $Y_{4}$ | $Y_{3}$ | $Y_{2}$ | $Y_{1}$ | $Y_{0}$ |
| V4L2_MBUS_FMT_YUYV10_2X10 0x200b | - | - | - | - | - | - | - | - | - | - | $Y_{9}$ | $Y_{8}$ | $Y_{7}$ | $Y_{6}$ | $Y_{5}$ | $Y_{4}$ | $Y_{3}$ | $Y_{2}$ | $Y_{1}$ | $Y_{0}$ |
| | - | - | - | - | - | - | - | - | - | - | $U_{9}$ | $U_{8}$ | $U_{7}$ | $U_{6}$ | $U_{5}$ | $U_{4}$ | $U_{3}$ | $U_{2}$ | $U_{1}$ | $U_{0}$ |

| Identifier Code | Bit 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | - | - | - | - | - | - | - | - | - | - | $y_9$ | $y_8$ | $y_7$ | $y_6$ | $y_5$ | $y_4$ | $y_3$ | $y_2$ | $y_1$ | $y_0$ |
| | - | - | - | - | - | - | - | - | - | - | $v_9$ | $v_8$ | $v_7$ | $v_6$ | $v_5$ | $v_4$ | $v_3$ | $v_2$ | $v_1$ | $v_0$ |
| V4L2_MBUS_FMT_YVYU10_2X10 0x200c | - | - | - | - | - | - | - | - | - | - | $y_9$ | $y_8$ | $y_7$ | $y_6$ | $y_5$ | $y_4$ | $y_3$ | $y_2$ | $y_1$ | $y_0$ |
| | - | - | - | - | - | - | - | - | - | - | $v_9$ | $v_8$ | $v_7$ | $v_6$ | $v_5$ | $v_4$ | $v_3$ | $v_2$ | $v_1$ | $v_0$ |
| | - | - | - | - | - | - | - | - | - | - | $y_9$ | $y_8$ | $y_7$ | $y_6$ | $y_5$ | $y_4$ | $y_3$ | $y_2$ | $y_1$ | $y_0$ |
| | - | - | - | - | - | - | - | - | - | - | $u_9$ | $u_8$ | $u_7$ | $u_6$ | $u_5$ | $u_4$ | $u_3$ | $u_2$ | $u_1$ | $u_0$ |
| V4L2_MBUS_FMT_Y12_1X12 0x2013 | - | - | - | - | - | - | - | - | $y_{11}$ | $y_{10}$ | $y_9$ | $y_8$ | $y_7$ | $y_6$ | $y_5$ | $y_4$ | $y_3$ | $y_2$ | $y_1$ | $y_0$ |
| V4L2_MBUS_FMT_UYVY8_1X16 0x200f | $v_7$ | $v_6$ | $v_5$ | $v_4$ | $v_3$ | $v_2$ | $v_1$ | $v_0$ | $u_7$ | $u_6$ | $u_5$ | $u_4$ | $u_3$ | $u_2$ | $u_1$ | $u_0$ | $y_7$ | $y_6$ | $y_5$ | $y_4$ | $y_3$ | $y_2$ | $y_1$ | $y_0$ |
| | - | - | - | - | $v_7$ | $v_6$ | $v_5$ | $v_4$ | $v_3$ | $v_2$ | $v_1$ | $v_0$ | $y_7$ | $y_6$ | $y_5$ | $y_4$ | $y_3$ | $y_2$ | $y_1$ | $y_0$ |
| V4L2_MBUS_FMT_VYUY8_1X16 0x2010 | $v_7$ | $v_6$ | $v_5$ | $v_4$ | $v_3$ | $v_2$ | $v_1$ | $v_0$ | $u_7$ | $u_6$ | $u_5$ | $u_4$ | $u_3$ | $u_2$ | $u_1$ | $u_0$ | $y_7$ | $y_6$ | $y_5$ | $y_4$ | $y_3$ | $y_2$ | $y_1$ | $y_0$ |
| | - | - | - | - | $u_7$ | $u_6$ | $u_5$ | $u_4$ | $u_3$ | $u_2$ | $u_1$ | $u_0$ | $y_7$ | $y_6$ | $y_5$ | $y_4$ | $y_3$ | $y_2$ | $y_1$ | $y_0$ |
| V4L2_MBUS_FMT_YUYV8_1X16 0x2011 | $y_7$ | $y_6$ | $y_5$ | $y_4$ | $y_3$ | $y_2$ | $y_1$ | $y_0$ | $u_7$ | $u_6$ | $u_5$ | $u_4$ | $u_3$ | $u_2$ | $u_1$ | $u_0$ | $v_7$ | $v_6$ | $v_5$ | $v_4$ | $v_3$ | $v_2$ | $v_1$ | $v_0$ |
| | - | - | - | - | $y_7$ | $y_6$ | $y_5$ | $y_4$ | $y_3$ | $y_2$ | $y_1$ | $y_0$ | $v_7$ | $v_6$ | $v_5$ | $v_4$ | $v_3$ | $v_2$ | $v_1$ | $v_0$ |
| V4L2_MBUS_FMT_YVYU8_1X16 0x2012 | $y_7$ | $y_6$ | $y_5$ | $y_4$ | $y_3$ | $y_2$ | $y_1$ | $y_0$ | $v_7$ | $v_6$ | $v_5$ | $v_4$ | $v_3$ | $v_2$ | $v_1$ | $v_0$ | $u_7$ | $u_6$ | $u_5$ | $u_4$ | $u_3$ | $u_2$ | $u_1$ | $u_0$ |

| Identifier | Code | Data organization | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Bit | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| | | - | - | - | - | $y_7$ | $y_6$ | $y_5$ | $y_4$ | $y_3$ | $y_2$ | $y_1$ | $y_0$ | $u_7$ | $u_6$ | $u_5$ | $u_4$ | $u_3$ | $u_2$ | $u_1$ | $u_0$ |
| V4L2_MBUS_FMT_YUYV10_1X20 0x200d | | $y_9$ | $y_8$ | $y_7$ | $y_6$ | $y_5$ | $y_4$ | $y_3$ | $y_2$ | $y_1$ | $y_0$ | $u_9$ | $u_8$ | $u_7$ | $u_6$ | $u_5$ | $u_4$ | $u_3$ | $u_2$ | $u_1$ | $u_0$ |
| | | $y_9$ | $y_8$ | $y_7$ | $y_6$ | $y_5$ | $y_4$ | $y_3$ | $y_2$ | $y_1$ | $y_0$ | $v_9$ | $v_8$ | $v_7$ | $v_6$ | $v_5$ | $v_4$ | $v_3$ | $v_2$ | $v_1$ | $v_0$ |
| V4L2_MBUS_FMT_YVYU10_1X20 0x200e | | $y_9$ | $y_8$ | $y_7$ | $y_6$ | $y_5$ | $y_4$ | $y_3$ | $y_2$ | $y_1$ | $y_0$ | $v_9$ | $v_8$ | $v_7$ | $v_6$ | $v_5$ | $v_4$ | $v_3$ | $v_2$ | $v_1$ | $v_0$ |
| | | $y_9$ | $y_8$ | $y_7$ | $y_6$ | $y_5$ | $y_4$ | $y_3$ | $y_2$ | $y_1$ | $y_0$ | $u_9$ | $u_8$ | $u_7$ | $u_6$ | $u_5$ | $u_4$ | $u_3$ | $u_2$ | $u_1$ | $u_0$ |

#### 4.13.4.1.4. JPEG Compressed Formats

Those data formats consist of an ordered sequence of 8-bit bytes obtained from JPEG compression process. Additionally to the `_JPEG` prefix the format code is made of the following information.

- The number of bus samples per entropy encoded byte.
- The bus width.

For instance, for a JPEG baseline process and an 8-bit bus width the format will be named `V4L2_MBUS_FMT_JPEG_1X8`.

The following table lists existing JPEG compressed formats.

**Table 4-23. JPEG Formats**

| Identifier | Code | Remarks |
|---|---|---|

| Identifier | Code | Remarks |
|---|---|---|
| V4L2_MBUS_FMT_JPEG_1X8 | 0x4001 | Besides of its usage for the parallel bus this format is recommended for transmission of JPEG data over MIPI CSI bus using the User Defined 8-bit Data types. |

# Notes

1. A common application of two file descriptors is the XFree86 Xv/V4L interface driver and a V4L2 application. While the X server controls video overlay, the application can take advantage of memory mapping and DMA.

   In the opinion of the designers of this API, no driver writer taking the efforts to support simultaneous capturing and overlay will restrict this ability by requiring a single file descriptor, as in V4L and earlier versions of V4L2. Making this optional means applications depending on two file descriptors need backup routines to be compatible with all drivers, which is considerable more work than using two fds in applications which do not. Also two fd's fit the general concept of one file descriptor for each logical stream. Hence as a complexity trade-off drivers *must* support two file descriptors and *may* support single fd operation.

2. The X Window system defines "regions" which are vectors of struct BoxRec { short x1, y1, x2, y2; } with width = x2 - x1 and height = y2 - y1, so one cannot pass X11 clip lists directly.

3. ASK: Amplitude-Shift Keying. A high signal level represents a '1' bit, a low level a '0' bit.

# Chapter 5. V4L2 Driver Programming

to do

# Chapter 6. Libv4l Userspace Library

## 6.1. Introduction

libv4l is a collection of libraries which adds a thin abstraction layer on top of video4linux2 devices. The purpose of this (thin) layer is to make it easy for application writers to support a wide variety of devices without having to write separate code for different devices in the same class.

An example of using libv4l is provided by v4l2grab.

libv4l consists of 3 different libraries:

### 6.1.1. libv4lconvert

libv4lconvert is a library that converts several different pixelformats found in V4L2 drivers into a few common RGB and YUY formats.

It currently accepts the following V4L2 driver formats: `V4L2_PIX_FMT_BGR24`, `V4L2_PIX_FMT_HM12`, `V4L2_PIX_FMT_JPEG`, `V4L2_PIX_FMT_MJPEG`, `V4L2_PIX_FMT_MR97310A`, `V4L2_PIX_FMT_OV511`, `V4L2_PIX_FMT_OV518`, `V4L2_PIX_FMT_PAC207`, `V4L2_PIX_FMT_PJPG`, `V4L2_PIX_FMT_RGB24`, `V4L2_PIX_FMT_SBGGR8`, `V4L2_PIX_FMT_SGBRG8`, `V4L2_PIX_FMT_SGRBG8`, `V4L2_PIX_FMT_SN9C10X`, `V4L2_PIX_FMT_SN9C20X_I420`, `V4L2_PIX_FMT_SPCA501`, `V4L2_PIX_FMT_SPCA505`, `V4L2_PIX_FMT_SPCA508`, `V4L2_PIX_FMT_SPCA561`, `V4L2_PIX_FMT_SQ905C`, `V4L2_PIX_FMT_SRGGB8`, `V4L2_PIX_FMT_UYVY`, `V4L2_PIX_FMT_YUV420`, `V4L2_PIX_FMT_YUYV`, `V4L2_PIX_FMT_YVU420`, and `V4L2_PIX_FMT_YVYU`.

Later on libv4lconvert was expanded to also be able to do various video processing functions to improve webcam video quality. The video processing is split in to 2 parts: libv4lconvert/control and libv4lconvert/processing.

The control part is used to offer video controls which can be used to control the video processing functions made available by libv4lconvert/processing. These controls are stored application wide (until reboot) by using a persistent shared memory object.

libv4lconvert/processing offers the actual video processing functionality.

### 6.1.2. libv4l1

This library offers functions that can be used to quickly make v4l1 applications work with v4l2 devices. These functions work exactly like the normal open/close/etc, except that libv4l1 does full emulation of the v4l1 api on top of v4l2 drivers, in case of v4l1 drivers it will just pass calls through.

Since those functions are emulations of the old V4L1 API, it shouldn't be used for new applications.

### 6.1.3. libv4l2

This library should be used for all modern V4L2 applications.

It provides handles to call V4L2 open/ioctl/close/poll methods. Instead of just providing the raw output of the device, it enhances the calls in the sense that it will use libv4lconvert to provide more video formats and to enhance the image quality.

In most cases, libv4l2 just passes the calls directly through to the v4l2 driver, intercepting the calls to `VIDIOC_TRY_FMT`, `VIDIOC_G_FMT` `VIDIOC_S_FMT` `VIDIOC_ENUM_FRAMESIZES` and `VIDIOC_ENUM_FRAMEINTERVALS` in order to emulate the formats `V4L2_PIX_FMT_BGR24`, `V4L2_PIX_FMT_RGB24`, `V4L2_PIX_FMT_YUV420`, and `V4L2_PIX_FMT_YVU420`, if they aren't available in the driver. `VIDIOC_ENUM_FMT` keeps enumerating the hardware supported formats, plus the emulated formats offered by libv4l at the end.

#### 6.1.3.1. Libv4l device control functions

The common file operation methods are provided by libv4l.

Those functions operate just like glibc open/close/dup/ioctl/read/mmap/munmap:

- int v4l2_open(const char *file, int oflag, ...) - operates like the standard open() function.
- int v4l2_close(int fd) - operates like the standard close() function.
- int v4l2_dup(int fd) - operates like the standard dup() function, duplicating a file handler.
- int v4l2_ioctl (int fd, unsigned long int request, ...) - operates like the standard ioctl() function.
- int v4l2_read (int fd, void* buffer, size_t n) - operates like the standard read() function.
- void v4l2_mmap(void *start, size_t length, int prot, int flags, int fd, int64_t offset); - operates like the standard mmap() function.

- int v4l2_munmap(void *_start, size_t length); - operates like the standard munmap() function.

Those functions provide additional control:

- int v4l2_fd_open(int fd, int v4l2_flags) - opens an already opened fd for further use through v4l2lib and possibly modify libv4l2's default behavior through the v4l2_flags argument. Currently, v4l2_flags can be `V4L2_DISABLE_CONVERSION`, to disable format conversion.
- int v4l2_set_control(int fd, int cid, int value) - This function takes a value of 0 - 65535, and then scales that range to the actual range of the given v4l control id, and then if the cid exists and is not locked sets the cid to the scaled value.
- int v4l2_get_control(int fd, int cid) - This function returns a value of 0 - 65535, scaled to from the actual range of the given v4l control id. when the cid does not exist, could not be accessed for some reason, or some error occurred 0 is returned.

### 6.1.4. v4l1compat.so wrapper library

This library intercepts calls to open/close/ioctl/mmap/mmunmap operations and redirects them to the libv4l counterparts, by using LD_PRELOAD=/usr/lib/v4l1compat.so. It also emulates V4L1 calls via V4L2 API.

It allows usage of binary legacy applications that still don't use libv4l.

# Chapter 7. Changes

The following chapters document the evolution of the V4L2 API, errata or extensions. They are also intended to help application and driver writers to port or update their code.

## 7.1. Differences between V4L and V4L2

The Video For Linux API was first introduced in Linux 2.1 to unify and replace various TV and radio device related interfaces, developed independently by driver writers in prior years. Starting with Linux 2.5 the much improved V4L2 API replaces the V4L API, although existing drivers will continue to support V4L applications in the future, either directly or through the V4L2 compatibility layer in the `videodev` kernel module translating ioctls on the fly. For a transition period not all drivers will support the V4L2 API.

### 7.1.1. Opening and Closing Devices

For compatibility reasons the character device file names recommended for V4L2 video capture, overlay, radio and raw vbi capture devices did not change from those used by V4L. They are listed in Chapter 4 and below in Table 7-1.

The teletext devices (minor range 192-223) have been removed in V4L2 and no longer exist. There is no hardware available anymore for handling pure teletext. Instead raw or sliced VBI is used.

The V4L `videodev` module automatically assigns minor numbers to drivers in load order, depending on the registered device type. We recommend that V4L2 drivers by default register devices with the same numbers, but the system administrator can assign arbitrary minor numbers using driver module options. The major device number remains 81.

**Table 7-1. V4L Device Types, Names and Numbers**

| Device Type | File Name | Minor Numbers |
|---|---|---|
| Video capture and overlay | `/dev/video` and `/dev/bttv0`[a], `/dev/video0` to `/dev/video63` | 0-63 |

| Device Type | File Name | Minor Numbers |
|---|---|---|
| Radio receiver | /dev/radio[b], /dev/radio0 to /dev/radio63 | 64-127 |
| Raw VBI capture | /dev/vbi, /dev/vbi0 to /dev/vbi31 | 224-255 |
| Notes:<br>a. According to Documentation/devices.txt these should be symbolic links to /dev/video0. Note the original bttv interface is not compatible with V4L or V4L2.<br>b. According to Documentation/devices.txt a symbolic link to /dev/radio0. | | |

V4L prohibits (or used to prohibit) multiple opens of a device file. V4L2 drivers *may* support multiple opens, see Section 1.1 for details and consequences.

V4L drivers respond to V4L2 ioctls with an EINVAL error code. The compatibility layer in the V4L2 videodev module can translate V4L ioctl requests to their V4L2 counterpart, however a V4L2 driver usually needs more preparation to become fully V4L compatible. This is covered in more detail in Chapter 5.

## 7.1.2. Querying Capabilities

The V4L VIDIOCGCAP ioctl is equivalent to V4L2's VIDIOC_QUERYCAP.

The *name* field in struct video_capability became *card* in struct v4l2_capability, *type* was replaced by *capabilities*. Note V4L2 does not distinguish between device types like this, better think of basic video input, video output and radio devices supporting a set of related functions like video capturing, video overlay and VBI capturing. See Section 1.1 for an introduction.

| struct video_capability *type* | struct v4l2_capability *capabilities* flags | Purpose |
|---|---|---|
| VID_TYPE_CAPTURE | V4L2_CAP_VIDEO_CAPTURE | The video capture interface is supported. |
| VID_TYPE_TUNER | V4L2_CAP_TUNER | The device has a tuner or modulator. |
| VID_TYPE_TELETEXT | V4L2_CAP_VBI_CAPTURE | The raw VBI capture interface is supported. |
| VID_TYPE_OVERLAY | V4L2_CAP_VIDEO_OVERLAY | The video overlay interface is supported. |

| struct video_capability *type* | struct v4l2_capability *capabilities* flags | Purpose |
|---|---|---|
| VID_TYPE_CHROMAKEY | V4L2_FBUF_CAP_CHROMAKEY in field *capability* of struct v4l2_framebuffer | Whether chromakey overlay is supported. For more information on overlay see Section 4.2. |
| VID_TYPE_CLIPPING | V4L2_FBUF_CAP_LIST_CLIPPING and V4L2_FBUF_CAP_BITMAP_CLIPPING in field *capability* of struct v4l2_framebuffer | Whether clipping the overlaid image is supported, see Section 4.2. |
| VID_TYPE_FRAMERAM | V4L2_FBUF_CAP_EXTERNOVERLAY *not set* in field *capability* of struct v4l2_framebuffer | Whether overlay overwrites frame buffer memory, see Section 4.2. |
| VID_TYPE_SCALES | – | This flag indicates if the hardware can scale images. The V4L2 API implies the scale factor by setting the cropping dimensions and image size with the VIDIOC_S_CROP and VIDIOC_S_FMT ioctl, respectively. The driver returns the closest sizes possible. For more information on cropping and scaling see Section 1.12. |
| VID_TYPE_MONOCHROME | – | Applications can enumerate the supported image formats with the VIDIOC_ENUM_FMT ioctl to determine if the device supports grey scale capturing only. For more information on image formats see Chapter 2. |

| struct video_capability *type* | struct v4l2_capability *capabilities* flags | Purpose |
|---|---|---|
| VID_TYPE_SUBCAPTURE | – | Applications can call the VIDIOC_G_CROP ioctl to determine if the device supports capturing a subsection of the full picture ("cropping" in V4L2). If not, the ioctl returns the EINVAL error code. For more information on cropping and scaling see Section 1.12. |
| VID_TYPE_MPEG_DECODER | – | Applications can enumerate the supported image formats with the VIDIOC_ENUM_FMT ioctl to determine if the device supports MPEG streams. |
| VID_TYPE_MPEG_ENCODER | – | See above. |
| VID_TYPE_MJPEG_DECODER | | See above. |
| VID_TYPE_MJPEG_ENCODER | | See above. |

The *audios* field was replaced by *capabilities* flag V4L2_CAP_AUDIO, indicating *if* the device has any audio inputs or outputs. To determine their number applications can enumerate audio inputs with the VIDIOC_G_AUDIO ioctl. The audio ioctls are described in Section 1.5.

The *maxwidth*, *maxheight*, *minwidth* and *minheight* fields were removed. Calling the VIDIOC_S_FMT or VIDIOC_TRY_FMT ioctl with the desired dimensions returns the closest size possible, taking into account the current video standard, cropping and scaling limitations.

## 7.1.3. Video Sources

V4L provides the VIDIOCGCHAN and VIDIOCSCHAN ioctl using struct video_channel to enumerate the video inputs of a V4L device. The equivalent V4L2

ioctls are VIDIOC_ENUMINPUT, VIDIOC_G_INPUT and VIDIOC_S_INPUT using struct v4l2_input as discussed in Section 1.4.

The *channel* field counting inputs was renamed to *index*, the video input types were renamed as follows:

| struct video_channel *type* | struct v4l2_input *type* |
|---|---|
| VIDEO_TYPE_TV | V4L2_INPUT_TYPE_TUNER |
| VIDEO_TYPE_CAMERA | V4L2_INPUT_TYPE_CAMERA |

Unlike the *tuners* field expressing the number of tuners of this input, V4L2 assumes each video input is connected to at most one tuner. However a tuner can have more than one input, i. e. RF connectors, and a device can have multiple tuners. The index number of the tuner associated with the input, if any, is stored in field *tuner* of struct v4l2_input. Enumeration of tuners is discussed in Section 1.6.

The redundant VIDEO_VC_TUNER flag was dropped. Video inputs associated with a tuner are of type V4L2_INPUT_TYPE_TUNER. The VIDEO_VC_AUDIO flag was replaced by the *audioset* field. V4L2 considers devices with up to 32 audio inputs. Each set bit in the *audioset* field represents one audio input this video input combines with. For information about audio inputs and how to switch between them see Section 1.5.

The *norm* field describing the supported video standards was replaced by *std*. The V4L specification mentions a flag VIDEO_VC_NORM indicating whether the standard can be changed. This flag was a later addition together with the *norm* field and has been removed in the meantime. V4L2 has a similar, albeit more comprehensive approach to video standards, see Section 1.7 for more information.

## 7.1.4. Tuning

The V4L VIDIOCGTUNER and VIDIOCSTUNER ioctl and struct video_tuner can be used to enumerate the tuners of a V4L TV or radio device. The equivalent V4L2 ioctls are VIDIOC_G_TUNER and VIDIOC_S_TUNER using struct v4l2_tuner. Tuners are covered in Section 1.6.

The *tuner* field counting tuners was renamed to *index*. The fields *name*, *rangelow* and *rangehigh* remained unchanged.

The VIDEO_TUNER_PAL, VIDEO_TUNER_NTSC and VIDEO_TUNER_SECAM flags indicating the supported video standards were dropped. This information is now contained in the associated struct v4l2_input. No replacement exists for the VIDEO_TUNER_NORM flag indicating whether the video standard can be switched. The *mode* field to select a different video standard was replaced by a whole new set

of ioctls and structures described in Section 1.7. Due to its ubiquity it should be mentioned the BTTV driver supports several standards in addition to the regular VIDEO_MODE_PAL (0), VIDEO_MODE_NTSC, VIDEO_MODE_SECAM and VIDEO_MODE_AUTO (3). Namely N/PAL Argentina, M/PAL, N/PAL, and NTSC Japan with numbers 3-6 (sic).

The VIDEO_TUNER_STEREO_ON flag indicating stereo reception became V4L2_TUNER_SUB_STEREO in field *rxsubchans*. This field also permits the detection of monaural and bilingual audio, see the definition of struct v4l2_tuner for details. Presently no replacement exists for the VIDEO_TUNER_RDS_ON and VIDEO_TUNER_MBS_ON flags.

The VIDEO_TUNER_LOW flag was renamed to V4L2_TUNER_CAP_LOW in the struct v4l2_tuner *capability* field.

The VIDIOCGFREQ and VIDIOCSFREQ ioctl to change the tuner frequency where renamed to VIDIOC_G_FREQUENCY and VIDIOC_S_FREQUENCY. They take a pointer to a struct v4l2_frequency instead of an unsigned long integer.

## 7.1.5. Image Properties

V4L2 has no equivalent of the VIDIOCGPICT and VIDIOCSPICT ioctl and struct video_picture. The following fields where replaced by V4L2 controls accessible with the VIDIOC_QUERYCTRL, VIDIOC_G_CTRL and VIDIOC_S_CTRL ioctls:

| struct video_picture | V4L2 Control ID |
|---|---|
| *brightness* | V4L2_CID_BRIGHTNESS |
| *hue* | V4L2_CID_HUE |
| *colour* | V4L2_CID_SATURATION |
| *contrast* | V4L2_CID_CONTRAST |
| *whiteness* | V4L2_CID_WHITENESS |

The V4L picture controls are assumed to range from 0 to 65535 with no particular reset value. The V4L2 API permits arbitrary limits and defaults which can be queried with the VIDIOC_QUERYCTRL ioctl. For general information about controls see Section 1.8.

The *depth* (average number of bits per pixel) of a video image is implied by the selected image format. V4L2 does not explicitely provide such information assuming applications recognizing the format are aware of the image depth and others need not know. The *palette* field moved into the struct v4l2_pix_format:

| struct video_picture *palette* | struct v4l2_pix_format *pixfmt* |
|---|---|
| VIDEO_PALETTE_GREY | V4L2_PIX_FMT_GREY |
| VIDEO_PALETTE_HI240 | V4L2_PIX_FMT_HI240[a] |
| VIDEO_PALETTE_RGB565 | V4L2_PIX_FMT_RGB565 |
| VIDEO_PALETTE_RGB555 | V4L2_PIX_FMT_RGB555 |
| VIDEO_PALETTE_RGB24 | V4L2_PIX_FMT_BGR24 |
| VIDEO_PALETTE_RGB32 | V4L2_PIX_FMT_BGR32[b] |
| VIDEO_PALETTE_YUV422 | V4L2_PIX_FMT_YUYV |
| VIDEO_PALETTE_YUYV[c] | V4L2_PIX_FMT_YUYV |
| VIDEO_PALETTE_UYVY | V4L2_PIX_FMT_UYVY |
| VIDEO_PALETTE_YUV420 | None |
| VIDEO_PALETTE_YUV411 | V4L2_PIX_FMT_Y41P[d] |
| VIDEO_PALETTE_RAW | None[e] |
| VIDEO_PALETTE_YUV422P | V4L2_PIX_FMT_YUV422P |
| VIDEO_PALETTE_YUV411P | V4L2_PIX_FMT_YUV411P[f] |
| VIDEO_PALETTE_YUV420P | V4L2_PIX_FMT_YVU420 |
| VIDEO_PALETTE_YUV410P | V4L2_PIX_FMT_YVU410 |

Notes:
a. This is a custom format used by the BTTV driver, not one of the V4L2 standard formats.
b. Presumably all V4L RGB formats are little-endian, although some drivers might interpret them according to machine endianess. V4L2 defines little-endian, big-endian and red/blue swapped variants. For details see Section 2.6.
c. VIDEO_PALETTE_YUV422 and VIDEO_PALETTE_YUYV are the same formats. Some V4L drivers respond to one, some to the other.
d. Not to be confused with V4L2_PIX_FMT_YUV411P, which is a planar format.
e. V4L explains this as: "RAW capture (BT848)"
f. Not to be confused with V4L2_PIX_FMT_Y41P, which is a packed format.

V4L2 image formats are defined in Chapter 2. The image format can be selected with the VIDIOC_S_FMT ioctl.

## 7.1.6. Audio

The VIDIOCGAUDIO and VIDIOCSAUDIO ioctl and struct video_audio are used to enumerate the audio inputs of a V4L device. The equivalent V4L2 ioctls are VIDIOC_G_AUDIO and VIDIOC_S_AUDIO using struct v4l2_audio as discussed in Section 1.5.

The *audio* "channel number" field counting audio inputs was renamed to *index*.

On VIDIOCSAUDIO the *mode* field selects *one* of the VIDEO_SOUND_MONO, VIDEO_SOUND_STEREO, VIDEO_SOUND_LANG1 or VIDEO_SOUND_LANG2 audio demodulation modes. When the current audio standard is BTSC VIDEO_SOUND_LANG2 refers to SAP and VIDEO_SOUND_LANG1 is meaningless. Also undocumented in the V4L specification, there is no way to query the selected mode. On VIDIOCGAUDIO the driver returns the *actually received* audio programmes in this field. In the V4L2 API this information is stored in the struct v4l2_tuner *rxsubchans* and *audmode* fields, respectively. See Section 1.6 for more information on tuners. Related to audio modes struct v4l2_audio also reports if this is a mono or stereo input, regardless if the source is a tuner.

The following fields where replaced by V4L2 controls accessible with the VIDIOC_QUERYCTRL, VIDIOC_G_CTRL and VIDIOC_S_CTRL ioctls:

| struct video_audio | V4L2 Control ID |
|---|---|
| *volume* | V4L2_CID_AUDIO_VOLUME |
| *bass* | V4L2_CID_AUDIO_BASS |
| *treble* | V4L2_CID_AUDIO_TREBLE |
| *balance* | V4L2_CID_AUDIO_BALANCE |

To determine which of these controls are supported by a driver V4L provides the *flags* VIDEO_AUDIO_VOLUME, VIDEO_AUDIO_BASS, VIDEO_AUDIO_TREBLE and VIDEO_AUDIO_BALANCE. In the V4L2 API the VIDIOC_QUERYCTRL ioctl reports if the respective control is supported. Accordingly the VIDEO_AUDIO_MUTABLE and VIDEO_AUDIO_MUTE flags where replaced by the boolean V4L2_CID_AUDIO_MUTE control.

All V4L2 controls have a *step* attribute replacing the struct video_audio *step* field. The V4L audio controls are assumed to range from 0 to 65535 with no particular reset value. The V4L2 API permits arbitrary limits and defaults which can be queried with the VIDIOC_QUERYCTRL ioctl. For general information about controls see Section 1.8.

## 7.1.7. Frame Buffer Overlay

The V4L2 ioctls equivalent to VIDIOCGFBUF and VIDIOCSFBUF are VIDIOC_G_FBUF and VIDIOC_S_FBUF. The *base* field of struct video_buffer remained unchanged, except V4L2 defines a flag to indicate non-destructive overlays instead of a NULL pointer. All other fields moved into the struct v4l2_pix_format *fmt* substructure of struct v4l2_framebuffer. The *depth* field was replaced by *pixelformat*. See Section 2.6 for a list of RGB formats and their respective color depths.

Instead of the special ioctls VIDIOCGWIN and VIDIOCSWIN V4L2 uses the general-purpose data format negotiation ioctls VIDIOC_G_FMT and VIDIOC_S_FMT. They take a pointer to a struct v4l2_format as argument. Here the *win* member of the *fmt* union is used, a struct v4l2_window.

The *x*, *y*, *width* and *height* fields of struct video_window moved into struct v4l2_rect substructure *w* of struct v4l2_window. The *chromakey*, *clips*, and *clipcount* fields remained unchanged. Struct video_clip was renamed to struct v4l2_clip, also containing a struct v4l2_rect, but the semantics are still the same.

The VIDEO_WINDOW_INTERLACE flag was dropped. Instead applications must set the *field* field to V4L2_FIELD_ANY or V4L2_FIELD_INTERLACED. The VIDEO_WINDOW_CHROMAKEY flag moved into struct v4l2_framebuffer, under the new name V4L2_FBUF_FLAG_CHROMAKEY.

In V4L, storing a bitmap pointer in *clips* and setting *clipcount* to VIDEO_CLIP_BITMAP (-1) requests bitmap clipping, using a fixed size bitmap of 1024 × 625 bits. Struct v4l2_window has a separate *bitmap* pointer field for this purpose and the bitmap size is determined by *w.width* and *w.height*.

The VIDIOCCAPTURE ioctl to enable or disable overlay was renamed to VIDIOC_OVERLAY.

## 7.1.8. Cropping

To capture only a subsection of the full picture V4L defines the VIDIOCGCAPTURE and VIDIOCSCAPTURE ioctls using struct video_capture. The equivalent V4L2 ioctls are VIDIOC_G_CROP and VIDIOC_S_CROP using struct v4l2_crop, and the related VIDIOC_CROPCAP ioctl. This is a rather complex matter, see Section 1.12 for details.

The *x*, *y*, *width* and *height* fields moved into struct v4l2_rect substructure *c* of struct v4l2_crop. The *decimation* field was dropped. In the V4L2 API the scaling factor is implied by the size of the cropping rectangle and the size of the captured or overlaid image.

The VIDEO_CAPTURE_ODD and VIDEO_CAPTURE_EVEN flags to capture only the odd or even field, respectively, were replaced by V4L2_FIELD_TOP and V4L2_FIELD_BOTTOM in the field named *field* of struct v4l2_pix_format and struct v4l2_window. These structures are used to select a capture or overlay format with the VIDIOC_S_FMT ioctl.

## 7.1.9. Reading Images, Memory Mapping

### 7.1.9.1. Capturing using the read method

There is no essential difference between reading images from a V4L or V4L2 device using the `read()` function, however V4L2 drivers are not required to support this I/O method. Applications can determine if the function is available with the `VIDIOC_QUERYCAP` ioctl. All V4L2 devices exchanging data with applications must support the `select()` and `poll()` functions.

To select an image format and size, V4L provides the `VIDIOCSPICT` and `VIDIOCSWIN` ioctls. V4L2 uses the general-purpose data format negotiation ioctls `VIDIOC_G_FMT` and `VIDIOC_S_FMT`. They take a pointer to a struct v4l2_format as argument, here the struct v4l2_pix_format named *`pix`* of its *`fmt`* union is used.

For more information about the V4L2 read interface see Section 3.1.

### 7.1.9.2. Capturing using memory mapping

Applications can read from V4L devices by mapping buffers in device memory, or more often just buffers allocated in DMA-able system memory, into their address space. This avoids the data copying overhead of the read method. V4L2 supports memory mapping as well, with a few differences.

| V4L | V4L2 |
|---|---|
| | The image format must be selected before buffers are allocated, with the `VIDIOC_S_FMT` ioctl. When no format is selected the driver may use the last, possibly by another application requested format. |
| Applications cannot change the number of buffers. The it is built into the driver, unless it has a module option to change the number when the driver module is loaded. | The `VIDIOC_REQBUFS` ioctl allocates the desired number of buffers, this is a required step in the initialization sequence. |

| V4L | V4L2 |
|---|---|
| Drivers map all buffers as one contiguous range of memory. The VIDIOCGMBUF ioctl is available to query the number of buffers, the offset of each buffer from the start of the virtual file, and the overall amount of memory used, which can be used as arguments for the mmap() function. | Buffers are individually mapped. The offset and size of each buffer can be determined with the VIDIOC_QUERYBUF ioctl. |
| The VIDIOCMCAPTURE ioctl prepares a buffer for capturing. It also determines the image format for this buffer. The ioctl returns immediately, eventually with an EAGAIN error code if no video signal had been detected. When the driver supports more than one buffer applications can call the ioctl multiple times and thus have multiple outstanding capture requests.<br>The VIDIOCSYNC ioctl suspends execution until a particular buffer has been filled. | Drivers maintain an incoming and outgoing queue. VIDIOC_QBUF enqueues any empty buffer into the incoming queue. Filled buffers are dequeued from the outgoing queue with the VIDIOC_DQBUF ioctl. To wait until filled buffers become available this function, select() or poll() can be used. The VIDIOC_STREAMON ioctl must be called once after enqueuing one or more buffers to start capturing. Its counterpart VIDIOC_STREAMOFF stops capturing and dequeues all buffers from both queues. Applications can query the signal status, if known, with the VIDIOC_ENUMINPUT ioctl. |

For a more in-depth discussion of memory mapping and examples, see Section 3.2.

## 7.1.10. Reading Raw VBI Data

Originally the V4L API did not specify a raw VBI capture interface, only the device file /dev/vbi was reserved for this purpose. The only driver supporting this interface was the BTTV driver, de-facto defining the V4L VBI interface. Reading from the device yields a raw VBI image with the following parameters:

| struct v4l2_vbi_format | V4L, BTTV driver |
|---|---|
| sampling_rate | 28636363 Hz NTSC (or any other 525-line standard); 35468950 Hz PAL and SECAM (625-line standards) |
| offset | ? |

| struct v4l2_vbi_format | V4L, BTTV driver |
|---|---|
| samples_per_line | 2048 |
| sample_format | V4L2_PIX_FMT_GREY. The last four bytes (a machine endianess integer) contain a frame counter. |
| start[] | 10, 273 NTSC; 22, 335 PAL and SECAM |
| count[] | 16, 16[a] |
| flags | 0 |
| Notes:<br>a. Old driver versions used different values, eventually the custom BTTV_VBISIZE ioctl was added to query the correct values. | |

Undocumented in the V4L specification, in Linux 2.3 the VIDIOCGVBIFMT and VIDIOCSVBIFMT ioctls using struct vbi_format were added to determine the VBI image parameters. These ioctls are only partially compatible with the V4L2 VBI interface specified in Section 4.7.

An *offset* field does not exist, *sample_format* is supposed to be VIDEO_PALETTE_RAW, equivalent to V4L2_PIX_FMT_GREY. The remaining fields are probably equivalent to struct v4l2_vbi_format.

Apparently only the Zoran (ZR 36120) driver implements these ioctls. The semantics differ from those specified for V4L2 in two ways. The parameters are reset on open() and VIDIOCSVBIFMT always returns an EINVAL error code if the parameters are invalid.

### 7.1.11. Miscellaneous

V4L2 has no equivalent of the VIDIOCGUNIT ioctl. Applications can find the VBI device associated with a video capture device (or vice versa) by reopening the device and requesting VBI data. For details see Section 1.1.

No replacement exists for VIDIOCKEY, and the V4L functions for microcode programming. A new interface for MPEG compression and playback devices is documented in Section 1.9.

## 7.2. Changes of the V4L2 API

Soon after the V4L API was added to the kernel it was criticised as too inflexible. In

August 1998 Bill Dirks proposed a number of improvements and began to work on documentation, example drivers and applications. With the help of other volunteers this eventually became the V4L2 API, not just an extension but a replacement for the V4L API. However it took another four years and two stable kernel releases until the new API was finally accepted for inclusion into the kernel in its present form.

## 7.2.1. Early Versions

1998-08-20: First version.

1998-08-27: The `select()` function was introduced.

1998-09-10: New video standard interface.

1998-09-18: The `VIDIOC_NONCAP` ioctl was replaced by the otherwise meaningless `O_TRUNC` `open()` flag, and the aliases `O_NONCAP` and `O_NOIO` were defined. Applications can set this flag if they intend to access controls only, as opposed to capture applications which need exclusive access. The `VIDEO_STD_XXX` identifiers are now ordinals instead of flags, and the `video_std_construct()` helper function takes id and transmission arguments.

1998-09-28: Revamped video standard. Made video controls individually enumerable.

1998-10-02: The *id* field was removed from struct video_standard and the color subcarrier fields were renamed. The `VIDIOC_QUERYSTD` ioctl was renamed to `VIDIOC_ENUMSTD`, `VIDIOC_G_INPUT` to `VIDIOC_ENUMINPUT`. A first draft of the Codec API was released.

1998-11-08: Many minor changes. Most symbols have been renamed. Some material changes to struct v4l2_capability.

1998-11-12: The read/write directon of some ioctls was misdefined.

1998-11-14: `V4L2_PIX_FMT_RGB24` changed to `V4L2_PIX_FMT_BGR24`, and `V4L2_PIX_FMT_RGB32` changed to `V4L2_PIX_FMT_BGR32`. Audio controls are now accessible with the `VIDIOC_G_CTRL` and `VIDIOC_S_CTRL` ioctls under names starting with `V4L2_CID_AUDIO`. The `V4L2_MAJOR` define was removed from `videodev.h` since it was only used once in the `videodev` kernel module. The `YUV422` and `YUV411` planar image formats were added.

1998-11-28: A few ioctl symbols changed. Interfaces for codecs and video output devices were added.

1999-01-14: A raw VBI capture interface was added.

1999-01-19: The `VIDIOC_NEXTBUF` ioctl was removed.

### 7.2.2. V4L2 Version 0.16 1999-01-31

1999-01-27: There is now one QBUF ioctl, VIDIOC_QWBUF and VIDIOC_QRBUF are gone. VIDIOC_QBUF takes a v4l2_buffer as a parameter. Added digital zoom (cropping) controls.

### 7.2.3. V4L2 Version 0.18 1999-03-16

Added a v4l to V4L2 ioctl compatibility layer to videodev.c. Driver writers, this changes how you implement your ioctl handler. See the Driver Writer's Guide. Added some more control id codes.

### 7.2.4. V4L2 Version 0.19 1999-06-05

1999-03-18: Fill in the category and catname fields of v4l2_queryctrl objects before passing them to the driver. Required a minor change to the VIDIOC_QUERYCTRL handlers in the sample drivers.

1999-03-31: Better compatibility for v4l memory capture ioctls. Requires changes to drivers to fully support new compatibility features, see Driver Writer's Guide and v4l2cap.c. Added new control IDs: V4L2_CID_HFLIP, _VFLIP. Changed V4L2_PIX_FMT_YUV422P to _YUV422P, and _YUV411P to _YUV411P.

1999-04-04: Added a few more control IDs.

1999-04-07: Added the button control type.

1999-05-02: Fixed a typo in videodev.h, and added the V4L2_CTRL_FLAG_GRAYED (later V4L2_CTRL_FLAG_GRABBED) flag.

1999-05-20: Definition of VIDIOC_G_CTRL was wrong causing a malfunction of this ioctl.

1999-06-05: Changed the value of V4L2_CID_WHITENESS.

### 7.2.5. V4L2 Version 0.20 (1999-09-10)

Version 0.20 introduced a number of changes which were *not backward compatible* with 0.19 and earlier versions. Purpose of these changes was to simplify the API, while making it more extensible and following common Linux driver API conventions.

1. Some typos in `V4L2_FMT_FLAG` symbols were fixed. struct v4l2_clip was changed for compatibility with v4l. (1999-08-30)

2. V4L2_TUNER_SUB_LANG1 was added. (1999-09-05)

3. All ioctl() commands that used an integer argument now take a pointer to an integer. Where it makes sense, ioctls will return the actual new value in the integer pointed to by the argument, a common convention in the V4L2 API. The affected ioctls are: VIDIOC_PREVIEW, VIDIOC_STREAMON, VIDIOC_STREAMOFF, VIDIOC_S_FREQ, VIDIOC_S_INPUT, VIDIOC_S_OUTPUT, VIDIOC_S_EFFECT. For example

   ```
   err = ioctl (fd, VIDIOC_XXX, V4L2_XXX);
   ```
   becomes
   ```
   int a = V4L2_XXX; err = ioctl(fd, VIDIOC_XXX, &a);
   ```

4. All the different get- and set-format commands were swept into one VIDIOC_G_FMT and VIDIOC_S_FMT ioctl taking a union and a type field selecting the union member as parameter. Purpose is to simplify the API by eliminating several ioctls and to allow new and driver private data streams without adding new ioctls.

   This change obsoletes the following ioctls: VIDIOC_S_INFMT, VIDIOC_G_INFMT, VIDIOC_S_OUTFMT, VIDIOC_G_OUTFMT, VIDIOC_S_VBIFMT and VIDIOC_G_VBIFMT. The image format structure v4l2_format was renamed to struct v4l2_pix_format, while struct v4l2_format is now the envelopping structure for all format negotiations.

5. Similar to the changes above, the VIDIOC_G_PARM and VIDIOC_S_PARM ioctls were merged with VIDIOC_G_OUTPARM and VIDIOC_S_OUTPARM. A *type* field in the new struct v4l2_streamparm selects the respective union member.

   This change obsoletes the VIDIOC_G_OUTPARM and VIDIOC_S_OUTPARM ioctls.

6. Control enumeration was simplified, and two new control flags were introduced and one dropped. The *catname* field was replaced by a *group* field.

   Drivers can now flag unsupported and temporarily unavailable controls with V4L2_CTRL_FLAG_DISABLED and V4L2_CTRL_FLAG_GRABBED respectively. The *group* name indicates a possibly narrower classification than the *category*. In other words, there may be multiple groups within a category. Controls within a group would typically be drawn within a group box. Controls in different categories might have a greater separation, or may even appear in separate windows.

7. The struct v4l2_buffer *timestamp* was changed to a 64 bit integer, containing the sampling or output time of the frame in nanoseconds. Additionally timestamps will be in absolute system time, not starting from zero at the beginning of a stream. The data type name for timestamps is stamp_t, defined as a signed 64-bit integer. Output devices should not send a buffer out until the

time in the timestamp field has arrived. I would like to follow SGI's lead, and adopt a multimedia timestamping system like their UST (Unadjusted System Time). See http://web.archive.org/web/*/http://reality.sgi.com /cpirazzi_engr/lg/time/intro.html. UST uses timestamps that are 64-bit signed integers (not struct timeval's) and given in nanosecond units. The UST clock starts at zero when the system is booted and runs continuously and uniformly. It takes a little over 292 years for UST to overflow. There is no way to set the UST clock. The regular Linux time-of-day clock can be changed periodically, which would cause errors if it were being used for timestamping a multimedia stream. A real UST style clock will require some support in the kernel that is not there yet. But in anticipation, I will change the timestamp field to a 64-bit integer, and I will change the v4l2_masterclock_gettime() function (used only by drivers) to return a 64-bit integer.

8. A *sequence* field was added to struct v4l2_buffer. The *sequence* field counts captured frames, it is ignored by output devices. When a capture driver drops a frame, the sequence number of that frame is skipped.

## 7.2.6. V4L2 Version 0.20 incremental changes

1999-12-23: In struct v4l2_vbi_format the *reserved1* field became *offset*. Previously drivers were required to clear the *reserved1* field.

2000-01-13: The V4L2_FMT_FLAG_NOT_INTERLACED flag was added.

2000-07-31: The linux/poll.h header is now included by videodev.h for compatibility with the original videodev.h file.

2000-11-20: V4L2_TYPE_VBI_OUTPUT and V4L2_PIX_FMT_Y41P were added.

2000-11-25: V4L2_TYPE_VBI_INPUT was added.

2000-12-04: A couple typos in symbol names were fixed.

2001-01-18: To avoid namespace conflicts the fourcc macro defined in the videodev.h header file was renamed to v4l2_fourcc.

2001-01-25: A possible driver-level compatibility problem between the videodev.h file in Linux 2.4.0 and the videodev.h file included in the videodevX patch was fixed. Users of an earlier version of videodevX on Linux 2.4.0 should recompile their V4L and V4L2 drivers.

2001-01-26: A possible kernel-level incompatibility between the videodev.h file in the videodevX patch and the videodev.h file in Linux 2.2.x with devfs patches applied was fixed.

2001-03-02: Certain V4L ioctls which pass data in both direction although they are defined with read-only parameter, did not work correctly through the backward

compatibility layer. [Solution?]

2001-04-13: Big endian 16-bit RGB formats were added.

2001-09-17: New YUV formats and the VIDIOC_G_FREQUENCY and VIDIOC_S_FREQUENCY ioctls were added. (The old VIDIOC_G_FREQ and VIDIOC_S_FREQ ioctls did not take multiple tuners into account.)

2000-09-18: V4L2_BUF_TYPE_VBI was added. This may *break compatibility* as the VIDIOC_G_FMT and VIDIOC_S_FMT ioctls may fail now if the struct v4l2_fmt *type* field does not contain V4L2_BUF_TYPE_VBI. In the documentation of the struct v4l2_vbi_format *offset* field the ambiguous phrase "rising edge" was changed to "leading edge".

## 7.2.7. V4L2 Version 0.20 2000-11-23

A number of changes were made to the raw VBI interface.

1. Figures clarifying the line numbering scheme were added to the V4L2 API specification. The *start*[0] and *start*[1] fields no longer count line numbers beginning at zero. Rationale: a) The previous definition was unclear. b) The *start*[] values are ordinal numbers. c) There is no point in inventing a new line numbering scheme. We now use line number as defined by ITU-R, period. Compatibility: Add one to the start values. Applications depending on the previous semantics may not function correctly.
2. The restriction "count[0] > 0 and count[1] > 0" has been relaxed to "(count[0] + count[1]) > 0". Rationale: Drivers may allocate resources at scan line granularity and some data services are transmitted only on the first field. The comment that both *count* values will usually be equal is misleading and pointless and has been removed. This change *breaks compatibility* with earlier versions: Drivers may return EINVAL, applications may not function correctly.
3. Drivers are again permitted to return negative (unknown) start values as proposed earlier. Why this feature was dropped is unclear. This change may *break compatibility* with applications depending on the start values being positive. The use of EBUSY and EINVAL error codes with the VIDIOC_S_FMT ioctl was clarified. The EBUSY error code was finally documented, and the *reserved2* field which was previously mentioned only in the videodev.h header file.
4. New buffer types V4L2_TYPE_VBI_INPUT and V4L2_TYPE_VBI_OUTPUT were added. The former is an alias for the old V4L2_TYPE_VBI, the latter was missing in the videodev.h file.

### 7.2.8. V4L2 Version 0.20 2002-07-25

Added sliced VBI interface proposal.

### 7.2.9. V4L2 in Linux 2.5.46, 2002-10

Around October-November 2002, prior to an announced feature freeze of Linux 2.5, the API was revised, drawing from experience with V4L2 0.20. This unnamed version was finally merged into Linux 2.5.46.

1. As specified in Section 1.1.2, drivers must make related device functions available under all minor device numbers.
2. The `open()` function requires access mode `O_RDWR` regardless of the device type. All V4L2 drivers exchanging data with applications must support the `O_NONBLOCK` flag. The `O_NOIO` flag, a V4L2 symbol which aliased the meaningless `O_TRUNC` to indicate accesses without data exchange (panel applications) was dropped. Drivers must stay in "panel mode" until the application attempts to initiate a data exchange, see Section 1.1.
3. The struct v4l2_capability changed dramatically. Note that also the size of the structure changed, which is encoded in the ioctl request code, thus older V4L2 devices will respond with an EINVAL error code to the new `VIDIOC_QUERYCAP` ioctl.

   There are new fields to identify the driver, a new RDS device function `V4L2_CAP_RDS_CAPTURE`, the `V4L2_CAP_AUDIO` flag indicates if the device has any audio connectors, another I/O capability `V4L2_CAP_ASYNCIO` can be flagged. In response to these changes the *`type`* field became a bit set and was merged into the *`flags`* field. `V4L2_FLAG_TUNER` was renamed to `V4L2_CAP_TUNER`, `V4L2_CAP_VIDEO_OVERLAY` replaced `V4L2_FLAG_PREVIEW` and `V4L2_CAP_VBI_CAPTURE` and `V4L2_CAP_VBI_OUTPUT` replaced `V4L2_FLAG_DATA_SERVICE`. `V4L2_FLAG_READ` and `V4L2_FLAG_WRITE` were merged into `V4L2_CAP_READWRITE`.

   The redundant fields *`inputs`*, *`outputs`* and *`audios`* were removed. These properties can be determined as described in Section 1.4 and Section 1.5.

   The somewhat volatile and therefore barely useful fields *`maxwidth`*, *`maxheight`*, *`minwidth`*, *`minheight`*, *`maxframerate`* were removed. This information is available as described in Section 1.10 and Section 1.7.

   `V4L2_FLAG_SELECT` was removed. We believe the select() function is important enough to require support of it in all V4L2 drivers exchanging data

with applications. The redundant V4L2_FLAG_MONOCHROME flag was removed, this information is available as described in Section 1.10.

4. In struct v4l2_input the *assoc_audio* field and the *capability* field and its only flag V4L2_INPUT_CAP_AUDIO was replaced by the new *audioset* field. Instead of linking one video input to one audio input this field reports all audio inputs this video input combines with.

   New fields are *tuner* (reversing the former link from tuners to video inputs), *std* and *status*.

   Accordingly struct v4l2_output lost its *capability* and *assoc_audio* fields. *audioset*, *modulator* and *std* where added instead.

5. The struct v4l2_audio field *audio* was renamed to *index*, for consistency with other structures. A new capability flag V4L2_AUDCAP_STEREO was added to indicated if the audio input in question supports stereo sound. V4L2_AUDCAP_EFFECTS and the corresponding V4L2_AUDMODE flags where removed. This can be easily implemented using controls. (However the same applies to AVL which is still there.)

   Again for consistency the struct v4l2_audioout field *audio* was renamed to *index*.

6. The struct v4l2_tuner *input* field was replaced by an *index* field, permitting devices with multiple tuners. The link between video inputs and tuners is now reversed, inputs point to their tuner. The *std* substructure became a simple set (more about this below) and moved into struct v4l2_input. A *type* field was added.

   Accordingly in struct v4l2_modulator the *output* was replaced by an *index* field.

   In struct v4l2_frequency the *port* field was replaced by a *tuner* field containing the respective tuner or modulator index number. A tuner *type* field was added and the *reserved* field became larger for future extensions (satellite tuners in particular).

7. The idea of completely transparent video standards was dropped. Experience showed that applications must be able to work with video standards beyond presenting the user a menu. Instead of enumerating supported standards with an ioctl applications can now refer to standards by v4l2_std_id and symbols defined in the videodev2.h header file. For details see Section 1.7. The VIDIOC_G_STD and VIDIOC_S_STD now take a pointer to this type as argument. VIDIOC_QUERYSTD was added to autodetect the received standard, if the hardware has this capability. In struct v4l2_standard an *index* field was added for VIDIOC_ENUMSTD. A v4l2_std_id field named *id* was added as machine readable identifier, also replacing the *transmission* field. The misleading *framerate* field was renamed to *frameperiod*. The now obsolete

*colorstandard* information, originally needed to distguish between variations of standards, were removed.

Struct v4l2_enumstd ceased to be. VIDIOC_ENUMSTD now takes a pointer to a struct v4l2_standard directly. The information which standards are supported by a particular video input or output moved into struct v4l2_input and struct v4l2_output fields named *std*, respectively.

8. The struct v4l2_queryctrl fields *category* and *group* did not catch on and/or were not implemented as expected and therefore removed.

9. The VIDIOC_TRY_FMT ioctl was added to negotiate data formats as with VIDIOC_S_FMT, but without the overhead of programming the hardware and regardless of I/O in progress.

   In struct v4l2_format the *fmt* union was extended to contain struct v4l2_window. All image format negotiations are now possible with VIDIOC_G_FMT, VIDIOC_S_FMT and VIDIOC_TRY_FMT; ioctl. The VIDIOC_G_WIN and VIDIOC_S_WIN ioctls to prepare for a video overlay were removed. The *type* field changed to type enum v4l2_buf_type and the buffer type names changed as follows.

| Old defines | enum v4l2_buf_type |
|---|---|
| V4L2_BUF_TYPE_CAPTURE | V4L2_BUF_TYPE_VIDEO_CAPTURE |
| V4L2_BUF_TYPE_CODECIN | Omitted for now |
| V4L2_BUF_TYPE_CODECOUT | Omitted for now |
| V4L2_BUF_TYPE_EFFECTSIN | Omitted for now |
| V4L2_BUF_TYPE_EFFECTSIN2 | Omitted for now |
| V4L2_BUF_TYPE_EFFECTSOUT | Omitted for now |
| V4L2_BUF_TYPE_VIDEOOUT | V4L2_BUF_TYPE_VIDEO_OUTPUT |
| – | V4L2_BUF_TYPE_VIDEO_OVERLAY |
| – | V4L2_BUF_TYPE_VBI_CAPTURE |
| – | V4L2_BUF_TYPE_VBI_OUTPUT |
| – | V4L2_BUF_TYPE_SLICED_VBI_CAPTURE |
| – | V4L2_BUF_TYPE_SLICED_VBI_OUTPUT |
| V4L2_BUF_TYPE_PRIVATE_BASE | V4L2_BUF_TYPE_PRIVATE |

10. In struct v4l2_fmtdesc a enum v4l2_buf_type field named *type* was added as in struct v4l2_format. The VIDIOC_ENUM_FBUFFMT ioctl is no longer needed and was removed. These calls can be replaced by VIDIOC_ENUM_FMT with type V4L2_BUF_TYPE_VIDEO_OVERLAY.

11. In struct v4l2_pix_format the *depth* field was removed, assuming applications which recognize the format by its four-character-code already know the color depth, and others do not care about it. The same rationale lead to the removal of the V4L2_FMT_FLAG_COMPRESSED flag. The V4L2_FMT_FLAG_SWCONVECOMPRESSED flag was removed because drivers are not supposed to convert images in kernel space. A user library of conversion functions should be provided instead. The V4L2_FMT_FLAG_BYTESPERLINE flag was redundant. Applications can set the *bytesperline* field to zero to get a reasonable default. Since the remaining flags were replaced as well, the *flags* field itself was removed.

    The interlace flags were replaced by a enum v4l2_field value in a newly added *field* field.

| Old flag | enum v4l2_field |
|---|---|
| V4L2_FMT_FLAG_NOT_INTERLACED | ? |
| V4L2_FMT_FLAG_INTERLACED = V4L2_FMT_FLAG_COMBINED | V4L2_FIELD_INTERLACED |
| V4L2_FMT_FLAG_TOPFIELD = V4L2_FMT_FLAG_ODDFIELD | V4L2_FIELD_TOP |
| V4L2_FMT_FLAG_BOTFIELD = V4L2_FMT_FLAG_EVENFIELD | V4L2_FIELD_BOTTOM |
| – | V4L2_FIELD_SEQ_TB |
| – | V4L2_FIELD_SEQ_BT |
| – | V4L2_FIELD_ALTERNATE |

    The color space flags were replaced by a enum v4l2_colorspace value in a newly added *colorspace* field, where one of V4L2_COLORSPACE_SMPTE170M, V4L2_COLORSPACE_BT878, V4L2_COLORSPACE_470_SYSTEM_M or V4L2_COLORSPACE_470_SYSTEM_BG replaces V4L2_FMT_CS_601YUV.

12. In struct v4l2_requestbuffers the *type* field was properly defined as enum v4l2_buf_type. Buffer types changed as mentioned above. A new *memory* field of type enum v4l2_memory was added to distinguish between I/O methods using buffers allocated by the driver or the application. See Chapter 3 for details.

13. In struct v4l2_buffer the *type* field was properly defined as enum v4l2_buf_type. Buffer types changed as mentioned above. A *field* field of type enum v4l2_field was added to indicate if a buffer contains a top or bottom field. The old field flags were removed. Since no unadjusted system time clock was added to the kernel as planned, the *timestamp* field changed

back from type stamp_t, an unsigned 64 bit integer expressing the sample time in nanoseconds, to struct timeval. With the addition of a second memory mapping method the *offset* field moved into union *m*, and a new *memory* field of type enum v4l2_memory was added to distinguish between I/O methods. See Chapter 3 for details.

The V4L2_BUF_REQ_CONTIG flag was used by the V4L compatibility layer, after changes to this code it was no longer needed. The V4L2_BUF_ATTR_DEVICEMEM flag would indicate if the buffer was indeed allocated in device memory rather than DMA-able system memory. It was barely useful and so was removed.

14. In struct v4l2_framebuffer the *base[3]* array anticipating double- and triple-buffering in off-screen video memory, however without defining a synchronization mechanism, was replaced by a single pointer. The V4L2_FBUF_CAP_SCALEUP and V4L2_FBUF_CAP_SCALEDOWN flags were removed. Applications can determine this capability more accurately using the new cropping and scaling interface. The V4L2_FBUF_CAP_CLIPPING flag was replaced by V4L2_FBUF_CAP_LIST_CLIPPING and V4L2_FBUF_CAP_BITMAP_CLIPPING.

15. In struct v4l2_clip the *x*, *y*, *width* and *height* field moved into a *c* substructure of type struct v4l2_rect. The *x* and *y* fields were renamed to *left* and *top*, i. e. offsets to a context dependent origin.

16. In struct v4l2_window the *x*, *y*, *width* and *height* field moved into a *w* substructure as above. A *field* field of type %v4l2-field; was added to distinguish between field and frame (interlaced) overlay.

17. The digital zoom interface, including struct v4l2_zoomcap, struct v4l2_zoom, V4L2_ZOOM_NONCAP and V4L2_ZOOM_WHILESTREAMING was replaced by a new cropping and scaling interface. The previously unused struct v4l2_cropcap and v4l2_crop where redefined for this purpose. See Section 1.12 for details.

18. In struct v4l2_vbi_format the *SAMPLE_FORMAT* field now contains a four-character-code as used to identify video image formats and V4L2_PIX_FMT_GREY replaces the V4L2_VBI_SF_UBYTE define. The *reserved* field was extended.

19. In struct v4l2_captureparm the type of the *timeperframe* field changed from unsigned long to struct v4l2_fract. This allows the accurate expression of multiples of the NTSC-M frame rate 30000 / 1001. A new field *readbuffers* was added to control the driver behaviour in read I/O mode.

    Similar changes were made to struct v4l2_outputparm.

20. The struct v4l2_performance and VIDIOC_G_PERF ioctl were dropped. Except when using the read/write I/O method, which is limited anyway, this information is already available to applications.

21. The example transformation from RGB to YCbCr color space in the old V4L2 documentation was inaccurate, this has been corrected in Chapter 2.

## 7.2.10. V4L2 2003-06-19

1. A new capability flag V4L2_CAP_RADIO was added for radio devices. Prior to this change radio devices would identify solely by having exactly one tuner whose type field reads V4L2_TUNER_RADIO.
2. An optional driver access priority mechanism was added, see Section 1.3 for details.
3. The audio input and output interface was found to be incomplete.

   Previously the VIDIOC_G_AUDIO ioctl would enumerate the available audio inputs. An ioctl to determine the current audio input, if more than one combines with the current video input, did not exist. So VIDIOC_G_AUDIO was renamed to VIDIOC_G_AUDIO_OLD, this ioctl was removed on Kernel 2.6.39. The VIDIOC_ENUMAUDIO ioctl was added to enumerate audio inputs, while VIDIOC_G_AUDIO now reports the current audio input.

   The same changes were made to VIDIOC_G_AUDOUT and VIDIOC_ENUMAUDOUT.

   Until further the "videodev" module will automatically translate between the old and new ioctls, but drivers and applications must be updated to successfully compile again.
4. The VIDIOC_OVERLAY ioctl was incorrectly defined with write-read parameter. It was changed to write-only, while the write-read version was renamed to VIDIOC_OVERLAY_OLD. The old ioctl was removed on Kernel 2.6.39. Until further the "videodev" kernel module will automatically translate to the new version, so drivers must be recompiled, but not applications.
5. Section 4.2 incorrectly stated that clipping rectangles define regions where the video can be seen. Correct is that clipping rectangles define regions where *no* video shall be displayed and so the graphics surface can be seen.
6. The VIDIOC_S_PARM and VIDIOC_S_CTRL ioctls were defined with write-only parameter, inconsistent with other ioctls modifying their argument. They were changed to write-read, while a _OLD suffix was added to the write-only versions. The old ioctls were removed on Kernel 2.6.39. Drivers and applications assuming a constant parameter need an update.

## 7.2.11. V4L2 2003-11-05

1. In Section 2.6 the following pixel formats were incorrectly transferred from Bill Dirks' V4L2 specification. Descriptions below refer to bytes in memory, in ascending address order.

| Symbol | In this document prior to revision 0.5 | Corrected |
|---|---|---|
| V4L2_PIX_FMT_RGB24 | B, G, R | R, G, B |
| V4L2_PIX_FMT_BGR24 | R, G, B | B, G, R |
| V4L2_PIX_FMT_RGB32 | B, G, R, X | R, G, B, X |
| V4L2_PIX_FMT_BGR32 | R, G, B, X | B, G, R, X |

   The V4L2_PIX_FMT_BGR24 example was always correct.

   In Section 7.1.5 the mapping of the V4L VIDEO_PALETTE_RGB24 and VIDEO_PALETTE_RGB32 formats to V4L2 pixel formats was accordingly corrected.

2. Unrelated to the fixes above, drivers may still interpret some V4L2 RGB pixel formats differently. These issues have yet to be addressed, for details see Section 2.6.

## 7.2.12. V4L2 in Linux 2.6.6, 2004-05-09

1. The VIDIOC_CROPCAP ioctl was incorrectly defined with read-only parameter. It is now defined as write-read ioctl, while the read-only version was renamed to VIDIOC_CROPCAP_OLD. The old ioctl was removed on Kernel 2.6.39.

## 7.2.13. V4L2 in Linux 2.6.8

1. A new field *input* (former *reserved[0]*) was added to the struct v4l2_buffer structure. Purpose of this field is to alternate between video inputs (e. g. cameras) in step with the video capturing process. This function must be enabled with the new V4L2_BUF_FLAG_INPUT flag. The *flags* field is no longer read-only.

## 7.2.14. V4L2 spec erratum 2004-08-01

1. The return value of the V4L2 open()(2) function was incorrectly documented.
2. Audio output ioctls end in -AUDOUT, not -AUDIOOUT.
3. In the Current Audio Input example the VIDIOC_G_AUDIO ioctl took the wrong argument.
4. The documentation of the VIDIOC_QBUF and VIDIOC_DQBUF ioctls did not mention the struct v4l2_buffer *memory* field. It was also missing from examples. Also on the VIDIOC_DQBUF page the EIO error code was not documented.

## 7.2.15. V4L2 in Linux 2.6.14

1. A new sliced VBI interface was added. It is documented in Section 4.8 and replaces the interface first proposed in V4L2 specification 0.8.

## 7.2.16. V4L2 in Linux 2.6.15

1. The VIDIOC_LOG_STATUS ioctl was added.
2. New video standards V4L2_STD_NTSC_443, V4L2_STD_SECAM_LC, V4L2_STD_SECAM_DK (a set of SECAM D, K and K1), and V4L2_STD_ATSC (a set of V4L2_STD_ATSC_8_VSB and V4L2_STD_ATSC_16_VSB) were defined. Note the V4L2_STD_525_60 set now includes V4L2_STD_NTSC_443. See also Table A-3.
3. The VIDIOC_G_COMP and VIDIOC_S_COMP ioctl were renamed to VIDIOC_G_MPEGCOMP and VIDIOC_S_MPEGCOMP respectively. Their argument was replaced by a struct v4l2_mpeg_compression pointer. (The VIDIOC_G_MPEGCOMP and VIDIOC_S_MPEGCOMP ioctls where removed in Linux 2.6.25.)

## 7.2.17. V4L2 spec erratum 2005-11-27

The capture example in Appendix C called the VIDIOC_S_CROP ioctl without checking if cropping is supported. In the video standard selection example in Section 1.7 the VIDIOC_S_STD call used the wrong argument type.

## 7.2.18. V4L2 spec erratum 2006-01-10

1. The V4L2_IN_ST_COLOR_KILL flag in struct v4l2_input not only indicates if the color killer is enabled, but also if it is active. (The color killer disables color decoding when it detects no color in the video signal to improve the image quality.)
2. VIDIOC_S_PARM is a write-read ioctl, not write-only as stated on its reference page. The ioctl changed in 2003 as noted above.

## 7.2.19. V4L2 spec erratum 2006-02-03

1. In struct v4l2_captureparm and struct v4l2_outputparm the *timeperframe* field gives the time in seconds, not microseconds.

## 7.2.20. V4L2 spec erratum 2006-02-04

1. The *clips* field in struct v4l2_window must point to an array of struct v4l2_clip, not a linked list, because drivers ignore the struct v4l2_clip.*next* pointer.

## 7.2.21. V4L2 in Linux 2.6.17

1. New video standard macros were added: V4L2_STD_NTSC_M_KR (NTSC M South Korea), and the sets V4L2_STD_MN, V4L2_STD_B, V4L2_STD_GH and V4L2_STD_DK. The V4L2_STD_NTSC and V4L2_STD_SECAM sets now include V4L2_STD_NTSC_M_KR and V4L2_STD_SECAM_LC respectively.
2. A new V4L2_TUNER_MODE_LANG1_LANG2 was defined to record both languages of a bilingual program. The use of V4L2_TUNER_MODE_STEREO for this purpose is deprecated now. See the VIDIOC_G_TUNER section for details.

## 7.2.22. V4L2 spec erratum 2006-09-23 (Draft 0.15)

1. In various places V4L2_BUF_TYPE_SLICED_VBI_CAPTURE and V4L2_BUF_TYPE_SLICED_VBI_OUTPUT of the sliced VBI interface were not mentioned along with other buffer types.
2. In ioctl VIDIOC_G_AUDIO, VIDIOC_S_AUDIO(2) it was clarified that the struct v4l2_audio *mode* field is a flags field.
3. ioctl VIDIOC_QUERYCAP(2) did not mention the sliced VBI and radio capability flags.
4. In ioctl VIDIOC_G_FREQUENCY, VIDIOC_S_FREQUENCY(2) it was clarified that applications must initialize the tuner *type* field of struct v4l2_frequency before calling VIDIOC_S_FREQUENCY.
5. The *reserved* array in struct v4l2_requestbuffers has 2 elements, not 32.
6. In Section 4.3 and Section 4.7 the device file names /dev/vout which never caught on were replaced by /dev/video.
7. With Linux 2.6.15 the possible range for VBI device minor numbers was extended from 224-239 to 224-255. Accordingly device file names /dev/vbi0 to /dev/vbi31 are possible now.

## 7.2.23. V4L2 in Linux 2.6.18

1. New ioctls VIDIOC_G_EXT_CTRLS, VIDIOC_S_EXT_CTRLS and VIDIOC_TRY_EXT_CTRLS were added, a flag to skip unsupported controls with VIDIOC_QUERYCTRL, new control types V4L2_CTRL_TYPE_INTEGER64 and V4L2_CTRL_TYPE_CTRL_CLASS (Table A-3), and new control flags V4L2_CTRL_FLAG_READ_ONLY, V4L2_CTRL_FLAG_UPDATE, V4L2_CTRL_FLAG_INACTIVE and V4L2_CTRL_FLAG_SLIDER (Table A-4). See Section 1.9 for details.

## 7.2.24. V4L2 in Linux 2.6.19

1. In struct v4l2_sliced_vbi_cap a buffer type field was added replacing a reserved field. Note on architectures where the size of enum types differs from int types the size of the structure changed. The VIDIOC_G_SLICED_VBI_CAP ioctl was redefined from being read-only to write-read. Applications must initialize the type field and clear the reserved fields now. These changes may *break the compatibility* with older drivers and applications.

2. The ioctls VIDIOC_ENUM_FRAMESIZES and VIDIOC_ENUM_FRAMEINTERVALS were added.
3. A new pixel format V4L2_PIX_FMT_RGB444 (Table 2-1) was added.

## 7.2.25. V4L2 spec erratum 2006-10-12 (Draft 0.17)

1. V4L2_PIX_FMT_HM12 (Table 2-10) is a YUV 4:2:0, not 4:2:2 format.

## 7.2.26. V4L2 in Linux 2.6.21

1. The videodev2.h header file is now dual licensed under GNU General Public License version two or later, and under a 3-clause BSD-style license.

## 7.2.27. V4L2 in Linux 2.6.22

1. Two new field orders V4L2_FIELD_INTERLACED_TB and V4L2_FIELD_INTERLACED_BT were added. See Table 3-9 for details.
2. Three new clipping/blending methods with a global or straight or inverted local alpha value were added to the video overlay interface. See the description of the VIDIOC_G_FBUF and VIDIOC_S_FBUF ioctls for details.

   A new *global_alpha* field was added to v4l2_window, extending the structure. This may *break compatibility* with applications using a struct v4l2_window directly. However the VIDIOC_G/S/TRY_FMT ioctls, which take a pointer to a v4l2_format parent structure with padding bytes at the end, are not affected.
3. The format of the *chromakey* field in struct v4l2_window changed from "host order RGB32" to a pixel value in the same format as the framebuffer. This may *break compatibility* with existing applications. Drivers supporting the "host order RGB32" format are not known.

## 7.2.28. V4L2 in Linux 2.6.24

1. The pixel formats V4L2_PIX_FMT_PAL8, V4L2_PIX_FMT_YUV444, V4L2_PIX_FMT_YUV555, V4L2_PIX_FMT_YUV565 and V4L2_PIX_FMT_YUV32 were added.

## 7.2.29. V4L2 in Linux 2.6.25

1. The pixel formats V4L2_PIX_FMT_Y16 and V4L2_PIX_FMT_SBGGR16 were added.
2. New controls V4L2_CID_POWER_LINE_FREQUENCY, V4L2_CID_HUE_AUTO, V4L2_CID_WHITE_BALANCE_TEMPERATURE, V4L2_CID_SHARPNESS and V4L2_CID_BACKLIGHT_COMPENSATION were added. The controls V4L2_CID_BLACK_LEVEL, V4L2_CID_WHITENESS, V4L2_CID_HCENTER and V4L2_CID_VCENTER were deprecated.
3. A Camera controls class was added, with the new controls V4L2_CID_EXPOSURE_AUTO, V4L2_CID_EXPOSURE_ABSOLUTE, V4L2_CID_EXPOSURE_AUTO_PRIORITY, V4L2_CID_PAN_RELATIVE, V4L2_CID_TILT_RELATIVE, V4L2_CID_PAN_RESET, V4L2_CID_TILT_RESET, V4L2_CID_PAN_ABSOLUTE, V4L2_CID_TILT_ABSOLUTE, V4L2_CID_FOCUS_ABSOLUTE, V4L2_CID_FOCUS_RELATIVE and V4L2_CID_FOCUS_AUTO.
4. The VIDIOC_G_MPEGCOMP and VIDIOC_S_MPEGCOMP ioctls, which were superseded by the extended controls interface in Linux 2.6.18, where finally removed from the videodev2.h header file.

## 7.2.30. V4L2 in Linux 2.6.26

1. The pixel formats V4L2_PIX_FMT_Y16 and V4L2_PIX_FMT_SBGGR16 were added.
2. Added user controls V4L2_CID_CHROMA_AGC and V4L2_CID_COLOR_KILLER.

## 7.2.31. V4L2 in Linux 2.6.27

1. The VIDIOC_S_HW_FREQ_SEEK ioctl and the V4L2_CAP_HW_FREQ_SEEK capability were added.

2. The pixel formats V4L2_PIX_FMT_YVYU, V4L2_PIX_FMT_PCA501, V4L2_PIX_FMT_PCA505, V4L2_PIX_FMT_PCA508, V4L2_PIX_FMT_PCA561, V4L2_PIX_FMT_SGBRG8, V4L2_PIX_FMT_PAC207 and V4L2_PIX_FMT_PJPG were added.

## 7.2.32. V4L2 in Linux 2.6.28

1. Added V4L2_MPEG_AUDIO_ENCODING_AAC and V4L2_MPEG_AUDIO_ENCODING_AC3 MPEG audio encodings.
2. Added V4L2_MPEG_VIDEO_ENCODING_MPEG_4_AVC MPEG video encoding.
3. The pixel formats V4L2_PIX_FMT_SGRBG10 and V4L2_PIX_FMT_SGRBG10DPCM8 were added.

## 7.2.33. V4L2 in Linux 2.6.29

1. The VIDIOC_G_CHIP_IDENT ioctl was renamed to VIDIOC_G_CHIP_IDENT_OLD and VIDIOC_DBG_G_CHIP_IDENT was introduced in its place. The old struct v4l2_chip_ident was renamed to v4l2_chip_ident_old.
2. The pixel formats V4L2_PIX_FMT_VYUY, V4L2_PIX_FMT_NV16 and V4L2_PIX_FMT_NV61 were added.
3. Added camera controls V4L2_CID_ZOOM_ABSOLUTE, V4L2_CID_ZOOM_RELATIVE, V4L2_CID_ZOOM_CONTINUOUS and V4L2_CID_PRIVACY.

## 7.2.34. V4L2 in Linux 2.6.30

1. New control flag V4L2_CTRL_FLAG_WRITE_ONLY was added.
2. New control V4L2_CID_COLORFX was added.

## 7.2.35. V4L2 in Linux 2.6.32

1. In order to be easier to compare a V4L2 API and a kernel version, now V4L2 API is numbered using the Linux Kernel version numeration.
2. Finalized the RDS capture API. See Section 4.11 for more information.
3. Added new capabilities for modulators and RDS encoders.
4. Add description for libv4l API.
5. Added support for string controls via new type V4L2_CTRL_TYPE_STRING.
6. Added V4L2_CID_BAND_STOP_FILTER documentation.
7. Added FM Modulator (FM TX) Extended Control Class: V4L2_CTRL_CLASS_FM_TX and their Control IDs.
8. Added Remote Controller chapter, describing the default Remote Controller mapping for media devices.

## 7.2.36. V4L2 in Linux 2.6.33

1. Added support for Digital Video timings in order to support HDTV receivers and transmitters.

## 7.2.37. V4L2 in Linux 2.6.34

1. Added V4L2_CID_IRIS_ABSOLUTE and V4L2_CID_IRIS_RELATIVE controls to the Camera controls class.

## 7.2.38. V4L2 in Linux 2.6.37

1. Remove the vtx (videotext/teletext) API. This API was no longer used and no hardware exists to verify the API. Nor were any userspace applications found that used it. It was originally scheduled for removal in 2.6.35.

## 7.2.39. V4L2 in Linux 2.6.39

1. The old VIDIOC_*_OLD symbols and V4L1 support were removed.

2. Multi-planar API added. Does not affect the compatibility of current drivers and applications. See multi-planar API for details.

# 7.2.40. Relation of V4L2 to other Linux multimedia APIs

## 7.2.40.1. X Video Extension

The X Video Extension (abbreviated XVideo or just Xv) is an extension of the X Window system, implemented for example by the XFree86 project. Its scope is similar to V4L2, an API to video capture and output devices for X clients. Xv allows applications to display live video in a window, send window contents to a TV output, and capture or output still images in XPixmaps[1]. With their implementation XFree86 makes the extension available across many operating systems and architectures.

Because the driver is embedded into the X server Xv has a number of advantages over the V4L2 video overlay interface. The driver can easily determine the overlay target, i. e. visible graphics memory or off-screen buffers for a destructive overlay. It can program the RAMDAC for a non-destructive overlay, scaling or color-keying, or the clipping functions of the video capture hardware, always in sync with drawing operations or windows moving or changing their stacking order.

To combine the advantages of Xv and V4L a special Xv driver exists in XFree86 and XOrg, just programming any overlay capable Video4Linux device it finds. To enable it `/etc/X11/XF86Config` must contain these lines:

```
Section "Module"
    Load "v4l"
EndSection
```

As of XFree86 4.2 this driver still supports only V4L ioctls, however it should work just fine with all V4L2 devices through the V4L2 backward-compatibility layer. Since V4L2 permits multiple opens it is possible (if supported by the V4L2 driver) to capture video while an X client requested video overlay. Restrictions of simultaneous capturing and overlay are discussed in Section 4.2 apply.

Only marginally related to V4L2, XFree86 extended Xv to support hardware YUV to RGB conversion and scaling for faster video playback, and added an interface to MPEG-2 decoding hardware. This API is useful to display images captured with V4L2 devices.

#### 7.2.40.2. Digital Video

V4L2 does not support digital terrestrial, cable or satellite broadcast. A separate project aiming at digital receivers exists. You can find its homepage at http://linuxtv.org. The Linux DVB API has no connection to the V4L2 API except that drivers for hybrid hardware may support both.

#### 7.2.40.3. Audio Interfaces

[to do - OSS/ALSA]

### 7.2.41. Experimental API Elements

The following V4L2 API elements are currently experimental and may change in the future.

- Video Output Overlay (OSD) Interface, Section 4.4.
- V4L2_BUF_TYPE_VIDEO_OUTPUT_OVERLAY, enum v4l2_buf_type, Table 3-3.
- V4L2_CAP_VIDEO_OUTPUT_OVERLAY, VIDIOC_QUERYCAP ioctl, Table A-2.
- VIDIOC_ENUM_FRAMESIZES and VIDIOC_ENUM_FRAMEINTERVALS ioctls.
- VIDIOC_G_ENC_INDEX ioctl.
- VIDIOC_ENCODER_CMD and VIDIOC_TRY_ENCODER_CMD ioctls.
- VIDIOC_DBG_G_REGISTER and VIDIOC_DBG_S_REGISTER ioctls.
- VIDIOC_DBG_G_CHIP_IDENT ioctl.

### 7.2.42. Obsolete API Elements

The following V4L2 API elements were superseded by new interfaces and should not be implemented in new drivers.

- VIDIOC_G_MPEGCOMP and VIDIOC_S_MPEGCOMP ioctls. Use Extended Controls, Section 1.9.

## Notes

1. This is not implemented in XFree86.

# Appendix A. Function Reference

## V4L2 close()

### Name

v4l2-close — Close a V4L2 device

### Synopsis

```
#include <unistd.h>
int close(int fd);
```

### Arguments

*fd*

File descriptor returned by open().

### Description

Closes the device. Any I/O in progress is terminated and resources associated with the file descriptor are freed. However data format parameters, current input or output, control values or other properties remain unchanged.

### Return Value

The function returns 0 on success, -1 on failure and the errno is set appropriately. Possible error codes:

EBADF

*fd* is not a valid open file descriptor.

# V4L2 ioctl()

## Name

v4l2-ioctl — Program a V4L2 device

## Synopsis

```
#include <sys/ioctl.h>
int ioctl(int fd, int request, void *argp);
```

## Arguments

*fd*

File descriptor returned by open().

*request*

V4L2 ioctl request code as defined in the videodev2.h header file, for example VIDIOC_QUERYCAP.

*argp*

Pointer to a function parameter, usually a structure.

## Description

The ioctl() function is used to program V4L2 devices. The argument *fd* must be an open file descriptor. An ioctl *request* has encoded in it whether the argument is an input, output or read/write parameter, and the size of the argument *argp* in bytes. Macros and defines specifying V4L2 ioctl requests are located in the videodev2.h header file. Applications should use their own copy, not include the version in the kernel sources on the system they compile on. All V4L2 ioctl requests, their respective function and parameters are specified in Appendix A.

## Return Value

On success the `ioctl()` function returns 0 and does not reset the `errno` variable. On failure -1 is returned, when the ioctl takes an output or read/write parameter it remains unmodified, and the `errno` variable is set appropriately. See below for possible error codes. Generic errors like EBADF or EFAULT are not listed in the sections discussing individual ioctl requests.

Note ioctls may return undefined error codes. Since errors may have side effects such as a driver reset applications should abort on unexpected errors.

EBADF

*`fd`* is not a valid open file descriptor.

EBUSY

The property cannot be changed right now. Typically this error code is returned when I/O is in progress or the driver supports multiple opens and another process locked the property.

EFAULT

*`argp`* references an inaccessible memory area.

ENOTTY

*`fd`* is not associated with a character special device.

EINVAL

The *`request`* or the data pointed to by *`argp`* is not valid. This is a very common error code, see the individual ioctl requests listed in Appendix A for actual causes.

ENOMEM

Not enough physical or virtual memory was available to complete the request.

ERANGE

The application attempted to set a control with the `VIDIOC_S_CTRL` ioctl to a value which is out of bounds.

# ioctl VIDIOC_CROPCAP

## Name

VIDIOC_CROPCAP — Information about the video cropping and scaling abilities

## Synopsis

```
int ioctl(int fd, int request, struct v4l2_cropcap *argp);
```

## Arguments

*fd*

File descriptor returned by open().

*request*

VIDIOC_CROPCAP

*argp*

## Description

Applications use this function to query the cropping limits, the pixel aspect of images and to calculate scale factors. They set the *type* field of a v4l2_cropcap structure to the respective buffer (stream) type and call the VIDIOC_CROPCAP ioctl with a pointer to this structure. Drivers fill the rest of the structure. The results are constant except when switching the video standard. Remember this switch can occur implicit when switching the video input or output.

**Table A-1. struct v4l2_cropcap**

| | | |
|---|---|---|
| enum v4l2_buf_type | *type* | Type of the data stream, set by the application. Only these types are valid here: V4L2_BUF_TYPE_VIDEO_CAPTURE, V4L2_BUF_TYPE_VIDEO_OUTPUT, V4L2_BUF_TYPE_VIDEO_OVERLAY, and custom (driver defined) types with code V4L2_BUF_TYPE_PRIVATE and higher. |
| struct v4l2_rect | *bounds* | Defines the window within capturing or output is possible, this may exclude for example the horizontal and vertical blanking areas. The cropping rectangle cannot exceed these limits. Width and height are defined in pixels, the driver writer is free to choose origin and units of the coordinate system in the analog domain. |
| struct v4l2_rect | *defrect* | Default cropping rectangle, it shall cover the "whole picture". Assuming pixel aspect 1/1 this could be for example a 640 × 480 rectangle for NTSC, a 768 × 576 rectangle for PAL and SECAM centered over the active picture area. The same co-ordinate system as for *bounds* is used. |
| struct v4l2_fract | *pixelaspect* | This is the pixel aspect (y / x) when no scaling is applied, the ratio of the actual sampling frequency and the frequency required to get square pixels.<br>When cropping coordinates refer to square pixels, the driver sets *pixelaspect* to 1/1. Other common values are 54/59 for PAL and SECAM, 11/10 for NTSC sampled according to [ITU BT.601]. |

**Table A-2. struct v4l2_rect**

| | | |
|---|---|---|
| __s32 | *left* | Horizontal offset of the top, left corner of the rectangle, in pixels. |
| __s32 | *top* | Vertical offset of the top, left corner of the rectangle, in pixels. |
| __s32 | *width* | Width of the rectangle, in pixels. |
| __s32 | *height* | Height of the rectangle, in pixels. Width and height cannot be negative, the fields are signed for hysterical reasons. |

## Return Value

On success 0 is returned, on error -1 and the `errno` variable is set appropriately:

EINVAL

The struct v4l2_cropcap *type* is invalid or the ioctl is not supported. This is not permitted for video capture, output and overlay devices, which must support `VIDIOC_CROPCAP`.

# ioctl VIDIOC_DBG_G_CHIP_IDENT

## Name

`VIDIOC_DBG_G_CHIP_IDENT` — Identify the chips on a TV card

## Synopsis

```
int ioctl(int fd, int request, struct v4l2_dbg_chip_ident
*argp);
```

## Arguments

*fd*

File descriptor returned by open().

*request*

VIDIOC_DBG_G_CHIP_IDENT

*argp*

## Description

**Experimental:** This is an experimental interface and may change in the future.

For driver debugging purposes this ioctl allows test applications to query the driver about the chips present on the TV card. Regular applications must not use it. When you found a chip specific bug, please contact the linux-media mailing list (http://www.linuxtv.org/lists.php) so it can be fixed.

To query the driver applications must initialize the *match.type* and *match.addr* or *match.name* fields of a struct v4l2_dbg_chip_ident and call VIDIOC_DBG_G_CHIP_IDENT with a pointer to this structure. On success the driver stores information about the selected chip in the *ident* and *revision* fields. On failure the structure remains unchanged.

When *match.type* is V4L2_CHIP_MATCH_HOST, *match.addr* selects the nth non-I$^2$C chip on the TV card. You can enumerate all chips by starting at zero and incrementing *match.addr* by one until VIDIOC_DBG_G_CHIP_IDENT fails with an EINVAL error code. The number zero always selects the host chip, e. g. the chip connected to the PCI or USB bus.

When *match.type* is V4L2_CHIP_MATCH_I2C_DRIVER, *match.name* contains the I2C driver name. For instance "saa7127" will match any chip supported by the saa7127 driver, regardless of its I$^2$C bus address. When multiple chips supported by the same driver are present, the ioctl will return V4L2_IDENT_AMBIGUOUS in the *ident* field.

When *match.type* is V4L2_CHIP_MATCH_I2C_ADDR, *match.addr* selects a chip by its 7 bit I$^2$C bus address.

When *match.type* is V4L2_CHIP_MATCH_AC97, *match.addr* selects the nth AC97 chip on the TV card. You can enumerate all chips by starting at zero and incrementing *match.addr* by one until VIDIOC_DBG_G_CHIP_IDENT fails with an EINVAL error code.

On success, the *ident* field will contain a chip ID from the Linux media/v4l2-chip-ident.h header file, and the *revision* field will contain a driver specific value, or zero if no particular revision is associated with this chip.

When the driver could not identify the selected chip, *ident* will contain V4L2_IDENT_UNKNOWN. When no chip matched the ioctl will succeed but the *ident* field will contain V4L2_IDENT_NONE. If multiple chips matched, *ident* will contain V4L2_IDENT_AMBIGUOUS. In all these cases the *revision* field remains unchanged.

This ioctl is optional, not all drivers may support it. It was introduced in Linux 2.6.21, but the API was changed to the one described here in 2.6.29.

We recommended the v4l2-dbg utility over calling this ioctl directly. It is available from the LinuxTV v4l-dvb repository; see http://linuxtv.org/repo/ for access instructions.

**Table A-1. struct v4l2_dbg_match**

| | | | |
|---|---|---|---|
| __u32 | *type* | See Table A-3 for a list of possible types. | |
| union | (anonymous) | | |
| | __u32 | *addr* | Match a chip by this number, interpreted according to the *type* field. |
| | char | *name[32]* | Match a chip by this name, interpreted according to the *type* field. |

**Table A-2. struct v4l2_dbg_chip_ident**

| | | |
|---|---|---|
| struct v4l2_dbg_match | *match* | How to match the chip, see Table A-1. |
| __u32 | *ident* | A chip identifier as defined in the Linux media/v4l2-chip-ident.h header file, or one of the values from Table A-4. |
| __u32 | *revision* | A chip revision, chip and driver specific. |

**Table A-3. Chip Match Types**

| | | |
|---|---|---|
| V4L2_CHIP_MATCH_HOST | 0 | Match the nth chip on the card, zero for the host chip. Does not match I2C chips. |
| V4L2_CHIP_MATCH_I2C_DRIVER | 1 | Match an I2C chip by its driver name. |
| V4L2_CHIP_MATCH_I2C_ADDR | 2 | Match a chip by its 7 bit I2C bus address. |
| V4L2_CHIP_MATCH_AC97 | 3 | Match the nth anciliary AC97 chip. |

**Table A-4. Chip Identifiers**

| | | |
|---|---|---|
| V4L2_IDENT_NONE | 0 | No chip matched. |
| V4L2_IDENT_AMBIGUOUS | 1 | Multiple chips matched. |
| V4L2_IDENT_UNKNOWN | 2 | A chip is present at this address, but the driver could not identify it. |

## Return Value

On success 0 is returned, on error -1 and the errno variable is set appropriately:

EINVAL

The driver does not support this ioctl, or the *match_type* is invalid.

# ioctl VIDIOC_DBG_G_REGISTER, VIDIOC_DBG_S_REGISTER

## Name

VIDIOC_DBG_G_REGISTER, VIDIOC_DBG_S_REGISTER — Read or write hardware registers

## Synopsis

```
int ioctl(int fd, int request, struct v4l2_dbg_register
*argp);
```

```
int ioctl(int fd, int request, const struct v4l2_dbg_register
*argp);
```

## Arguments

*fd*

File descriptor returned by `open()`.

*request*

VIDIOC_DBG_G_REGISTER, VIDIOC_DBG_S_REGISTER

*argp*

## Description

**Experimental:** This is an experimental interface and may change in the future.

For driver debugging purposes these ioctls allow test applications to access hardware registers directly. Regular applications must not use them.

Since writing or even reading registers can jeopardize the system security, its stability and damage the hardware, both ioctls require superuser privileges. Additionally the Linux kernel must be compiled with the `CONFIG_VIDEO_ADV_DEBUG` option to enable these ioctls.

To write a register applications must initialize all fields of a struct v4l2_dbg_register and call `VIDIOC_DBG_S_REGISTER` with a pointer to this structure. The *`match.type`* and *`match.addr`* or *`match.name`* fields select a chip on the TV card,

the *reg* field specifies a register number and the *val* field the value to be written into the register.

To read a register applications must initialize the *match.type*, *match.chip* or *match.name* and *reg* fields, and call VIDIOC_DBG_G_REGISTER with a pointer to this structure. On success the driver stores the register value in the *val* field. On failure the structure remains unchanged.

When *match.type* is V4L2_CHIP_MATCH_HOST, *match.addr* selects the nth non-I²C chip on the TV card. The number zero always selects the host chip, e. g. the chip connected to the PCI or USB bus. You can find out which chips are present with the VIDIOC_DBG_G_CHIP_IDENT ioctl.

When *match.type* is V4L2_CHIP_MATCH_I2C_DRIVER, *match.name* contains the I2C driver name. For instance "saa7127" will match any chip supported by the saa7127 driver, regardless of its I²C bus address. When multiple chips supported by the same driver are present, the effect of these ioctls is undefined. Again with the VIDIOC_DBG_G_CHIP_IDENT ioctl you can find out which I²C chips are present.

When *match.type* is V4L2_CHIP_MATCH_I2C_ADDR, *match.addr* selects a chip by its 7 bit I²C bus address.

When *match.type* is V4L2_CHIP_MATCH_AC97, *match.addr* selects the nth AC97 chip on the TV card.

> **Success not guaranteed:** Due to a flaw in the Linux I²C bus driver these ioctls may return successfully without actually reading or writing a register. To catch the most likely failure we recommend a VIDIOC_DBG_G_CHIP_IDENT call confirming the presence of the selected I²C chip.

These ioctls are optional, not all drivers may support them. However when a driver supports these ioctls it must also support VIDIOC_DBG_G_CHIP_IDENT. Conversely it may support VIDIOC_DBG_G_CHIP_IDENT but not these ioctls.

VIDIOC_DBG_G_REGISTER and VIDIOC_DBG_S_REGISTER were introduced in Linux 2.6.21, but their API was changed to the one described here in kernel 2.6.29.

We recommended the v4l2-dbg utility over calling these ioctls directly. It is available from the LinuxTV v4l-dvb repository; see http://linuxtv.org/repo/ for access instructions.

**Table A-1. struct v4l2_dbg_match**

| | | |
|---|---|---|
| __u32 | *type* | See Table A-3 for a list of possible types. |

| | | | |
|---|---|---|---|
| union | (anonymous) | | |
| | __u32 | *addr* | Match a chip by this number, interpreted according to the *type* field. |
| | char | *name[32]* | Match a chip by this name, interpreted according to the *type* field. |

**Table A-2. struct v4l2_dbg_register**

| | | |
|---|---|---|
| struct v4l2_dbg_match | *match* | How to match the chip, see Table A-1. |
| __u64 | *reg* | A register number. |
| __u64 | *val* | The value read from, or to be written into the register. |

**Table A-3. Chip Match Types**

| | | |
|---|---|---|
| V4L2_CHIP_MATCH_HOST | 0 | Match the nth chip on the card, zero for the host chip. Does not match I2C chips. |
| V4L2_CHIP_MATCH_I2C_DRIVER | 1 | Match an I2C chip by its driver name. |
| V4L2_CHIP_MATCH_I2C_ADDR | 2 | Match a chip by its 7 bit I2C bus address. |
| V4L2_CHIP_MATCH_AC97 | 3 | Match the nth anciliary AC97 chip. |

## Return Value

On success 0 is returned, on error -1 and the `errno` variable is set appropriately:

EINVAL

The driver does not support this ioctl, or the kernel was not compiled with the `CONFIG_VIDEO_ADV_DEBUG` option, or the *match_type* is invalid, or the selected chip or register does not exist.

EPERM

Insufficient permissions. Root privileges are required to execute these ioctls.

# ioctl VIDIOC_DQEVENT

## Name

VIDIOC_DQEVENT — Dequeue event

## Synopsis

```
int ioctl(int fd, int request, struct v4l2_event *argp);
```

## Arguments

*fd*

File descriptor returned by open().

*request*

VIDIOC_DQEVENT

*argp*

## Description

Dequeue an event from a video device. No input is required for this ioctl. All the fields of the struct v4l2_event structure are filled by the driver. The file handle will also receive exceptions which the application may get by e.g. using the select system call.

**Table A-1. struct v4l2_event**

| | | | |
|---|---|---|---|
| __u32 | *type* | | Type of the event. |
| union | *u* | | |
| | struct v4l2_event_vsync | | Event data for event V4L2_EVENT_VSYNC. |
| | __u8 | *data*[64] | Event data. Defined by the event type. The union should be used to define easily accessible type for events. |
| __u32 | *pending* | | Number of pending events excluding this one. |
| __u32 | *sequence* | | Event sequence number. The sequence number is incremented for every subscribed event that takes place. If sequence numbers are not contiguous it means that events have been lost. |
| struct timespec | *timestamp* | | Event timestamp. |

| | | |
|---|---|---|
| __u32 | *reserved*[9] | Reserved for future extensions. Drivers must set the array to zero. |

# ioctl VIDIOC_ENCODER_CMD, VIDIOC_TRY_ENCODER_CMD

## Name

VIDIOC_ENCODER_CMD, VIDIOC_TRY_ENCODER_CMD — Execute an encoder command

## Synopsis

```
int ioctl(int fd, int request, struct v4l2_encoder_cmd *argp);
```

## Arguments

*fd*

File descriptor returned by open().

*request*

VIDIOC_ENCODER_CMD, VIDIOC_TRY_ENCODER_CMD

*argp*

# Description

**Experimental:** This is an experimental interface and may change in the future.

These ioctls control an audio/video (usually MPEG-) encoder. VIDIOC_ENCODER_CMD sends a command to the encoder, VIDIOC_TRY_ENCODER_CMD can be used to try a command without actually executing it.

To send a command applications must initialize all fields of a struct v4l2_encoder_cmd and call VIDIOC_ENCODER_CMD or VIDIOC_TRY_ENCODER_CMD with a pointer to this structure.

The *cmd* field must contain the command code. The *flags* field is currently only used by the STOP command and contains one bit: If the V4L2_ENC_CMD_STOP_AT_GOP_END flag is set, encoding will continue until the end of the current *Group Of Pictures*, otherwise it will stop immediately.

A read() call sends a START command to the encoder if it has not been started yet. After a STOP command, read() calls will read the remaining data buffered by the driver. When the buffer is empty, read() will return zero and the next read() call will restart the encoder.

A close() call sends an immediate STOP to the encoder, and all buffered data is discarded.

These ioctls are optional, not all drivers may support them. They were introduced in Linux 2.6.21.

**Table A-1. struct v4l2_encoder_cmd**

| | | |
|---|---|---|
| __u32 | *cmd* | The encoder command, see Table A-2. |
| __u32 | *flags* | Flags to go with the command, see Table A-3. If no flags are defined for this command, drivers and applications must set this field to zero. |
| __u32 | *data*[8] | Reserved for future extensions. Drivers and applications must set the array to zero. |

**Table A-2. Encoder Commands**

| | | |
|---|---|---|
| V4L2_ENC_CMD_START | 0 | Start the encoder. When the encoder is already running or paused, this command does nothing. No flags are defined for this command. |
| V4L2_ENC_CMD_STOP | 1 | Stop the encoder. When the V4L2_ENC_CMD_STOP_AT_GOP_END flag is set, encoding will continue until the end of the current *Group Of Pictures*, otherwise encoding will stop immediately. When the encoder is already stopped, this command does nothing. |
| V4L2_ENC_CMD_PAUSE | 2 | Pause the encoder. When the encoder has not been started yet, the driver will return an EPERM error code. When the encoder is already paused, this command does nothing. No flags are defined for this command. |
| V4L2_ENC_CMD_RESUME | 3 | Resume encoding after a PAUSE command. When the encoder has not been started yet, the driver will return an EPERM error code. When the encoder is already running, this command does nothing. No flags are defined for this command. |

**Table A-3. Encoder Command Flags**

| | | |
|---|---|---|
| V4L2_ENC_CMD_STOP_AT_GOP_END | 0x0001 | Stop encoding at the end of the current *Group Of Pictures*, rather than immediately. |

## Return Value

On success 0 is returned, on error -1 and the `errno` variable is set appropriately:

EINVAL

The driver does not support this ioctl, or the *cmd* field is invalid.

EPERM

The application sent a PAUSE or RESUME command when the encoder was not running.

# ioctl VIDIOC_ENUMAUDIO

## Name

VIDIOC_ENUMAUDIO — Enumerate audio inputs

## Synopsis

```
int ioctl(int fd, int request, struct v4l2_audio *argp);
```

## Arguments

*fd*

File descriptor returned by open().

*request*

VIDIOC_ENUMAUDIO

*argp*

## Description

To query the attributes of an audio input applications initialize the *index* field and zero out the *reserved* array of a struct v4l2_audio and call the VIDIOC_ENUMAUDIO ioctl with a pointer to this structure. Drivers fill the rest of the structure or return an EINVAL error code when the index is out of bounds. To

enumerate all audio inputs applications shall begin at index zero, incrementing by one until the driver returns EINVAL.

See ioctl VIDIOC_G_AUDIO, VIDIOC_S_AUDIO(2) for a description of struct v4l2_audio.

## Return Value

On success 0 is returned, on error -1 and the `errno` variable is set appropriately:

EINVAL

The number of the audio input is out of bounds, or there are no audio inputs at all and this ioctl is not supported.

# ioctl VIDIOC_ENUMAUDOUT

## Name

`VIDIOC_ENUMAUDOUT` — Enumerate audio outputs

## Synopsis

```
int ioctl(int fd, int request, struct v4l2_audioout *argp);
```

## Arguments

*`fd`*

File descriptor returned by `open()`.

*`request`*

VIDIOC_ENUMAUDOUT

*argp*

## Description

To query the attributes of an audio output applications initialize the *index* field and zero out the *reserved* array of a struct v4l2_audioout and call the VIDIOC_G_AUDOUT ioctl with a pointer to this structure. Drivers fill the rest of the structure or return an EINVAL error code when the index is out of bounds. To enumerate all audio outputs applications shall begin at index zero, incrementing by one until the driver returns EINVAL.

Note connectors on a TV card to loop back the received audio signal to a sound card are not audio outputs in this sense.

See ioctl VIDIOC_G_AUDOUT, VIDIOC_S_AUDOUT(2) for a description of struct v4l2_audioout.

## Return Value

On success 0 is returned, on error -1 and the errno variable is set appropriately:

EINVAL

The number of the audio output is out of bounds, or there are no audio outputs at all and this ioctl is not supported.

# ioctl VIDIOC_ENUM_DV_PRESETS

## Name

VIDIOC_ENUM_DV_PRESETS — Enumerate supported Digital Video presets

# Synopsis

```
int ioctl(int fd, int request, struct v4l2_dv_enum_preset
*argp);
```

# Arguments

*fd*

File descriptor returned by `open()`.

*request*

VIDIOC_ENUM_DV_PRESETS

*argp*

# Description

To query the attributes of a DV preset, applications initialize the *index* field and zero the reserved array of struct v4l2_dv_enum_preset and call the `VIDIOC_ENUM_DV_PRESETS` ioctl with a pointer to this structure. Drivers fill the rest of the structure or return an EINVAL error code when the index is out of bounds. To enumerate all DV Presets supported, applications shall begin at index zero, incrementing by one until the driver returns EINVAL. Drivers may enumerate a different set of DV presets after switching the video input or output.

**Table A-1. struct v4l2_dv_enum_presets**

| | | |
|---|---|---|
| __u32 | *index* | Number of the DV preset, set by the application. |
| __u32 | *preset* | This field identifies one of the DV preset values listed in Table A-2. |
| __u8 | *name*[24] | Name of the preset, a NUL-terminated ASCII string, for example: "720P-60", "1080I-60". This information is intended for the user. |

| | | |
|---|---|---|
| __u32 | *width* | Width of the active video in pixels for the DV preset. |
| __u32 | *height* | Height of the active video in lines for the DV preset. |
| __u32 | *reserved*[4] | Reserved for future extensions. Drivers must set the array to zero. |

**Table A-2. struct DV Presets**

| Preset | Preset value | Description |
|---|---|---|
| V4L2_DV_INVALID | 0 | Invalid preset value. |
| V4L2_DV_480P59_94 | 1 | 720x480 progressive video at 59.94 fps as per BT.1362. |
| V4L2_DV_576P50 | 2 | 720x576 progressive video at 50 fps as per BT.1362. |
| V4L2_DV_720P24 | 3 | 1280x720 progressive video at 24 fps as per SMPTE 296M. |
| V4L2_DV_720P25 | 4 | 1280x720 progressive video at 25 fps as per SMPTE 296M. |
| V4L2_DV_720P30 | 5 | 1280x720 progressive video at 30 fps as per SMPTE 296M. |
| V4L2_DV_720P50 | 6 | 1280x720 progressive video at 50 fps as per SMPTE 296M. |
| V4L2_DV_720P59_94 | 7 | 1280x720 progressive video at 59.94 fps as per SMPTE 274M. |
| V4L2_DV_720P60 | 8 | 1280x720 progressive video at 60 fps as per SMPTE 274M/296M. |
| V4L2_DV_1080I29_97 | 9 | 1920x1080 interlaced video at 29.97 fps as per BT.1120/SMPTE 274M. |
| V4L2_DV_1080I30 | 10 | 1920x1080 interlaced video at 30 fps as per BT.1120/SMPTE 274M. |
| V4L2_DV_1080I25 | 11 | 1920x1080 interlaced video at 25 fps as per BT.1120. |
| V4L2_DV_1080I50 | 12 | 1920x1080 interlaced video at 50 fps as per SMPTE 296M. |
| V4L2_DV_1080I60 | 13 | 1920x1080 interlaced video at 60 fps as per SMPTE 296M. |

| | |
|---|---|
| V4L2_DV_1080P24 14 | 1920x1080 progressive video at 24 fps as per SMPTE 296M. |
| V4L2_DV_1080P25 15 | 1920x1080 progressive video at 25 fps as per SMPTE 296M. |
| V4L2_DV_1080P30 16 | 1920x1080 progressive video at 30 fps as per SMPTE 296M. |
| V4L2_DV_1080P50 17 | 1920x1080 progressive video at 50 fps as per BT.1120. |
| V4L2_DV_1080P60 18 | 1920x1080 progressive video at 60 fps as per BT.1120. |

## Return Value

On success 0 is returned, on error -1 and the `errno` variable is set appropriately:

EINVAL

The struct v4l2_dv_enum_preset *index* is out of bounds.

# ioctl VIDIOC_ENUM_FMT

## Name

`VIDIOC_ENUM_FMT` — Enumerate image formats

## Synopsis

```
int ioctl(int fd, int request, struct v4l2_fmtdesc *argp);
```

## Arguments

*fd*

File descriptor returned by open().

*request*

VIDIOC_ENUM_FMT

*argp*

## Description

To enumerate image formats applications initialize the *type* and *index* field of struct v4l2_fmtdesc and call the VIDIOC_ENUM_FMT ioctl with a pointer to this structure. Drivers fill the rest of the structure or return an EINVAL error code. All formats are enumerable by beginning at index zero and incrementing by one until EINVAL is returned.

**Table A-1. struct v4l2_fmtdesc**

| | | |
|---|---|---|
| __u32 | *index* | Number of the format in the enumeration, set by the application. This is in no way related to the *pixelformat* field. |
| enum v4l2_buf_type | *type* | Type of the data stream, set by the application. Only these types are valid here: V4L2_BUF_TYPE_VIDEO_CAPTURE, V4L2_BUF_TYPE_VIDEO_CAPTURE_MPLANE, V4L2_BUF_TYPE_VIDEO_OUTPUT, V4L2_BUF_TYPE_VIDEO_OUTPUT_MPLANE, V4L2_BUF_TYPE_VIDEO_OVERLAY, and custom (driver defined) types with code V4L2_BUF_TYPE_PRIVATE and higher. |
| __u32 | *flags* | See Table A-2 |
| __u8 | *description*[32] | Description of the format, a NUL-terminated ASCII string. This information is intended for the user, for example: "YUV 4:2:2". |

| | | |
|---|---|---|
| __u32 | *pixelformat* | The image format identifier. This is a four character code as computed by the v4l2_fourcc() macro: |

```
#define v4l2_fourcc(a,b,c,d) (((__u32)(a)<<0)|((__u32)(b)<<8)|((__u32)(c)
```

Several image formats are already defined by this specification in Chapter 2. Note these codes are not the same as those used in the Windows world.

| | | |
|---|---|---|
| __u32 | *reserved*[4] | Reserved for future extensions. Drivers must set the array to zero. |

**Table A-2. Image Format Description Flags**

| | | |
|---|---|---|
| V4L2_FMT_FLAG_COMPRESSED | 0x0001 | This is a compressed format. |
| V4L2_FMT_FLAG_EMULATED | 0x0002 | This format is not native to the device but emulated through software (usually libv4l2), where possible try to use a native format instead for better performance. |

## Return Value

On success 0 is returned, on error -1 and the `errno` variable is set appropriately:

EINVAL

The struct v4l2_fmtdesc *type* is not supported or the *index* is out of bounds.

# ioctl VIDIOC_ENUM_FRAMESIZES

## Name

VIDIOC_ENUM_FRAMESIZES — Enumerate frame sizes

# Synopsis

```
int ioctl(int fd, int request, struct v4l2_frmsizeenum *argp);
```

# Arguments

*fd*

File descriptor returned by open().

*request*

VIDIOC_ENUM_FRAMESIZES

*argp*

Pointer to a struct v4l2_frmsizeenum that contains an index and pixel format and receives a frame width and height.

# Description

**Experimental:** This is an experimental interface and may change in the future.

This ioctl allows applications to enumerate all frame sizes (i. e. width and height in pixels) that the device supports for the given pixel format.

The supported pixel formats can be obtained by using the VIDIOC_ENUM_FMT function.

The return value and the content of the *v4l2_frmsizeenum.type* field depend on the type of frame sizes the device supports. Here are the semantics of the function for the different cases:

- **Discrete:** The function returns success if the given index value (zero-based) is valid. The application should increase the index by one for each call until EINVAL is returned. The *v4l2_frmsizeenum.type* field is set to

V4L2_FRMSIZE_TYPE_DISCRETE by the driver. Of the union only the *discrete* member is valid.

- **Step-wise:** The function returns success if the given index value is zero and EINVAL for any other index value. The *v4l2_frmsizeenum.type* field is set to V4L2_FRMSIZE_TYPE_STEPWISE by the driver. Of the union only the *stepwise* member is valid.
- **Continuous:** This is a special case of the step-wise type above. The function returns success if the given index value is zero and EINVAL for any other index value. The *v4l2_frmsizeenum.type* field is set to V4L2_FRMSIZE_TYPE_CONTINUOUS by the driver. Of the union only the *stepwise* member is valid and the *step_width* and *step_height* values are set to 1.

When the application calls the function with index zero, it must check the *type* field to determine the type of frame size enumeration the device supports. Only for the V4L2_FRMSIZE_TYPE_DISCRETE type does it make sense to increase the index value to receive more frame sizes.

Note that the order in which the frame sizes are returned has no special meaning. In particular does it not say anything about potential default format sizes.

Applications can assume that the enumeration data does not change without any interaction from the application itself. This means that the enumeration data is consistent if the application does not perform any other ioctl calls while it runs the frame size enumeration.

## Structs

In the structs below, *IN* denotes a value that has to be filled in by the application, *OUT* denotes values that the driver fills in. The application should zero out all members except for the *IN* fields.

**Table A-1. struct v4l2_frmsize_discrete**

| | | |
|---|---|---|
| __u32 | *width* | Width of the frame [pixel]. |
| __u32 | *height* | Height of the frame [pixel]. |

**Table A-2. struct v4l2_frmsize_stepwise**

| | | |
|---|---|---|
| __u32 | *min_width* | Minimum frame width [pixel]. |
| __u32 | *max_width* | Maximum frame width [pixel]. |
| __u32 | *step_width* | Frame width step size [pixel]. |

| | | |
|---|---|---|
| __u32 | *min_height* | Minimum frame height [pixel]. |
| __u32 | *max_height* | Maximum frame height [pixel]. |
| __u32 | *step_height* | Frame height step size [pixel]. |

**Table A-3. struct v4l2_frmsizeenum**

| | | |
|---|---|---|
| __u32 | *index* | IN: Index of the given frame size in the enumeration. |
| __u32 | *pixel_format* | IN: Pixel format for which the frame sizes are enumerated. |
| __u32 | *type* | OUT: Frame size type the device supports. |
| union | | OUT: Frame size with the given index. |
| | struct v4l2_frmsize_discrete *discrete* | |
| | struct v4l2_frmsize_stepwise *stepwise* | |
| __u32 | *reserved[2]* | Reserved space for future use. |

# Enums

**Table A-4. enum v4l2_frmsizetypes**

| | | |
|---|---|---|
| V4L2_FRMSIZE_TYPE_DISCRETE | 1 | Discrete frame size. |
| V4L2_FRMSIZE_TYPE_CONTINUOUS | 2 | Continuous frame size. |
| V4L2_FRMSIZE_TYPE_STEPWISE | 3 | Step-wise defined frame size. |

## Return Value

On success 0 is returned, on error -1 and the `errno` variable is set appropriately:

See the description section above for a list of return values that `errno` can have.

# ioctl VIDIOC_ENUM_FRAMEINTERVALS

## Name

`VIDIOC_ENUM_FRAMEINTERVALS` — Enumerate frame intervals

## Synopsis

```
int ioctl(int fd, int request, struct v4l2_frmivalenum *argp);
```

## Arguments

*`fd`*

File descriptor returned by `open()`.

*`request`*

VIDIOC_ENUM_FRAMEINTERVALS

*`argp`*

Pointer to a struct v4l2_frmivalenum structure that contains a pixel format and size and receives a frame interval.

## Description

This ioctl allows applications to enumerate all frame intervals that the device supports for the given pixel format and frame size.

The supported pixel formats and frame sizes can be obtained by using the VIDIOC_ENUM_FMT and VIDIOC_ENUM_FRAMESIZES functions.

The return value and the content of the *v4l2_frmivalenum.type* field depend on the type of frame intervals the device supports. Here are the semantics of the function for the different cases:

- **Discrete:** The function returns success if the given index value (zero-based) is valid. The application should increase the index by one for each call until EINVAL is returned. The ‘v4l2_frmivalenum.type‘ field is set to ‘V4L2_FRMIVAL_TYPE_DISCRETE‘ by the driver. Of the union only the ‘discrete‘ member is valid.
- **Step-wise:** The function returns success if the given index value is zero and EINVAL for any other index value. The *v4l2_frmivalenum.type* field is set to V4L2_FRMIVAL_TYPE_STEPWISE by the driver. Of the union only the *stepwise* member is valid.
- **Continuous:** This is a special case of the step-wise type above. The function returns success if the given index value is zero and EINVAL for any other index value. The *v4l2_frmivalenum.type* field is set to V4L2_FRMIVAL_TYPE_CONTINUOUS by the driver. Of the union only the *stepwise* member is valid and the *step* value is set to 1.

When the application calls the function with index zero, it must check the *type* field to determine the type of frame interval enumeration the device supports. Only for the V4L2_FRMIVAL_TYPE_DISCRETE type does it make sense to increase the index value to receive more frame intervals.

Note that the order in which the frame intervals are returned has no special meaning. In particular does it not say anything about potential default frame intervals.

Applications can assume that the enumeration data does not change without any interaction from the application itself. This means that the enumeration data is consistent if the application does not perform any other ioctl calls while it runs the frame interval enumeration.

## Notes

- **Frame intervals and frame rates:** The V4L2 API uses frame intervals instead of frame rates. Given the frame interval the frame rate can be computed as follows:

```
frame_rate = 1 / frame_interval
```

# Structs

In the structs below, *IN* denotes a value that has to be filled in by the application, *OUT* denotes values that the driver fills in. The application should zero out all members except for the *IN* fields.

**Table A-1. struct v4l2_frmival_stepwise**

| | | |
|---|---|---|
| struct v4l2_fract | *min* | Minimum frame interval [s]. |
| struct v4l2_fract | *max* | Maximum frame interval [s]. |
| struct v4l2_fract | *step* | Frame interval step size [s]. |

**Table A-2. struct v4l2_frmivalenum**

| | | | |
|---|---|---|---|
| __u32 | *index* | | IN: Index of the given frame interval in the enumeration. |
| __u32 | *pixel_format* | | IN: Pixel format for which the frame intervals are enumerated. |
| __u32 | *width* | | IN: Frame width for which the frame intervals are enumerated. |
| __u32 | *height* | | IN: Frame height for which the frame intervals are enumerated. |
| __u32 | *type* | | OUT: Frame interval type the device supports. |
| union | | | OUT: Frame interval with the given index. |
| | struct v4l2_fract | *discrete* | Frame interval [s]. |
| | struct v4l2_frmival_stepwise | *stepwise* | |
| __u32 | *reserved[2]* | | Reserved space for future use. |

## Enums

**Table A-3. enum v4l2_frmivaltypes**

| | | |
|---|---|---|
| V4L2_FRMIVAL_TYPE_DISCRETE | 1 | Discrete frame interval. |
| V4L2_FRMIVAL_TYPE_CONTINUOUS | 2 | Continuous frame interval. |
| V4L2_FRMIVAL_TYPE_STEPWISE | 3 | Step-wise defined frame interval. |

## Return Value

On success 0 is returned, on error -1 and the `errno` variable is set appropriately:

See the description section above for a list of return values that `errno` can have.

# ioctl VIDIOC_ENUMINPUT

## Name

`VIDIOC_ENUMINPUT` — Enumerate video inputs

## Synopsis

```
int ioctl(int fd, int request, struct v4l2_input *argp);
```

## Arguments

*`fd`*

File descriptor returned by `open()`.

*request*

VIDIOC_ENUMINPUT

*argp*

## Description

To query the attributes of a video input applications initialize the *index* field of struct v4l2_input and call the VIDIOC_ENUMINPUT ioctl with a pointer to this structure. Drivers fill the rest of the structure or return an EINVAL error code when the index is out of bounds. To enumerate all inputs applications shall begin at index zero, incrementing by one until the driver returns EINVAL.

**Table A-1. struct v4l2_input**

| | | |
|---|---|---|
| __u32 | *index* | Identifies the input, set by the application. |
| __u8 | *name*[32] | Name of the video input, a NUL-terminated ASCII string, for example: "Vin (Composite 2)". This information is intended for the user, preferably the connector label on the device itself. |
| __u32 | *type* | Type of the input, see Table A-2. |

| | | |
|---|---|---|
| __u32 | *audioset* | Drivers can enumerate up to 32 video and audio inputs. This field shows which audio inputs were selectable as audio source if this was the currently selected video input. It is a bit mask. The LSB corresponds to audio input 0, the MSB to input 31. Any number of bits can be set, or none.<br>When the driver does not enumerate audio inputs no bits must be set. Applications shall not interpret this as lack of audio support. Some drivers automatically select audio sources and do not enumerate them since there is no choice anyway.<br><br>For details on audio inputs and how to select the current input see Section 1.5. |
| __u32 | *tuner* | Capture devices can have zero or more tuners (RF demodulators). When the *type* is set to V4L2_INPUT_TYPE_TUNER this is an RF connector and this field identifies the tuner. It corresponds to struct v4l2_tuner field *index*. For details on tuners see Section 1.6. |
| v4l2_std_id | *std* | Every video input supports one or more different video standards. This field is a set of all supported standards. For details on video standards and how to switch see Section 1.7. |
| __u32 | *status* | This field provides status information about the input. See Table A-3 for flags. With the exception of the sensor orientation bits *status* is only valid when this is the current input. |
| __u32 | *capabilities* | This field provides capabilities for the input. See Table A-4 for flags. |
| __u32 | *reserved*[3] | Reserved for future extensions. Drivers must set the array to zero. |

**Table A-2. Input Types**

| | | |
|---|---|---|
| V4L2_INPUT_TYPE_TUNER | 1 | This input uses a tuner (RF demodulator). |
| V4L2_INPUT_TYPE_CAMERA | 2 | Analog baseband input, for example CVBS / Composite Video, S-Video, RGB. |

**Table A-3. Input Status Flags**

| | | |
|---|---|---|
| General | | |
| V4L2_IN_ST_NO_POWER | 0x00000001 | Attached device is off. |
| V4L2_IN_ST_NO_SIGNAL | 0x00000002 | |
| V4L2_IN_ST_NO_COLOR | 0x00000004 | The hardware supports color decoding, but does not detect color modulation in the signal. |
| Sensor Orientation | | |
| V4L2_IN_ST_HFLIP | 0x00000010 | The input is connected to a device that produces a signal that is flipped horizontally and does not correct this before passing the signal to userspace. |
| V4L2_IN_ST_VFLIP | 0x00000020 | The input is connected to a device that produces a signal that is flipped vertically and does not correct this before passing the signal to userspace. Note that a 180 degree rotation is the same as HFLIP \| VFLIP |
| Analog Video | | |
| V4L2_IN_ST_NO_H_LOCK | 0x00000100 | No horizontal sync lock. |

| | | |
|---|---|---|
| V4L2_IN_ST_COLOR_KILL | 0x00000200 | A color killer circuit automatically disables color decoding when it detects no color modulation. When this flag is set the color killer is enabled *and* has shut off color decoding. |
| Digital Video | | |
| V4L2_IN_ST_NO_SYNC | 0x00010000 | No synchronization lock. |
| V4L2_IN_ST_NO_EQU | 0x00020000 | No equalizer lock. |
| V4L2_IN_ST_NO_CARRIER | 0x00040000 | Carrier recovery failed. |
| VCR and Set-Top Box | | |
| V4L2_IN_ST_MACROVISION | 0x01000000 | Macrovision is an analog copy prevention system mangling the video signal to confuse video recorders. When this flag is set Macrovision has been detected. |
| V4L2_IN_ST_NO_ACCESS | 0x02000000 | Conditional access denied. |
| V4L2_IN_ST_VTR | 0x04000000 | VTR time constant. [?] |

**Table A-4. Input capabilities**

| | | |
|---|---|---|
| V4L2_IN_CAP_PRESETS | 0x00000001 | This input supports setting DV presets by using VIDIOC_S_DV_PRESET. |
| V4L2_IN_CAP_CUSTOM_TIMINGS | 0x00000002 | This input supports setting custom video timings by using VIDIOC_S_DV_TIMINGS. |
| V4L2_IN_CAP_STD | 0x00000004 | This input supports setting the TV standard by using VIDIOC_S_STD. |

# Return Value

On success 0 is returned, on error -1 and the `errno` variable is set appropriately:

EINVAL

The struct v4l2_input *index* is out of bounds.

# ioctl VIDIOC_ENUMOUTPUT

## Name

VIDIOC_ENUMOUTPUT — Enumerate video outputs

## Synopsis

```
int ioctl(int fd, int request, struct v4l2_output *argp);
```

## Arguments

*fd*

File descriptor returned by open().

*request*

VIDIOC_ENUMOUTPUT

*argp*

## Description

To query the attributes of a video outputs applications initialize the *index* field of struct v4l2_output and call the VIDIOC_ENUMOUTPUT ioctl with a pointer to this structure. Drivers fill the rest of the structure or return an EINVAL error code when the index is out of bounds. To enumerate all outputs applications shall begin at index zero, incrementing by one until the driver returns EINVAL.

**Table A-1. struct v4l2_output**

| | | |
|---|---|---|
| __u32 | *index* | Identifies the output, set by the application. |
| __u8 | *name*[32] | Name of the video output, a NUL-terminated ASCII string, for example: "Vout". This information is intended for the user, preferably the connector label on the device itself. |
| __u32 | *type* | Type of the output, see Table A-2. |
| __u32 | *audioset* | Drivers can enumerate up to 32 video and audio outputs. This field shows which audio outputs were selectable as the current output if this was the currently selected video output. It is a bit mask. The LSB corresponds to audio output 0, the MSB to output 31. Any number of bits can be set, or none. When the driver does not enumerate audio outputs no bits must be set. Applications shall not interpret this as lack of audio support. Drivers may automatically select audio outputs without enumerating them.<br><br>For details on audio outputs and how to select the current output see Section 1.5. |
| __u32 | *modulator* | Output devices can have zero or more RF modulators. When the *type* is V4L2_OUTPUT_TYPE_MODULATOR this is an RF connector and this field identifies the modulator. It corresponds to struct v4l2_modulator field *index*. For details on modulators see Section 1.6. |
| v4l2_std_id | *std* | Every video output supports one or more different video standards. This field is a set of all supported standards. For details on video standards and how to switch see Section 1.7. |

| | | |
|---|---|---|
| __u32 | *capabilities* | This field provides capabilities for the output. See Table A-3 for flags. |
| __u32 | *reserved*[3] | Reserved for future extensions. Drivers must set the array to zero. |

**Table A-2. Output Type**

| | | |
|---|---|---|
| V4L2_OUTPUT_TYPE_MODULATOR | 1 | This output is an analog TV modulator. |
| V4L2_OUTPUT_TYPE_ANALOG | 2 | Analog baseband output, for example Composite / CVBS, S-Video, RGB. |
| V4L2_OUTPUT_TYPE_ANALOGVGAOVERLAY | 3 | [?] |

**Table A-3. Output capabilities**

| | | |
|---|---|---|
| V4L2_OUT_CAP_PRESETS | 0x00000001 | This output supports setting DV presets by using VIDIOC_S_DV_PRESET. |
| V4L2_OUT_CAP_CUSTOM_TIMINGS | 0x00000002 | This output supports setting custom video timings by using VIDIOC_S_DV_TIMINGS. |
| V4L2_OUT_CAP_STD | 0x00000004 | This output supports setting the TV standard by using VIDIOC_S_STD. |

# Return Value

On success 0 is returned, on error -1 and the `errno` variable is set appropriately:

EINVAL

The struct v4l2_output *index* is out of bounds.

# ioctl VIDIOC_ENUMSTD

## Name

VIDIOC_ENUMSTD — Enumerate supported video standards

## Synopsis

```
int ioctl(int fd, int request, struct v4l2_standard *argp);
```

## Arguments

*fd*

File descriptor returned by open().

*request*

VIDIOC_ENUMSTD

*argp*

## Description

To query the attributes of a video standard, especially a custom (driver defined) one, applications initialize the *index* field of struct v4l2_standard and call the VIDIOC_ENUMSTD ioctl with a pointer to this structure. Drivers fill the rest of the structure or return an EINVAL error code when the index is out of bounds. To enumerate all standards applications shall begin at index zero, incrementing by one until the driver returns EINVAL. Drivers may enumerate a different set of standards after switching the video input or output.[1]

**Table A-1. struct v4l2_standard**

| | | |
|---|---|---|
| __u32 | *index* | Number of the video standard, set by the application. |

| | | |
|---|---|---|
| v4l2_std_id | *id* | The bits in this field identify the standard as one of the common standards listed in Table A-3, or if bits 32 to 63 are set as custom standards. Multiple bits can be set if the hardware does not distinguish between these standards, however separate indices do not indicate the opposite. The *id* must be unique. No other enumerated v4l2_standard structure, for this input or output anyway, can contain the same set of bits. |
| __u8 | *name*[24] | Name of the standard, a NUL-terminated ASCII string, for example: "PAL-B/G", "NTSC Japan". This information is intended for the user. |
| struct v4l2_fract | *frameperiod* | The frame period (not field period) is numerator / denominator. For example M/NTSC has a frame period of 1001 / 30000 seconds. |
| __u32 | *framelines* | Total lines per frame including blanking, e. g. 625 for B/PAL. |
| __u32 | *reserved*[4] | Reserved for future extensions. Drivers must set the array to zero. |

**Table A-2. struct v4l2_fract**

| | |
|---|---|
| __u32 | *numerator* |
| __u32 | *denominator* |

**Table A-3. typedef v4l2_std_id**

| | | |
|---|---|---|
| __u64 | *v4l2_std_id* | This type is a set, each bit representing another video standard as listed below and in Table A-4. The 32 most significant bits are reserved for custom (driver defined) video standards. |

```
#define V4L2_STD_PAL_B          ((v4l2_std_id)0x00000001)
```

```
#define V4L2_STD_PAL_B1         ((v4l2_std_id)0x00000002)
#define V4L2_STD_PAL_G          ((v4l2_std_id)0x00000004)
#define V4L2_STD_PAL_H          ((v4l2_std_id)0x00000008)
#define V4L2_STD_PAL_I          ((v4l2_std_id)0x00000010)
#define V4L2_STD_PAL_D          ((v4l2_std_id)0x00000020)
#define V4L2_STD_PAL_D1         ((v4l2_std_id)0x00000040)
#define V4L2_STD_PAL_K          ((v4l2_std_id)0x00000080)

#define V4L2_STD_PAL_M          ((v4l2_std_id)0x00000100)
#define V4L2_STD_PAL_N          ((v4l2_std_id)0x00000200)
#define V4L2_STD_PAL_Nc         ((v4l2_std_id)0x00000400)
#define V4L2_STD_PAL_60         ((v4l2_std_id)0x00000800)
```

V4L2_STD_PAL_60 is a hybrid standard with 525 lines, 60 Hz refresh rate, and PAL color modulation with a 4.43 MHz color subcarrier. Some PAL video recorders can play back NTSC tapes in this mode for display on a 50/60 Hz agnostic PAL TV.

```
#define V4L2_STD_NTSC_M         ((v4l2_std_id)0x00001000)
#define V4L2_STD_NTSC_M_JP      ((v4l2_std_id)0x00002000)
#define V4L2_STD_NTSC_443       ((v4l2_std_id)0x00004000)
```

V4L2_STD_NTSC_443 is a hybrid standard with 525 lines, 60 Hz refresh rate, and NTSC color modulation with a 4.43 MHz color subcarrier.

```
#define V4L2_STD_NTSC_M_KR      ((v4l2_std_id)0x00008000)

#define V4L2_STD_SECAM_B        ((v4l2_std_id)0x00010000)
#define V4L2_STD_SECAM_D        ((v4l2_std_id)0x00020000)
#define V4L2_STD_SECAM_G        ((v4l2_std_id)0x00040000)
#define V4L2_STD_SECAM_H        ((v4l2_std_id)0x00080000)
#define V4L2_STD_SECAM_K        ((v4l2_std_id)0x00100000)
#define V4L2_STD_SECAM_K1       ((v4l2_std_id)0x00200000)
#define V4L2_STD_SECAM_L        ((v4l2_std_id)0x00400000)
#define V4L2_STD_SECAM_LC       ((v4l2_std_id)0x00800000)

/* ATSC/HDTV */
#define V4L2_STD_ATSC_8_VSB     ((v4l2_std_id)0x01000000)
#define V4L2_STD_ATSC_16_VSB    ((v4l2_std_id)0x02000000)
```

V4L2_STD_ATSC_8_VSB and V4L2_STD_ATSC_16_VSB are U.S. terrestrial digital TV standards. Presently the V4L2 API does not support digital TV. See also the Linux DVB API at http://linuxtv.org.

```
#define V4L2_STD_PAL_BG         (V4L2_STD_PAL_B         |\
     V4L2_STD_PAL_B1        |\
     V4L2_STD_PAL_G)
#define V4L2_STD_B              (V4L2_STD_PAL_B         |\
     V4L2_STD_PAL_B1        |\
     V4L2_STD_SECAM_B)
#define V4L2_STD_GH             (V4L2_STD_PAL_G         |\
     V4L2_STD_PAL_H         |\
     V4L2_STD_SECAM_G       |\
     V4L2_STD_SECAM_H)
#define V4L2_STD_PAL_DK         (V4L2_STD_PAL_D         |\
     V4L2_STD_PAL_D1        |\
     V4L2_STD_PAL_K)
#define V4L2_STD_PAL            (V4L2_STD_PAL_BG        |\
     V4L2_STD_PAL_DK        |\
     V4L2_STD_PAL_H         |\
     V4L2_STD_PAL_I)
#define V4L2_STD_NTSC           (V4L2_STD_NTSC_M        |\
     V4L2_STD_NTSC_M_JP     |\
     V4L2_STD_NTSC_M_KR)
#define V4L2_STD_MN             (V4L2_STD_PAL_M         |\
     V4L2_STD_PAL_N         |\
     V4L2_STD_PAL_Nc        |\
     V4L2_STD_NTSC)
#define V4L2_STD_SECAM_DK       (V4L2_STD_SECAM_D       |\
     V4L2_STD_SECAM_K       |\
     V4L2_STD_SECAM_K1)
#define V4L2_STD_DK             (V4L2_STD_PAL_DK        |\
     V4L2_STD_SECAM_DK)

#define V4L2_STD_SECAM          (V4L2_STD_SECAM_B       |\
     V4L2_STD_SECAM_G       |\
     V4L2_STD_SECAM_H       |\
     V4L2_STD_SECAM_DK      |\
     V4L2_STD_SECAM_L       |\
     V4L2_STD_SECAM_LC)

#define V4L2_STD_525_60         (V4L2_STD_PAL_M         |\
     V4L2_STD_PAL_60        |\
     V4L2_STD_NTSC          |\
     V4L2_STD_NTSC_443)
#define V4L2_STD_625_50         (V4L2_STD_PAL           |\
     V4L2_STD_PAL_N         |\
     V4L2_STD_PAL_Nc        |\
     V4L2_STD_SECAM)

#define V4L2_STD_UNKNOWN        0
#define V4L2_STD_ALL            (V4L2_STD_525_60        |\
```

V4L2_STD_625_50)

**Table A-4. Video Standards (based on [ITU BT.470])**

| Characteristics | M/NTSC[a] | M/PAL | N/PAL | B, B1, G/PAL | D, D1, K/PAL | H/PAL | I/PAL | B, G/SECAM | D, K/SECAM | K1/SECAM | L/SECAM |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Frame lines | 525 | | | 625 | | | | | | | |
| Frame period (s) | 1001/30000 | | | 1/25 | | | | | | | |
| Chrominance sub-carrier frequency (Hz) | 3579545 ± 10 | 3579611.49 ± 10 | 3582056.25 ± 5 | 4433618.75 ± 5 (3582056.25 ± 5) | | | 4433618.75 ± 1 | $f_{OR}$ = 4406250 ± 2000, $f_{OB}$ = 4250000 ± 2000 | | | |
| Nominal radio-frequency channel bandwidth (MHz) | 6 | 6 | 6 | B: 7; B1, G: 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 |
| Sound carrier relative to vision carrier (MHz) | + 4.5 | + 4.5 | + 4.5 | + 5.5 ± 0.001[c d e f] | + 6.5 ± 0.001 | + 5.5 ± 0.001 | + 5.9996 ± 0.0005 | + 5.5 ± 0.001[e] | + 6.5 ± 0.001 | + 6.5 ± 0.001 | + 6.5[g] |

| Characteristics | M/NTSC[a] | M/PAL | N/PAL[b] | B, B1, G/PAL | D, D1, K/PAL | H/PAL | I/PAL | B, G/SECAM | D, K/SECAM | K1/SECAM | L/SECAM |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Notes:<br>a. Japan uses a standard similar to M/NTSC (V4L2_STD_NTSC_M_JP).<br>b. The values in brackets apply to the combination N/PAL a.k.a. Nc used in Argentina (V4L2_STD_PAL_Nc).<br>c. In the Federal Republic of Germany, Austria, Italy, the Netherlands, Slovakia and Switzerland a system of two sound carriers is used, the frequency of the second carrier being 242.1875 kHz above the frequency of the first sound carrier. For stereophonic sound transmissions a similar system is used in Australia.<br>d. New Zealand uses a sound carrier displaced 5.4996 ± 0.0005 MHz from the vision carrier.<br>e. In Denmark, Finland, New Zealand, Sweden and Spain a system of two sound carriers is used. In Iceland, Norway and Poland the same system is being introduced. The second carrier is 5.85 MHz above the vision carrier and is DQPSK modulated with 728 kbit/s sound and data multiplex. (NICAM system)<br>f. In the United Kingdom, a system of two sound carriers is used. The second sound carrier is 6.552 MHz above the vision carrier and is DQPSK modulated with a 728 kbit/s sound and data multiplex able to carry two sound channels. (NICAM system)<br>g. In France, a digital carrier 5.85 MHz away from the vision carrier may be used in addition to the main sound carrier. It is modulated in differentially encoded QPSK with a 728 kbit/s sound and data multiplexer capable of carrying two sound channels. (NICAM system) | | | | | | | | | | | |

## Return Value

On success 0 is returned, on error -1 and the `errno` variable is set appropriately:

EINVAL

The struct v4l2_standard *`index`* is out of bounds.

## Notes

1. The supported standards may overlap and we need an unambiguous set to find the current standard returned by `VIDIOC_G_STD`.

# ioctl VIDIOC_G_AUDIO, VIDIOC_S_AUDIO

## Name

VIDIOC_G_AUDIO, VIDIOC_S_AUDIO — Query or select the current audio input and its attributes

## Synopsis

```
int ioctl(int fd, int request, struct v4l2_audio *argp);
```

```
int ioctl(int fd, int request, const struct v4l2_audio *argp);
```

## Arguments

*fd*

File descriptor returned by open().

*request*

VIDIOC_G_AUDIO, VIDIOC_S_AUDIO

*argp*

## Description

To query the current audio input applications zero out the *reserved* array of a struct v4l2_audio and call the VIDIOC_G_AUDIO ioctl with a pointer to this structure. Drivers fill the rest of the structure or return an EINVAL error code when the device has no audio inputs, or none which combine with the current video input.

Audio inputs have one writable property, the audio mode. To select the current audio input *and* change the audio mode, applications initialize the *index* and *mode* fields, and the *reserved* array of a v4l2_audio structure and call the VIDIOC_S_AUDIO

ioctl. Drivers may switch to a different audio mode if the request cannot be satisfied. However, this is a write-only ioctl, it does not return the actual new audio mode.

**Table A-1. struct v4l2_audio**

| | | |
|---|---|---|
| __u32 | *index* | Identifies the audio input, set by the driver or application. |
| __u8 | *name*[32] | Name of the audio input, a NUL-terminated ASCII string, for example: "Line In". This information is intended for the user, preferably the connector label on the device itself. |
| __u32 | *capability* | Audio capability flags, see Table A-2. |
| __u32 | *mode* | Audio mode flags set by drivers and applications (on VIDIOC_S_AUDIO ioctl), see Table A-3. |
| __u32 | *reserved*[2] | Reserved for future extensions. Drivers and applications must set the array to zero. |

**Table A-2. Audio Capability Flags**

| | | |
|---|---|---|
| V4L2_AUDCAP_STEREO | 0x00001 | This is a stereo input. The flag is intended to automatically disable stereo recording etc. when the signal is always monaural. The API provides no means to detect if stereo is *received*, unless the audio input belongs to a tuner. |
| V4L2_AUDCAP_AVL | 0x00002 | Automatic Volume Level mode is supported. |

**Table A-3. Audio Mode Flags**

| | | |
|---|---|---|
| V4L2_AUDMODE_AVL | 0x00001 | AVL mode is on. |

## Return Value

On success 0 is returned, on error -1 and the errno variable is set appropriately:

EINVAL

No audio inputs combine with the current video input, or the number of the selected audio input is out of bounds or it does not combine, or there are no audio inputs at all and the ioctl is not supported.

EBUSY

I/O is in progress, the input cannot be switched.

# ioctl VIDIOC_G_AUDOUT, VIDIOC_S_AUDOUT

## Name

VIDIOC_G_AUDOUT, VIDIOC_S_AUDOUT — Query or select the current audio output

## Synopsis

```
int ioctl(int fd, int request, struct v4l2_audioout *argp);
```

```
int ioctl(int fd, int request, const struct v4l2_audioout
*argp);
```

## Arguments

*fd*

File descriptor returned by open().

*request*

VIDIOC_G_AUDOUT, VIDIOC_S_AUDOUT

*argp*

## Description

To query the current audio output applications zero out the *reserved* array of a struct v4l2_audioout and call the VIDIOC_G_AUDOUT ioctl with a pointer to this structure. Drivers fill the rest of the structure or return an EINVAL error code when the device has no audio inputs, or none which combine with the current video output.

Audio outputs have no writable properties. Nevertheless, to select the current audio output applications can initialize the *index* field and *reserved* array (which in the future may contain writable properties) of a v4l2_audioout structure and call the VIDIOC_S_AUDOUT ioctl. Drivers switch to the requested output or return the EINVAL error code when the index is out of bounds. This is a write-only ioctl, it does not return the current audio output attributes as VIDIOC_G_AUDOUT does.

Note connectors on a TV card to loop back the received audio signal to a sound card are not audio outputs in this sense.

**Table A-1. struct v4l2_audioout**

| | | |
|---|---|---|
| __u32 | *index* | Identifies the audio output, set by the driver or application. |
| __u8 | *name*[32] | Name of the audio output, a NUL-terminated ASCII string, for example: "Line Out". This information is intended for the user, preferably the connector label on the device itself. |
| __u32 | *capability* | Audio capability flags, none defined yet. Drivers must set this field to zero. |
| __u32 | *mode* | Audio mode, none defined yet. Drivers and applications (on VIDIOC_S_AUDOUT) must set this field to zero. |
| __u32 | *reserved*[2] | Reserved for future extensions. Drivers and applications must set the array to zero. |

### Return Value

On success 0 is returned, on error -1 and the `errno` variable is set appropriately:

EINVAL

No audio outputs combine with the current video output, or the number of the selected audio output is out of bounds or it does not combine, or there are no audio outputs at all and the ioctl is not supported.

EBUSY

I/O is in progress, the output cannot be switched.

## ioctl VIDIOC_G_CROP, VIDIOC_S_CROP

### Name

`VIDIOC_G_CROP, VIDIOC_S_CROP` — Get or set the current cropping rectangle

### Synopsis

```
int ioctl(int fd, int request, struct v4l2_crop *argp);
```

```
int ioctl(int fd, int request, const struct v4l2_crop *argp);
```

### Arguments

*fd*

File descriptor returned by `open()`.

*request*

VIDIOC_G_CROP, VIDIOC_S_CROP

*argp*

## Description

To query the cropping rectangle size and position applications set the *type* field of a v4l2_crop structure to the respective buffer (stream) type and call the VIDIOC_G_CROP ioctl with a pointer to this structure. The driver fills the rest of the structure or returns the EINVAL error code if cropping is not supported.

To change the cropping rectangle applications initialize the *type* and struct v4l2_rect substructure named *c* of a v4l2_crop structure and call the VIDIOC_S_CROP ioctl with a pointer to this structure.

The driver first adjusts the requested dimensions against hardware limits, i. e. the bounds given by the capture/output window, and it rounds to the closest possible values of horizontal and vertical offset, width and height. In particular the driver must round the vertical offset of the cropping rectangle to frame lines modulo two, such that the field order cannot be confused.

Second the driver adjusts the image size (the opposite rectangle of the scaling process, source or target depending on the data direction) to the closest size possible while maintaining the current horizontal and vertical scaling factor.

Finally the driver programs the hardware with the actual cropping and image parameters. VIDIOC_S_CROP is a write-only ioctl, it does not return the actual parameters. To query them applications must call VIDIOC_G_CROP and VIDIOC_G_FMT. When the parameters are unsuitable the application may modify the cropping or image parameters and repeat the cycle until satisfactory parameters have been negotiated.

When cropping is not supported then no parameters are changed and VIDIOC_S_CROP returns the EINVAL error code.

**Table A-1. struct v4l2_crop**

| | | |
|---|---|---|
| enum v4l2_buf_type | *type* | Type of the data stream, set by the application. Only these types are valid here: V4L2_BUF_TYPE_VIDEO_CAPTURE, V4L2_BUF_TYPE_VIDEO_OUTPUT, V4L2_BUF_TYPE_VIDEO_OVERLAY, and custom (driver defined) types with code V4L2_BUF_TYPE_PRIVATE and higher. |
| struct v4l2_rect | *c* | Cropping rectangle. The same co-ordinate system as for struct v4l2_cropcap *bounds* is used. |

## Return Value

On success 0 is returned, on error -1 and the `errno` variable is set appropriately:

EINVAL

Cropping is not supported.

# ioctl VIDIOC_G_CTRL, VIDIOC_S_CTRL

## Name

VIDIOC_G_CTRL, VIDIOC_S_CTRL — Get or set the value of a control

## Synopsis

```
int ioctl(int fd, int request, struct v4l2_control *argp);
```

## Arguments

*fd*

File descriptor returned by open().

*request*

VIDIOC_G_CTRL, VIDIOC_S_CTRL

*argp*

## Description

To get the current value of a control applications initialize the *id* field of a struct v4l2_control and call the VIDIOC_G_CTRL ioctl with a pointer to this structure. To change the value of a control applications initialize the *id* and *value* fields of a struct v4l2_control and call the VIDIOC_S_CTRL ioctl.

When the *id* is invalid drivers return an EINVAL error code. When the *value* is out of bounds drivers can choose to take the closest valid value or return an ERANGE error code, whatever seems more appropriate. However, VIDIOC_S_CTRL is a write-only ioctl, it does not return the actual new value.

These ioctls work only with user controls. For other control classes the VIDIOC_G_EXT_CTRLS, VIDIOC_S_EXT_CTRLS or VIDIOC_TRY_EXT_CTRLS must be used.

**Table A-1. struct v4l2_control**

| | | |
|---|---|---|
| __u32 | *id* | Identifies the control, set by the application. |
| __s32 | *value* | New value or current value. |

## Return Value

On success 0 is returned, on error -1 and the errno variable is set appropriately:

EINVAL

The struct v4l2_control *id* is invalid.

ERANGE

The struct v4l2_control *value* is out of bounds.

EBUSY

The control is temporarily not changeable, possibly because another applications took over control of the device function this control belongs to.

# ioctl VIDIOC_G_DV_PRESET, VIDIOC_S_DV_PRESET

## Name

VIDIOC_G_DV_PRESET, VIDIOC_S_DV_PRESET — Query or select the DV preset of the current input or output

## Synopsis

```
int ioctl(int fd, int request, struct v4l2_dv_preset *argp);
```

## Arguments

*fd*

File descriptor returned by `open()`.

*request*

VIDIOC_G_DV_PRESET, VIDIOC_S_DV_PRESET

*argp*

## Description

To query and select the current DV preset, applications use the VIDIOC_G_DV_PRESET and VIDIOC_S_DV_PRESET ioctls which take a pointer to a struct v4l2_dv_preset type as argument. Applications must zero the reserved array in struct v4l2_dv_preset. VIDIOC_G_DV_PRESET returns a dv preset in the field *preset* of struct v4l2_dv_preset.

VIDIOC_S_DV_PRESET accepts a pointer to a struct v4l2_dv_preset that has the preset value to be set. Applications must zero the reserved array in struct v4l2_dv_preset. If the preset is not supported, it returns an EINVAL error code

## Return Value

On success 0 is returned, on error -1 and the errno variable is set appropriately:

EINVAL

This ioctl is not supported, or the VIDIOC_S_DV_PRESET,VIDIOC_S_DV_PRESET parameter was unsuitable.

EBUSY

The device is busy and therefore can not change the preset.

**Table A-1. struct v4l2_dv_preset**

| | | |
|---|---|---|
| __u32 | *preset* | Preset value to represent the digital video timings |
| __u32 | *reserved[4]* | Reserved fields for future use |

# ioctl VIDIOC_G_DV_TIMINGS, VIDIOC_S_DV_TIMINGS

## Name

VIDIOC_G_DV_TIMINGS, VIDIOC_S_DV_TIMINGS — Get or set custom

DV timings for input or output

## Synopsis

```
int ioctl(int fd, int request, struct v4l2_dv_timings *argp);
```

## Arguments

*fd*

File descriptor returned by `open()`.

*request*

VIDIOC_G_DV_TIMINGS, VIDIOC_S_DV_TIMINGS

*argp*

## Description

To set custom DV timings for the input or output, applications use the `VIDIOC_S_DV_TIMINGS` ioctl and to get the current custom timings, applications use the `VIDIOC_G_DV_TIMINGS` ioctl. The detailed timing information is filled in using the structure struct v4l2_dv_timings. These ioctls take a pointer to the struct v4l2_dv_timings structure as argument. If the ioctl is not supported or the timing values are not correct, the driver returns EINVAL error code.

## Return Value

On success 0 is returned, on error -1 and the `errno` variable is set appropriately:

EINVAL

This ioctl is not supported, or the `VIDIOC_S_DV_TIMINGS` parameter was unsuitable.

EBUSY

The device is busy and therefore can not change the timings.

**Table A-1. struct v4l2_bt_timings**

| | | |
|---|---|---|
| __u32 | *width* | Width of the active video in pixels |
| __u32 | *height* | Height of the active video in lines |
| __u32 | *interlaced* | Progressive (0) or interlaced (1) |
| __u32 | *polarities* | This is a bit mask that defines polarities of sync signals. bit 0 (V4L2_DV_VSYNC_POS_POL) is for vertical sync polarity and bit 1 (V4L2_DV_HSYNC_POS_POL) is for horizontal sync polarity. If the bit is set (1) it is positive polarity and if is cleared (0), it is negative polarity. |
| __u64 | *pixelclock* | Pixel clock in Hz. Ex. 74.25MHz->74250000 |
| __u32 | *hfrontporch* | Horizontal front porch in pixels |
| __u32 | *hsync* | Horizontal sync length in pixels |
| __u32 | *hbackporch* | Horizontal back porch in pixels |
| __u32 | *vfrontporch* | Vertical front porch in lines |
| __u32 | *vsync* | Vertical sync length in lines |
| __u32 | *vbackporch* | Vertical back porch in lines |
| __u32 | *il_vfrontporch* | Vertical front porch in lines for bottom field of interlaced field formats |
| __u32 | *il_vsync* | Vertical sync length in lines for bottom field of interlaced field formats |
| __u32 | *il_vbackporch* | Vertical back porch in lines for bottom field of interlaced field formats |

**Table A-2. struct v4l2_dv_timings**

| | | |
|---|---|---|
| __u32 | *type* | Type of DV timings as listed in Table A-3. |
| union | | |

| | | |
|---|---|---|
| struct v4l2_bt_timings | bt | Timings defined by BT.656/1120 specifications |
| __u32 | *reserved*[32] | |

**Table A-3. DV Timing types**

| Timing type | value | Description |
|---|---|---|
| V4L2_DV_BT_656_1120 | 0 | BT.656/1120 timings |

# ioctl VIDIOC_G_ENC_INDEX

## Name

VIDIOC_G_ENC_INDEX — Get meta data about a compressed video stream

## Synopsis

```
int ioctl(int fd, int request, struct v4l2_enc_idx *argp);
```

## Arguments

*fd*

File descriptor returned by open().

*request*

VIDIOC_G_ENC_INDEX

*argp*

# Description

**Experimental:** This is an experimental interface and may change in the future.

The VIDIOC_G_ENC_INDEX ioctl provides meta data about a compressed video stream the same or another application currently reads from the driver, which is useful for random access into the stream without decoding it.

To read the data applications must call VIDIOC_G_ENC_INDEX with a pointer to a struct v4l2_enc_idx. On success the driver fills the *entry* array, stores the number of elements written in the *entries* field, and initializes the *entries_cap* field.

Each element of the *entry* array contains meta data about one picture. A VIDIOC_G_ENC_INDEX call reads up to V4L2_ENC_IDX_ENTRIES entries from a driver buffer, which can hold up to *entries_cap* entries. This number can be lower or higher than V4L2_ENC_IDX_ENTRIES, but not zero. When the application fails to read the meta data in time the oldest entries will be lost. When the buffer is empty or no capturing/encoding is in progress, *entries* will be zero.

Currently this ioctl is only defined for MPEG-2 program streams and video elementary streams.

**Table A-1. struct v4l2_enc_idx**

| | | |
|---|---|---|
| __u32 | *entries* | The number of entries the driver stored in the *entry* array. |
| __u32 | *entries_cap* | The number of entries the driver can buffer. Must be greater than zero. |
| __u32 | | Reserved for future extensions. Drivers must set the array to zero. |
| struct v4l2_enc_idx_entry | *entry*[V4L2_ENC_IDX_ENTRIES] | Meta data about a compressed video stream. Each element of the array corresponds to one picture, sorted in ascending order by their *offset*. |

**Table A-2. struct v4l2_enc_idx_entry**

| | | |
|---|---|---|
| __u64 | *offset* | The offset in bytes from the beginning of the compressed video stream to the beginning of this picture, that is a *PES packet header* as defined in ISO 13818-1 or a *picture header* as defined in ISO 13818-2. When the encoder is stopped, the driver resets the offset to zero. |
| __u64 | *pts* | The 33 bit *Presentation Time Stamp* of this picture as defined in ISO 13818-1. |
| __u32 | *length* | The length of this picture in bytes. |
| __u32 | *flags* | Flags containing the coding type of this picture, see Table A-3. |
| __u32 | *reserved*[2] | Reserved for future extensions. Drivers must set the array to zero. |

**Table A-3. Index Entry Flags**

| | | |
|---|---|---|
| V4L2_ENC_IDX_FRAME_I | 0x00 | This is an Intra-coded picture. |
| V4L2_ENC_IDX_FRAME_P | 0x01 | This is a Predictive-coded picture. |
| V4L2_ENC_IDX_FRAME_B | 0x02 | This is a Bidirectionally predictive-coded picture. |
| V4L2_ENC_IDX_FRAME_MASK | 0x0F | *AND* the flags field with this mask to obtain the picture coding type. |

## Return Value

On success 0 is returned, on error -1 and the `errno` variable is set appropriately:

EINVAL

The driver does not support this ioctl.

# ioctl VIDIOC_G_EXT_CTRLS, VIDIOC_S_EXT_CTRLS,

# VIDIOC_TRY_EXT_CTRLS

## Name

VIDIOC_G_EXT_CTRLS, VIDIOC_S_EXT_CTRLS, VIDIOC_TRY_EXT_CTRLS — Get or set the value of several controls, try control values

## Synopsis

```
int ioctl(int fd, int request, struct v4l2_ext_controls *argp);
```

## Arguments

*fd*

File descriptor returned by open().

*request*

VIDIOC_G_EXT_CTRLS, VIDIOC_S_EXT_CTRLS, VIDIOC_TRY_EXT_CTRLS

*argp*

## Description

These ioctls allow the caller to get or set multiple controls atomically. Control IDs are grouped into control classes (see Table A-3) and all controls in the control array must belong to the same control class.

Applications must always fill in the *count*, *ctrl_class*, *controls* and *reserved* fields of struct v4l2_ext_controls, and initialize the struct v4l2_ext_control array pointed to by the *controls* fields.

To get the current value of a set of controls applications initialize the *id*, *size* and *reserved2* fields of each struct v4l2_ext_control and call the VIDIOC_G_EXT_CTRLS ioctl. String controls controls must also set the *string* field.

If the *size* is too small to receive the control result (only relevant for pointer-type controls like strings), then the driver will set *size* to a valid value and return an ENOSPC error code. You should re-allocate the string memory to this new size and try again. It is possible that the same issue occurs again if the string has grown in the meantime. It is recommended to call VIDIOC_QUERYCTRL first and use *maximum*+1 as the new *size* value. It is guaranteed that that is sufficient memory.

To change the value of a set of controls applications initialize the *id*, *size*, *reserved2* and *value/string* fields of each struct v4l2_ext_control and call the VIDIOC_S_EXT_CTRLS ioctl. The controls will only be set if *all* control values are valid.

To check if a set of controls have correct values applications initialize the *id*, *size*, *reserved2* and *value/string* fields of each struct v4l2_ext_control and call the VIDIOC_TRY_EXT_CTRLS ioctl. It is up to the driver whether wrong values are automatically adjusted to a valid value or if an error is returned.

When the *id* or *ctrl_class* is invalid drivers return an EINVAL error code. When the value is out of bounds drivers can choose to take the closest valid value or return an ERANGE error code, whatever seems more appropriate. In the first case the new value is set in struct v4l2_ext_control.

The driver will only set/get these controls if all control values are correct. This prevents the situation where only some of the controls were set/get. Only low-level errors (e. g. a failed i2c command) can still cause this situation.

**Table A-1. struct v4l2_ext_control**

| | | |
|---|---|---|
| __u32 | *id* | Identifies the control, set by the application. |

| | | | |
|---|---|---|---|
| __u32 | *size* | | The total size in bytes of the payload of this control. This is normally 0, but for pointer controls this should be set to the size of the memory containing the payload, or that will receive the payload. If VIDIOC_G_EXT_CTRLS finds that this value is less than is required to store the payload result, then it is set to a value large enough to store the payload result and ENOSPC is returned. Note that for string controls this *size* field should not be confused with the length of the string. This field refers to the size of the memory that contains the string. The actual *length* of the string may well be much smaller. |
| __u32 | *reserved2*[1] | | Reserved for future extensions. Drivers and applications must set the array to zero. |
| union | (anonymous) | | |
| | __s32 | *value* | New value or current value. |
| | __s64 | *value64* | New value or current value. |
| | char * | *string* | A pointer to a string. |

**Table A-2. struct v4l2_ext_controls**

| | | |
|---|---|---|
| __u32 | *ctrl_class* | The control class to which all controls belong, see Table A-3. |
| __u32 | *count* | The number of controls in the controls array. May also be zero. |
| __u32 | *error_idx* | Set by the driver in case of an error. It is the index of the control causing the error or equal to 'count' when the error is not associated with a particular control. Undefined when the ioctl returns 0 (success). |

| | | |
|---|---|---|
| __u32 | *reserved*[2] | Reserved for future extensions. Drivers and applications must set the array to zero. |
| struct v4l2_ext_control * | *bntrols* | Pointer to an array of *count* v4l2_ext_control structures. Ignored if *count* equals zero. |

**Table A-3. Control classes**

| | | |
|---|---|---|
| V4L2_CTRL_CLASS_USER | 0x980000 | The class containing user controls. These controls are described in Section 1.8. All controls that can be set using the VIDIOC_S_CTRL and VIDIOC_G_CTRL ioctl belong to this class. |
| V4L2_CTRL_CLASS_MPEG | 0x990000 | The class containing MPEG compression controls. These controls are described in Section 1.9.5. |
| V4L2_CTRL_CLASS_CAMERA | 0x9a0000 | The class containing camera controls. These controls are described in Section 1.9.6. |
| V4L2_CTRL_CLASS_FM_TX | 0x9b0000 | The class containing FM Transmitter (FM TX) controls. These controls are described in Section 1.9.7. |

## Return Value

On success 0 is returned, on error -1 and the errno variable is set appropriately:

EINVAL

The struct v4l2_ext_control *id* is invalid or the struct v4l2_ext_controls *ctrl_class* is invalid. This error code is also returned by the VIDIOC_S_EXT_CTRLS and VIDIOC_TRY_EXT_CTRLS ioctls if two or more control values are in conflict.

ERANGE

The struct v4l2_ext_control *value* is out of bounds.

EBUSY

The control is temporarily not changeable, possibly because another applications took over control of the device function this control belongs to.

ENOSPC

The space reserved for the control's payload is insufficient. The field *size* is set to a value that is enough to store the payload and this error code is returned.

# ioctl VIDIOC_G_FBUF, VIDIOC_S_FBUF

## Name

VIDIOC_G_FBUF, VIDIOC_S_FBUF — Get or set frame buffer overlay parameters

## Synopsis

```
int ioctl(int fd, int request, struct v4l2_framebuffer *argp);
```

```
int ioctl(int fd, int request, const struct v4l2_framebuffer
*argp);
```

## Arguments

*fd*

File descriptor returned by open().

*request*

VIDIOC_G_FBUF, VIDIOC_S_FBUF

*argp*

## Description

Applications can use the VIDIOC_G_FBUF and VIDIOC_S_FBUF ioctl to get and set the framebuffer parameters for a Video Overlay or Video Output Overlay (OSD). The type of overlay is implied by the device type (capture or output device) and can be determined with the VIDIOC_QUERYCAP ioctl. One /dev/videoN device must not support both kinds of overlay.

The V4L2 API distinguishes destructive and non-destructive overlays. A destructive overlay copies captured video images into the video memory of a graphics card. A non-destructive overlay blends video images into a VGA signal or graphics into a video signal. *Video Output Overlays* are always non-destructive.

To get the current parameters applications call the VIDIOC_G_FBUF ioctl with a pointer to a v4l2_framebuffer structure. The driver fills all fields of the structure or returns an EINVAL error code when overlays are not supported.

To set the parameters for a *Video Output Overlay*, applications must initialize the *flags* field of a struct v4l2_framebuffer. Since the framebuffer is implemented on the TV card all other parameters are determined by the driver. When an application calls VIDIOC_S_FBUF with a pointer to this structure, the driver prepares for the overlay and returns the framebuffer parameters as VIDIOC_G_FBUF does, or it returns an error code.

To set the parameters for a *non-destructive Video Overlay*, applications must initialize the *flags* field, the *fmt* substructure, and call VIDIOC_S_FBUF. Again the driver prepares for the overlay and returns the framebuffer parameters as VIDIOC_G_FBUF does, or it returns an error code.

For a *destructive Video Overlay* applications must additionally provide a *base* address. Setting up a DMA to a random memory location can jeopardize the system security, its stability or even damage the hardware, therefore only the superuser can set the parameters for a destructive video overlay.

**Table A-1. struct v4l2_framebuffer**

| | | |
|---|---|---|
| __u32 | *capability* | Overlay capability flags set by the driver, see Table A-2. |
| __u32 | *flags* | Overlay control flags set by application and driver, see Table A-3 |
| void * | *base* | Physical base address of the framebuffer, that is the address of the pixel in the top left corner of the framebuffer.[a] |

| | | |
|---|---|---|
| | | This field is irrelevant to *non-destructive Video Overlays*. For *destructive Video Overlays* applications must provide a base address. The driver may accept only base addresses which are a multiple of two, four or eight bytes. For *Video Output Overlays* the driver must return a valid base address, so applications can find the corresponding Linux framebuffer device (see Section 4.4). |
| struct v4l2_pix_format | *fmt* | Layout of the frame buffer. The v4l2_pix_format structure is defined in Chapter 2, for clarification the fields and acceptable values are listed below: |
| __u32 | *width* | Width of the frame buffer in pixels. |
| __u32 | *height* | Height of the frame buffer in pixels. |
| __u32 | *pixelformat* | The pixel format of the framebuffer.<br>For *non-destructive Video Overlays* this field only defines a format for the struct v4l2_window *chromakey* field.<br>For *destructive Video Overlays* applications must initialize this field. For *Video Output Overlays* the driver must return a valid format. |

| | | |
|---|---|---|
| | | Usually this is an RGB format (for example V4L2_PIX_FMT_RGB565) but YUV formats (only packed YUV formats when chroma keying is used, not including V4L2_PIX_FMT_YUYV and V4L2_PIX_FMT_UYVY) and the V4L2_PIX_FMT_PAL8 format are also permitted. The behavior of the driver when an application requests a compressed format is undefined. See Chapter 2 for information on pixel formats. |
| enum v4l2_field | *field* | Drivers and applications shall ignore this field. If applicable, the field order is selected with the VIDIOC_S_FMT ioctl, using the *field* field of struct v4l2_window. |
| __u32 | *bytesperline* | Distance in bytes between the leftmost pixels in two adjacent lines. |

This field is irrelevant to *non-destructive Video Overlays*.
For *destructive Video Overlays* both applications and drivers can set this field to request padding bytes at the end of each line. Drivers however may ignore the requested value, 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.

For *Video Output Overlays* the driver must return a valid value.

Video hardware may access padding bytes, therefore they must reside in accessible memory. Consider for example the case where padding bytes after the last line of an image cross a system page boundary. Capture 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` | This field is irrelevant to *non-destructive Video Overlays*. For *destructive Video Overlays* applications must initialize this field. For *Video Output Overlays* the driver must return a valid format. Together with `base` it defines the framebuffer memory accessible by the driver. |
| enum v4l2_colorspace | `colorspace` | This information supplements the `pixelformat` and must be set by the driver, see Section 2.4. |
| __u32 | `priv` | Reserved for additional information about custom (driver defined) formats. When not used drivers and applications must set this field to zero. |

Notes:
a. A physical base address may not suit all platforms. GK notes in theory we should pass something like PCI device + memory region + offset instead. If you encounter problems please discuss on the linux-media mailing list: http://www.linuxtv.org/lists.php.

**Table A-2. Frame Buffer Capability Flags**

| | | |
|---|---|---|
| V4L2_FBUF_CAP_EXTERNOVERLAY | 0x0001 | The device is capable of non-destructive overlays. When the driver clears this flag, only destructive overlays are supported. There are no drivers yet which support both destructive and non-destructive overlays. |
| V4L2_FBUF_CAP_CHROMAKEY | 0x0002 | The device supports clipping by chroma-keying the images. That is, image pixels replace pixels in the VGA or video signal only where the latter assume a certain color. Chroma-keying makes no sense for destructive overlays. |
| V4L2_FBUF_CAP_LIST_CLIPPING | 0x0004 | The device supports clipping using a list of clip rectangles. |
| V4L2_FBUF_CAP_BITMAP_CLIPPING | 0x0008 | The device supports clipping using a bit mask. |
| V4L2_FBUF_CAP_LOCAL_ALPHA | 0x0010 | The device supports clipping/blending using the alpha channel of the framebuffer or VGA signal. Alpha blending makes no sense for destructive overlays. |
| V4L2_FBUF_CAP_GLOBAL_ALPHA | 0x0020 | The device supports alpha blending using a global alpha value. Alpha blending makes no sense for destructive overlays. |
| V4L2_FBUF_CAP_LOCAL_INV_ALPHA | 0x0040 | The device supports clipping/blending using the inverted alpha channel of the framebuffer or VGA signal. Alpha blending makes no sense for destructive overlays. |

| | | |
|---|---|---|
| V4L2_FBUF_CAP_SRC_CHROMAKEY | 0x0080 | The device supports Source Chroma-keying. Framebuffer pixels with the chroma-key colors are replaced by video pixels, which is exactly opposite of V4L2_FBUF_CAP_CHROMAKEY |

**Table A-3. Frame Buffer Flags**

| | | |
|---|---|---|
| V4L2_FBUF_FLAG_PRIMARY | 0x0001 | The framebuffer is the primary graphics surface. In other words, the overlay is destructive. [?] |
| V4L2_FBUF_FLAG_OVERLAY | 0x0002 | The frame buffer is an overlay surface the same size as the capture. [?] |

The purpose of V4L2_FBUF_FLAG_PRIMARY and V4L2_FBUF_FLAG_OVERLAY was never quite clear. Most drivers seem to ignore these flags. For compatibility with the *bttv* driver applications should set the V4L2_FBUF_FLAG_OVERLAY flag.

| | | |
|---|---|---|
| V4L2_FBUF_FLAG_CHROMAKEY | 0x0004 | Use chroma-keying. The chroma-key color is determined by the *chromakey* field of struct v4l2_window and negotiated with the VIDIOC_S_FMT ioctl, see Section 4.2 and Section 4.4. |

There are no flags to enable clipping using a list of clip rectangles or a bitmap. These methods are negotiated with the VIDIOC_S_FMT ioctl, see Section 4.2 and Section 4.4.

| | | |
|---|---|---|
| V4L2_FBUF_FLAG_LOCAL_ALPHA | 0x0008 | Use the alpha channel of the framebuffer to clip or blend framebuffer pixels with video images. The blend function is: output = framebuffer pixel * alpha + video pixel * (1 - alpha). The actual alpha depth depends on the framebuffer pixel format. |
| V4L2_FBUF_FLAG_GLOBAL_ALPHA | 0x0010 | Use a global alpha value to blend the framebuffer with video images. The blend function is: output = (framebuffer pixel * alpha + video pixel * (255 - alpha)) / 255. The alpha value is determined by the *global_alpha* field of struct v4l2_window and negotiated with the VIDIOC_S_FMT ioctl, see Section 4.2 and Section 4.4. |

| | | |
|---|---|---|
| V4L2_FBUF_FLAG_LOCAL_INV_ALPHA | 0x0020 | Like V4L2_FBUF_FLAG_LOCAL_ALPHA, use the alpha channel of the framebuffer to clip or blend framebuffer pixels with video images, but with an inverted alpha value. The blend function is: output = framebuffer pixel * (1 - alpha) + video pixel * alpha. The actual alpha depth depends on the framebuffer pixel format. |
| V4L2_FBUF_FLAG_SRC_CHROMAKEY | 0x0040 | Use source chroma-keying. The source chroma-key color is determined by the *chromakey* field of struct v4l2_window and negotiated with the VIDIOC_S_FMT ioctl, see Section 4.2 and Section 4.4. Both chroma-keying are mutual exclusive to each other, so same *chromakey* field of struct v4l2_window is being used. |

## Return Value

On success 0 is returned, on error -1 and the errno variable is set appropriately:

EPERM

VIDIOC_S_FBUF can only be called by a privileged user to negotiate the parameters for a destructive overlay.

EBUSY

The framebuffer parameters cannot be changed at this time because overlay is already enabled, or capturing is enabled and the hardware cannot capture and overlay simultaneously.

EINVAL

The ioctl is not supported or the VIDIOC_S_FBUF parameters are unsuitable.

# ioctl VIDIOC_G_FMT, VIDIOC_S_FMT, VIDIOC_TRY_FMT

## Name

VIDIOC_G_FMT, VIDIOC_S_FMT, VIDIOC_TRY_FMT — Get or set the data format, try a format

## Synopsis

```
int ioctl(int fd, int request, struct v4l2_format *argp);
```

## Arguments

*fd*

File descriptor returned by open().

*request*

VIDIOC_G_FMT, VIDIOC_S_FMT, VIDIOC_TRY_FMT

*argp*

## Description

These ioctls are used to negotiate the format of data (typically image format) exchanged between driver and application.

To query the current parameters applications set the *type* field of a struct v4l2_format to the respective buffer (stream) type. For example video capture devices use V4L2_BUF_TYPE_VIDEO_CAPTURE or V4L2_BUF_TYPE_VIDEO_CAPTURE_MPLANE. When the application calls the VIDIOC_G_FMT ioctl with a pointer to this structure the driver fills the respective member of the *fmt* union. In case of video capture devices that is either the struct v4l2_pix_format *pix* or the struct v4l2_pix_format_mplane *pix_mp*

member. When the requested buffer type is not supported drivers return an EINVAL error code.

To change the current format parameters applications initialize the *type* field and all fields of the respective *fmt* union member. For details see the documentation of the various devices types in Chapter 4. Good practice is to query the current parameters first, and to modify only those parameters not suitable for the application. When the application calls the VIDIOC_S_FMT ioctl with a pointer to a v4l2_format structure the driver checks and adjusts the parameters against hardware abilities. Drivers should not return an error code unless the input is ambiguous, this is a mechanism to fathom device capabilities and to approach parameters acceptable for both the application and driver. On success the driver may program the hardware, allocate resources and generally prepare for data exchange. Finally the VIDIOC_S_FMT ioctl returns the current format parameters as VIDIOC_G_FMT does. Very simple, inflexible devices may even ignore all input and always return the default parameters. However all V4L2 devices exchanging data with the application must implement the VIDIOC_G_FMT and VIDIOC_S_FMT ioctl. When the requested buffer type is not supported drivers return an EINVAL error code on a VIDIOC_S_FMT attempt. When I/O is already in progress or the resource is not available for other reasons drivers return the EBUSY error code.

The VIDIOC_TRY_FMT ioctl is equivalent to VIDIOC_S_FMT with one exception: it does not change driver state. It can also be called at any time, never returning EBUSY. This function is provided to negotiate parameters, to learn about hardware limitations, without disabling I/O or possibly time consuming hardware preparations. Although strongly recommended drivers are not required to implement this ioctl.

**Table A-1. struct v4l2_format**

| | | |
|---|---|---|
| enum v4l2_buf_type *type* | | Type of the data stream, see Table 3-3. |
| union | *fmt* | |
| | struct v4l2_pix_format *pix* | Definition of an image format, see Chapter 2, used by video capture and output devices. |

| | | |
|---|---|---|
| struct v4l2_pix_format_mplane | *pix_mp* | Definition of an image format, see Chapter 2, used by video capture and output devices that support the multi-planar version of the API. |
| struct v4l2_window | *win* | Definition of an overlaid image, see Section 4.2, used by video overlay devices. |
| struct v4l2_vbi_format | *vbi* | Raw VBI capture or output parameters. This is discussed in more detail in Section 4.7. Used by raw VBI capture and output devices. |
| struct v4l2_sliced_vbi_format | *sliced* | Sliced VBI capture or output parameters. See Section 4.8 for details. Used by sliced VBI capture and output devices. |
| __u8 | *raw_data*[200] | Place holder for future extensions and custom (driver defined) formats with *type* V4L2_BUF_TYPE_PRIVATE and higher. |

## Return Value

On success 0 is returned, on error -1 and the `errno` variable is set appropriately:

EBUSY

The data format cannot be changed at this time, for example because I/O is already in progress.

EINVAL

The struct v4l2_format *type* field is invalid, the requested buffer type not supported, or VIDIOC_TRY_FMT was called and is not supported with this buffer type.

# ioctl VIDIOC_G_FREQUENCY, VIDIOC_S_FREQUENCY

## Name

VIDIOC_G_FREQUENCY, VIDIOC_S_FREQUENCY — Get or set tuner or modulator radio frequency

## Synopsis

```
int ioctl(int fd, int request, struct v4l2_frequency *argp);
```

```
int ioctl(int fd, int request, const struct v4l2_frequency
*argp);
```

## Arguments

*fd*

File descriptor returned by open().

*request*

VIDIOC_G_FREQUENCY, VIDIOC_S_FREQUENCY

*argp*

# Description

To get the current tuner or modulator radio frequency applications set the *tuner* field of a struct v4l2_frequency to the respective tuner or modulator number (only input devices have tuners, only output devices have modulators), zero out the *reserved* array and call the VIDIOC_G_FREQUENCY ioctl with a pointer to this structure. The driver stores the current frequency in the *frequency* field.

To change the current tuner or modulator radio frequency applications initialize the *tuner*, *type* and *frequency* fields, and the *reserved* array of a struct v4l2_frequency and call the VIDIOC_S_FREQUENCY ioctl with a pointer to this structure. When the requested frequency is not possible the driver assumes the closest possible value. However VIDIOC_S_FREQUENCY is a write-only ioctl, it does not return the actual new frequency.

**Table A-1. struct v4l2_frequency**

| | | |
|---|---|---|
| __u32 | *tuner* | The tuner or modulator index number. This is the same value as in the struct v4l2_input *tuner* field and the struct v4l2_tuner *index* field, or the struct v4l2_output *modulator* field and the struct v4l2_modulator *index* field. |
| enum v4l2_tuner_type | *type* | The tuner type. This is the same value as in the struct v4l2_tuner *type* field. The field is not applicable to modulators, i. e. ignored by drivers. |
| __u32 | *frequency* | Tuning frequency in units of 62.5 kHz, or if the struct v4l2_tuner or struct v4l2_modulator *capabilities* flag V4L2_TUNER_CAP_LOW is set, in units of 62.5 Hz. |
| __u32 | *reserved*[8] | Reserved for future extensions. Drivers and applications must set the array to zero. |

### Return Value

On success 0 is returned, on error -1 and the `errno` variable is set appropriately:

EINVAL

The *`tuner`* index is out of bounds or the value in the *`type`* field is wrong.

# ioctl VIDIOC_G_INPUT, VIDIOC_S_INPUT

## Name

`VIDIOC_G_INPUT, VIDIOC_S_INPUT` — Query or select the current video input

## Synopsis

```
int ioctl(int fd, int request, int *argp);
```

## Arguments

*`fd`*

File descriptor returned by `open()`.

*`request`*

VIDIOC_G_INPUT, VIDIOC_S_INPUT

*`argp`*

## Description

To query the current video input applications call the VIDIOC_G_INPUT ioctl with a pointer to an integer where the driver stores the number of the input, as in the struct v4l2_input *index* field. This ioctl will fail only when there are no video inputs, returning EINVAL.

To select a video input applications store the number of the desired input in an integer and call the VIDIOC_S_INPUT ioctl with a pointer to this integer. Side effects are possible. For example inputs may support different video standards, so the driver may implicitly switch the current standard. It is good practice to select an input before querying or negotiating any other parameters.

Information about video inputs is available using the VIDIOC_ENUMINPUT ioctl.

## Return Value

On success 0 is returned, on error -1 and the errno variable is set appropriately:

EINVAL

The number of the video input is out of bounds, or there are no video inputs at all and this ioctl is not supported.

EBUSY

I/O is in progress, the input cannot be switched.

# ioctl VIDIOC_G_JPEGCOMP, VIDIOC_S_JPEGCOMP

## Name

VIDIOC_G_JPEGCOMP, VIDIOC_S_JPEGCOMP —

## Synopsis

```
int ioctl(int fd, int request, v4l2_jpegcompression *argp);
```

```
int ioctl(int fd, int request, const v4l2_jpegcompression
*argp);
```

## Arguments

*fd*

File descriptor returned by `open()`.

*request*

VIDIOC_G_JPEGCOMP, VIDIOC_S_JPEGCOMP

*argp*

## Description

[to do]

Ronald Bultje elaborates:

APP is some application-specific information. The application can set it itself, and it'll be stored in the JPEG-encoded fields (eg; interlacing information for in an AVI or so). COM is the same, but it's comments, like 'encoded by me' or so.

jpeg_markers describes whether the huffman tables, quantization tables and the restart interval information (all JPEG-specific stuff) should be stored in the JPEG-encoded fields. These define how the JPEG field is encoded. If you omit them, applications assume you've used standard encoding. You usually do want to add them.

**Table A-1. struct v4l2_jpegcompression**

| | |
|---|---|
| int | *quality* |
| int | *APPn* |

| | | |
|---|---|---|
| int | *APP_len* | |
| char | *APP_data*[60] | |
| int | *COM_len* | |
| char | *COM_data*[60] | |
| __u32 | *jpeg_markers* | See Table A-2. |

**Table A-2. JPEG Markers Flags**

| | | |
|---|---|---|
| V4L2_JPEG_MARKER_DHT | (1<<3) | Define Huffman Tables |
| V4L2_JPEG_MARKER_DQT | (1<<4) | Define Quantization Tables |
| V4L2_JPEG_MARKER_DRI | (1<<5) | Define Restart Interval |
| V4L2_JPEG_MARKER_COM | (1<<6) | Comment segment |
| V4L2_JPEG_MARKER_APP | (1<<7) | App segment, driver will always use APP0 |

## Return Value

On success 0 is returned, on error -1 and the `errno` variable is set appropriately:

EINVAL

This ioctl is not supported.

# ioctl VIDIOC_G_MODULATOR, VIDIOC_S_MODULATOR

## Name

`VIDIOC_G_MODULATOR, VIDIOC_S_MODULATOR` — Get or set modulator attributes

## Synopsis

```
int ioctl(int fd, int request, struct v4l2_modulator *argp);
```

```
int ioctl(int fd, int request, const struct v4l2_modulator
*argp);
```

## Arguments

*fd*

File descriptor returned by open().

*request*

VIDIOC_G_MODULATOR, VIDIOC_S_MODULATOR

*argp*

## Description

To query the attributes of a modulator applications initialize the *index* field and zero out the *reserved* array of a struct v4l2_modulator and call the VIDIOC_G_MODULATOR ioctl with a pointer to this structure. Drivers fill the rest of the structure or return an EINVAL error code when the index is out of bounds. To enumerate all modulators applications shall begin at index zero, incrementing by one until the driver returns EINVAL.

Modulators have two writable properties, an audio modulation set and the radio frequency. To change the modulated audio subprograms, applications initialize the *index* and *txsubchans* fields and the *reserved* array and call the VIDIOC_S_MODULATOR ioctl. Drivers may choose a different audio modulation if the request cannot be satisfied. However this is a write-only ioctl, it does not return the actual audio modulation selected.

To change the radio frequency the VIDIOC_S_FREQUENCY ioctl is available.

**Table A-1. struct v4l2_modulator**

| | | |
|---|---|---|
| __u32 | *index* | Identifies the modulator, set by the application. |
| __u8 | *name*[32] | Name of the modulator, a NUL-terminated ASCII string. This information is intended for the user. |
| __u32 | *capability* | Modulator capability flags. No flags are defined for this field, the tuner flags in struct v4l2_tuner are used accordingly. The audio flags indicate the ability to encode audio subprograms. They will *not* change for example with the current video standard. |
| __u32 | *rangelow* | The lowest tunable frequency in units of 62.5 KHz, or if the *capability* flag V4L2_TUNER_CAP_LOW is set, in units of 62.5 Hz. |
| __u32 | *rangehigh* | The highest tunable frequency in units of 62.5 KHz, or if the *capability* flag V4L2_TUNER_CAP_LOW is set, in units of 62.5 Hz. |
| __u32 | *txsubchans* | With this field applications can determine how audio sub-carriers shall be modulated. It contains a set of flags as defined in Table A-2. Note the tuner *rxsubchans* flags are reused, but the semantics are different. Video output devices are assumed to have an analog or PCM audio input with 1-3 channels. The *txsubchans* flags select one or more channels for modulation, together with some audio subprogram indicator, for example a stereo pilot tone. |
| __u32 | *reserved*[4] | Reserved for future extensions. Drivers and applications must set the array to zero. |

**Table A-2. Modulator Audio Transmission Flags**

| | | |
|---|---|---|
| V4L2_TUNER_SUB_MONO | 0x0001 | Modulate channel 1 as mono audio, when the input has more channels, a down-mix of channel 1 and 2. This flag does not combine with V4L2_TUNER_SUB_STEREO or V4L2_TUNER_SUB_LANG1. |
| V4L2_TUNER_SUB_STEREO | 0x0002 | Modulate channel 1 and 2 as left and right channel of a stereo audio signal. When the input has only one channel or two channels and V4L2_TUNER_SUB_SAP is also set, channel 1 is encoded as left and right channel. This flag does not combine with V4L2_TUNER_SUB_MONO or V4L2_TUNER_SUB_LANG1. When the driver does not support stereo audio it shall fall back to mono. |
| V4L2_TUNER_SUB_LANG1 | 0x0008 | Modulate channel 1 and 2 as primary and secondary language of a bilingual audio signal. When the input has only one channel it is used for both languages. It is not possible to encode the primary or secondary language only. This flag does not combine with V4L2_TUNER_SUB_MONO, V4L2_TUNER_SUB_STEREO or V4L2_TUNER_SUB_SAP. If the hardware does not support the respective audio matrix, or the current video standard does not permit bilingual audio the VIDIOC_S_MODULATOR ioctl shall return an EINVAL error code and the driver shall fall back to mono or stereo mode. |
| V4L2_TUNER_SUB_LANG2 | 0x0004 | Same effect as V4L2_TUNER_SUB_SAP. |

| | | |
|---|---|---|
| V4L2_TUNER_SUB_SAP | 0x0004 | When combined with V4L2_TUNER_SUB_MONO the first channel is encoded as mono audio, the last channel as Second Audio Program. When the input has only one channel it is used for both audio tracks. When the input has three channels the mono track is a down-mix of channel 1 and 2. When combined with V4L2_TUNER_SUB_STEREO channel 1 and 2 are encoded as left and right stereo audio, channel 3 as Second Audio Program. When the input has only two channels, the first is encoded as left and right channel and the second as SAP. When the input has only one channel it is used for all audio tracks. It is not possible to encode a Second Audio Program only. This flag must combine with V4L2_TUNER_SUB_MONO or V4L2_TUNER_SUB_STEREO. If the hardware does not support the respective audio matrix, or the current video standard does not permit SAP the VIDIOC_S_MODULATOR ioctl shall return an EINVAL error code and driver shall fall back to mono or stereo mode. |
| V4L2_TUNER_SUB_RDS | 0x0010 | Enable the RDS encoder for a radio FM transmitter. |

## Return Value

On success 0 is returned, on error -1 and the errno variable is set appropriately:

EINVAL

The struct v4l2_modulator *index* is out of bounds.

# ioctl VIDIOC_G_OUTPUT, VIDIOC_S_OUTPUT

## Name

VIDIOC_G_OUTPUT, VIDIOC_S_OUTPUT — Query or select the current video output

## Synopsis

```
int ioctl(int fd, int request, int *argp);
```

## Arguments

*fd*

File descriptor returned by open().

*request*

VIDIOC_G_OUTPUT, VIDIOC_S_OUTPUT

*argp*

## Description

To query the current video output applications call the VIDIOC_G_OUTPUT ioctl with a pointer to an integer where the driver stores the number of the output, as in the struct v4l2_output *index* field. This ioctl will fail only when there are no video outputs, returning the EINVAL error code.

To select a video output applications store the number of the desired output in an integer and call the VIDIOC_S_OUTPUT ioctl with a pointer to this integer. Side effects are possible. For example outputs may support different video standards, so the driver may implicitly switch the current standard. It is good practice to select an output before querying or negotiating any other parameters.

Information about video outputs is available using the VIDIOC_ENUMOUTPUT ioctl.

## Return Value

On success 0 is returned, on error -1 and the `errno` variable is set appropriately:

EINVAL

The number of the video output is out of bounds, or there are no video outputs at all and this ioctl is not supported.

EBUSY

I/O is in progress, the output cannot be switched.

# ioctl VIDIOC_G_PARM, VIDIOC_S_PARM

## Name

`VIDIOC_G_PARM, VIDIOC_S_PARM` — Get or set streaming parameters

## Synopsis

```
int ioctl(int fd, int request, v412_streamparm *argp);
```

## Arguments

*`fd`*

File descriptor returned by `open()`.

*`request`*

VIDIOC_G_PARM, VIDIOC_S_PARM

*`argp`*

## Description

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 unnecessary data copying.

Further 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.

**Table A-1. struct v4l2_streamparm**

| | | | |
|---|---|---|---|
| enum v4l2_buf_type | *type* | | The buffer (stream) type, same as struct v4l2_format *type*, set by the application. |
| union | *parm* | | |
| | struct v4l2_captureparm | *capture* | Parameters for capture devices, used when *type* is `V4L2_BUF_TYPE_VIDEO_CAPTURE`. |
| | struct v4l2_outputparm | *output* | Parameters for output devices, used when *type* is `V4L2_BUF_TYPE_VIDEO_OUTPUT`. |
| | __u8 | *raw_data*[200] | A place holder for future extensions and custom (driver defined) buffer types `V4L2_BUF_TYPE_PRIVATE` and higher. |

**Table A-2. struct v4l2_captureparm**

| | | |
|---|---|---|
| __u32 | *capability* | See Table A-4. |
| __u32 | *capturemode* | Set by drivers and applications, see Table A-5. |

| | | |
|---|---|---|
| struct v4l2_fract | *timeperframe* | This is is the desired period between successive frames captured by the driver, in seconds. The field is intended to skip frames on the driver side, saving I/O bandwidth.<br>Applications store here the desired frame period, drivers return the actual frame period, which must be greater or equal to the nominal frame period determined by the current video standard (struct v4l2_standard *frameperiod* field). Changing the video standard (also implicitly by switching the video input) may reset this parameter to the nominal frame period. To reset manually applications can just set this field to zero.<br><br>Drivers support this function only when they set the V4L2_CAP_TIMEPERFRAME flag in the *capability* field. |
| __u32 | *extendedmode* | Custom (driver specific) streaming parameters. When unused, applications and drivers must set this field to zero. Applications using this field should check the driver name and version, see Section 1.2. |
| __u32 | *readbuffers* | Applications set this field to the desired number of buffers used internally by the driver in read() mode. Drivers return the actual number of buffers. When an application requests zero buffers, drivers should just return the current setting rather than the minimum or an error code. For details see Section 3.1. |
| __u32 | *reserved*[4] | Reserved for future extensions. Drivers and applications must set the array to zero. |

**Table A-3. struct v4l2_outputparm**

| | | |
|---|---|---|
| __u32 | *capability* | See Table A-4. |
| __u32 | *outputmode* | Set by drivers and applications, see Table A-5. |
| struct v4l2_fract | *timeperframe* | This is is the desired period between successive frames output by the driver, in seconds. |

The field is intended to repeat frames on the driver side in `write()` mode (in streaming mode timestamps can be used to throttle the output), saving I/O bandwidth.
Applications store here the desired frame period, drivers return the actual frame period, which must be greater or equal to the nominal frame period determined by the current video standard (struct v4l2_standard *frameperiod* field). Changing the video standard (also implicitly by switching the video output) may reset this parameter to the nominal frame period. To reset manually applications can just set this field to zero.

Drivers support this function only when they set the `V4L2_CAP_TIMEPERFRAME` flag in the *capability* field.

| | | |
|---|---|---|
| __u32 | *extendedmode* | Custom (driver specific) streaming parameters. When unused, applications and drivers must set this field to zero. Applications using this field should check the driver name and version, see Section 1.2. |
| __u32 | *writebuffers* | Applications set this field to the desired number of buffers used internally by the driver in `write()` mode. Drivers return the actual number of buffers. When an application requests zero buffers, drivers should just return the current setting rather than the minimum or an error code. For details see Section 3.1. |
| __u32 | *reserved*[4] | Reserved for future extensions. Drivers and applications must set the array to zero. |

**Table A-4. Streaming Parameters Capabilites**

| | | |
|---|---|---|
| V4L2_CAP_TIMEPERFRAME | 0x1000 | The frame skipping/repeating controlled by the *timeperframe* field is supported. |

**Table A-5. Capture Parameters Flags**

| | | |
|---|---|---|
| V4L2_MODE_HIGHQUALITY | 0x0001 | High quality imaging mode. High quality mode is intended for still imaging applications. The idea is to get the best possible image quality that the hardware can deliver. It is not defined how the driver writer may achieve that; it will depend on the hardware and the ingenuity of the driver writer. High quality mode is a different mode from the the regular motion video capture modes. In high quality mode:• The driver may be able to capture higher resolutions than for motion capture.<br>• The driver may support fewer pixel formats than motion capture (eg; true color).<br>• The driver may capture and arithmetically combine multiple successive fields or frames to remove color edge artifacts and reduce the noise in the video data.<br>• The driver may capture images in slices like a scanner in order to handle larger format images than would otherwise be possible.<br>• An image capture operation may be significantly slower than motion capture.<br>• Moving objects in the image might have excessive motion blur.<br>• Capture might only work through the `read()` call. |

## Return Value

On success 0 is returned, on error -1 and the `errno` variable is set appropriately:

EINVAL

This ioctl is not supported.

# ioctl VIDIOC_G_PRIORITY, VIDIOC_S_PRIORITY

## Name

`VIDIOC_G_PRIORITY, VIDIOC_S_PRIORITY` — Query or request the access priority associated with a file descriptor

## Synopsis

```
int ioctl(int fd, int request, enum v4l2_priority *argp);
```

```
int ioctl(int fd, int request, const enum v4l2_priority
*argp);
```

## Arguments

*fd*

File descriptor returned by `open()`.

*request*

VIDIOC_G_PRIORITY, VIDIOC_S_PRIORITY

*argp*

Pointer to an enum v4l2_priority type.

## Description

To query the current access priority applications call the VIDIOC_G_PRIORITY ioctl with a pointer to an enum v4l2_priority variable where the driver stores the current priority.

To request an access priority applications store the desired priority in an enum v4l2_priority variable and call VIDIOC_S_PRIORITY ioctl with a pointer to this variable.

**Table A-1. enum v4l2_priority**

| | | |
|---|---|---|
| V4L2_PRIORITY_UNSET | 0 | |
| V4L2_PRIORITY_BACKGROUND | 1 | Lowest priority, usually applications running in background, for example monitoring VBI transmissions. A proxy application running in user space will be necessary if multiple applications want to read from a device at this priority. |
| V4L2_PRIORITY_INTERACTIVE | 2 | |
| V4L2_PRIORITY_DEFAULT | 2 | Medium priority, usually applications started and interactively controlled by the user. For example TV viewers, Teletext browsers, or just "panel" applications to change the channel or video controls. This is the default priority unless an application requests another. |
| V4L2_PRIORITY_RECORD | 3 | Highest priority. Only one file descriptor can have this priority, it blocks any other fd from changing device properties. Usually applications which must not be interrupted, like video recording. |

## Return Value

On success 0 is returned, on error -1 and the `errno` variable is set appropriately:

EINVAL

The requested priority value is invalid, or the driver does not support access priorities.

EBUSY

Another application already requested higher priority.

# ioctl VIDIOC_G_SLICED_VBI_CAP

## Name

`VIDIOC_G_SLICED_VBI_CAP` — Query sliced VBI capabilities

## Synopsis

```
int ioctl(int fd, int request, struct v4l2_sliced_vbi_cap
*argp);
```

## Arguments

*`fd`*

File descriptor returned by `open()`.

*`request`*

VIDIOC_G_SLICED_VBI_CAP

*`argp`*

## Description

To find out which data services are supported by a sliced VBI capture or output device, applications initialize the *type* field of a struct v4l2_sliced_vbi_cap, clear the *reserved* array and call the VIDIOC_G_SLICED_VBI_CAP ioctl. The driver fills in the remaining fields or returns an EINVAL error code if the sliced VBI API is unsupported or *type* is invalid.

Note the *type* field was added, and the ioctl changed from read-only to write-read, in Linux 2.6.19.

**Table A-1. struct v4l2_sliced_vbi_cap**

| | | |
|---|---|---|
| __u16 | *service_set* | A set of all data services supported by the driver. Equal to the union of all elements of the *service_lines* array. |
| __u16 | *service_lines*[2][24] | Each element of this array contains a set of data services the hardware can look for or insert into a particular scan line. Data services are defined in Table A-2. Array indices map to ITU-R line numbers (see also Figure 4-2 and Figure 4-3) as follows: |

| Element | 525 line systems | 625 line systems |
|---|---|---|
| *service_lines*[0][1] | 1 | |
| *service_lines*[0][23] | 23 | |
| *service_lines*[1][1] | 264 | 314 |
| *service_lines*[1][23] | 286 | 336 |

| | | |
|---|---|---|
| | | The number of VBI lines the hardware can capture or output per frame, or the number of services it can identify on a given line may be limited. For example on PAL line 16 the hardware may be able to look for a VPS or Teletext signal, but not both at the same time. Applications can learn about these limits using the VIDIOC_S_FMT ioctl as described in Section 4.8.<br><br>Drivers must set *service_lines*[0][0] and *service_lines*[1][0] to zero. |
| enum v4l2_buf_type | *type* | Type of the data stream, see Table 3-3. Should be V4L2_BUF_TYPE_SLICED_VBI_CAPTURE or V4L2_BUF_TYPE_SLICED_VBI_OUTPUT. |
| __u32 | *reserved*[3] | This array is reserved for future extensions. Applications and drivers must set it to zero. |

**Table A-2. Sliced VBI services**

| Symbol | Value | Reference | lines, usually | Payload |
|---|---|---|---|---|
| V4L2_SLICED_TELETEXT_B (Teletext System B) | 0x0001 | ETS 300 706, ITU BT.653 | 6-22, 318-335 (second field 7-22) | Last 42 of the 45 byte Teletext packet, that is without clock run-in and framing code, lsb first transmitted. |

| Symbol | Value | Reference | Lines, usually | Payload |
|---|---|---|---|---|
| V4L2_SLICED_VPS | 0x0400 | ETS 300 231 | PAL line 16 | Byte number 3 to 15 according to Figure 9 of ETS 300 231, lsb first transmitted. |
| V4L2_SLICED_CAPTION_525 | 0x1000 | EIA 608-B | NTSC line 21, 284 (second field 21) | Two bytes in transmission order, including parity bit, lsb first transmitted. |
| V4L2_SLICED_WSS_625 | 0x4000 | EN 300 294, ITU BT.1119 | PAL/SECAM line 23 | Byte 0<br>msb lsb msb<br>Bit 7 6 5 4 3 2 1 0 x x |
| V4L2_SLICED_VBI_525 | 0x1000 | Set of services applicable to 525 line systems. | | |
| V4L2_SLICED_VBI_625 | 0x4401 | Set of services applicable to 625 line systems. | | |

## Return Value

On success 0 is returned, on error -1 and the `errno` variable is set appropriately:

EINVAL

The device does not support sliced VBI capturing or output, or the value in the *`type`* field is wrong.

# ioctl VIDIOC_G_STD, VIDIOC_S_STD

## Name

`VIDIOC_G_STD, VIDIOC_S_STD` — Query or select the video standard of the current input

## Synopsis

```
int ioctl(int fd, int request, v4l2_std_id *argp);
```

```
int ioctl(int fd, int request, const v4l2_std_id *argp);
```

## Arguments

*fd*

File descriptor returned by open().

*request*

VIDIOC_G_STD, VIDIOC_S_STD

*argp*

## Description

To query and select the current video standard applications use the VIDIOC_G_STD and VIDIOC_S_STD ioctls which take a pointer to a v4l2_std_id type as argument. VIDIOC_G_STD can return a single flag or a set of flags as in struct v4l2_standard field *id*. The flags must be unambiguous such that they appear in only one enumerated v4l2_standard structure.

VIDIOC_S_STD accepts one or more flags, being a write-only ioctl it does not return the actual new standard as VIDIOC_G_STD does. When no flags are given or the current input does not support the requested standard the driver returns an EINVAL error code. When the standard set is ambiguous drivers may return EINVAL or choose any of the requested standards.

## Return Value

On success 0 is returned, on error -1 and the errno variable is set appropriately:

EINVAL

This ioctl is not supported, or the VIDIOC_S_STD parameter was unsuitable.

EBUSY

The device is busy and therefore can not change the standard

# ioctl VIDIOC_G_TUNER, VIDIOC_S_TUNER

## Name

VIDIOC_G_TUNER, VIDIOC_S_TUNER — Get or set tuner attributes

## Synopsis

```
int ioctl(int fd, int request, struct v4l2_tuner *argp);
```

```
int ioctl(int fd, int request, const struct v4l2_tuner *argp);
```

## Arguments

*fd*

File descriptor returned by open().

*request*

VIDIOC_G_TUNER, VIDIOC_S_TUNER

*argp*

# Description

To query the attributes of a tuner applications initialize the *index* field and zero out the *reserved* array of a struct v4l2_tuner and call the VIDIOC_G_TUNER ioctl with a pointer to this structure. Drivers fill the rest of the structure or return an EINVAL error code when the index is out of bounds. To enumerate all tuners applications shall begin at index zero, incrementing by one until the driver returns EINVAL.

Tuners have two writable properties, the audio mode and the radio frequency. To change the audio mode, applications initialize the *index*, *audmode* and *reserved* fields and call the VIDIOC_S_TUNER ioctl. This will *not* change the current tuner, which is determined by the current video input. Drivers may choose a different audio mode if the requested mode is invalid or unsupported. Since this is a write-only ioctl, it does not return the actually selected audio mode.

To change the radio frequency the VIDIOC_S_FREQUENCY ioctl is available.

**Table A-1. struct v4l2_tuner**

| | | |
|---|---|---|
| __u32 | *index* | Identifies the tuner, set by the application. |
| __u8 | *name*[32] | Name of the tuner, a NUL-terminated ASCII string. This information is intended for the user. |
| enum v4l2_tuner_type | *type* | Type of the tuner, see Table A-2. |
| __u32 | *capability* | Tuner capability flags, see Table A-3. Audio flags indicate the ability to decode audio subprograms. They will *not* change, for example with the current video standard.<br>When the structure refers to a radio tuner only the V4L2_TUNER_CAP_LOW, V4L2_TUNER_CAP_STEREO and V4L2_TUNER_CAP_RDS flags can be set. |
| __u32 | *rangelow* | The lowest tunable frequency in units of 62.5 kHz, or if the *capability* flag V4L2_TUNER_CAP_LOW is set, in units of 62.5 Hz. |

| | | |
|---|---|---|
| __u32 | *rangehigh* | The highest tunable frequency in units of 62.5 kHz, or if the *capability* flag V4L2_TUNER_CAP_LOW is set, in units of 62.5 Hz. |
| __u32 | *rxsubchans* | Some tuners or audio decoders can determine the received audio subprograms by analyzing audio carriers, pilot tones or other indicators. To pass this information drivers set flags defined in Table A-4 in this field. For example:<br>V4L2_TUNER_SUB_MONO receiving mono audio<br>STEREO \| SAP receiving stereo audio and a secondary audio program<br>MONO \| STEREO receiving mono or stereo audio, the hardware cannot distinguish<br>LANG1 \| LANG2 receiving bilingual audio<br>MONO \| STEREO \| LANG1 \| LANG2 receiving mono, stereo or bilingual audio<br>When the V4L2_TUNER_CAP_STEREO, _LANG1, _LANG2 or _SAP flag is cleared in the *capability* field, the corresponding V4L2_TUNER_SUB_ flag must not be set here.<br>This field is valid only if this is the tuner of the current video input, or when the structure refers to a radio tuner. |

| | | |
|---|---|---|
| __u32 | *audmode* | The selected audio mode, see Table A-5 for valid values. The audio mode does not affect audio subprogram detection, and like a control it does not automatically change unless the requested mode is invalid or unsupported. See Table A-6 for possible results when the selected and received audio programs do not match. Currently this is the only field of struct v4l2_tuner applications can change. |
| __u32 | *signal* | The signal strength if known, ranging from 0 to 65535. Higher values indicate a better signal. |
| __s32 | *afc* | Automatic frequency control: When the *afc* value is negative, the frequency is too low, when positive too high. |
| __u32 | *reserved*[4] | Reserved for future extensions. Drivers and applications must set the array to zero. |

**Table A-2. enum v4l2_tuner_type**

| | |
|---|---|
| V4L2_TUNER_RADIO | 1 |
| V4L2_TUNER_ANALOG_TV | 2 |

**Table A-3. Tuner and Modulator Capability Flags**

| | | |
|---|---|---|
| V4L2_TUNER_CAP_LOW | 0x0001 | When set, tuning frequencies are expressed in units of 62.5 Hz, otherwise in units of 62.5 kHz. |

| | | |
|---|---|---|
| V4L2_TUNER_CAP_NORM | 0x0002 | This is a multi-standard tuner; the video standard can or must be switched. (B/G PAL tuners for example are typically not considered multi-standard because the video standard is automatically determined from the frequency band.) The set of supported video standards is available from the struct v4l2_input pointing to this tuner, see the description of ioctl VIDIOC_ENUMINPUT for details. Only V4L2_TUNER_ANALOG_TV tuners can have this capability. |
| V4L2_TUNER_CAP_STEREO | 0x0010 | Stereo audio reception is supported. |
| V4L2_TUNER_CAP_LANG1 | 0x0040 | Reception of the primary language of a bilingual audio program is supported. Bilingual audio is a feature of two-channel systems, transmitting the primary language monaural on the main audio carrier and a secondary language monaural on a second carrier. Only V4L2_TUNER_ANALOG_TV tuners can have this capability. |
| V4L2_TUNER_CAP_LANG2 | 0x0020 | Reception of the secondary language of a bilingual audio program is supported. Only V4L2_TUNER_ANALOG_TV tuners can have this capability. |

| | | |
|---|---|---|
| V4L2_TUNER_CAP_SAP | 0x0020 | Reception of a secondary audio program is supported. This is a feature of the BTSC system which accompanies the NTSC video standard. Two audio carriers are available for mono or stereo transmissions of a primary language, and an independent third carrier for a monaural secondary language. Only V4L2_TUNER_ANALOG_TV tuners can have this capability.<br>Note the V4L2_TUNER_CAP_LANG2 and V4L2_TUNER_CAP_SAP flags are synonyms. V4L2_TUNER_CAP_SAP applies when the tuner supports the V4L2_STD_NTSC_M video standard. |
| V4L2_TUNER_CAP_RDS | 0x0080 | RDS capture is supported. This capability is only valid for radio tuners. |

**Table A-4. Tuner Audio Reception Flags**

| | | |
|---|---|---|
| V4L2_TUNER_SUB_MONO | 0x0001 | The tuner receives a mono audio signal. |
| V4L2_TUNER_SUB_STEREO | 0x0002 | The tuner receives a stereo audio signal. |
| V4L2_TUNER_SUB_LANG1 | 0x0008 | The tuner receives the primary language of a bilingual audio signal. Drivers must clear this flag when the current video standard is V4L2_STD_NTSC_M. |
| V4L2_TUNER_SUB_LANG2 | 0x0004 | The tuner receives the secondary language of a bilingual audio signal (or a second audio program). |
| V4L2_TUNER_SUB_SAP | 0x0004 | The tuner receives a Second Audio Program. Note the V4L2_TUNER_SUB_LANG2 and V4L2_TUNER_SUB_SAP flags are synonyms. The V4L2_TUNER_SUB_SAP flag applies when the current video standard is V4L2_STD_NTSC_M. |
| V4L2_TUNER_SUB_RDS | 0x0010 | The tuner receives an RDS channel. |

**Table A-5. Tuner Audio Modes**

| | | |
|---|---|---|
| V4L2_TUNER_MODE_MONO | 0 | Play mono audio. When the tuner receives a stereo signal this a down-mix of the left and right channel. When the tuner receives a bilingual or SAP signal this mode selects the primary language. |
| V4L2_TUNER_MODE_STEREO | 1 | Play stereo audio. When the tuner receives bilingual audio it may play different languages on the left and right channel or the primary language is played on both channels.<br>Playing different languages in this mode is deprecated. New drivers should do this only in MODE_LANG1_LANG2.<br><br>When the tuner receives no stereo signal or does not support stereo reception the driver shall fall back to MODE_MONO. |
| V4L2_TUNER_MODE_LANG1 | 3 | Play the primary language, mono or stereo. Only V4L2_TUNER_ANALOG_TV tuners support this mode. |
| V4L2_TUNER_MODE_LANG2 | 2 | Play the secondary language, mono. When the tuner receives no bilingual audio or SAP, or their reception is not supported the driver shall fall back to mono or stereo mode. Only V4L2_TUNER_ANALOG_TV tuners support this mode. |
| V4L2_TUNER_MODE_SAP | 2 | Play the Second Audio Program. When the tuner receives no bilingual audio or SAP, or their reception is not supported the driver shall fall back to mono or stereo mode. Only V4L2_TUNER_ANALOG_TV tuners support this mode. Note the V4L2_TUNER_MODE_LANG2 and V4L2_TUNER_MODE_SAP are synonyms. |

| | |
|---|---|
| V4L2_TUNER_MODE_LANG1_LANG2 | Play the primary language on the left channel, the secondary language on the right channel. When the tuner receives no bilingual audio or SAP, it shall fall back to MODE_LANG1 or MODE_MONO. Only V4L2_TUNER_ANALOG_TV tuners support this mode. |

**Table A-6. Tuner Audio Matrix**

| | Selected V4L2_TUNER_MODE_ | | | | |
|---|---|---|---|---|---|
| **Received V4L2_TUNER_SUB_** | **MONO** | **STEREO** | **LANG1** | **LANG2 = SAP** | **LANG1_LANG2**[a] |
| MONO | Mono | Mono/Mono | Mono | Mono | Mono/Mono |
| MONO \| SAP | Mono | Mono/Mono | Mono | SAP | Mono/SAP (preferred) or Mono/Mono |
| STEREO | L+R | L/R | Stereo L/R (preferred) or Mono L+R | Stereo L/R (preferred) or Mono L+R | L/R (preferred) or L+R/L+R |
| STEREO \| SAP | L+R | L/R | Stereo L/R (preferred) or Mono L+R | SAP | L+R/SAP (preferred) or L/R or L+R/L+R |
| LANG1 \| LANG2 | Language 1 | Lang1/Lang2 (deprecated[b]) or Lang1/Lang1 | Language 1 | Language 2 | Lang1/Lang2 (preferred) or Lang1/Lang1 |

Notes:
a. This mode has been added in Linux 2.6.17 and may not be supported by older drivers.
b. Playback of both languages in MODE_STEREO is deprecated. In the future drivers should produce only the primary language in this mode. Applications should request MODE_LANG1_LANG2 to record both languages or a stereo signal.

## Return Value

On success 0 is returned, on error -1 and the `errno` variable is set appropriately:

EINVAL

The struct v4l2_tuner *index* is out of bounds.

# ioctl VIDIOC_LOG_STATUS

## Name

`VIDIOC_LOG_STATUS` — Log driver status information

## Synopsis

```
int ioctl(int fd, int request);
```

## Description

As the video/audio devices become more complicated it becomes harder to debug problems. When this ioctl is called the driver will output the current device status to the kernel log. This is particular useful when dealing with problems like no sound, no video and incorrectly tuned channels. Also many modern devices autodetect video and audio standards and this ioctl will report what the device thinks what the standard is. Mismatches may give an indication where the problem is.

This ioctl is optional and not all drivers support it. It was introduced in Linux 2.6.15.

## Return Value

On success 0 is returned, on error -1 and the `errno` variable is set appropriately:

EINVAL

The driver does not support this ioctl.

# ioctl VIDIOC_OVERLAY

## Name

VIDIOC_OVERLAY — Start or stop video overlay

## Synopsis

```
int ioctl(int fd, int request, const int *argp);
```

## Arguments

*fd*

File descriptor returned by open().

*request*

VIDIOC_OVERLAY

*argp*

## Description

This ioctl is part of the video overlay I/O method. Applications call VIDIOC_OVERLAY to start or stop the overlay. It takes a pointer to an integer which must be set to zero by the application to stop overlay, to one to start.

Drivers do not support VIDIOC_STREAMON or VIDIOC_STREAMOFF with V4L2_BUF_TYPE_VIDEO_OVERLAY.

### Return Value

On success 0 is returned, on error -1 and the `errno` variable is set appropriately:

EINVAL

Video overlay is not supported, or the parameters have not been set up. See Section 4.2 for the necessary steps.

## ioctl VIDIOC_QBUF, VIDIOC_DQBUF

### Name

`VIDIOC_QBUF, VIDIOC_DQBUF` — Exchange a buffer with the driver

### Synopsis

```
int ioctl(int fd, int request, struct v4l2_buffer *argp);
```

### Arguments

*`fd`*

File descriptor returned by `open()`.

*`request`*

VIDIOC_QBUF, VIDIOC_DQBUF

*`argp`*

# Description

Applications call the VIDIOC_QBUF ioctl to enqueue an empty (capturing) or filled (output) buffer in the driver's incoming queue. The semantics depend on the selected I/O method.

To enqueue a buffer applications set the *type* field of a struct v4l2_buffer to the same buffer type as was previously used with struct v4l2_format *type* and struct v4l2_requestbuffers *type*. Applications must also set the *index* field. Valid index numbers range from zero to the number of buffers allocated with VIDIOC_REQBUFS (struct v4l2_requestbuffers *count*) minus one. The contents of the struct v4l2_buffer returned by a VIDIOC_QUERYBUF ioctl will do as well. When the buffer is intended for output (*type* is V4L2_BUF_TYPE_VIDEO_OUTPUT, V4L2_BUF_TYPE_VIDEO_OUTPUT_MPLANE, or V4L2_BUF_TYPE_VBI_OUTPUT) applications must also initialize the *bytesused*, *field* and *timestamp* fields, see Section 3.5 for details. Applications must also set *flags* to 0. If a driver supports capturing from specific video inputs and you want to specify a video input, then *flags* should be set to V4L2_BUF_FLAG_INPUT and the field *input* must be initialized to the desired input. The *reserved* field must be set to 0. When using the multi-planar API, the *m.planes* field must contain a userspace pointer to a filled-in array of struct v4l2_plane and the *length* field must be set to the number of elements in that array.

To enqueue a memory mapped buffer applications set the *memory* field to V4L2_MEMORY_MMAP. When VIDIOC_QBUF is called with a pointer to this structure the driver sets the V4L2_BUF_FLAG_MAPPED and V4L2_BUF_FLAG_QUEUED flags and clears the V4L2_BUF_FLAG_DONE flag in the *flags* field, or it returns an EINVAL error code.

To enqueue a user pointer buffer applications set the *memory* field to V4L2_MEMORY_USERPTR, the *m.userptr* field to the address of the buffer and *length* to its size. When the multi-planar API is used, *m.userptr* and *length* members of the passed array of struct v4l2_plane have to be used instead. When VIDIOC_QBUF is called with a pointer to this structure the driver sets the V4L2_BUF_FLAG_QUEUED flag and clears the V4L2_BUF_FLAG_MAPPED and V4L2_BUF_FLAG_DONE flags in the *flags* field, or it returns an error code. This ioctl locks the memory pages of the buffer in physical memory, they cannot be swapped out to disk. Buffers remain locked until dequeued, until the VIDIOC_STREAMOFF or VIDIOC_REQBUFS ioctl is called, or until the device is closed.

Applications call the VIDIOC_DQBUF ioctl to dequeue a filled (capturing) or displayed (output) buffer from the driver's outgoing queue. They just set the *type*, *memory* and *reserved* fields of a struct v4l2_buffer as above, when VIDIOC_DQBUF is called with a pointer to this structure the driver fills the remaining fields or returns an error code. The driver may also set

V4L2_BUF_FLAG_ERROR in the *flags* field. It indicates a non-critical (recoverable) streaming error. In such case the application may continue as normal, but should be aware that data in the dequeued buffer might be corrupted. When using the multi-planar API, the planes array does not have to be passed; the *m.planes* member must be set to NULL in that case.

By default VIDIOC_DQBUF blocks when no buffer is in the outgoing queue. When the O_NONBLOCK flag was given to the open() function, VIDIOC_DQBUF returns immediately with an EAGAIN error code when no buffer is available.

The v4l2_buffer structure is specified in Section 3.5.

## Return Value

On success 0 is returned, on error -1 and the errno variable is set appropriately:

EAGAIN

Non-blocking I/O has been selected using O_NONBLOCK and no buffer was in the outgoing queue.

EINVAL

The buffer *type* is not supported, or the *index* is out of bounds, or no buffers have been allocated yet, or the *userptr* or *length* are invalid.

ENOMEM

Not enough physical or virtual memory was available to enqueue a user pointer buffer.

EIO

VIDIOC_DQBUF failed due to an internal error. Can also indicate temporary problems like signal loss. Note the driver might dequeue an (empty) buffer despite returning an error, or even stop capturing. Reusing such buffer may be unsafe though and its details (e.g. *index*) may not be returned either. It is recommended that drivers indicate recoverable errors by setting the V4L2_BUF_FLAG_ERROR and returning 0 instead. In that case the application should be able to safely reuse the buffer and continue streaming.

# ioctl VIDIOC_QUERYBUF

## Name

VIDIOC_QUERYBUF — Query the status of a buffer

## Synopsis

```
int ioctl(int fd, int request, struct v4l2_buffer *argp);
```

## Arguments

*fd*

File descriptor returned by open().

*request*

VIDIOC_QUERYBUF

*argp*

## Description

This ioctl is part of the memory mapping I/O method. It can be used to query the status of a buffer at any time after buffers have been allocated with the VIDIOC_REQBUFS ioctl.

Applications set the *type* field of a struct v4l2_buffer to the same buffer type as was previously used with struct v4l2_format *type* and struct v4l2_requestbuffers *type*, and the *index* field. Valid index numbers range from zero to the number of buffers allocated with VIDIOC_REQBUFS (struct v4l2_requestbuffers *count*) minus one. The *reserved* field should to set to 0. When using the multi-planar API, the *m.planes* field must contain a userspace pointer to an array of struct v4l2_plane and the *length* field has to be set to the number of elements in that array. After

calling VIDIOC_QUERYBUF with a pointer to this structure drivers return an error code or fill the rest of the structure.

In the *flags* field the V4L2_BUF_FLAG_MAPPED, V4L2_BUF_FLAG_QUEUED and V4L2_BUF_FLAG_DONE flags will be valid. The *memory* field will be set to the current I/O method. For the single-planar API, the *m.offset* contains the offset of the buffer from the start of the device memory, the *length* field its size. For the multi-planar API, fields *m.mem_offset* and *length* in the *m.planes* array elements will be used instead. The driver may or may not set the remaining fields and flags, they are meaningless in this context.

The v4l2_buffer structure is specified in Section 3.5.

### Return Value

On success 0 is returned, on error -1 and the errno variable is set appropriately:

EINVAL

The buffer *type* is not supported, or the *index* is out of bounds.

## ioctl VIDIOC_QUERYCAP

### Name

VIDIOC_QUERYCAP — Query device capabilities

### Synopsis

```
int ioctl(int fd, int request, struct v4l2_capability *argp);
```

### Arguments

*fd*

File descriptor returned by open().

*request*

VIDIOC_QUERYCAP

*argp*

## Description

All V4L2 devices support the VIDIOC_QUERYCAP ioctl. It is used to identify kernel devices compatible with this specification and to obtain information about driver and hardware capabilities. The ioctl takes a pointer to a struct v4l2_capability which is filled by the driver. When the driver is not compatible with this specification the ioctl returns an EINVAL error code.

**Table A-1. struct v4l2_capability**

| | | |
|---|---|---|
| __u8 | *driver*[16] | Name of the driver, a unique NUL-terminated ASCII string. For example: "bttv". Driver specific applications can use this information to verify the driver identity. It is also useful to work around known bugs, or to identify drivers in error reports. The driver version is stored in the *version* field.<br>Storing strings in fixed sized arrays is bad practice but unavoidable here. Drivers and applications should take precautions to never read or write beyond the end of the array and to make sure the strings are properly NUL-terminated. |

| | | |
|---|---|---|
| __u8 | *card*[32] | Name of the device, a NUL-terminated ASCII string. For example: "Yoyodyne TV/FM". One driver may support different brands or models of video hardware. This information is intended for users, for example in a menu of available devices. Since multiple TV cards of the same brand may be installed which are supported by the same driver, this name should be combined with the character device file name (e. g. */dev/video2*) or the *bus_info* string to avoid ambiguities. |
| __u8 | *bus_info*[32] | Location of the device in the system, a NUL-terminated ASCII string. For example: "PCI Slot 4". This information is intended for users, to distinguish multiple identical devices. If no such information is available the field may simply count the devices controlled by the driver, or contain the empty string (*bus_info*[0] = 0). |
| __u32 | *version* | Version number of the driver. Together with the *driver* field this identifies a particular driver. The version number is formatted using the KERNEL_VERSION() macro: |

```
#define KERNEL_VERSION(a,b,c) (((a) << 16) + ((b) << 8) + (c))
__u32 version = KERNEL_VERSION(0, 8, 1);
printf ("Version: %u.%u.%u\n",  (version >> 16) & 0xFF,
(version >> 8) & 0xFF,   version & 0xFF);
```

| | | |
|---|---|---|
| __u32 | *capabilities* | Device capabilities, see Table A-2. |
| __u32 | *reserved*[4] | Reserved for future extensions. Drivers must set this array to zero. |

**Table A-2. Device Capabilities Flags**

| | | |
|---|---|---|
| V4L2_CAP_VIDEO_CAPTURE | 0x00000001 | The device supports the single-planar API through the Video Capture interface. |
| V4L2_CAP_VIDEO_CAPTURE_MPLANE | 0x00001000 | The device supports the multi-planar API through the Video Capture interface. |
| V4L2_CAP_VIDEO_OUTPUT | 0x00000002 | The device supports the single-planar API through the Video Output interface. |
| V4L2_CAP_VIDEO_OUTPUT_MPLANE | 0x00002000 | The device supports the multi-planar API through the Video Output interface. |
| V4L2_CAP_VIDEO_OVERLAY | 0x00000004 | The device supports the Video Overlay interface. A video overlay device typically stores captured images directly in the video memory of a graphics card, with hardware clipping and scaling. |
| V4L2_CAP_VBI_CAPTURE | 0x00000010 | The device supports the Raw VBI Capture interface, providing Teletext and Closed Caption data. |
| V4L2_CAP_VBI_OUTPUT | 0x00000020 | The device supports the Raw VBI Output interface. |
| V4L2_CAP_SLICED_VBI_CAPTURE | 0x00000040 | The device supports the Sliced VBI Capture interface. |
| V4L2_CAP_SLICED_VBI_OUTPUT | 0x00000080 | The device supports the Sliced VBI Output interface. |
| V4L2_CAP_RDS_CAPTURE | 0x00000100 | The device supports the RDS capture interface. |
| V4L2_CAP_VIDEO_OUTPUT_OVERLAY | 0x00000200 | The device supports the Video Output Overlay (OSD) interface. Unlike the *Video Overlay* interface, this is a secondary function of video output devices and overlays an image onto an outgoing video signal. When the driver sets this flag, it must clear the V4L2_CAP_VIDEO_OVERLAY flag and vice versa.[a] |
| V4L2_CAP_HW_FREQ_SEEK | 0x00000400 | The device supports the VIDIOC_S_HW_FREQ_SEEK ioctl for hardware frequency seeking. |
| V4L2_CAP_RDS_OUTPUT | 0x00000800 | The device supports the RDS output interface. |

| | | |
|---|---|---|
| V4L2_CAP_TUNER | 0x00010000 | The device has some sort of tuner to receive RF-modulated video signals. For more information about tuner programming see Section 1.6. |
| V4L2_CAP_AUDIO | 0x00020000 | The device has audio inputs or outputs. It may or may not support audio recording or playback, in PCM or compressed formats. PCM audio support must be implemented as ALSA or OSS interface. For more information on audio inputs and outputs see Section 1.5. |
| V4L2_CAP_RADIO | 0x00040000 | This is a radio receiver. |
| V4L2_CAP_MODULATOR | 0x00080000 | The device has some sort of modulator to emit RF-modulated video/audio signals. For more information about modulator programming see Section 1.6. |
| V4L2_CAP_READWRITE | 0x01000000 | The device supports the read() and/or write() I/O methods. |
| V4L2_CAP_ASYNCIO | 0x02000000 | The device supports the asynchronous I/O methods. |
| V4L2_CAP_STREAMING | 0x04000000 | The device supports the streaming I/O method. |

Notes:
a. The struct v4l2_framebuffer lacks an enum v4l2_buf_type field, therefore the type of overlay is implied by the driver capabilities.

## Return Value

On success 0 is returned, on error -1 and the `errno` variable is set appropriately:

EINVAL

The device is not compatible with this specification.

# ioctl VIDIOC_QUERYCTRL,

# VIDIOC_QUERYMENU

## Name

VIDIOC_QUERYCTRL, VIDIOC_QUERYMENU — Enumerate controls and menu control items

## Synopsis

```
int ioctl(int fd, int request, struct v4l2_queryctrl *argp);
```

```
int ioctl(int fd, int request, struct v4l2_querymenu *argp);
```

## Arguments

*fd*

File descriptor returned by open().

*request*

VIDIOC_QUERYCTRL, VIDIOC_QUERYMENU

*argp*

## Description

To query the attributes of a control applications set the *id* field of a struct v4l2_queryctrl and call the VIDIOC_QUERYCTRL ioctl with a pointer to this structure. The driver fills the rest of the structure or returns an EINVAL error code when the *id* is invalid.

It is possible to enumerate controls by calling VIDIOC_QUERYCTRL with successive *id* values starting from V4L2_CID_BASE up to and exclusive V4L2_CID_BASE_LASTP1. Drivers may return EINVAL if a control in this range is

not supported. Further applications can enumerate private controls, which are not defined in this specification, by starting at V4L2_CID_PRIVATE_BASE and incrementing *id* until the driver returns EINVAL.

In both cases, when the driver sets the V4L2_CTRL_FLAG_DISABLED flag in the *flags* field this control is permanently disabled and should be ignored by the application.[1]

When the application ORs *id* with V4L2_CTRL_FLAG_NEXT_CTRL the driver returns the next supported control, or EINVAL if there is none. Drivers which do not support this flag yet always return EINVAL.

Additional information is required for menu controls: the names of the menu items. To query them applications set the *id* and *index* fields of struct v4l2_querymenu and call the VIDIOC_QUERYMENU ioctl with a pointer to this structure. The driver fills the rest of the structure or returns an EINVAL error code when the *id* or *index* is invalid. Menu items are enumerated by calling VIDIOC_QUERYMENU with successive *index* values from struct v4l2_queryctrl *minimum* to *maximum*, inclusive. Note that it is possible for VIDIOC_QUERYMENU to return an EINVAL error code for some indices between *minimum* and *maximum*. In that case that particular menu item is not supported by this driver. Also note that the *minimum* value is not necessarily 0.

See also the examples in Section 1.8.

**Table A-1. struct v4l2_queryctrl**

| | | |
|---|---|---|
| __u32 | *id* | Identifies the control, set by the application. See Table 1-1 for predefined IDs. When the ID is ORed with V4L2_CTRL_FLAG_NEXT_CTRL the driver clears the flag and returns the first control with a higher ID. Drivers which do not support this flag yet always return an EINVAL error code. |
| enum v4l2_ctrl_type | *type* | Type of control, see Table A-3. |
| __u8 | *name*[32] | Name of the control, a NUL-terminated ASCII string. This information is intended for the user. |

| | | |
|---|---|---|
| __s32 | *minimum* | Minimum value, inclusive. This field gives a lower bound for V4L2_CTRL_TYPE_INTEGER controls and the lowest valid index for V4L2_CTRL_TYPE_MENU controls. For V4L2_CTRL_TYPE_STRING controls the minimum value gives the minimum length of the string. This length *does not include the terminating zero*. It may not be valid for any other type of control, including V4L2_CTRL_TYPE_INTEGER64 controls. Note that this is a signed value. |
| __s32 | *maximum* | Maximum value, inclusive. This field gives an upper bound for V4L2_CTRL_TYPE_INTEGER controls and the highest valid index for V4L2_CTRL_TYPE_MENU controls. For V4L2_CTRL_TYPE_STRING controls the maximum value gives the maximum length of the string. This length *does not include the terminating zero*. It may not be valid for any other type of control, including V4L2_CTRL_TYPE_INTEGER64 controls. Note that this is a signed value. |

| | | |
|---|---|---|
| __s32 | *step* | This field gives a step size for V4L2_CTRL_TYPE_INTEGER controls. For V4L2_CTRL_TYPE_STRING controls this field refers to the string length that has to be a multiple of this step size. It may not be valid for any other type of control, including V4L2_CTRL_TYPE_INTEGER64 controls.<br>Generally drivers should not scale hardware control values. It may be necessary for example when the *name* or *id* imply a particular unit and the hardware actually accepts only multiples of said unit. If so, drivers must take care values are properly rounded when scaling, such that errors will not accumulate on repeated read-write cycles.<br><br>This field gives the smallest change of an integer control actually affecting hardware. Often the information is needed when the user can change controls by keyboard or GUI buttons, rather than a slider. When for example a hardware register accepts values 0-511 and the driver reports 0-65535, step should be 128.<br><br>Note that although signed, the step value is supposed to be always positive. |
| __s32 | *default_value* | The default value of a V4L2_CTRL_TYPE_INTEGER, _BOOLEAN or _MENU control. Not valid for other types of controls. Drivers reset controls only when the driver is loaded, not later, in particular not when the func-open; is called. |
| __u32 | *flags* | Control flags, see Table A-4. |

| | | |
|---|---|---|
| __u32 | *reserved*[2] | Reserved for future extensions. Drivers must set the array to zero. |

**Table A-2. struct v4l2_querymenu**

| | | |
|---|---|---|
| __u32 | *id* | Identifies the control, set by the application from the respective struct v4l2_queryctrl *id*. |
| __u32 | *index* | Index of the menu item, starting at zero, set by the application. |
| __u8 | *name*[32] | Name of the menu item, a NUL-terminated ASCII string. This information is intended for the user. |
| __u32 | *reserved* | Reserved for future extensions. Drivers must set the array to zero. |

**Table A-3. enum v4l2_ctrl_type**

| Type | *minimum* | *step* | *maximum* | Description |
|---|---|---|---|---|
| V4L2_CTRL_TYPE_INTEGER | any | any | any | An integer-valued control ranging from minimum to maximum inclusive. The step value indicates the increment between values which are actually different on the hardware. |
| V4L2_CTRL_TYPE_BOOLEAN | 0 | 1 | 1 | A boolean-valued control. Zero corresponds to "disabled", and one means "enabled". |
| V4L2_CTRL_TYPE_MENU | ≥ 0 | 1 | N-1 | The control has a menu of N choices. The names of the menu items can be enumerated with the VIDIOC_QUERYMENU ioctl. |
| V4L2_CTRL_TYPE_BUTTON | 0 | 0 | 0 | A control which performs an action when set. Drivers must ignore the value passed with VIDIOC_S_CTRL and return an EINVAL error code on a VIDIOC_G_CTRL attempt. |
| V4L2_CTRL_TYPE_INTEGER64 | n/a | n/a | n/a | A 64-bit integer valued control. Minimum, maximum and step size cannot be queried. |

| Type | minimum | step | maximum | Description |
|---|---|---|---|---|
| V4L2_CTRL_TYPE_STRING | ≥ 0 | ≥ 1 | ≥ 0 | The minimum and maximum string lengths. The step size means that the string must be (minimum + N * step) characters long for N ≥ 0. These lengths do not include the terminating zero, so in order to pass a string of length 8 to VIDIOC_S_EXT_CTRLS you need to set the *size* field of struct v4l2_ext_control to 9. For VIDIOC_G_EXT_CTRLS you can set the *size* field to *maximum* + 1. Which character encoding is used will depend on the string control itself and should be part of the control documentation. |
| V4L2_CTRL_TYPE_CTRL_CLASS | n/a | n/a | n/a | This is not a control. When VIDIOC_QUERYCTRL is called with a control ID equal to a control class code (see Table A-3) + 1, the ioctl returns the name of the control class and this control type. Older drivers which do not support this feature return an EINVAL error code. |

**Table A-4. Control Flags**

| | | |
|---|---|---|
| V4L2_CTRL_FLAG_DISABLED | 0x0001 | This control is permanently disabled and should be ignored by the application. Any attempt to change the control will result in an EINVAL error code. |
| V4L2_CTRL_FLAG_GRABBED | 0x0002 | This control is temporarily unchangeable, for example because another application took over control of the respective resource. Such controls may be displayed specially in a user interface. Attempts to change the control may result in an EBUSY error code. |
| V4L2_CTRL_FLAG_READ_ONLY | 0x0004 | This control is permanently readable only. Any attempt to change the control will result in an EINVAL error code. |

| | | |
|---|---|---|
| V4L2_CTRL_FLAG_UPDATE | 0x0008 | A hint that changing this control may affect the value of other controls within the same control class. Applications should update their user interface accordingly. |
| V4L2_CTRL_FLAG_INACTIVE | 0x0010 | This control is not applicable to the current configuration and should be displayed accordingly in a user interface. For example the flag may be set on a MPEG audio level 2 bitrate control when MPEG audio encoding level 1 was selected with another control. |
| V4L2_CTRL_FLAG_SLIDER | 0x0020 | A hint that this control is best represented as a slider-like element in a user interface. |
| V4L2_CTRL_FLAG_WRITE_ONLY | 0x0040 | This control is permanently writable only. Any attempt to read the control will result in an EACCES error code error code. This flag is typically present for relative controls or action controls where writing a value will cause the device to carry out a given action (e. g. motor control) but no meaningful value can be returned. |

## Return Value

On success 0 is returned, on error -1 and the `errno` variable is set appropriately:

EINVAL

The struct v4l2_queryctrl *id* is invalid. The struct v4l2_querymenu *id* is invalid or *index* is out of range (less than *minimum* or greater than *maximum*) or this particular menu item is not supported by the driver.

EACCES

An attempt was made to read a write-only control.

## Notes

1. V4L2_CTRL_FLAG_DISABLED was intended for two purposes: Drivers can skip predefined controls not supported by the hardware (although returning EINVAL would do as well), or disable predefined and private controls after hardware detection without the trouble of reordering control arrays and indices (EINVAL cannot be used to skip private controls because it would prematurely end the enumeration).

# ioctl VIDIOC_QUERY_DV_PRESET

## Name

VIDIOC_QUERY_DV_PRESET — Sense the DV preset received by the current input

## Synopsis

```
int ioctl(int fd, int request, struct v4l2_dv_preset *argp);
```

## Arguments

*fd*

File descriptor returned by open().

*request*

VIDIOC_QUERY_DV_PRESET

*argp*

## Description

The hardware may be able to detect the current DV preset automatically, similar to sensing the video standard. To do so, applications call

VIDIOC_QUERY_DV_PRESET with a pointer to a struct v4l2_dv_preset type. Once the hardware detects a preset, that preset is returned in the preset field of struct v4l2_dv_preset. If the preset could not be detected because there was no signal, or the signal was unreliable, or the signal did not map to a supported preset, then the value V4L2_DV_INVALID is returned.

## Return Value

On success 0 is returned, on error -1 and the errno variable is set appropriately:

EINVAL

This ioctl is not supported.

EBUSY

The device is busy and therefore can not sense the preset

# ioctl VIDIOC_QUERYSTD

## Name

VIDIOC_QUERYSTD — Sense the video standard received by the current input

## Synopsis

```
int ioctl(int fd, int request, v4l2_std_id *argp);
```

## Arguments

*fd*

File descriptor returned by open().

*request*

VIDIOC_QUERYSTD

*argp*

## Description

The hardware may be able to detect the current video standard automatically. To do so, applications call VIDIOC_QUERYSTD with a pointer to a v4l2_std_id type. The driver stores here a set of candidates, this can be a single flag or a set of supported standards if for example the hardware can only distinguish between 50 and 60 Hz systems. When detection is not possible or fails, the set must contain all standards supported by the current video input or output.

## Return Value

On success 0 is returned, on error -1 and the errno variable is set appropriately:

EINVAL

This ioctl is not supported.

EBUSY

The device is busy and therefore can not detect the standard

# ioctl VIDIOC_REQBUFS

## Name

VIDIOC_REQBUFS — Initiate Memory Mapping or User Pointer I/O

## Synopsis

```
int ioctl(int fd, int request, struct v4l2_requestbuffers
*argp);
```

## Arguments

*fd*

File descriptor returned by `open()`.

*request*

VIDIOC_REQBUFS

*argp*

## Description

This ioctl is used to initiate memory mapped or user pointer I/O. Memory mapped buffers are located in device memory and must be allocated with this ioctl before they can be mapped into the application's address space. User buffers are allocated by applications themselves, and this ioctl is merely used to switch the driver into user pointer I/O mode and to setup some internal structures.

To allocate device buffers applications initialize all fields of the v4l2_requestbuffers structure. They set the *type* field to the respective stream or buffer type, the *count* field to the desired number of buffers, *memory* must be set to the requested I/O method and the *reserved* array must be zeroed. When the ioctl is called with a pointer to this structure the driver will attempt to allocate the requested number of buffers and it stores the actual number allocated in the *count* field. It can be smaller than the number requested, even zero, when the driver runs out of free memory. A larger number is also possible when the driver requires more buffers to function correctly. For example video output requires at least two buffers, one displayed and one filled by the application.

When the I/O method is not supported the ioctl returns an EINVAL error code.

Applications can call `VIDIOC_REQBUFS` again to change the number of buffers, however this cannot succeed when any buffers are still mapped. A *count* value of

zero frees all buffers, after aborting or finishing any DMA in progress, an implicit VIDIOC_STREAMOFF.

**Table A-1. struct v4l2_requestbuffers**

| | | |
|---|---|---|
| __u32 | *count* | The number of buffers requested or granted. |
| enum v4l2_buf_type | *type* | Type of the stream or buffers, this is the same as the struct v4l2_format *type* field. See Table 3-3 for valid values. |
| enum v4l2_memory | *memory* | Applications set this field to V4L2_MEMORY_MMAP or V4L2_MEMORY_USERPTR. |
| __u32 | *reserved*[2] | A place holder for future extensions and custom (driver defined) buffer types V4L2_BUF_TYPE_PRIVATE and higher. This array should be zeroed by applications. |

## Return Value

On success 0 is returned, on error -1 and the errno variable is set appropriately:

EBUSY

The driver supports multiple opens and I/O is already in progress, or reallocation of buffers was attempted although one or more are still mapped.

EINVAL

The buffer type (*type* field) or the requested I/O method (*memory*) is not supported.

# ioctl VIDIOC_S_HW_FREQ_SEEK

## Name

VIDIOC_S_HW_FREQ_SEEK — Perform a hardware frequency seek

# Synopsis

```
int ioctl(int fd, int request, struct v4l2_hw_freq_seek
*argp);
```

# Arguments

*fd*

File descriptor returned by `open()`.

*request*

VIDIOC_S_HW_FREQ_SEEK

*argp*

# Description

Start a hardware frequency seek from the current frequency. To do this applications initialize the *tuner*, *type*, *seek_upward*, *spacing* and *wrap_around* fields, and zero out the *reserved* array of a struct v4l2_hw_freq_seek and call the VIDIOC_S_HW_FREQ_SEEK ioctl with a pointer to this structure.

This ioctl is supported if the V4L2_CAP_HW_FREQ_SEEK capability is set.

**Table A-1. struct v4l2_hw_freq_seek**

| | | |
|---|---|---|
| __u32 | *tuner* | The tuner index number. This is the same value as in the struct v4l2_input *tuner* field and the struct v4l2_tuner *index* field. |
| enum v4l2_tuner_type | *type* | The tuner type. This is the same value as in the struct v4l2_tuner *type* field. |
| __u32 | *seek_upward* | If non-zero, seek upward from the current frequency, else seek downward. |
| __u32 | *wrap_around* | If non-zero, wrap around when at the end of the frequency range, else stop seeking. |

| | | |
|---|---|---|
| __u32 | *spacing* | If non-zero, defines the hardware seek resolution in Hz. The driver selects the nearest value that is supported by the device. If spacing is zero a reasonable default value is used. |
| __u32 | *reserved*[7] | Reserved for future extensions. Drivers and applications must set the array to zero. |

### Return Value

On success 0 is returned, on error -1 and the `errno` variable is set appropriately:

EINVAL

The *tuner* index is out of bounds or the value in the *type* field is wrong.

EAGAIN

The ioctl timed-out. Try again.

## ioctl VIDIOC_STREAMON, VIDIOC_STREAMOFF

### Name

`VIDIOC_STREAMON, VIDIOC_STREAMOFF` — Start or stop streaming I/O

### Synopsis

```
int ioctl(int fd, int request, const int *argp);
```

## Arguments

*fd*

File descriptor returned by open().

*request*

VIDIOC_STREAMON, VIDIOC_STREAMOFF

*argp*

## Description

The VIDIOC_STREAMON and VIDIOC_STREAMOFF ioctl start and stop the capture or output process during streaming (memory mapping or user pointer) I/O.

Specifically the capture hardware is disabled and no input buffers are filled (if there are any empty buffers in the incoming queue) until VIDIOC_STREAMON has been called. Accordingly the output hardware is disabled, no video signal is produced until VIDIOC_STREAMON has been called. The ioctl will succeed only when at least one output buffer is in the incoming queue.

The VIDIOC_STREAMOFF ioctl, apart of aborting or finishing any DMA in progress, unlocks any user pointer buffers locked in physical memory, and it removes all buffers from the incoming and outgoing queues. That means all images captured but not dequeued yet will be lost, likewise all images enqueued for output but not transmitted yet. I/O returns to the same state as after calling VIDIOC_REQBUFS and can be restarted accordingly.

Both ioctls take a pointer to an integer, the desired buffer or stream type. This is the same as struct v4l2_requestbuffers *type*.

Note applications can be preempted for unknown periods right before or after the VIDIOC_STREAMON or VIDIOC_STREAMOFF calls, there is no notion of starting or stopping "now". Buffer timestamps can be used to synchronize with other events.

## Return Value

On success 0 is returned, on error -1 and the errno variable is set appropriately:

EINVAL

Streaming I/O is not supported, the buffer *type* is not supported, or no buffers have been allocated (memory mapping) or enqueued (output) yet.

EPIPE

The driver implements pad-level format configuration and the pipeline configuration is invalid.

# ioctl VIDIOC_SUBDEV_ENUM_FRAME_INTERVAL

## Name

VIDIOC_SUBDEV_ENUM_FRAME_INTERVAL — Enumerate frame intervals

## Synopsis

```
int ioctl(int fd, int request, struct
v4l2_subdev_frame_interval_enum * argp);
```

## Arguments

*fd*

File descriptor returned by open().

*request*

VIDIOC_SUBDEV_ENUM_FRAME_INTERVAL

*argp*

# Description

**Experimental:** This is an experimental interface and may change in the future.

This ioctl lets applications enumerate available frame intervals on a given sub-device pad. Frame intervals only makes sense for sub-devices that can control the frame period on their own. This includes, for instance, image sensors and TV tuners.

For the common use case of image sensors, the frame intervals available on the sub-device output pad depend on the frame format and size on the same pad. Applications must thus specify the desired format and size when enumerating frame intervals.

To enumerate frame intervals applications initialize the *index*, *pad*, *code*, *width* and *height* fields of struct v4l2_subdev_frame_interval_enum and call the VIDIOC_SUBDEV_ENUM_FRAME_INTERVAL ioctl with a pointer to this structure. Drivers fill the rest of the structure or return an EINVAL error code if one of the input fields is invalid. All frame intervals are enumerable by beginning at index zero and incrementing by one until EINVAL is returned.

Available frame intervals may depend on the current 'try' formats at other pads of the sub-device, as well as on the current active links. See VIDIOC_SUBDEV_G_FMT for more information about the try formats.

Sub-devices that support the frame interval enumeration ioctl should implemented it on a single pad only. Its behaviour when supported on multiple pads of the same sub-device is not defined.

**Table A-1. struct v4l2_subdev_frame_interval_enum**

| | | |
|---|---|---|
| __u32 | *index* | Number of the format in the enumeration, set by the application. |
| __u32 | *pad* | Pad number as reported by the media controller API. |
| __u32 | *code* | The media bus format code, as defined in Section 4.13.4. |
| __u32 | *width* | Frame width, in pixels. |
| __u32 | *height* | Frame height, in pixels. |
| struct v4l2_fract | *interval* | Period, in seconds, between consecutive video frames. |
| __u32 | *reserved*[9] | Reserved for future extensions. Applications and drivers must set the array to zero. |

### Return Value

On success 0 is returned, on error -1 and the `errno` variable is set appropriately:

EINVAL

The struct v4l2_subdev_frame_interval_enum *pad* references a non-existing pad, one of the *code*, *width* or *height* fields are invalid for the given pad or the *index* field is out of bounds.

## ioctl VIDIOC_SUBDEV_ENUM_FRAME_SIZE

### Name

`VIDIOC_SUBDEV_ENUM_FRAME_SIZE` — Enumerate media bus frame sizes

### Synopsis

```
int ioctl(int fd, int request, struct
v4l2_subdev_frame_size_enum * argp);
```

### Arguments

*fd*

File descriptor returned by `open()`.

*request*

VIDIOC_SUBDEV_ENUM_FRAME_SIZE

*argp*

# Description

**Experimental:** This is an experimental interface and may change in the future.

This ioctl allows applications to enumerate all frame sizes supported by a sub-device on the given pad for the given media bus format. Supported formats can be retrieved with the VIDIOC_SUBDEV_ENUM_MBUS_CODE ioctl.

To enumerate frame sizes applications initialize the *pad*, *code* and *index* fields of the struct v4l2_subdev_mbus_code_enum and call the VIDIOC_SUBDEV_ENUM_FRAME_SIZE ioctl with a pointer to the structure. Drivers fill the minimum and maximum frame sizes or return an EINVAL error code if one of the input parameters is invalid.

Sub-devices that only support discrete frame sizes (such as most sensors) will return one or more frame sizes with identical minimum and maximum values.

Not all possible sizes in given [minimum, maximum] ranges need to be supported. For instance, a scaler that uses a fixed-point scaling ratio might not be able to produce every frame size between the minimum and maximum values. Applications must use the VIDIOC_SUBDEV_S_FMT ioctl to try the sub-device for an exact supported frame size.

Available frame sizes may depend on the current 'try' formats at other pads of the sub-device, as well as on the current active links and the current values of V4L2 controls. See VIDIOC_SUBDEV_G_FMT for more information about try formats.

**Table A-1. struct v4l2_subdev_frame_size_enum**

| | | |
|---|---|---|
| __u32 | *index* | Number of the format in the enumeration, set by the application. |
| __u32 | *pad* | Pad number as reported by the media controller API. |
| __u32 | *code* | The media bus format code, as defined in Section 4.13.4. |
| __u32 | *min_width* | Minimum frame width, in pixels. |
| __u32 | *max_width* | Maximum frame width, in pixels. |
| __u32 | *min_height* | Minimum frame height, in pixels. |
| __u32 | *max_height* | Maximum frame height, in pixels. |
| __u32 | *reserved*[9] | Reserved for future extensions. Applications and drivers must set the array to zero. |

## Return Value

On success 0 is returned, on error -1 and the `errno` variable is set appropriately:

EINVAL

The struct v4l2_subdev_frame_size_enum *pad* references a non-existing pad, the *code* is invalid for the given pad or the *index* field is out of bounds.

# ioctl VIDIOC_SUBDEV_ENUM_MBUS_CODE

## Name

`VIDIOC_SUBDEV_ENUM_MBUS_CODE` — Enumerate media bus formats

## Synopsis

```
int ioctl(int fd, int request, struct
v4l2_subdev_mbus_code_enum * argp);
```

## Arguments

*fd*

File descriptor returned by `open()`.

*request*

VIDIOC_SUBDEV_ENUM_MBUS_CODE

*argp*

## Description

**Experimental:** This is an experimental interface and may change in the future.

To enumerate media bus formats available at a given sub-device pad applications initialize the *pad* and *index* fields of struct v4l2_subdev_mbus_code_enum and call the VIDIOC_SUBDEV_ENUM_MBUS_CODE ioctl with a pointer to this structure. Drivers fill the rest of the structure or return an EINVAL error code if either the *pad* or *index* are invalid. All media bus formats are enumerable by beginning at index zero and incrementing by one until EINVAL is returned.

Available media bus formats may depend on the current 'try' formats at other pads of the sub-device, as well as on the current active links. See VIDIOC_SUBDEV_G_FMT for more information about the try formats.

**Table A-1. struct v4l2_subdev_mbus_code_enum**

| | | |
|---|---|---|
| __u32 | *pad* | Pad number as reported by the media controller API. |
| __u32 | *index* | Number of the format in the enumeration, set by the application. |
| __u32 | *code* | The media bus format code, as defined in Section 4.13.4. |
| __u32 | *reserved*[9] | Reserved for future extensions. Applications and drivers must set the array to zero. |

## Return Value

On success 0 is returned, on error -1 and the errno variable is set appropriately:

EINVAL

The struct v4l2_subdev_mbus_code_enum *pad* references a non-existing pad, or the *index* field is out of bounds.

# ioctl VIDIOC_SUBDEV_G_CROP,

# VIDIOC_SUBDEV_S_CROP

## Name

VIDIOC_SUBDEV_G_CROP, VIDIOC_SUBDEV_S_CROP — Get or set the crop rectangle on a subdev pad

## Synopsis

```
int ioctl(int fd, int request, struct v4l2_subdev_crop *argp);
```

```
int ioctl(int fd, int request, const struct v4l2_subdev_crop
*argp);
```

## Arguments

*fd*

File descriptor returned by open().

*request*

VIDIOC_SUBDEV_G_CROP, VIDIOC_SUBDEV_S_CROP

*argp*

## Description

**Experimental:** This is an experimental interface and may change in the future.

To retrieve the current crop rectangle applications set the *pad* field of a struct v4l2_subdev_crop to the desired pad number as reported by the media API and the *which* field to V4L2_SUBDEV_FORMAT_ACTIVE. They then call the

VIDIOC_SUBDEV_G_CROP ioctl with a pointer to this structure. The driver fills the members of the *rect* field or returns EINVAL error code if the input arguments are invalid, or if cropping is not supported on the given pad.

To change the current crop rectangle applications set both the *pad* and *which* fields and all members of the *rect* field. They then call the VIDIOC_SUBDEV_S_CROP ioctl with a pointer to this structure. The driver verifies the requested crop rectangle, adjusts it based on the hardware capabilities and configures the device. Upon return the struct v4l2_subdev_crop contains the current format as would be returned by a VIDIOC_SUBDEV_G_CROP call.

Applications can query the device capabilities by setting the *which* to V4L2_SUBDEV_FORMAT_TRY. When set, 'try' crop rectangles are not applied to the device by the driver, but are mangled exactly as active crop rectangles and stored in the sub-device file handle. Two applications querying the same sub-device would thus not interact with each other.

Drivers must not return an error solely because the requested crop rectangle doesn't match the device capabilities. They must instead modify the rectangle to match what the hardware can provide. The modified format should be as close as possible to the original request.

**Table A-1. struct v4l2_subdev_crop**

| | | |
|---|---|---|
| __u32 | *pad* | Pad number as reported by the media framework. |
| __u32 | *which* | Crop rectangle to get or set, from enum v4l2_subdev_format_whence. |
| struct v4l2_rect | *rect* | Crop rectangle boundaries, in pixels. |
| __u32 | *reserved*[8] | Reserved for future extensions. Applications and drivers must set the array to zero. |

# Return Value

On success 0 is returned, on error -1 and the errno variable is set appropriately:

EBUSY

The crop rectangle can't be changed because the pad is currently busy. This can be caused, for instance, by an active video stream on the pad. The ioctl must not be retried without performing another action to fix the problem first. Only returned by VIDIOC_SUBDEV_S_CROP

EINVAL

The struct v4l2_subdev_crop *pad* references a non-existing pad, the *which* field references a non-existing format, or cropping is not supported on the given subdev pad.

# ioctl VIDIOC_SUBDEV_G_FMT, VIDIOC_SUBDEV_S_FMT

## Name

VIDIOC_SUBDEV_G_FMT, VIDIOC_SUBDEV_S_FMT — Get or set the data format on a subdev pad

## Synopsis

```
int ioctl(int fd, int request, struct v4l2_subdev_format *argp
);
```

## Arguments

*fd*

File descriptor returned by open().

*request*

VIDIOC_SUBDEV_G_FMT, VIDIOC_SUBDEV_S_FMT

*argp*

# Description

**Experimental:** This is an experimental interface and may change in the future.

These ioctls are used to negotiate the frame format at specific subdev pads in the image pipeline.

To retrieve the current format applications set the *pad* field of a struct v4l2_subdev_format to the desired pad number as reported by the media API and the *which* field to V4L2_SUBDEV_FORMAT_ACTIVE. When they call the VIDIOC_SUBDEV_G_FMT ioctl with a pointer to this structure the driver fills the members of the *format* field.

To change the current format applications set both the *pad* and *which* fields and all members of the *format* field. When they call the VIDIOC_SUBDEV_S_FMT ioctl with a pointer to this structure the driver verifies the requested format, adjusts it based on the hardware capabilities and configures the device. Upon return the struct v4l2_subdev_format contains the current format as would be returned by a VIDIOC_SUBDEV_G_FMT call.

Applications can query the device capabilities by setting the *which* to V4L2_SUBDEV_FORMAT_TRY. When set, 'try' formats are not applied to the device by the driver, but are changed exactly as active formats and stored in the sub-device file handle. Two applications querying the same sub-device would thus not interact with each other.

For instance, to try a format at the output pad of a sub-device, applications would first set the try format at the sub-device input with the VIDIOC_SUBDEV_S_FMT ioctl. They would then either retrieve the default format at the output pad with the VIDIOC_SUBDEV_G_FMT ioctl, or set the desired output pad format with the VIDIOC_SUBDEV_S_FMT ioctl and check the returned value.

Try formats do not depend on active formats, but can depend on the current links configuration or sub-device controls value. For instance, a low-pass noise filter might crop pixels at the frame boundaries, modifying its output frame size.

Drivers must not return an error solely because the requested format doesn't match the device capabilities. They must instead modify the format to match what the hardware can provide. The modified format should be as close as possible to the original request.

**Table A-1. struct v4l2_subdev_format**

| | | |
|---|---|---|
| __u32 | *pad* | Pad number as reported by the media controller API. |

| | | |
|---|---|---|
| __u32 | *which* | Format to modified, from enum v4l2_subdev_format_whence. |
| struct v4l2_mbus_framefmt | *format* | Definition of an image format, see Table 4-19 for details. |
| __u32 | *reserved*[8] | Reserved for future extensions. Applications and drivers must set the array to zero. |

**Table A-2. enum v4l2_subdev_format_whence**

| | |
|---|---|
| V4L2_SUBDEV_FORMAT_TRY | Try formats, used for querying device capabilities. |
| V4L2_SUBDEV_FORMAT_ACTIVE | Active formats, applied to the hardware. |

## Return Value

On success 0 is returned, on error -1 and the `errno` variable is set appropriately:

EBUSY

The format can't be changed because the pad is currently busy. This can be caused, for instance, by an active video stream on the pad. The ioctl must not be retried without performing another action to fix the problem first. Only returned by `VIDIOC_SUBDEV_S_FMT`

EINVAL

The struct v4l2_subdev_format *pad* references a non-existing pad, or the *which* field references a non-existing format.

# ioctl VIDIOC_SUBDEV_G_FRAME_INTERVAL, VIDIOC_SUBDEV_S_FRAME_INTERVAL

## Name

VIDIOC_SUBDEV_G_FRAME_INTERVAL, VIDIOC_SUBDEV_S_FRAME_INTERVAL — Get or set the frame interval on a subdev pad

# Synopsis

```
int ioctl(int fd, int request, struct
v4l2_subdev_frame_interval *argp );
```

# Arguments

*fd*

File descriptor returned by open().

*request*

VIDIOC_SUBDEV_G_FRAME_INTERVAL, VIDIOC_SUBDEV_S_FRAME_INTERVAL

*argp*

# Description

**Experimental:** This is an experimental interface and may change in the future.

These ioctls are used to get and set the frame interval at specific subdev pads in the image pipeline. The frame interval only makes sense for sub-devices that can control the frame period on their own. This includes, for instance, image sensors and TV tuners. Sub-devices that don't support frame intervals must not implement these ioctls.

To retrieve the current frame interval applications set the *pad* field of a struct v4l2_subdev_frame_interval to the desired pad number as reported by the media controller API. When they call the VIDIOC_SUBDEV_G_FRAME_INTERVAL

ioctl with a pointer to this structure the driver fills the members of the *interval* field.

To change the current frame interval applications set both the *pad* field and all members of the *interval* field. When they call the VIDIOC_SUBDEV_S_FRAME_INTERVAL ioctl with a pointer to this structure the driver verifies the requested interval, adjusts it based on the hardware capabilities and configures the device. Upon return the struct v4l2_subdev_frame_interval contains the current frame interval as would be returned by a VIDIOC_SUBDEV_G_FRAME_INTERVAL call.

Drivers must not return an error solely because the requested interval doesn't match the device capabilities. They must instead modify the interval to match what the hardware can provide. The modified interval should be as close as possible to the original request.

Sub-devices that support the frame interval ioctls should implement them on a single pad only. Their behaviour when supported on multiple pads of the same sub-device is not defined.

**Table A-1. struct v4l2_subdev_frame_interval**

| | | |
|---|---|---|
| __u32 | *pad* | Pad number as reported by the media controller API. |
| struct v4l2_fract | *interval* | Period, in seconds, between consecutive video frames. |
| __u32 | *reserved*[9] | Reserved for future extensions. Applications and drivers must set the array to zero. |

## Return Value

On success 0 is returned, on error -1 and the errno variable is set appropriately:

EBUSY

The frame interval can't be changed because the pad is currently busy. This can be caused, for instance, by an active video stream on the pad. The ioctl must not be retried without performing another action to fix the problem first. Only returned by VIDIOC_SUBDEV_S_FRAME_INTERVAL

EINVAL

The struct v4l2_subdev_frame_interval *pad* references a non-existing pad, or the pad doesn't support frame intervals.

# ioctl VIDIOC_SUBSCRIBE_EVENT, VIDIOC_UNSUBSCRIBE_EVENT

## Name

VIDIOC_SUBSCRIBE_EVENT, VIDIOC_UNSUBSCRIBE_EVENT — Subscribe or unsubscribe event

## Synopsis

```
int ioctl(int fd, int request, struct v4l2_event_subscription *argp);
```

## Arguments

*fd*

File descriptor returned by open().

*request*

VIDIOC_SUBSCRIBE_EVENT, VIDIOC_UNSUBSCRIBE_EVENT

*argp*

## Description

Subscribe or unsubscribe V4L2 event. Subscribed events are dequeued by using the VIDIOC_DQEVENT ioctl.

**Table A-1. struct v4l2_event_subscription**

| | | |
|---|---|---|
| __u32 | *type* | Type of the event. |
| __u32 | *reserved*[7] | Reserved for future extensions. Drivers and applications must set the array to zero. |

**Table A-2. Event Types**

| | | |
|---|---|---|
| V4L2_EVENT_ALL | 0 | All events. V4L2_EVENT_ALL is valid only for VIDIOC_UNSUBSCRIBE_EVENT for unsubscribing all events at once. |
| V4L2_EVENT_VSYNC | 1 | This event is triggered on the vertical sync. This event has struct v4l2_event_vsync associated with it. |
| V4L2_EVENT_EOS | 2 | This event is triggered when the end of a stream is reached. This is typically used with MPEG decoders to report to the application when the last of the MPEG stream has been decoded. |
| V4L2_EVENT_PRIVATE_START | 0x08000000 | Base event number for driver-private events. |

**Table A-3. struct v4l2_event_vsync**

| | | |
|---|---|---|
| __u8 | *field* | The upcoming field. See enum v4l2_field. |

# V4L2 mmap()

## Name

v4l2-mmap — Map device memory into application address space

# Synopsis

```
#include <unistd.h>
#include <sys/mman.h>
void *mmap(void *start, size_t length, int prot, int flags,
int fd, off_t offset);
```

# Arguments

*start*

Map the buffer to this address in the application's address space. When the MAP_FIXED flag is specified, *start* must be a multiple of the pagesize and mmap will fail when the specified address cannot be used. Use of this option is discouraged; applications should just specify a NULL pointer here.

*length*

Length of the memory area to map. This must be the same value as returned by the driver in the struct v4l2_buffer *length* field for the single-planar API, and the same value as returned by the driver in the struct v4l2_plane *length* field for the multi-planar API.

*prot*

The *prot* argument describes the desired memory protection. Regardless of the device type and the direction of data exchange it should be set to PROT_READ | PROT_WRITE, permitting read and write access to image buffers. Drivers should support at least this combination of flags. Note the Linux video-buf kernel module, which is used by the bttv, saa7134, saa7146, cx88 and vivi driver supports only PROT_READ | PROT_WRITE. When the driver does not support the desired protection the mmap() function fails.

Note device memory accesses (e. g. the memory on a graphics card with video capturing hardware) may incur a performance penalty compared to main memory accesses, or reads may be significantly slower than writes or vice versa. Other I/O methods may be more efficient in this case.

*flags*

The *flags* parameter specifies the type of the mapped object, mapping options and whether modifications made to the mapped copy of the page are private to the process or are to be shared with other references.

MAP_FIXED requests that the driver selects no other address than the one specified. If the specified address cannot be used, mmap() will fail. If MAP_FIXED is specified, *start* must be a multiple of the pagesize. Use of this option is discouraged.

One of the MAP_SHARED or MAP_PRIVATE flags must be set. MAP_SHARED allows applications to share the mapped memory with other (e. g. child-) processes. Note the Linux video-buf module which is used by the bttv, saa7134, saa7146, cx88 and vivi driver supports only MAP_SHARED. MAP_PRIVATE requests copy-on-write semantics. V4L2 applications should not set the MAP_PRIVATE, MAP_DENYWRITE, MAP_EXECUTABLE or MAP_ANON flag.

*fd*

File descriptor returned by open().

*offset*

Offset of the buffer in device memory. This must be the same value as returned by the driver in the struct v4l2_buffer *m* union *offset* field for the single-planar API, and the same value as returned by the driver in the struct v4l2_plane *m* union *mem_offset* field for the multi-planar API.

## Description

The mmap() function asks to map *length* bytes starting at *offset* in the memory of the device specified by *fd* into the application address space, preferably at address *start*. This latter address is a hint only, and is usually specified as 0.

Suitable length and offset parameters are queried with the VIDIOC_QUERYBUF ioctl. Buffers must be allocated with the VIDIOC_REQBUFS ioctl before they can be queried.

To unmap buffers the munmap() function is used.

## Return Value

On success mmap() returns a pointer to the mapped buffer. On error MAP_FAILED (-1) is returned, and the errno variable is set appropriately. Possible error codes are:

EBADF

*fd* is not a valid file descriptor.

EACCES

*fd* is not open for reading and writing.

EINVAL

The *start* or *length* or *offset* are not suitable. (E. g. they are too large, or not aligned on a PAGESIZE boundary.)

The *flags* or *prot* value is not supported.

No buffers have been allocated with the VIDIOC_REQBUFS ioctl.

ENOMEM

Not enough physical or virtual memory was available to complete the request.

# V4L2 munmap()

## Name

v4l2-munmap — Unmap device memory

## Synopsis

```
#include <unistd.h>
#include <sys/mman.h>
int munmap(void *start, size_t length);
```

## Arguments

*start*

Address of the mapped buffer as returned by the mmap() function.

*length*

Length of the mapped buffer. This must be the same value as given to `mmap()` and returned by the driver in the struct v4l2_buffer *length* field for the single-planar API and in the struct v4l2_plane *length* field for the multi-planar API.

## Description

Unmaps a previously with the `mmap()` function mapped buffer and frees it, if possible.

## Return Value

On success `munmap()` returns 0, on failure -1 and the `errno` variable is set appropriately:

EINVAL

The *start* or *length* is incorrect, or no buffers have been mapped yet.

# V4L2 open()

## Name

`v4l2-open` — Open a V4L2 device

## Synopsis

```
#include <fcntl.h>
int open(const char *device_name, int flags);
```

## Arguments

*device_name*

Device to be opened.

*flags*

Open flags. Access mode must be O_RDWR. This is just a technicality, input devices still support only reading and output devices only writing.

When the O_NONBLOCK flag is given, the read() function and the VIDIOC_DQBUF ioctl will return the EAGAIN error code when no data is available or no buffer is in the driver outgoing queue, otherwise these functions block until data becomes available. All V4L2 drivers exchanging data with applications must support the O_NONBLOCK flag.

Other flags have no effect.

## Description

To open a V4L2 device applications call open() with the desired device name. This function has no side effects; all data format parameters, current input or output, control values or other properties remain unchanged. At the first open() call after loading the driver they will be reset to default values, drivers are never in an undefined state.

## Return Value

On success open returns the new file descriptor. On error -1 is returned, and the errno variable is set appropriately. Possible error codes are:

EACCES

The caller has no permission to access the device.

EBUSY

The driver does not support multiple opens and the device is already in use.

ENXIO

No device corresponding to this device special file exists.

ENOMEM

Not enough kernel memory was available to complete the request.

EMFILE

The process already has the maximum number of files open.

ENFILE

The limit on the total number of files open on the system has been reached.

# V4L2 poll()

## Name

v4l2-poll — Wait for some event on a file descriptor

## Synopsis

```
#include <sys/poll.h>
int poll(struct pollfd *ufds, unsigned int nfds, int timeout);
```

## Description

With the poll() function applications can suspend execution until the driver has captured data or is ready to accept data for output.

When streaming I/O has been negotiated this function waits until a buffer has been filled or displayed and can be dequeued with the VIDIOC_DQBUF ioctl. When buffers are already in the outgoing queue of the driver the function returns immediately.

On success poll() returns the number of file descriptors that have been selected (that is, file descriptors for which the *revents* field of the respective pollfd structure is non-zero). Capture devices set the POLLIN and POLLRDNORM flags in the *revents* field, output devices the POLLOUT and POLLWRNORM flags. When the function timed out it returns a value of zero, on failure it returns -1 and the errno variable is set appropriately. When the application did not call VIDIOC_QBUF or

VIDIOC_STREAMON yet the poll() function succeeds, but sets the POLLERR flag in the *revents* field.

When use of the read() function has been negotiated and the driver does not capture yet, the poll function starts capturing. When that fails it returns a POLLERR as above. Otherwise it waits until data has been captured and can be read. When the driver captures continuously (as opposed to, for example, still images) the function may return immediately.

When use of the write() function has been negotiated the poll function just waits until the driver is ready for a non-blocking write() call.

All drivers implementing the read() or write() function or streaming I/O must also support the poll() function.

For more details see the poll() manual page.

## Return Value

On success, poll() returns the number structures which have non-zero *revents* fields, or zero if the call timed out. On error -1 is returned, and the errno variable is set appropriately:

EBADF

One or more of the *ufds* members specify an invalid file descriptor.

EBUSY

The driver does not support multiple read or write streams and the device is already in use.

EFAULT

*ufds* references an inaccessible memory area.

EINTR

The call was interrupted by a signal.

EINVAL

The *nfds* argument is greater than OPEN_MAX.

# V4L2 read()

## Name

v4l2-read — Read from a V4L2 device

## Synopsis

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
```

## Arguments

*fd*

File descriptor returned by open().

*buf*

*count*

## Description

read() attempts to read up to *count* bytes from file descriptor *fd* into the buffer starting at *buf*. The layout of the data in the buffer is discussed in the respective device interface section, see ##. If *count* is zero, read() returns zero and has no other results. If *count* is greater than SSIZE_MAX, the result is unspecified. Regardless of the *count* value each read() call will provide at most one frame (two fields) worth of data.

By default read() blocks until data becomes available. When the O_NONBLOCK flag was given to the open() function it returns immediately with an EAGAIN error code when no data is available. The select() or poll() functions can always be used to suspend execution until data becomes available. All drivers supporting the read() function must also support select() and poll().

Drivers can implement read functionality in different ways, using a single or multiple buffers and discarding the oldest or newest frames once the internal buffers are filled.

`read()` never returns a "snapshot" of a buffer being filled. Using a single buffer the driver will stop capturing when the application starts reading the buffer until the read is finished. Thus only the period of the vertical blanking interval is available for reading, or the capture rate must fall below the nominal frame rate of the video standard.

The behavior of `read()` when called during the active picture period or the vertical blanking separating the top and bottom field depends on the discarding policy. A driver discarding the oldest frames keeps capturing into an internal buffer, continuously overwriting the previously, not read frame, and returns the frame being received at the time of the `read()` call as soon as it is complete.

A driver discarding the newest frames stops capturing until the next `read()` call. The frame being received at `read()` time is discarded, returning the following frame instead. Again this implies a reduction of the capture rate to one half or less of the nominal frame rate. An example of this model is the video read mode of the bttv driver, initiating a DMA to user memory when `read()` is called and returning when the DMA finished.

In the multiple buffer model drivers maintain a ring of internal buffers, automatically advancing to the next free buffer. This allows continuous capturing when the application can empty the buffers fast enough. Again, the behavior when the driver runs out of free buffers depends on the discarding policy.

Applications can get and set the number of buffers used internally by the driver with the `VIDIOC_G_PARM` and `VIDIOC_S_PARM` ioctls. They are optional, however. The discarding policy is not reported and cannot be changed. For minimum requirements see Chapter 4.

## Return Value

On success, the number of bytes read is returned. It is not an error if this number is smaller than the number of bytes requested, or the amount of data required for one frame. This may happen for example because `read()` was interrupted by a signal. On error, -1 is returned, and the `errno` variable is set appropriately. In this case the next read will start at the beginning of a new frame. Possible error codes are:

EAGAIN

Non-blocking I/O has been selected using O_NONBLOCK and no data was immediately available for reading.

EBADF

*fd* is not a valid file descriptor or is not open for reading, or the process already has the maximum number of files open.

EBUSY

The driver does not support multiple read streams and the device is already in use.

EFAULT

*buf* references an inaccessible memory area.

EINTR

The call was interrupted by a signal before any data was read.

EIO

I/O error. This indicates some hardware problem or a failure to communicate with a remote device (USB camera etc.).

EINVAL

The `read()` function is not supported by this driver, not on this device, or generally not on this type of device.

# V4L2 select()

## Name

`v4l2-select` — Synchronous I/O multiplexing

## Synopsis

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set
*exceptfds, struct timeval *timeout);
```

## Description

With the select() function applications can suspend execution until the driver has captured data or is ready to accept data for output.

When streaming I/O has been negotiated this function waits until a buffer has been filled or displayed and can be dequeued with the VIDIOC_DQBUF ioctl. When buffers are already in the outgoing queue of the driver the function returns immediately.

On success select() returns the total number of bits set in the fd_sets. When the function timed out it returns a value of zero. On failure it returns -1 and the errno variable is set appropriately. When the application did not call VIDIOC_QBUF or VIDIOC_STREAMON yet the select() function succeeds, setting the bit of the file descriptor in *readfds* or *writefds*, but subsequent VIDIOC_DQBUF calls will fail.[1]

When use of the read() function has been negotiated and the driver does not capture yet, the select() function starts capturing. When that fails, select() returns successful and a subsequent read() call, which also attempts to start capturing, will return an appropriate error code. When the driver captures continuously (as opposed to, for example, still images) and data is already available the select() function returns immediately.

When use of the write() function has been negotiated the select() function just waits until the driver is ready for a non-blocking write() call.

All drivers implementing the read() or write() function or streaming I/O must also support the select() function.

For more details see the select() manual page.

## Return Value

On success, select() returns the number of descriptors contained in the three returned descriptor sets, which will be zero if the timeout expired. On error -1 is returned, and the errno variable is set appropriately; the sets and *timeout* are undefined. Possible error codes are:

EBADF

One or more of the file descriptor sets specified a file descriptor that is not open.

EBUSY

The driver does not support multiple read or write streams and the device is already in use.

EFAULT

The *readfds*, *writefds*, *exceptfds* or *timeout* pointer references an inaccessible memory area.

EINTR

The call was interrupted by a signal.

EINVAL

The *nfds* argument is less than zero or greater than FD_SETSIZE.

### Notes

1. The Linux kernel implements select() like the poll() function, but select() cannot return a POLLERR.

## V4L2 write()

### Name

v4l2-write — Write to a V4L2 device

### Synopsis

```
#include <unistd.h>
ssize_t write(int fd, void *buf, size_t count);
```

### Arguments

*fd*

File descriptor returned by `open()`.

*`buf`*

*`count`*

## Description

`write()` writes up to *`count`* bytes to the device referenced by the file descriptor *`fd`* from the buffer starting at *`buf`*. When the hardware outputs are not active yet, this function enables them. When *`count`* is zero, `write()` returns 0 without any other effect.

When the application does not provide more data in time, the previous video frame, raw VBI image, sliced VPS or WSS data is displayed again. Sliced Teletext or Closed Caption data is not repeated, the driver inserts a blank line instead.

## Return Value

On success, the number of bytes written are returned. Zero indicates nothing was written. On error, -1 is returned, and the `errno` variable is set appropriately. In this case the next write will start at the beginning of a new frame. Possible error codes are:

EAGAIN

Non-blocking I/O has been selected using the `O_NONBLOCK` flag and no buffer space was available to write the data immediately.

EBADF

*`fd`* is not a valid file descriptor or is not open for writing.

EBUSY

The driver does not support multiple write streams and the device is already in use.

EFAULT

*`buf`* references an inaccessible memory area.

EINTR

The call was interrupted by a signal before any data was written.

EIO

I/O error. This indicates some hardware problem.

EINVAL

The `write()` function is not supported by this driver, not on this device, or generally not on this type of device.

# Notes

1. The supported standards may overlap and we need an unambiguous set to find the current standard returned by `VIDIOC_G_STD`.

1. `V4L2_CTRL_FLAG_DISABLED` was intended for two purposes: Drivers can skip predefined controls not supported by the hardware (although returning EINVAL would do as well), or disable predefined and private controls after hardware detection without the trouble of reordering control arrays and indices (EINVAL cannot be used to skip private controls because it would prematurely end the enumeration).

1. The Linux kernel implements `select()` like the `poll()` function, but `select()` cannot return a `POLLERR`.

# Appendix B. Video For Linux Two Header File

```
/*
 *  Video for Linux Two header file
 *
 *  Copyright (C) 1999-2007 the contributors
 *
 *  This program is free software; you can redistribute it and/or modify
 *  it under the terms of the GNU General Public License as published by
 *  the Free Software Foundation; either version 2 of the License, or
 *  (at your option) any later version.
 *
 *  This program is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *  GNU General Public License for more details.
 *
 *  Alternatively you can redistribute this file under the terms of the
 *  BSD license as stated below:
 *
 *  Redistribution and use in source and binary forms, with or without
 *  modification, are permitted provided that the following conditions
 *  are met:
 *  1. Redistributions of source code must retain the above copyright
 *     notice, this list of conditions and the following disclaimer.
 *  2. Redistributions in binary form must reproduce the above copyright
 *     notice, this list of conditions and the following disclaimer in
 *     the documentation and/or other materials provided with the
 *     distribution.
 *  3. The names of its contributors may not be used to endorse or promote
 *     products derived from this software without specific prior written
 *     permission.
 *
 *  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 *  "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 *  LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
 *  A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
 *  OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
 *  SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMI
 *  TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
 *  PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
 *  LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
 *  NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
 *  SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

```
 *
 *      Header file for v4l or V4L2 drivers and applications
 * with public API.
 * All kernel-specific stuff were moved to media/v4l2-dev.h, so
 * no #if __KERNEL tests are allowed here
 *
 *      See http://linuxtv.org for more info
 *
 *      Author: Bill Dirks <bill@thedirks.org>
 *              Justin Schoeman
 *              Hans Verkuil <hverkuil@xs4all.nl>
 *              et al.
 */
#ifndef __LINUX_VIDEODEV2_H
#define __LINUX_VIDEODEV2_H

#ifdef __KERNEL__
#include <linux/time.h>     /* need struct timeval */
#else
#include <sys/time.h>
#endif
#include <linux/compiler.h>
#include <linux/ioctl.h>
#include <linux/types.h>

/*
 * Common stuff for both V4L1 and V4L2
 * Moved from videodev.h
 */
#define VIDEO_MAX_FRAME               32
#define VIDEO_MAX_PLANES               8

#ifndef __KERNEL__

/* These defines are V4L1 specific and should not be used with the V4L2 AP
   They will be removed from this header in the future. */

#define VID_TYPE_CAPTURE        1       /* Can capture */
#define VID_TYPE_TUNER          2       /* Can tune */
#define VID_TYPE_TELETEXT       4       /* Does teletext */
#define VID_TYPE_OVERLAY        8       /* Overlay onto frame buffer */
#define VID_TYPE_CHROMAKEY      16      /* Overlay by chromakey */
#define VID_TYPE_CLIPPING       32      /* Can clip */
#define VID_TYPE_FRAMERAM       64      /* Uses the frame buffer memory */
#define VID_TYPE_SCALES         128     /* Scalable */
#define VID_TYPE_MONOCHROME     256     /* Monochrome only */
#define VID_TYPE_SUBCAPTURE     512     /* Can capture subareas of the ima
#define VID_TYPE_MPEG_DECODER   1024    /* Can decode MPEG streams */
```

```
#define VID_TYPE_MPEG_ENCODER   2048    /* Can encode MPEG streams */
#define VID_TYPE_MJPEG_DECODER  4096    /* Can decode MJPEG streams */
#define VID_TYPE_MJPEG_ENCODER  8192    /* Can encode MJPEG streams */
#endif

/*
 *      M I S C E L L A N E O U S
 */

/*  Four-character-code (FOURCC) */
#define v4l2_fourcc(a, b, c, d)\
        ((__u32)(a) | ((__u32)(b) << 8) | ((__u32)(c) << 16) | ((__u32)(d)

/*
 *      E N U M S
 */
enum v4l2_field {
        V4L2_FIELD_ANY           = 0, /* driver can choose from none,
                                         top, bottom, interlaced
                                         depending on whatever it thinks
                                         is approximate ... */
        V4L2_FIELD_NONE          = 1, /* this device has no fields ... */
        V4L2_FIELD_TOP           = 2, /* top field only */
        V4L2_FIELD_BOTTOM        = 3, /* bottom field only */
        V4L2_FIELD_INTERLACED    = 4, /* both fields interlaced */
        V4L2_FIELD_SEQ_TB        = 5, /* both fields sequential into one
                                         buffer, top-bottom order */
        V4L2_FIELD_SEQ_BT        = 6, /* same as above + bottom-top order
        V4L2_FIELD_ALTERNATE     = 7, /* both fields alternating into
                                         separate buffers */
        V4L2_FIELD_INTERLACED_TB = 8, /* both fields interlaced, top field
                                         first and the top field is
                                         transmitted first */
        V4L2_FIELD_INTERLACED_BT = 9, /* both fields interlaced, top field
                                         first and the bottom field is
                                         transmitted first */
};
#define V4L2_FIELD_HAS_TOP(field)       \
        ((field) == V4L2_FIELD_TOP      ||\
         (field) == V4L2_FIELD_INTERLACED ||\
         (field) == V4L2_FIELD_INTERLACED_TB ||\
         (field) == V4L2_FIELD_INTERLACED_BT ||\
         (field) == V4L2_FIELD_SEQ_TB   ||\
         (field) == V4L2_FIELD_SEQ_BT)
#define V4L2_FIELD_HAS_BOTTOM(field)    \
        ((field) == V4L2_FIELD_BOTTOM   ||\
         (field) == V4L2_FIELD_INTERLACED ||\
         (field) == V4L2_FIELD_INTERLACED_TB ||\
```

```
         (field) == V4L2_FIELD_INTERLACED_BT ||\
         (field) == V4L2_FIELD_SEQ_TB   ||\
         (field) == V4L2_FIELD_SEQ_BT)
#define V4L2_FIELD_HAS_BOTH(field)      \
        ((field) == V4L2_FIELD_INTERLACED ||\
         (field) == V4L2_FIELD_INTERLACED_TB ||\
         (field) == V4L2_FIELD_INTERLACED_BT ||\
         (field) == V4L2_FIELD_SEQ_TB ||\
         (field) == V4L2_FIELD_SEQ_BT)

enum v4l2_buf_type {
        V4L2_BUF_TYPE_VIDEO_CAPTURE        = 1,
        V4L2_BUF_TYPE_VIDEO_OUTPUT         = 2,
        V4L2_BUF_TYPE_VIDEO_OVERLAY        = 3,
        V4L2_BUF_TYPE_VBI_CAPTURE          = 4,
        V4L2_BUF_TYPE_VBI_OUTPUT           = 5,
        V4L2_BUF_TYPE_SLICED_VBI_CAPTURE   = 6,
        V4L2_BUF_TYPE_SLICED_VBI_OUTPUT    = 7,
#if 1
        /* Experimental */
        V4L2_BUF_TYPE_VIDEO_OUTPUT_OVERLAY = 8,
#endif
        V4L2_BUF_TYPE_VIDEO_CAPTURE_MPLANE = 9,
        V4L2_BUF_TYPE_VIDEO_OUTPUT_MPLANE  = 10,
        V4L2_BUF_TYPE_PRIVATE              = 0x80,
};

#define V4L2_TYPE_IS_MULTIPLANAR(type)                  \
        ((type) == V4L2_BUF_TYPE_VIDEO_CAPTURE_MPLANE   \
         || (type) == V4L2_BUF_TYPE_VIDEO_OUTPUT_MPLANE)

#define V4L2_TYPE_IS_OUTPUT(type)                               \
        ((type) == V4L2_BUF_TYPE_VIDEO_OUTPUT                   \
         || (type) == V4L2_BUF_TYPE_VIDEO_OUTPUT_MPLANE         \
         || (type) == V4L2_BUF_TYPE_VIDEO_OVERLAY               \
         || (type) == V4L2_BUF_TYPE_VIDEO_OUTPUT_OVERLAY        \
         || (type) == V4L2_BUF_TYPE_VBI_OUTPUT                  \
         || (type) == V4L2_BUF_TYPE_SLICED_VBI_OUTPUT)

enum v4l2_tuner_type {
        V4L2_TUNER_RADIO            = 1,
        V4L2_TUNER_ANALOG_TV        = 2,
        V4L2_TUNER_DIGITAL_TV       = 3,
};

enum v4l2_memory {
        V4L2_MEMORY_MMAP            = 1,
        V4L2_MEMORY_USERPTR         = 2,
```

```
        V4L2_MEMORY_OVERLAY          = 3,
};

/* see also http://vektor.theorem.ca/graphics/ycbcr/ */
enum v4l2_colorspace {
        /* ITU-R 601 -- broadcast NTSC/PAL */
        V4L2_COLORSPACE_SMPTE170M     = 1,

        /* 1125-Line (US) HDTV */
        V4L2_COLORSPACE_SMPTE240M     = 2,

        /* HD and modern captures. */
        V4L2_COLORSPACE_REC709        = 3,

        /* broken BT878 extents (601, luma range 16-253 instead of 16-235)
        V4L2_COLORSPACE_BT878         = 4,

        /* These should be useful.  Assume 601 extents. */
        V4L2_COLORSPACE_470_SYSTEM_M  = 5,
        V4L2_COLORSPACE_470_SYSTEM_BG = 6,

        /* I know there will be cameras that send this.  So, this is
         * unspecified chromaticities and full 0-255 on each of the
         * Y'CbCr components
         */
        V4L2_COLORSPACE_JPEG          = 7,

        /* For RGB colourspaces, this is probably a good start. */
        V4L2_COLORSPACE_SRGB          = 8,
};

enum v4l2_priority {
        V4L2_PRIORITY_UNSET       = 0,  /* not initialized */
        V4L2_PRIORITY_BACKGROUND  = 1,
        V4L2_PRIORITY_INTERACTIVE = 2,
        V4L2_PRIORITY_RECORD      = 3,
        V4L2_PRIORITY_DEFAULT     = V4L2_PRIORITY_INTERACTIVE,
};

struct v4l2_rect {
        __s32   left;
        __s32   top;
        __s32   width;
        __s32   height;
};

struct v4l2_fract {
        __u32   numerator;
```

```
        __u32   denominator;
};

/*
 *      D R I V E R   C A P A B I L I T I E S
 */
struct v4l2_capability {
        __u8    driver[16];     /* i.e.ie; "bttv" */
        __u8    card[32];       /* i.e.ie; "Hauppauge WinTV" */
        __u8    bus_info[32];   /* "PCI:" + pci_name(pci_dev) */
        __u32   version;        /* should use KERNEL_VERSION() */
        __u32   capabilities;   /* Device capabilities */
        __u32   reserved[4];
};

/* Values for 'capabilities' field */
#define V4L2_CAP_VIDEO_CAPTURE          0x00000001  /* Is a video capture
#define V4L2_CAP_VIDEO_OUTPUT           0x00000002  /* Is a video output
#define V4L2_CAP_VIDEO_OVERLAY          0x00000004  /* Can do video overla
#define V4L2_CAP_VBI_CAPTURE            0x00000010  /* Is a raw VBI captur
#define V4L2_CAP_VBI_OUTPUT             0x00000020  /* Is a raw VBI output
#define V4L2_CAP_SLICED_VBI_CAPTURE     0x00000040  /* Is a sliced VBI cap
#define V4L2_CAP_SLICED_VBI_OUTPUT      0x00000080  /* Is a sliced VBI out
#define V4L2_CAP_RDS_CAPTURE            0x00000100  /* RDS data capture */
#define V4L2_CAP_VIDEO_OUTPUT_OVERLAY   0x00000200  /* Can do video output
#define V4L2_CAP_HW_FREQ_SEEK           0x00000400  /* Can do hardware fre
#define V4L2_CAP_RDS_OUTPUT             0x00000800  /* Is an RDS encoder *

/* Is a video capture device that supports multiplanar formats */
#define V4L2_CAP_VIDEO_CAPTURE_MPLANE   0x00001000
/* Is a video output device that supports multiplanar formats */
#define V4L2_CAP_VIDEO_OUTPUT_MPLANE    0x00002000

#define V4L2_CAP_TUNER                  0x00010000  /* has a tuner */
#define V4L2_CAP_AUDIO                  0x00020000  /* has audio support *
#define V4L2_CAP_RADIO                  0x00040000  /* is a radio device *
#define V4L2_CAP_MODULATOR              0x00080000  /* has a modulator */

#define V4L2_CAP_READWRITE              0x01000000  /* read/write systemca
#define V4L2_CAP_ASYNCIO                0x02000000  /* async I/O */
#define V4L2_CAP_STREAMING              0x04000000  /* streaming I/O ioctl

/*
 *      V I D E O   I M A G E   F O R M A T
 */
struct v4l2_pix_format {
        __u32                   width;
        __u32                   height;
```

```
        __u32                   pixelformat;
        enum v4l2_field         field;
        __u32                   bytesperline;   /* for padding, zero if un
        __u32                   sizeimage;
        enum v4l2_colorspace    colorspace;
        __u32                   priv;           /* private data, depends o
};

/*      Pixel format         FOURCC                          depth  Descri

/* RGB formats */
#define V4L2_PIX_FMT_RGB332  v4l2_fourcc('R', 'G', 'B', '1') /*  8  RGB-3-
#define V4L2_PIX_FMT_RGB444  v4l2_fourcc('R', '4', '4', '4') /* 16  xxxxrr
#define V4L2_PIX_FMT_RGB555  v4l2_fourcc('R', 'G', 'B', 'O') /* 16  RGB-5-
#define V4L2_PIX_FMT_RGB565  v4l2_fourcc('R', 'G', 'B', 'P') /* 16  RGB-5-
#define V4L2_PIX_FMT_RGB555X v4l2_fourcc('R', 'G', 'B', 'Q') /* 16  RGB-5-
#define V4L2_PIX_FMT_RGB565X v4l2_fourcc('R', 'G', 'B', 'R') /* 16  RGB-5-
#define V4L2_PIX_FMT_BGR666  v4l2_fourcc('B', 'G', 'R', 'H') /* 18  BGR-6-
#define V4L2_PIX_FMT_BGR24   v4l2_fourcc('B', 'G', 'R', '3') /* 24  BGR-8-
#define V4L2_PIX_FMT_RGB24   v4l2_fourcc('R', 'G', 'B', '3') /* 24  RGB-8-
#define V4L2_PIX_FMT_BGR32   v4l2_fourcc('B', 'G', 'R', '4') /* 32  BGR-8-
#define V4L2_PIX_FMT_RGB32   v4l2_fourcc('R', 'G', 'B', '4') /* 32  RGB-8-

/* Grey formats */
#define V4L2_PIX_FMT_GREY    v4l2_fourcc('G', 'R', 'E', 'Y') /*  8  Greysc
#define V4L2_PIX_FMT_Y4      v4l2_fourcc('Y', '0', '4', ' ') /*  4  Greysc
#define V4L2_PIX_FMT_Y6      v4l2_fourcc('Y', '0', '6', ' ') /*  6  Greysc
#define V4L2_PIX_FMT_Y10     v4l2_fourcc('Y', '1', '0', ' ') /* 10  Greysc
#define V4L2_PIX_FMT_Y16     v4l2_fourcc('Y', '1', '6', ' ') /* 16  Greysc

/* Grey bit-packed formats */
#define V4L2_PIX_FMT_Y10BPACK    v4l2_fourcc('Y', '1', '0', 'B') /* 10  Gr

/* Palette formats */
#define V4L2_PIX_FMT_PAL8    v4l2_fourcc('P', 'A', 'L', '8') /*  8  8-bit

/* Luminance+Chrominance formats */
#define V4L2_PIX_FMT_YVU410  v4l2_fourcc('Y', 'V', 'U', '9') /*  9  YVU 4
#define V4L2_PIX_FMT_YVU420  v4l2_fourcc('Y', 'V', '1', '2') /* 12  YVU 4
#define V4L2_PIX_FMT_YUYV    v4l2_fourcc('Y', 'U', 'Y', 'V') /* 16  YUV 4
#define V4L2_PIX_FMT_YYUV    v4l2_fourcc('Y', 'Y', 'U', 'V') /* 16  YUV 4
#define V4L2_PIX_FMT_YVYU    v4l2_fourcc('Y', 'V', 'Y', 'U') /* 16 YVU 4:2
#define V4L2_PIX_FMT_UYVY    v4l2_fourcc('U', 'Y', 'V', 'Y') /* 16  YUV 4
#define V4L2_PIX_FMT_VYUY    v4l2_fourcc('V', 'Y', 'U', 'Y') /* 16  YUV 4
#define V4L2_PIX_FMT_YUV422P v4l2_fourcc('4', '2', '2', 'P') /* 16  YVU422
#define V4L2_PIX_FMT_YUV411P v4l2_fourcc('4', '1', '1', 'P') /* 16  YVU411
#define V4L2_PIX_FMT_Y41P    v4l2_fourcc('Y', '4', '1', 'P') /* 12  YUV 4
#define V4L2_PIX_FMT_YUV444  v4l2_fourcc('Y', '4', '4', '4') /* 16  xxxxyy
```

```
#define V4L2_PIX_FMT_YUV555  v4l2_fourcc('Y', 'U', 'V', 'O') /* 16  YUV-5-
#define V4L2_PIX_FMT_YUV565  v4l2_fourcc('Y', 'U', 'V', 'P') /* 16  YUV-5-
#define V4L2_PIX_FMT_YUV32   v4l2_fourcc('Y', 'U', 'V', '4') /* 32  YUV-8-
#define V4L2_PIX_FMT_YUV410  v4l2_fourcc('Y', 'U', 'V', '9') /*  9  YUV 4
#define V4L2_PIX_FMT_YUV420  v4l2_fourcc('Y', 'U', '1', '2') /* 12  YUV 4
#define V4L2_PIX_FMT_HI240   v4l2_fourcc('H', 'I', '2', '4') /*  8  8-bit
#define V4L2_PIX_FMT_HM12    v4l2_fourcc('H', 'M', '1', '2') /*  8  YUV 4
#define V4L2_PIX_FMT_M420    v4l2_fourcc('M', '4', '2', '0') /* 12  YUV 4

/* two planes -- one Y, one Cr + Cb interleaved  */
#define V4L2_PIX_FMT_NV12    v4l2_fourcc('N', 'V', '1', '2') /* 12  Y/CbC
#define V4L2_PIX_FMT_NV21    v4l2_fourcc('N', 'V', '2', '1') /* 12  Y/CrC
#define V4L2_PIX_FMT_NV16    v4l2_fourcc('N', 'V', '1', '6') /* 16  Y/CbC
#define V4L2_PIX_FMT_NV61    v4l2_fourcc('N', 'V', '6', '1') /* 16  Y/CrC

/* two non contiguous planes - one Y, one Cr + Cb interleaved  */
#define V4L2_PIX_FMT_NV12M   v4l2_fourcc('N', 'M', '1', '2') /* 12  Y/CbC
#define V4L2_PIX_FMT_NV12MT  v4l2_fourcc('T', 'M', '1', '2') /* 12  Y/CbC

/* three non contiguous planes - Y, Cb, Cr */
#define V4L2_PIX_FMT_YUV420M v4l2_fourcc('Y', 'M', '1', '2') /* 12  YUV42

/* Bayer formats - see http://www.siliconimaging.com/RGB%20Bayer.htm */
#define V4L2_PIX_FMT_SBGGR8  v4l2_fourcc('B', 'A', '8', '1') /*  8  BGBG.
#define V4L2_PIX_FMT_SGBRG8  v4l2_fourcc('G', 'B', 'R', 'G') /*  8  GBGB.
#define V4L2_PIX_FMT_SGRBG8  v4l2_fourcc('G', 'R', 'B', 'G') /*  8  GRGR.
#define V4L2_PIX_FMT_SRGGB8  v4l2_fourcc('R', 'G', 'G', 'B') /*  8  RGRG.
#define V4L2_PIX_FMT_SBGGR10 v4l2_fourcc('B', 'G', '1', '0') /* 10  BGBG.
#define V4L2_PIX_FMT_SGBRG10 v4l2_fourcc('G', 'B', '1', '0') /* 10  GBGB.
#define V4L2_PIX_FMT_SGRBG10 v4l2_fourcc('B', 'A', '1', '0') /* 10  GRGR.
#define V4L2_PIX_FMT_SRGGB10 v4l2_fourcc('R', 'G', '1', '0') /* 10  RGRG.
        /* 10bit raw bayer DPCM compressed to 8 bits */
#define V4L2_PIX_FMT_SGRBG10DPCM8 v4l2_fourcc('B', 'D', '1', '0')
        /*
         * 10bit raw bayer, expanded to 16 bits
         * xxxxrrrrrrrrrrxxxxgggggggggg xxxxggggggggggxxxxbbbbbbbbbb...
         */
#define V4L2_PIX_FMT_SBGGR16 v4l2_fourcc('B', 'Y', 'R', '2') /* 16  BGBG.

/* compressed formats */
#define V4L2_PIX_FMT_MJPEG    v4l2_fourcc('M', 'J', 'P', 'G') /* Motion-J
#define V4L2_PIX_FMT_JPEG     v4l2_fourcc('J', 'P', 'E', 'G') /* JFIF JPE
#define V4L2_PIX_FMT_DV       v4l2_fourcc('d', 'v', 's', 'd') /* 1394
#define V4L2_PIX_FMT_MPEG     v4l2_fourcc('M', 'P', 'E', 'G') /* MPEG-1/2

/*  Vendor-specific formats   */
#define V4L2_PIX_FMT_CPIA1    v4l2_fourcc('C', 'P', 'I', 'A') /* cpia1 YU
#define V4L2_PIX_FMT_WNVA     v4l2_fourcc('W', 'N', 'V', 'A') /* Winnov h
```

```
#define V4L2_PIX_FMT_SN9C10X  v4l2_fourcc('S', '9', '1', '0') /* SN9C10x
#define V4L2_PIX_FMT_SN9C20X_I420 v4l2_fourcc('S', '9', '2', '0') /* SN9C2
#define V4L2_PIX_FMT_PWC1     v4l2_fourcc('P', 'W', 'C', '1') /* pwc olde
#define V4L2_PIX_FMT_PWC2     v4l2_fourcc('P', 'W', 'C', '2') /* pwc newe
#define V4L2_PIX_FMT_ET61X251 v4l2_fourcc('E', '6', '2', '5') /* ET61X251
#define V4L2_PIX_FMT_SPCA501  v4l2_fourcc('S', '5', '0', '1') /* YUYV per
#define V4L2_PIX_FMT_SPCA505  v4l2_fourcc('S', '5', '0', '5') /* YYUV per
#define V4L2_PIX_FMT_SPCA508  v4l2_fourcc('S', '5', '0', '8') /* YUVY per
#define V4L2_PIX_FMT_SPCA561  v4l2_fourcc('S', '5', '6', '1') /* compresse
#define V4L2_PIX_FMT_PAC207   v4l2_fourcc('P', '2', '0', '7') /* compresse
#define V4L2_PIX_FMT_MR97310A v4l2_fourcc('M', '3', '1', '0') /* compresse
#define V4L2_PIX_FMT_SN9C2028 v4l2_fourcc('S', 'O', 'N', 'X') /* compresse
#define V4L2_PIX_FMT_SQ905C   v4l2_fourcc('9', '0', '5', 'C') /* compresse
#define V4L2_PIX_FMT_PJPG     v4l2_fourcc('P', 'J', 'P', 'G') /* Pixart 7
#define V4L2_PIX_FMT_OV511    v4l2_fourcc('O', '5', '1', '1') /* ov511 JPE
#define V4L2_PIX_FMT_OV518    v4l2_fourcc('O', '5', '1', '8') /* ov518 JPE
#define V4L2_PIX_FMT_STV0680  v4l2_fourcc('S', '6', '8', '0') /* stv0680 b
#define V4L2_PIX_FMT_TM6000   v4l2_fourcc('T', 'M', '6', '0') /* tm5600/tm
#define V4L2_PIX_FMT_CIT_YYVYUY v4l2_fourcc('C', 'I', 'T', 'V') /* one li
#define V4L2_PIX_FMT_KONICA420  v4l2_fourcc('K', 'O', 'N', 'I') /* YUV420

/*
 *      F O R M A T   E N U M E R A T I O N
 */
struct v4l2_fmtdesc {
        __u32               index;             /* Format number      */
        enum v4l2_buf_type  type;              /* buffer type        */
        __u32               flags;
        __u8                description[32];   /* Description string */
        __u32               pixelformat;       /* Format fourcc      */
        __u32               reserved[4];
};

#define V4L2_FMT_FLAG_COMPRESSED 0x0001
#define V4L2_FMT_FLAG_EMULATED   0x0002

#if 1
        /* Experimental Frame Size and frame rate enumeration */
/*
 *      F R A M E   S I Z E   E N U M E R A T I O N
 */
enum v4l2_frmsizetypes {
        V4L2_FRMSIZE_TYPE_DISCRETE      = 1,
        V4L2_FRMSIZE_TYPE_CONTINUOUS    = 2,
        V4L2_FRMSIZE_TYPE_STEPWISE      = 3,
};

struct v4l2_frmsize_discrete {
```

```
        __u32                   width;          /* Frame width [pixel] */
        __u32                   height;         /* Frame height [pixel] */
};

struct v4l2_frmsize_stepwise {
        __u32                   min_width;      /* Minimum frame width [p
        __u32                   max_width;      /* Maximum frame width [p
        __u32                   step_width;     /* Frame width step size
        __u32                   min_height;     /* Minimum frame height [
        __u32                   max_height;     /* Maximum frame height [
        __u32                   step_height;    /* Frame height step size
};

struct v4l2_frmsizeenum {
        __u32                   index;          /* Frame size number */
        __u32                   pixel_format;   /* Pixel format */
        __u32                   type;           /* Frame size type the dev

        union {                                 /* Frame size */
                struct v4l2_frmsize_discrete    discrete;
                struct v4l2_frmsize_stepwise    stepwise;
        };

        __u32   reserved[2];                    /* Reserved space for futu
};

/*
 *      F R A M E   R A T E   E N U M E R A T I O N
 */
enum v4l2_frmivaltypes {
        V4L2_FRMIVAL_TYPE_DISCRETE      = 1,
        V4L2_FRMIVAL_TYPE_CONTINUOUS    = 2,
        V4L2_FRMIVAL_TYPE_STEPWISE      = 3,
};

struct v4l2_frmival_stepwise {
        struct v4l2_fract       min;            /* Minimum frame interval
        struct v4l2_fract       max;            /* Maximum frame interval
        struct v4l2_fract       step;           /* Frame interval step si
};

struct v4l2_frmivalenum {
        __u32                   index;          /* Frame format index */
        __u32                   pixel_format;   /* Pixel format */
        __u32                   width;          /* Frame width */
        __u32                   height;         /* Frame height */
        __u32                   type;           /* Frame interval type the
```

```
        union {                                 /* Frame interval */
                struct v4l2_fract               discrete;
                struct v4l2_frmival_stepwise    stepwise;
        };

        __u32   reserved[2];                    /* Reserved space for futu
};
#endif

/*
 *      T I M E C O D E
 */
struct v4l2_timecode {
        __u32   type;
        __u32   flags;
        __u8    frames;
        __u8    seconds;
        __u8    minutes;
        __u8    hours;
        __u8    userbits[4];
};

/*  Type  */
#define V4L2_TC_TYPE_24FPS              1
#define V4L2_TC_TYPE_25FPS              2
#define V4L2_TC_TYPE_30FPS              3
#define V4L2_TC_TYPE_50FPS              4
#define V4L2_TC_TYPE_60FPS              5

/*  Flags  */
#define V4L2_TC_FLAG_DROPFRAME          0x0001 /* "drop-frame" mode */
#define V4L2_TC_FLAG_COLORFRAME         0x0002
#define V4L2_TC_USERBITS_field          0x000C
#define V4L2_TC_USERBITS_USERDEFINED    0x0000
#define V4L2_TC_USERBITS_8BITCHARS      0x0008
/* The above is based on SMPTE timecodes */

struct v4l2_jpegcompression {
        int quality;

        int  APPn;              /* Number of APP segment to be written,
                                 * must be 0..15 */
        int  APP_len;           /* Length of data in JPEG APPn segment */
        char APP_data[60];      /* Data in the JPEG APPn segment. */

        int  COM_len;           /* Length of data in JPEG COM segment */
        char COM_data[60];      /* Data in JPEG COM segment */
```

```
        __u32 jpeg_markers;     /* Which markers should go into the JPEG
                                 * output. Unless you exactly know what
                                 * you do, leave them untouched.
                                 * Inluding less markers will make the
                                 * resulting code smaller, but there will
                                 * be fewer applications which can read it
                                 * The presence of the APP and COM marker
                                 * is influenced by APP_len and COM_len
                                 * ONLY, not by this property! */

#define V4L2_JPEG_MARKER_DHT (1<<3)    /* Define Huffman Tables */
#define V4L2_JPEG_MARKER_DQT (1<<4)    /* Define Quantization Tables */
#define V4L2_JPEG_MARKER_DRI (1<<5)    /* Define Restart Interval */
#define V4L2_JPEG_MARKER_COM (1<<6)    /* Comment segment */
#define V4L2_JPEG_MARKER_APP (1<<7)    /* App segment, driver will
                                         * allways use APP0 */

};

/*
 *      M E M O R Y - M A P P I N G   B U F F E R S
 */
struct v4l2_requestbuffers {
        __u32                   count;
        enum v4l2_buf_type      type;
        enum v4l2_memory        memory;
        __u32                   reserved[2];
};

/**
 * struct v4l2_plane - plane info for multi-planar buffers
 * @bytesused:          number of bytes occupied by data in the plane (pay
 * @length:             size of this plane (NOT the payload) in bytes
 * @mem_offset:         when memory in the associated struct v4l2_buffer i
 *                      V4L2_MEMORY_MMAP, equals the offset from the start
 *                      the device memory for this plane (or is a "cookie"
 *                      should be passed to mmap() called on the video nod
 * @userptr:            when memory is V4L2_MEMORY_USERPTR, a userspace po
 *                      pointing to this plane
 * @data_offset:        offset in the plane to the start of data; usually
 *                      unless there is a header in front of the data
 *
 * Multi-planar buffers consist of one or more planes, e.g. an YCbCr buffe
 * with two planes can have one plane for Y, and another for interleaved C
 * components. Each plane can reside in a separate memory buffer, or even
 * a completely separate memory node (e.g. in embedded devices).
 */
struct v4l2_plane {
        __u32                   bytesused;
```

```
        __u32                   length;
        union {
                __u32           mem_offset;
                unsigned long   userptr;
        } m;
        __u32                   data_offset;
        __u32                   reserved[11];
};

/**
 * struct v4l2_buffer - video buffer info
 * @index:      id number of the buffer
 * @type:       buffer type (type == *_MPLANE for multiplanar buffers)
 * @bytesused:  number of bytes occupied by data in the buffer (payload);
 *              unused (set to 0) for multiplanar buffers
 * @flags:      buffer informational flags
 * @field:      field order of the image in the buffer
 * @timestamp:  frame timestamp
 * @timecode:   frame timecode
 * @sequence:   sequence count of this frame
 * @memory:     the method, in which the actual video data is passed
 * @offset:     for non-multiplanar buffers with memory == V4L2_MEMORY_MMA
 *              offset from the start of the device memory for this plane,
 *              (or a "cookie" that should be passed to mmap() as offset)
 * @userptr:    for non-multiplanar buffers with memory == V4L2_MEMORY_USE
 *              a userspace pointer pointing to this buffer
 * @planes:     for multiplanar buffers; userspace pointer to the array of
 *              info structs for this buffer
 * @length:     size in bytes of the buffer (NOT its payload) for single-p
 *              buffers (when type != *_MPLANE); number of elements in the
 *              planes array for multi-plane buffers
 * @input:      input number from which the video data has has been captur
 *
 * Contains data exchanged by application and driver using one of the Stre
 * I/O methods.
 */
struct v4l2_buffer {
        __u32                   index;
        enum v4l2_buf_type      type;
        __u32                   bytesused;
        __u32                   flags;
        enum v4l2_field         field;
        struct timeval          timestamp;
        struct v4l2_timecode    timecode;
        __u32                   sequence;

        /* memory location */
        enum v4l2_memory        memory;
```

```
        union {
                __u32           offset;
                unsigned long   userptr;
                struct v4l2_plane *planes;
        } m;
        __u32                   length;
        __u32                   input;
        __u32                   reserved;
};

/*  Flags for 'flags' field */
#define V4L2_BUF_FLAG_MAPPED    0x0001  /* Buffer is mapped (flag) */
#define V4L2_BUF_FLAG_QUEUED    0x0002  /* Buffer is queued for processing
#define V4L2_BUF_FLAG_DONE      0x0004  /* Buffer is ready */
#define V4L2_BUF_FLAG_KEYFRAME  0x0008  /* Image is a keyframe (I-frame) *
#define V4L2_BUF_FLAG_PFRAME    0x0010  /* Image is a P-frame */
#define V4L2_BUF_FLAG_BFRAME    0x0020  /* Image is a B-frame */
/* Buffer is ready, but the data contained within is corrupted. */
#define V4L2_BUF_FLAG_ERROR     0x0040
#define V4L2_BUF_FLAG_TIMECODE  0x0100  /* timecode field is valid */
#define V4L2_BUF_FLAG_INPUT     0x0200  /* input field is valid */

/*
 *      O V E R L A Y   P R E V I E W
 */
struct v4l2_framebuffer {
        __u32                   capability;
        __u32                   flags;
/* FIXME: in theory we should pass something like PCI device + memory
 * region + offset instead of some physical address */
        void                    *base;
        struct v4l2_pix_format  fmt;
};
/*  Flags for the 'capability' field. Read only */
#define V4L2_FBUF_CAP_EXTERNOVERLAY     0x0001
#define V4L2_FBUF_CAP_CHROMAKEY         0x0002
#define V4L2_FBUF_CAP_LIST_CLIPPING     0x0004
#define V4L2_FBUF_CAP_BITMAP_CLIPPING   0x0008
#define V4L2_FBUF_CAP_LOCAL_ALPHA       0x0010
#define V4L2_FBUF_CAP_GLOBAL_ALPHA      0x0020
#define V4L2_FBUF_CAP_LOCAL_INV_ALPHA   0x0040
#define V4L2_FBUF_CAP_SRC_CHROMAKEY     0x0080
/*  Flags for the 'flags' field. */
#define V4L2_FBUF_FLAG_PRIMARY          0x0001
#define V4L2_FBUF_FLAG_OVERLAY          0x0002
#define V4L2_FBUF_FLAG_CHROMAKEY        0x0004
#define V4L2_FBUF_FLAG_LOCAL_ALPHA      0x0008
#define V4L2_FBUF_FLAG_GLOBAL_ALPHA     0x0010
```

```
#define V4L2_FBUF_FLAG_LOCAL_INV_ALPHA  0x0020
#define V4L2_FBUF_FLAG_SRC_CHROMAKEY    0x0040

struct v4l2_clip {
        struct v4l2_rect        c;
        struct v4l2_clip        __user *next;
};

struct v4l2_window {
        struct v4l2_rect        w;
        enum v4l2_field         field;
        __u32                   chromakey;
        struct v4l2_clip        __user *clips;
        __u32                   clipcount;
        void                    __user *bitmap;
        __u8                    global_alpha;
};

/*
 *      C A P T U R E   P A R A M E T E R S
 */
struct v4l2_captureparm {
        __u32               capability;    /*  Supported modes */
        __u32               capturemode;   /*  Current mode */
        struct v4l2_fract   timeperframe;  /*  Time per frame in .1us units
        __u32               extendedmode;  /*  Driver-specific extensions
        __u32               readbuffers;   /*  # of buffers for read */
        __u32               reserved[4];
};

/*  Flags for 'capability' and 'capturemode' fields */
#define V4L2_MODE_HIGHQUALITY   0x0001  /*  High quality imaging mode */
#define V4L2_CAP_TIMEPERFRAME   0x1000  /*  timeperframe field is supporte

struct v4l2_outputparm {
        __u32               capability;   /*  Supported modes */
        __u32               outputmode;   /*  Current mode */
        struct v4l2_fract   timeperframe; /*  Time per frame in seconds */
        __u32               extendedmode; /*  Driver-specific extensions */
        __u32               writebuffers; /*  # of buffers for write */
        __u32               reserved[4];
};

/*
 *      I N P U T   I M A G E   C R O P P I N G
 */
struct v4l2_cropcap {
        enum v4l2_buf_type      type;
```

```
        struct v4l2_rect        bounds;
        struct v4l2_rect        defrect;
        struct v4l2_fract       pixelaspect;
};

struct v4l2_crop {
        enum v4l2_buf_type      type;
        struct v4l2_rect        c;
};

/*
 *      A N A L O G   V I D E O   S T A N D A R D
 */

typedef __u64 v4l2_std_id;

/* one bit for each */
#define V4L2_STD_PAL_B          ((v4l2_std_id)0x00000001)
#define V4L2_STD_PAL_B1         ((v4l2_std_id)0x00000002)
#define V4L2_STD_PAL_G          ((v4l2_std_id)0x00000004)
#define V4L2_STD_PAL_H          ((v4l2_std_id)0x00000008)
#define V4L2_STD_PAL_I          ((v4l2_std_id)0x00000010)
#define V4L2_STD_PAL_D          ((v4l2_std_id)0x00000020)
#define V4L2_STD_PAL_D1         ((v4l2_std_id)0x00000040)
#define V4L2_STD_PAL_K          ((v4l2_std_id)0x00000080)

#define V4L2_STD_PAL_M          ((v4l2_std_id)0x00000100)
#define V4L2_STD_PAL_N          ((v4l2_std_id)0x00000200)
#define V4L2_STD_PAL_Nc         ((v4l2_std_id)0x00000400)
#define V4L2_STD_PAL_60         ((v4l2_std_id)0x00000800)

#define V4L2_STD_NTSC_M         ((v4l2_std_id)0x00001000)
#define V4L2_STD_NTSC_M_JP      ((v4l2_std_id)0x00002000)
#define V4L2_STD_NTSC_443       ((v4l2_std_id)0x00004000)
#define V4L2_STD_NTSC_M_KR      ((v4l2_std_id)0x00008000)

#define V4L2_STD_SECAM_B        ((v4l2_std_id)0x00010000)
#define V4L2_STD_SECAM_D        ((v4l2_std_id)0x00020000)
#define V4L2_STD_SECAM_G        ((v4l2_std_id)0x00040000)
#define V4L2_STD_SECAM_H        ((v4l2_std_id)0x00080000)
#define V4L2_STD_SECAM_K        ((v4l2_std_id)0x00100000)
#define V4L2_STD_SECAM_K1       ((v4l2_std_id)0x00200000)
#define V4L2_STD_SECAM_L        ((v4l2_std_id)0x00400000)
#define V4L2_STD_SECAM_LC       ((v4l2_std_id)0x00800000)

/* ATSC/HDTV */
#define V4L2_STD_ATSC_8_VSB     ((v4l2_std_id)0x01000000)
#define V4L2_STD_ATSC_16_VSB    ((v4l2_std_id)0x02000000)
```

```
/* FIXME:
   Although std_id is 64 bits, there is an issue on PPC32 architecture tha
   makes switch(__u64) to break. So, there's a hack on v4l2-common.c round
   this value to 32 bits.
   As, currently, the max value is for V4L2_STD_ATSC_16_VSB (30 bits wide)
   it should work fine. However, if needed to add more than two standards,
   v4l2-common.c should be fixed.
 */

/* some merged standards */
#define V4L2_STD_MN     (V4L2_STD_PAL_M|V4L2_STD_PAL_N|V4L2_STD_PAL_Nc|V4L
#define V4L2_STD_B      (V4L2_STD_PAL_B|V4L2_STD_PAL_B1|V4L2_STD_SECAM_B)
#define V4L2_STD_GH     (V4L2_STD_PAL_G|V4L2_STD_PAL_H|V4L2_STD_SECAM_G|V4
#define V4L2_STD_DK     (V4L2_STD_PAL_DK|V4L2_STD_SECAM_DK)

/* some common needed stuff */
#define V4L2_STD_PAL_BG         (V4L2_STD_PAL_B         |\
                                 V4L2_STD_PAL_B1        |\
                                 V4L2_STD_PAL_G)
#define V4L2_STD_PAL_DK         (V4L2_STD_PAL_D         |\
                                 V4L2_STD_PAL_D1        |\
                                 V4L2_STD_PAL_K)
#define V4L2_STD_PAL            (V4L2_STD_PAL_BG        |\
                                 V4L2_STD_PAL_DK        |\
                                 V4L2_STD_PAL_H         |\
                                 V4L2_STD_PAL_I)
#define V4L2_STD_NTSC           (V4L2_STD_NTSC_M        |\
                                 V4L2_STD_NTSC_M_JP     |\
                                 V4L2_STD_NTSC_M_KR)
#define V4L2_STD_SECAM_DK       (V4L2_STD_SECAM_D       |\
                                 V4L2_STD_SECAM_K       |\
                                 V4L2_STD_SECAM_K1)
#define V4L2_STD_SECAM          (V4L2_STD_SECAM_B       |\
                                 V4L2_STD_SECAM_G       |\
                                 V4L2_STD_SECAM_H       |\
                                 V4L2_STD_SECAM_DK      |\
                                 V4L2_STD_SECAM_L       |\
                                 V4L2_STD_SECAM_LC)

#define V4L2_STD_525_60         (V4L2_STD_PAL_M         |\
                                 V4L2_STD_PAL_60        |\
                                 V4L2_STD_NTSC          |\
                                 V4L2_STD_NTSC_443)
#define V4L2_STD_625_50         (V4L2_STD_PAL           |\
                                 V4L2_STD_PAL_N         |\
                                 V4L2_STD_PAL_Nc        |\
                                 V4L2_STD_SECAM)
```

```
#define V4L2_STD_ATSC           (V4L2_STD_ATSC_8_VSB    |\
                                 V4L2_STD_ATSC_16_VSB)

#define V4L2_STD_UNKNOWN        0
#define V4L2_STD_ALL            (V4L2_STD_525_60        |\
                                 V4L2_STD_625_50)

struct v4l2_standard {
        __u32                index;
        v4l2_std_id          id;
        __u8                 name[24];
        struct v4l2_fract    frameperiod; /* Frames, not fields */
        __u32                framelines;
        __u32                reserved[4];
};

/*
 *      V I D E O       T I M I N G S   D V     P R E S E T
 */
struct v4l2_dv_preset {
        __u32   preset;
        __u32   reserved[4];
};

/*
 *      D V     P R E S E T S   E N U M E R A T I O N
 */
struct v4l2_dv_enum_preset {
        __u32   index;
        __u32   preset;
        __u8    name[32]; /* Name of the preset timing */
        __u32   width;
        __u32   height;
        __u32   reserved[4];
};

/*
 *      D V     P R E S E T     V A L U E S
 */
#define         V4L2_DV_INVALID         0
#define         V4L2_DV_480P59_94       1 /* BT.1362 */
#define         V4L2_DV_576P50          2 /* BT.1362 */
#define         V4L2_DV_720P24          3 /* SMPTE 296M */
#define         V4L2_DV_720P25          4 /* SMPTE 296M */
#define         V4L2_DV_720P30          5 /* SMPTE 296M */
#define         V4L2_DV_720P50          6 /* SMPTE 296M */
#define         V4L2_DV_720P59_94       7 /* SMPTE 274M */
#define         V4L2_DV_720P60          8 /* SMPTE 274M/296M */
```

```
#define         V4L2_DV_1080I29_97      9 /* BT.1120/ SMPTE 274M */
#define         V4L2_DV_1080I30         10 /* BT.1120/ SMPTE 274M */
#define         V4L2_DV_1080I25         11 /* BT.1120 */
#define         V4L2_DV_1080I50         12 /* SMPTE 296M */
#define         V4L2_DV_1080I60         13 /* SMPTE 296M */
#define         V4L2_DV_1080P24         14 /* SMPTE 296M */
#define         V4L2_DV_1080P25         15 /* SMPTE 296M */
#define         V4L2_DV_1080P30         16 /* SMPTE 296M */
#define         V4L2_DV_1080P50         17 /* BT.1120 */
#define         V4L2_DV_1080P60         18 /* BT.1120 */

/*
 *      D V     B T     T I M I N G S
 */

/* BT.656/BT.1120 timing data */
struct v4l2_bt_timings {
        __u32   width;          /* width in pixels */
        __u32   height;         /* height in lines */
        __u32   interlaced;     /* Interlaced or progressive */
        __u32   polarities;     /* Positive or negative polarity */
        __u64   pixelclock;     /* Pixel clock in HZ. Ex. 74.25MHz->74250
        __u32   hfrontporch;    /* Horizpontal front porch in pixels */
        __u32   hsync;          /* Horizontal Sync length in pixels */
        __u32   hbackporch;     /* Horizontal back porch in pixels */
        __u32   vfrontporch;    /* Vertical front porch in pixels */
        __u32   vsync;          /* Vertical Sync length in lines */
        __u32   vbackporch;     /* Vertical back porch in lines */
        __u32   il_vfrontporch; /* Vertical front porch for bottom field
                                 * interlaced field formats
                                 */
        __u32   il_vsync;       /* Vertical sync length for bottom field
                                 * interlaced field formats
                                 */
        __u32   il_vbackporch;  /* Vertical back porch for bottom field o
                                 * interlaced field formats
                                 */
        __u32   reserved[16];
} __attribute__ ((packed));

/* Interlaced or progressive format */
#define V4L2_DV_PROGRESSIVE     0
#define V4L2_DV_INTERLACED      1

/* Polarities. If bit is not set, it is assumed to be negative polarity */
#define V4L2_DV_VSYNC_POS_POL   0x00000001
#define V4L2_DV_HSYNC_POS_POL   0x00000002
```

```
/* DV timings */
struct v4l2_dv_timings {
        __u32 type;
        union {
                struct v4l2_bt_timings  bt;
                __u32   reserved[32];
        };
} __attribute__ ((packed));

/* Values for the type field */
#define V4L2_DV_BT_656_1120     0       /* BT.656/1120 timing type */

/*
 *      V I D E O   I N P U T S
 */
struct v4l2_input {
        __u32        index;             /*  Which input */
        __u8         name[32];          /*  Label */
        __u32        type;              /*  Type of input */
        __u32        audioset;          /*  Associated audios (bitfield) */
        __u32        tuner;             /*  Associated tuner */
        v4l2_std_id  std;
        __u32        status;
        __u32        capabilities;
        __u32        reserved[3];
};

/*  Values for the 'type' field */
#define V4L2_INPUT_TYPE_TUNER           1
#define V4L2_INPUT_TYPE_CAMERA          2

/* field 'status' - general */
#define V4L2_IN_ST_NO_POWER    0x00000001  /* Attached device is off */
#define V4L2_IN_ST_NO_SIGNAL   0x00000002
#define V4L2_IN_ST_NO_COLOR    0x00000004

/* field 'status' - sensor orientation */
/* If sensor is mounted upside down set both bits */
#define V4L2_IN_ST_HFLIP       0x00000010 /* Frames are flipped horizontal
#define V4L2_IN_ST_VFLIP       0x00000020 /* Frames are flipped vertically

/* field 'status' - analog */
#define V4L2_IN_ST_NO_H_LOCK   0x00000100  /* No horizontal sync lock */
#define V4L2_IN_ST_COLOR_KILL  0x00000200  /* Color killer is active */

/* field 'status' - digital */
#define V4L2_IN_ST_NO_SYNC     0x00010000  /* No synchronization lock */
```

```
#define V4L2_IN_ST_NO_EQU      0x00020000  /* No equalizer lock */
#define V4L2_IN_ST_NO_CARRIER  0x00040000  /* Carrier recovery failed */

/* field 'status' - VCR and set-top box */
#define V4L2_IN_ST_MACROVISION 0x01000000  /* Macrovision detected */
#define V4L2_IN_ST_NO_ACCESS   0x02000000  /* Conditional access denied */
#define V4L2_IN_ST_VTR         0x04000000  /* VTR time constant */

/* capabilities flags */
#define V4L2_IN_CAP_PRESETS             0x00000001 /* Supports S_DV_PRESET
#define V4L2_IN_CAP_CUSTOM_TIMINGS      0x00000002 /* Supports S_DV_TIMING
#define V4L2_IN_CAP_STD                 0x00000004 /* Supports S_STD */

/*
 *      V I D E O   O U T P U T S
 */
struct v4l2_output {
        __u32        index;             /*  Which output */
        __u8         name[32];          /*  Label */
        __u32        type;              /*  Type of output */
        __u32        audioset;          /*  Associated audios (bitfield)
        __u32        modulator;         /*  Associated modulator */
        v4l2_std_id  std;
        __u32        capabilities;
        __u32        reserved[3];
};
/*  Values for the 'type' field */
#define V4L2_OUTPUT_TYPE_MODULATOR              1
#define V4L2_OUTPUT_TYPE_ANALOG                 2
#define V4L2_OUTPUT_TYPE_ANALOGVGAOVERLAY       3

/* capabilities flags */
#define V4L2_OUT_CAP_PRESETS            0x00000001 /* Supports S_DV_PRESET
#define V4L2_OUT_CAP_CUSTOM_TIMINGS     0x00000002 /* Supports S_DV_TIMING
#define V4L2_OUT_CAP_STD                0x00000004 /* Supports S_STD */

/*
 *      C O N T R O L S
 */
struct v4l2_control {
        __u32                id;
        __s32                value;
};

struct v4l2_ext_control {
        __u32 id;
        __u32 size;
        __u32 reserved2[1];
```

```
        union {
                __s32 value;
                __s64 value64;
                char *string;
        };
} __attribute__ ((packed));

struct v4l2_ext_controls {
        __u32 ctrl_class;
        __u32 count;
        __u32 error_idx;
        __u32 reserved[2];
        struct v4l2_ext_control *controls;
};

/*  Values for ctrl_class field */
#define V4L2_CTRL_CLASS_USER 0x00980000 /* Old-style 'user' controls */
#define V4L2_CTRL_CLASS_MPEG 0x00990000 /* MPEG-compression controls */
#define V4L2_CTRL_CLASS_CAMERA 0x009a0000       /* Camera class controls
#define V4L2_CTRL_CLASS_FM_TX 0x009b0000        /* FM Modulator control cl

#define V4L2_CTRL_ID_MASK         (0x0fffffff)
#define V4L2_CTRL_ID2CLASS(id)    ((id) & 0x0fff0000UL)
#define V4L2_CTRL_DRIVER_PRIV(id) (((id) & 0xffff) >= 0x1000)

enum v4l2_ctrl_type {
        V4L2_CTRL_TYPE_INTEGER       = 1,
        V4L2_CTRL_TYPE_BOOLEAN       = 2,
        V4L2_CTRL_TYPE_MENU          = 3,
        V4L2_CTRL_TYPE_BUTTON        = 4,
        V4L2_CTRL_TYPE_INTEGER64     = 5,
        V4L2_CTRL_TYPE_CTRL_CLASS    = 6,
        V4L2_CTRL_TYPE_STRING        = 7,
};

/*  Used in the VIDIOC_QUERYCTRL ioctl for querying controls */
struct v4l2_queryctrl {
        __u32                id;
        enum v4l2_ctrl_type  type;
        __u8                 name[32];  /* Whatever */
        __s32                minimum;   /* Note signedness */
        __s32                maximum;
        __s32                step;
        __s32                default_value;
        __u32                flags;
        __u32                reserved[2];
};
```

```
/*  Used in the VIDIOC_QUERYMENU ioctl for querying menu items */
struct v4l2_querymenu {
        __u32           id;
        __u32           index;
        __u8            name[32];       /* Whatever */
        __u32           reserved;
};

/*  Control flags  */
#define V4L2_CTRL_FLAG_DISABLED         0x0001
#define V4L2_CTRL_FLAG_GRABBED          0x0002
#define V4L2_CTRL_FLAG_READ_ONLY        0x0004
#define V4L2_CTRL_FLAG_UPDATE           0x0008
#define V4L2_CTRL_FLAG_INACTIVE         0x0010
#define V4L2_CTRL_FLAG_SLIDER           0x0020
#define V4L2_CTRL_FLAG_WRITE_ONLY       0x0040

/*  Query flag, to be ORed with the control ID */
#define V4L2_CTRL_FLAG_NEXT_CTRL        0x80000000

/*  User-class control IDs defined by V4L2 */
#define V4L2_CID_BASE                   (V4L2_CTRL_CLASS_USER | 0x900)
#define V4L2_CID_USER_BASE              V4L2_CID_BASE
/*  IDs reserved for driver specific controls */
#define V4L2_CID_PRIVATE_BASE           0x08000000

#define V4L2_CID_USER_CLASS             (V4L2_CTRL_CLASS_USER | 1)
#define V4L2_CID_BRIGHTNESS             (V4L2_CID_BASE+0)
#define V4L2_CID_CONTRAST               (V4L2_CID_BASE+1)
#define V4L2_CID_SATURATION             (V4L2_CID_BASE+2)
#define V4L2_CID_HUE                    (V4L2_CID_BASE+3)
#define V4L2_CID_AUDIO_VOLUME           (V4L2_CID_BASE+5)
#define V4L2_CID_AUDIO_BALANCE          (V4L2_CID_BASE+6)
#define V4L2_CID_AUDIO_BASS             (V4L2_CID_BASE+7)
#define V4L2_CID_AUDIO_TREBLE           (V4L2_CID_BASE+8)
#define V4L2_CID_AUDIO_MUTE             (V4L2_CID_BASE+9)
#define V4L2_CID_AUDIO_LOUDNESS         (V4L2_CID_BASE+10)
#define V4L2_CID_BLACK_LEVEL            (V4L2_CID_BASE+11) /* Deprecated
#define V4L2_CID_AUTO_WHITE_BALANCE     (V4L2_CID_BASE+12)
#define V4L2_CID_DO_WHITE_BALANCE       (V4L2_CID_BASE+13)
#define V4L2_CID_RED_BALANCE            (V4L2_CID_BASE+14)
#define V4L2_CID_BLUE_BALANCE           (V4L2_CID_BASE+15)
#define V4L2_CID_GAMMA                  (V4L2_CID_BASE+16)
#define V4L2_CID_WHITENESS              (V4L2_CID_GAMMA) /* Deprecated */
#define V4L2_CID_EXPOSURE               (V4L2_CID_BASE+17)
#define V4L2_CID_AUTOGAIN               (V4L2_CID_BASE+18)
#define V4L2_CID_GAIN                   (V4L2_CID_BASE+19)
#define V4L2_CID_HFLIP                  (V4L2_CID_BASE+20)
```

```
#define V4L2_CID_VFLIP                  (V4L2_CID_BASE+21)

/* Deprecated; use V4L2_CID_PAN_RESET and V4L2_CID_TILT_RESET */
#define V4L2_CID_HCENTER                (V4L2_CID_BASE+22)
#define V4L2_CID_VCENTER                (V4L2_CID_BASE+23)

#define V4L2_CID_POWER_LINE_FREQUENCY   (V4L2_CID_BASE+24)
enum v4l2_power_line_frequency {
        V4L2_CID_POWER_LINE_FREQUENCY_DISABLED  = 0,
        V4L2_CID_POWER_LINE_FREQUENCY_50HZ      = 1,
        V4L2_CID_POWER_LINE_FREQUENCY_60HZ      = 2,
};
#define V4L2_CID_HUE_AUTO                       (V4L2_CID_BASE+25)
#define V4L2_CID_WHITE_BALANCE_TEMPERATURE      (V4L2_CID_BASE+26)
#define V4L2_CID_SHARPNESS                      (V4L2_CID_BASE+27)
#define V4L2_CID_BACKLIGHT_COMPENSATION         (V4L2_CID_BASE+28)
#define V4L2_CID_CHROMA_AGC                     (V4L2_CID_BASE+29)
#define V4L2_CID_COLOR_KILLER                   (V4L2_CID_BASE+30)
#define V4L2_CID_COLORFX                        (V4L2_CID_BASE+31)
enum v4l2_colorfx {
        V4L2_COLORFX_NONE       = 0,
        V4L2_COLORFX_BW         = 1,
        V4L2_COLORFX_SEPIA      = 2,
        V4L2_COLORFX_NEGATIVE = 3,
        V4L2_COLORFX_EMBOSS = 4,
        V4L2_COLORFX_SKETCH = 5,
        V4L2_COLORFX_SKY_BLUE = 6,
        V4L2_COLORFX_GRASS_GREEN = 7,
        V4L2_COLORFX_SKIN_WHITEN = 8,
        V4L2_COLORFX_VIVID = 9,
};
#define V4L2_CID_AUTOBRIGHTNESS                 (V4L2_CID_BASE+32)
#define V4L2_CID_BAND_STOP_FILTER               (V4L2_CID_BASE+33)

#define V4L2_CID_ROTATE                         (V4L2_CID_BASE+34)
#define V4L2_CID_BG_COLOR                       (V4L2_CID_BASE+35)

#define V4L2_CID_CHROMA_GAIN                    (V4L2_CID_BASE+36)

#define V4L2_CID_ILLUMINATORS_1                 (V4L2_CID_BASE+37)
#define V4L2_CID_ILLUMINATORS_2                 (V4L2_CID_BASE+38)

/* last CID + 1 */
#define V4L2_CID_LASTP1                         (V4L2_CID_BASE+39)

/*  MPEG-class control IDs defined by V4L2 */
#define V4L2_CID_MPEG_BASE                      (V4L2_CTRL_CLASS_MPEG | 0x
#define V4L2_CID_MPEG_CLASS                     (V4L2_CTRL_CLASS_MPEG | 1)
```

```
/*  MPEG streams */
#define V4L2_CID_MPEG_STREAM_TYPE                (V4L2_CID_MPEG_BASE+0)
enum v4l2_mpeg_stream_type {
        V4L2_MPEG_STREAM_TYPE_MPEG2_PS    = 0, /* MPEG-2 program stream */
        V4L2_MPEG_STREAM_TYPE_MPEG2_TS    = 1, /* MPEG-2 transport stream
        V4L2_MPEG_STREAM_TYPE_MPEG1_SS    = 2, /* MPEG-1 system stream */
        V4L2_MPEG_STREAM_TYPE_MPEG2_DVD   = 3, /* MPEG-2 DVD-compatible st
        V4L2_MPEG_STREAM_TYPE_MPEG1_VCD   = 4, /* MPEG-1 VCD-compatible st
        V4L2_MPEG_STREAM_TYPE_MPEG2_SVCD  = 5, /* MPEG-2 SVCD-compatible s
};
#define V4L2_CID_MPEG_STREAM_PID_PMT             (V4L2_CID_MPEG_BASE+1)
#define V4L2_CID_MPEG_STREAM_PID_AUDIO           (V4L2_CID_MPEG_BASE+2)
#define V4L2_CID_MPEG_STREAM_PID_VIDEO           (V4L2_CID_MPEG_BASE+3)
#define V4L2_CID_MPEG_STREAM_PID_PCR             (V4L2_CID_MPEG_BASE+4)
#define V4L2_CID_MPEG_STREAM_PES_ID_AUDIO        (V4L2_CID_MPEG_BASE+5)
#define V4L2_CID_MPEG_STREAM_PES_ID_VIDEO        (V4L2_CID_MPEG_BASE+6)
#define V4L2_CID_MPEG_STREAM_VBI_FMT             (V4L2_CID_MPEG_BASE+7)
enum v4l2_mpeg_stream_vbi_fmt {
        V4L2_MPEG_STREAM_VBI_FMT_NONE = 0,  /* No VBI in the MPEG stream
        V4L2_MPEG_STREAM_VBI_FMT_IVTV = 1,  /* VBI in private packets, IV
};

/*  MPEG audio */
#define V4L2_CID_MPEG_AUDIO_SAMPLING_FREQ        (V4L2_CID_MPEG_BASE+100)
enum v4l2_mpeg_audio_sampling_freq {
        V4L2_MPEG_AUDIO_SAMPLING_FREQ_44100 = 0,
        V4L2_MPEG_AUDIO_SAMPLING_FREQ_48000 = 1,
        V4L2_MPEG_AUDIO_SAMPLING_FREQ_32000 = 2,
};
#define V4L2_CID_MPEG_AUDIO_ENCODING             (V4L2_CID_MPEG_BASE+101)
enum v4l2_mpeg_audio_encoding {
        V4L2_MPEG_AUDIO_ENCODING_LAYER_1 = 0,
        V4L2_MPEG_AUDIO_ENCODING_LAYER_2 = 1,
        V4L2_MPEG_AUDIO_ENCODING_LAYER_3 = 2,
        V4L2_MPEG_AUDIO_ENCODING_AAC     = 3,
        V4L2_MPEG_AUDIO_ENCODING_AC3     = 4,
};
#define V4L2_CID_MPEG_AUDIO_L1_BITRATE           (V4L2_CID_MPEG_BASE+102)
enum v4l2_mpeg_audio_l1_bitrate {
        V4L2_MPEG_AUDIO_L1_BITRATE_32K  = 0,
        V4L2_MPEG_AUDIO_L1_BITRATE_64K  = 1,
        V4L2_MPEG_AUDIO_L1_BITRATE_96K  = 2,
        V4L2_MPEG_AUDIO_L1_BITRATE_128K = 3,
        V4L2_MPEG_AUDIO_L1_BITRATE_160K = 4,
        V4L2_MPEG_AUDIO_L1_BITRATE_192K = 5,
        V4L2_MPEG_AUDIO_L1_BITRATE_224K = 6,
        V4L2_MPEG_AUDIO_L1_BITRATE_256K = 7,
```

```
        V4L2_MPEG_AUDIO_L1_BITRATE_288K = 8,
        V4L2_MPEG_AUDIO_L1_BITRATE_320K = 9,
        V4L2_MPEG_AUDIO_L1_BITRATE_352K = 10,
        V4L2_MPEG_AUDIO_L1_BITRATE_384K = 11,
        V4L2_MPEG_AUDIO_L1_BITRATE_416K = 12,
        V4L2_MPEG_AUDIO_L1_BITRATE_448K = 13,
};
#define V4L2_CID_MPEG_AUDIO_L2_BITRATE          (V4L2_CID_MPEG_BASE+103)
enum v4l2_mpeg_audio_l2_bitrate {
        V4L2_MPEG_AUDIO_L2_BITRATE_32K  = 0,
        V4L2_MPEG_AUDIO_L2_BITRATE_48K  = 1,
        V4L2_MPEG_AUDIO_L2_BITRATE_56K  = 2,
        V4L2_MPEG_AUDIO_L2_BITRATE_64K  = 3,
        V4L2_MPEG_AUDIO_L2_BITRATE_80K  = 4,
        V4L2_MPEG_AUDIO_L2_BITRATE_96K  = 5,
        V4L2_MPEG_AUDIO_L2_BITRATE_112K = 6,
        V4L2_MPEG_AUDIO_L2_BITRATE_128K = 7,
        V4L2_MPEG_AUDIO_L2_BITRATE_160K = 8,
        V4L2_MPEG_AUDIO_L2_BITRATE_192K = 9,
        V4L2_MPEG_AUDIO_L2_BITRATE_224K = 10,
        V4L2_MPEG_AUDIO_L2_BITRATE_256K = 11,
        V4L2_MPEG_AUDIO_L2_BITRATE_320K = 12,
        V4L2_MPEG_AUDIO_L2_BITRATE_384K = 13,
};
#define V4L2_CID_MPEG_AUDIO_L3_BITRATE          (V4L2_CID_MPEG_BASE+104)
enum v4l2_mpeg_audio_l3_bitrate {
        V4L2_MPEG_AUDIO_L3_BITRATE_32K  = 0,
        V4L2_MPEG_AUDIO_L3_BITRATE_40K  = 1,
        V4L2_MPEG_AUDIO_L3_BITRATE_48K  = 2,
        V4L2_MPEG_AUDIO_L3_BITRATE_56K  = 3,
        V4L2_MPEG_AUDIO_L3_BITRATE_64K  = 4,
        V4L2_MPEG_AUDIO_L3_BITRATE_80K  = 5,
        V4L2_MPEG_AUDIO_L3_BITRATE_96K  = 6,
        V4L2_MPEG_AUDIO_L3_BITRATE_112K = 7,
        V4L2_MPEG_AUDIO_L3_BITRATE_128K = 8,
        V4L2_MPEG_AUDIO_L3_BITRATE_160K = 9,
        V4L2_MPEG_AUDIO_L3_BITRATE_192K = 10,
        V4L2_MPEG_AUDIO_L3_BITRATE_224K = 11,
        V4L2_MPEG_AUDIO_L3_BITRATE_256K = 12,
        V4L2_MPEG_AUDIO_L3_BITRATE_320K = 13,
};
#define V4L2_CID_MPEG_AUDIO_MODE                (V4L2_CID_MPEG_BASE+105)
enum v4l2_mpeg_audio_mode {
        V4L2_MPEG_AUDIO_MODE_STEREO       = 0,
        V4L2_MPEG_AUDIO_MODE_JOINT_STEREO = 1,
        V4L2_MPEG_AUDIO_MODE_DUAL         = 2,
        V4L2_MPEG_AUDIO_MODE_MONO         = 3,
};
```

```
#define V4L2_CID_MPEG_AUDIO_MODE_EXTENSION      (V4L2_CID_MPEG_BASE+106)
enum v4l2_mpeg_audio_mode_extension {
        V4L2_MPEG_AUDIO_MODE_EXTENSION_BOUND_4  = 0,
        V4L2_MPEG_AUDIO_MODE_EXTENSION_BOUND_8  = 1,
        V4L2_MPEG_AUDIO_MODE_EXTENSION_BOUND_12 = 2,
        V4L2_MPEG_AUDIO_MODE_EXTENSION_BOUND_16 = 3,
};
#define V4L2_CID_MPEG_AUDIO_EMPHASIS            (V4L2_CID_MPEG_BASE+107)
enum v4l2_mpeg_audio_emphasis {
        V4L2_MPEG_AUDIO_EMPHASIS_NONE         = 0,
        V4L2_MPEG_AUDIO_EMPHASIS_50_DIV_15_uS = 1,
        V4L2_MPEG_AUDIO_EMPHASIS_CCITT_J17    = 2,
};
#define V4L2_CID_MPEG_AUDIO_CRC                 (V4L2_CID_MPEG_BASE+108)
enum v4l2_mpeg_audio_crc {
        V4L2_MPEG_AUDIO_CRC_NONE  = 0,
        V4L2_MPEG_AUDIO_CRC_CRC16 = 1,
};
#define V4L2_CID_MPEG_AUDIO_MUTE                (V4L2_CID_MPEG_BASE+109)
#define V4L2_CID_MPEG_AUDIO_AAC_BITRATE         (V4L2_CID_MPEG_BASE+110)
#define V4L2_CID_MPEG_AUDIO_AC3_BITRATE         (V4L2_CID_MPEG_BASE+111)
enum v4l2_mpeg_audio_ac3_bitrate {
        V4L2_MPEG_AUDIO_AC3_BITRATE_32K  = 0,
        V4L2_MPEG_AUDIO_AC3_BITRATE_40K  = 1,
        V4L2_MPEG_AUDIO_AC3_BITRATE_48K  = 2,
        V4L2_MPEG_AUDIO_AC3_BITRATE_56K  = 3,
        V4L2_MPEG_AUDIO_AC3_BITRATE_64K  = 4,
        V4L2_MPEG_AUDIO_AC3_BITRATE_80K  = 5,
        V4L2_MPEG_AUDIO_AC3_BITRATE_96K  = 6,
        V4L2_MPEG_AUDIO_AC3_BITRATE_112K = 7,
        V4L2_MPEG_AUDIO_AC3_BITRATE_128K = 8,
        V4L2_MPEG_AUDIO_AC3_BITRATE_160K = 9,
        V4L2_MPEG_AUDIO_AC3_BITRATE_192K = 10,
        V4L2_MPEG_AUDIO_AC3_BITRATE_224K = 11,
        V4L2_MPEG_AUDIO_AC3_BITRATE_256K = 12,
        V4L2_MPEG_AUDIO_AC3_BITRATE_320K = 13,
        V4L2_MPEG_AUDIO_AC3_BITRATE_384K = 14,
        V4L2_MPEG_AUDIO_AC3_BITRATE_448K = 15,
        V4L2_MPEG_AUDIO_AC3_BITRATE_512K = 16,
        V4L2_MPEG_AUDIO_AC3_BITRATE_576K = 17,
        V4L2_MPEG_AUDIO_AC3_BITRATE_640K = 18,
};

/*  MPEG video */
#define V4L2_CID_MPEG_VIDEO_ENCODING            (V4L2_CID_MPEG_BASE+200)
enum v4l2_mpeg_video_encoding {
        V4L2_MPEG_VIDEO_ENCODING_MPEG_1     = 0,
        V4L2_MPEG_VIDEO_ENCODING_MPEG_2     = 1,
```

```
        V4L2_MPEG_VIDEO_ENCODING_MPEG_4_AVC = 2,
};
#define V4L2_CID_MPEG_VIDEO_ASPECT              (V4L2_CID_MPEG_BASE+201)
enum v4l2_mpeg_video_aspect {
        V4L2_MPEG_VIDEO_ASPECT_1x1     = 0,
        V4L2_MPEG_VIDEO_ASPECT_4x3     = 1,
        V4L2_MPEG_VIDEO_ASPECT_16x9    = 2,
        V4L2_MPEG_VIDEO_ASPECT_221x100 = 3,
};
#define V4L2_CID_MPEG_VIDEO_B_FRAMES            (V4L2_CID_MPEG_BASE+202)
#define V4L2_CID_MPEG_VIDEO_GOP_SIZE            (V4L2_CID_MPEG_BASE+203)
#define V4L2_CID_MPEG_VIDEO_GOP_CLOSURE         (V4L2_CID_MPEG_BASE+204)
#define V4L2_CID_MPEG_VIDEO_PULLDOWN            (V4L2_CID_MPEG_BASE+205)
#define V4L2_CID_MPEG_VIDEO_BITRATE_MODE        (V4L2_CID_MPEG_BASE+206)
enum v4l2_mpeg_video_bitrate_mode {
        V4L2_MPEG_VIDEO_BITRATE_MODE_VBR = 0,
        V4L2_MPEG_VIDEO_BITRATE_MODE_CBR = 1,
};
#define V4L2_CID_MPEG_VIDEO_BITRATE             (V4L2_CID_MPEG_BASE+207)
#define V4L2_CID_MPEG_VIDEO_BITRATE_PEAK        (V4L2_CID_MPEG_BASE+208)
#define V4L2_CID_MPEG_VIDEO_TEMPORAL_DECIMATION (V4L2_CID_MPEG_BASE+209)
#define V4L2_CID_MPEG_VIDEO_MUTE                (V4L2_CID_MPEG_BASE+210)
#define V4L2_CID_MPEG_VIDEO_MUTE_YUV            (V4L2_CID_MPEG_BASE+211)

/*  MPEG-class control IDs specific to the CX2341x driver as defined by V4
#define V4L2_CID_MPEG_CX2341X_BASE                              (V4L2_CTRI
#define V4L2_CID_MPEG_CX2341X_VIDEO_SPATIAL_FILTER_MODE         (V4L2_CID_
enum v4l2_mpeg_cx2341x_video_spatial_filter_mode {
        V4L2_MPEG_CX2341X_VIDEO_SPATIAL_FILTER_MODE_MANUAL = 0,
        V4L2_MPEG_CX2341X_VIDEO_SPATIAL_FILTER_MODE_AUTO   = 1,
};
#define V4L2_CID_MPEG_CX2341X_VIDEO_SPATIAL_FILTER              (V4L2_CID_
#define V4L2_CID_MPEG_CX2341X_VIDEO_LUMA_SPATIAL_FILTER_TYPE    (V4L2_CID_
enum v4l2_mpeg_cx2341x_video_luma_spatial_filter_type {
        V4L2_MPEG_CX2341X_VIDEO_LUMA_SPATIAL_FILTER_TYPE_OFF
        V4L2_MPEG_CX2341X_VIDEO_LUMA_SPATIAL_FILTER_TYPE_1D_HOR
        V4L2_MPEG_CX2341X_VIDEO_LUMA_SPATIAL_FILTER_TYPE_1D_VERT
        V4L2_MPEG_CX2341X_VIDEO_LUMA_SPATIAL_FILTER_TYPE_2D_HV_SEPARABLE
        V4L2_MPEG_CX2341X_VIDEO_LUMA_SPATIAL_FILTER_TYPE_2D_SYM_NON_SEPARA
};
#define V4L2_CID_MPEG_CX2341X_VIDEO_CHROMA_SPATIAL_FILTER_TYPE  (V4L2_CID_
enum v4l2_mpeg_cx2341x_video_chroma_spatial_filter_type {
        V4L2_MPEG_CX2341X_VIDEO_CHROMA_SPATIAL_FILTER_TYPE_OFF    = 0,
        V4L2_MPEG_CX2341X_VIDEO_CHROMA_SPATIAL_FILTER_TYPE_1D_HOR = 1,
};
#define V4L2_CID_MPEG_CX2341X_VIDEO_TEMPORAL_FILTER_MODE        (V4L2_CID_
enum v4l2_mpeg_cx2341x_video_temporal_filter_mode {
        V4L2_MPEG_CX2341X_VIDEO_TEMPORAL_FILTER_MODE_MANUAL = 0,
```

```
        V4L2_MPEG_CX2341X_VIDEO_TEMPORAL_FILTER_MODE_AUTO   = 1,
};
#define V4L2_CID_MPEG_CX2341X_VIDEO_TEMPORAL_FILTER             (V4L2_CID_
#define V4L2_CID_MPEG_CX2341X_VIDEO_MEDIAN_FILTER_TYPE          (V4L2_CID_
enum v4l2_mpeg_cx2341x_video_median_filter_type {
        V4L2_MPEG_CX2341X_VIDEO_MEDIAN_FILTER_TYPE_OFF      = 0,
        V4L2_MPEG_CX2341X_VIDEO_MEDIAN_FILTER_TYPE_HOR      = 1,
        V4L2_MPEG_CX2341X_VIDEO_MEDIAN_FILTER_TYPE_VERT     = 2,
        V4L2_MPEG_CX2341X_VIDEO_MEDIAN_FILTER_TYPE_HOR_VERT = 3,
        V4L2_MPEG_CX2341X_VIDEO_MEDIAN_FILTER_TYPE_DIAG     = 4,
};
#define V4L2_CID_MPEG_CX2341X_VIDEO_LUMA_MEDIAN_FILTER_BOTTOM   (V4L2_CID_
#define V4L2_CID_MPEG_CX2341X_VIDEO_LUMA_MEDIAN_FILTER_TOP      (V4L2_CID_
#define V4L2_CID_MPEG_CX2341X_VIDEO_CHROMA_MEDIAN_FILTER_BOTTOM (V4L2_CID_
#define V4L2_CID_MPEG_CX2341X_VIDEO_CHROMA_MEDIAN_FILTER_TOP    (V4L2_CID_
#define V4L2_CID_MPEG_CX2341X_STREAM_INSERT_NAV_PACKETS         (V4L2_CID_

/*  Camera class control IDs */
#define V4L2_CID_CAMERA_CLASS_BASE      (V4L2_CTRL_CLASS_CAMERA | 0x900)
#define V4L2_CID_CAMERA_CLASS           (V4L2_CTRL_CLASS_CAMERA | 1)

#define V4L2_CID_EXPOSURE_AUTO                  (V4L2_CID_CAMERA_CLASS_BAS
enum  v4l2_exposure_auto_type {
        V4L2_EXPOSURE_AUTO = 0,
        V4L2_EXPOSURE_MANUAL = 1,
        V4L2_EXPOSURE_SHUTTER_PRIORITY = 2,
        V4L2_EXPOSURE_APERTURE_PRIORITY = 3
};
#define V4L2_CID_EXPOSURE_ABSOLUTE              (V4L2_CID_CAMERA_CLASS_BAS
#define V4L2_CID_EXPOSURE_AUTO_PRIORITY         (V4L2_CID_CAMERA_CLASS_BAS

#define V4L2_CID_PAN_RELATIVE                   (V4L2_CID_CAMERA_CLASS_BAS
#define V4L2_CID_TILT_RELATIVE                  (V4L2_CID_CAMERA_CLASS_BAS
#define V4L2_CID_PAN_RESET                      (V4L2_CID_CAMERA_CLASS_BAS
#define V4L2_CID_TILT_RESET                     (V4L2_CID_CAMERA_CLASS_BAS

#define V4L2_CID_PAN_ABSOLUTE                   (V4L2_CID_CAMERA_CLASS_BAS
#define V4L2_CID_TILT_ABSOLUTE                  (V4L2_CID_CAMERA_CLASS_BAS

#define V4L2_CID_FOCUS_ABSOLUTE                 (V4L2_CID_CAMERA_CLASS_BAS
#define V4L2_CID_FOCUS_RELATIVE                 (V4L2_CID_CAMERA_CLASS_BAS
#define V4L2_CID_FOCUS_AUTO                     (V4L2_CID_CAMERA_CLASS_BAS

#define V4L2_CID_ZOOM_ABSOLUTE                  (V4L2_CID_CAMERA_CLASS_BAS
#define V4L2_CID_ZOOM_RELATIVE                  (V4L2_CID_CAMERA_CLASS_BAS
#define V4L2_CID_ZOOM_CONTINUOUS                (V4L2_CID_CAMERA_CLASS_BAS

#define V4L2_CID_PRIVACY                        (V4L2_CID_CAMERA_CLASS_BAS
```

```
#define V4L2_CID_IRIS_ABSOLUTE                  (V4L2_CID_CAMERA_CLASS_BAS
#define V4L2_CID_IRIS_RELATIVE                  (V4L2_CID_CAMERA_CLASS_BAS

/* FM Modulator class control IDs */
#define V4L2_CID_FM_TX_CLASS_BASE               (V4L2_CTRL_CLASS_FM_TX | 
#define V4L2_CID_FM_TX_CLASS                    (V4L2_CTRL_CLASS_FM_TX | 

#define V4L2_CID_RDS_TX_DEVIATION               (V4L2_CID_FM_TX_CLASS_BASE
#define V4L2_CID_RDS_TX_PI                      (V4L2_CID_FM_TX_CLASS_BASE
#define V4L2_CID_RDS_TX_PTY                     (V4L2_CID_FM_TX_CLASS_BASE
#define V4L2_CID_RDS_TX_PS_NAME                 (V4L2_CID_FM_TX_CLASS_BASE
#define V4L2_CID_RDS_TX_RADIO_TEXT              (V4L2_CID_FM_TX_CLASS_BASE

#define V4L2_CID_AUDIO_LIMITER_ENABLED          (V4L2_CID_FM_TX_CLASS_BASE
#define V4L2_CID_AUDIO_LIMITER_RELEASE_TIME     (V4L2_CID_FM_TX_CLASS_BASE
#define V4L2_CID_AUDIO_LIMITER_DEVIATION        (V4L2_CID_FM_TX_CLASS_BASE

#define V4L2_CID_AUDIO_COMPRESSION_ENABLED      (V4L2_CID_FM_TX_CLASS_BASE
#define V4L2_CID_AUDIO_COMPRESSION_GAIN         (V4L2_CID_FM_TX_CLASS_BASE
#define V4L2_CID_AUDIO_COMPRESSION_THRESHOLD    (V4L2_CID_FM_TX_CLASS_BASE
#define V4L2_CID_AUDIO_COMPRESSION_ATTACK_TIME  (V4L2_CID_FM_TX_CLASS_BASE
#define V4L2_CID_AUDIO_COMPRESSION_RELEASE_TIME (V4L2_CID_FM_TX_CLASS_BASE

#define V4L2_CID_PILOT_TONE_ENABLED             (V4L2_CID_FM_TX_CLASS_BASE
#define V4L2_CID_PILOT_TONE_DEVIATION           (V4L2_CID_FM_TX_CLASS_BASE
#define V4L2_CID_PILOT_TONE_FREQUENCY           (V4L2_CID_FM_TX_CLASS_BASE

#define V4L2_CID_TUNE_PREEMPHASIS               (V4L2_CID_FM_TX_CLASS_BASE
enum v4l2_preemphasis {
        V4L2_PREEMPHASIS_DISABLED       = 0,
        V4L2_PREEMPHASIS_50_uS          = 1,
        V4L2_PREEMPHASIS_75_uS          = 2,
};
#define V4L2_CID_TUNE_POWER_LEVEL               (V4L2_CID_FM_TX_CLASS_BASE
#define V4L2_CID_TUNE_ANTENNA_CAPACITOR         (V4L2_CID_FM_TX_CLASS_BASE

/*
 *      T U N I N G
 */
struct v4l2_tuner {
        __u32                   index;
        __u8                    name[32];
        enum v4l2_tuner_type    type;
        __u32                   capability;
        __u32                   rangelow;
        __u32                   rangehigh;
        __u32                   rxsubchans;
```

```
        __u32                   audmode;
        __s32                   signal;
        __s32                   afc;
        __u32                   reserved[4];
};

struct v4l2_modulator {
        __u32                   index;
        __u8                    name[32];
        __u32                   capability;
        __u32                   rangelow;
        __u32                   rangehigh;
        __u32                   txsubchans;
        __u32                   reserved[4];
};

/*  Flags for the 'capability' field */
#define V4L2_TUNER_CAP_LOW              0x0001
#define V4L2_TUNER_CAP_NORM             0x0002
#define V4L2_TUNER_CAP_STEREO           0x0010
#define V4L2_TUNER_CAP_LANG2            0x0020
#define V4L2_TUNER_CAP_SAP              0x0020
#define V4L2_TUNER_CAP_LANG1            0x0040
#define V4L2_TUNER_CAP_RDS              0x0080
#define V4L2_TUNER_CAP_RDS_BLOCK_IO     0x0100
#define V4L2_TUNER_CAP_RDS_CONTROLS     0x0200

/*  Flags for the 'rxsubchans' field */
#define V4L2_TUNER_SUB_MONO             0x0001
#define V4L2_TUNER_SUB_STEREO           0x0002
#define V4L2_TUNER_SUB_LANG2            0x0004
#define V4L2_TUNER_SUB_SAP              0x0004
#define V4L2_TUNER_SUB_LANG1            0x0008
#define V4L2_TUNER_SUB_RDS              0x0010

/*  Values for the 'audmode' field */
#define V4L2_TUNER_MODE_MONO            0x0000
#define V4L2_TUNER_MODE_STEREO          0x0001
#define V4L2_TUNER_MODE_LANG2           0x0002
#define V4L2_TUNER_MODE_SAP             0x0002
#define V4L2_TUNER_MODE_LANG1           0x0003
#define V4L2_TUNER_MODE_LANG1_LANG2     0x0004

struct v4l2_frequency {
        __u32                   tuner;
        enum v4l2_tuner_type    type;
        __u32                   frequency;
        __u32                   reserved[8];
```

```
};

struct v4l2_hw_freq_seek {
        __u32                   tuner;
        enum v4l2_tuner_type    type;
        __u32                   seek_upward;
        __u32                   wrap_around;
        __u32                   spacing;
        __u32                   reserved[7];
};

/*
 *      R D S
 */

struct v4l2_rds_data {
        __u8    lsb;
        __u8    msb;
        __u8    block;
} __attribute__ ((packed));

#define V4L2_RDS_BLOCK_MSK       0x7
#define V4L2_RDS_BLOCK_A         0
#define V4L2_RDS_BLOCK_B         1
#define V4L2_RDS_BLOCK_C         2
#define V4L2_RDS_BLOCK_D         3
#define V4L2_RDS_BLOCK_C_ALT     4
#define V4L2_RDS_BLOCK_INVALID   7

#define V4L2_RDS_BLOCK_CORRECTED 0x40
#define V4L2_RDS_BLOCK_ERROR     0x80

/*
 *      A U D I O
 */
struct v4l2_audio {
        __u32   index;
        __u8    name[32];
        __u32   capability;
        __u32   mode;
        __u32   reserved[2];
};

/*  Flags for the 'capability' field */
#define V4L2_AUDCAP_STEREO              0x00001
#define V4L2_AUDCAP_AVL                 0x00002

/*  Flags for the 'mode' field */
```

```
#define V4L2_AUDMODE_AVL                0x00001

struct v4l2_audioout {
        __u32   index;
        __u8    name[32];
        __u32   capability;
        __u32   mode;
        __u32   reserved[2];
};

/*
 *      M P E G   S E R V I C E S
 *
 *      NOTE: EXPERIMENTAL API
 */
#if 1
#define V4L2_ENC_IDX_FRAME_I    (0)
#define V4L2_ENC_IDX_FRAME_P    (1)
#define V4L2_ENC_IDX_FRAME_B    (2)
#define V4L2_ENC_IDX_FRAME_MASK (0xf)

struct v4l2_enc_idx_entry {
        __u64 offset;
        __u64 pts;
        __u32 length;
        __u32 flags;
        __u32 reserved[2];
};

#define V4L2_ENC_IDX_ENTRIES (64)
struct v4l2_enc_idx {
        __u32 entries;
        __u32 entries_cap;
        __u32 reserved[4];
        struct v4l2_enc_idx_entry entry[V4L2_ENC_IDX_ENTRIES];
};


#define V4L2_ENC_CMD_START      (0)
#define V4L2_ENC_CMD_STOP       (1)
#define V4L2_ENC_CMD_PAUSE      (2)
#define V4L2_ENC_CMD_RESUME     (3)

/* Flags for V4L2_ENC_CMD_STOP */
#define V4L2_ENC_CMD_STOP_AT_GOP_END    (1 << 0)

struct v4l2_encoder_cmd {
        __u32 cmd;
```

```
        __u32 flags;
        union {
                struct {
                        __u32 data[8];
                } raw;
        };
};

#endif


/*
 *      D A T A   S E R V I C E S   ( V B I )
 *
 *      Data services API by Michael Schimek
 */

/* Raw VBI */
struct v4l2_vbi_format {
        __u32   sampling_rate;          /* in 1 Hz */
        __u32   offset;
        __u32   samples_per_line;
        __u32   sample_format;          /* V4L2_PIX_FMT_* */
        __s32   start[2];
        __u32   count[2];
        __u32   flags;                  /* V4L2_VBI_* */
        __u32   reserved[2];            /* must be zero */
};

/*  VBI flags  */
#define V4L2_VBI_UNSYNC         (1 << 0)
#define V4L2_VBI_INTERLACED     (1 << 1)

/* Sliced VBI
 *
 *    This implements is a proposal V4L2 API to allow SLICED VBI
 * required for some hardware encoders. It should change without
 * notice in the definitive implementation.
 */

struct v4l2_sliced_vbi_format {
        __u16   service_set;
        /* service_lines[0][...] specifies lines 0-23 (1-23 used) of the
           service_lines[1][...] specifies lines 0-23 (1-23 used) of the
                                 (equals frame lines 313-336 for 625 line
                                  standards, 263-286 for 525 line standar
        __u16   service_lines[2][24];
        __u32   io_size;
```

```
        __u32   reserved[2];            /* must be zero */
};

/* Teletext World System Teletext
   (WST), defined on ITU-R BT.653-2 */
#define V4L2_SLICED_TELETEXT_B          (0x0001)
/* Video Program System, defined on ETS 300 231*/
#define V4L2_SLICED_VPS                 (0x0400)
/* Closed Caption, defined on EIA-608 */
#define V4L2_SLICED_CAPTION_525         (0x1000)
/* Wide Screen System, defined on ITU-R BT1119.1 */
#define V4L2_SLICED_WSS_625             (0x4000)

#define V4L2_SLICED_VBI_525             (V4L2_SLICED_CAPTION_525)
#define V4L2_SLICED_VBI_625             (V4L2_SLICED_TELETEXT_B | V4L2_SL

struct v4l2_sliced_vbi_cap {
        __u16   service_set;
        /* service_lines[0][...] specifies lines 0-23 (1-23 used) of the
           service_lines[1][...] specifies lines 0-23 (1-23 used) of the
                                 (equals frame lines 313-336 for 625 line
                                  standards, 263-286 for 525 line standar
        __u16   service_lines[2][24];
        enum v4l2_buf_type type;
        __u32   reserved[3];    /* must be 0 */
};

struct v4l2_sliced_vbi_data {
        __u32   id;
        __u32   field;          /* 0: first field, 1: second field */
        __u32   line;           /* 1-23 */
        __u32   reserved;       /* must be 0 */
        __u8    data[48];
};

/*
 * Sliced VBI data inserted into MPEG Streams
 */

/*
 * V4L2_MPEG_STREAM_VBI_FMT_IVTV:
 *
 * Structure of payload contained in an MPEG 2 Private Stream 1 PES Packet
 * MPEG-2 Program Pack that contains V4L2_MPEG_STREAM_VBI_FMT_IVTV Sliced
 * data
 *
 * Note, the MPEG-2 Program Pack and Private Stream 1 PES packet header
 * definitions are not included here.  See the MPEG-2 specifications for
```

```
 * on these headers.
 */

/* Line type IDs */
#define V4L2_MPEG_VBI_IVTV_TELETEXT_B     (1)
#define V4L2_MPEG_VBI_IVTV_CAPTION_525    (4)
#define V4L2_MPEG_VBI_IVTV_WSS_625        (5)
#define V4L2_MPEG_VBI_IVTV_VPS            (7)

struct v4l2_mpeg_vbi_itv0_line {
        __u8 id;        /* One of V4L2_MPEG_VBI_IVTV_* above */
        __u8 data[42];  /* Sliced VBI data for the line */
} __attribute__ ((packed));

struct v4l2_mpeg_vbi_itv0 {
        __le32 linemask[2]; /* Bitmasks of VBI service lines present */
        struct v4l2_mpeg_vbi_itv0_line line[35];
} __attribute__ ((packed));

struct v4l2_mpeg_vbi_ITV0 {
        struct v4l2_mpeg_vbi_itv0_line line[36];
} __attribute__ ((packed));

#define V4L2_MPEG_VBI_IVTV_MAGIC0       "itv0"
#define V4L2_MPEG_VBI_IVTV_MAGIC1       "ITV0"

struct v4l2_mpeg_vbi_fmt_ivtv {
        __u8 magic[4];
        union {
                struct v4l2_mpeg_vbi_itv0 itv0;
                struct v4l2_mpeg_vbi_ITV0 ITV0;
        };
} __attribute__ ((packed));

/*
 *      A G G R E G A T E   S T R U C T U R E S
 */

/**
 * struct v4l2_plane_pix_format - additional, per-plane format definition
 * @sizeimage:          maximum size in bytes required for data, for which
 *                      this plane will be used
 * @bytesperline:       distance in bytes between the leftmost pixels in t
 *                      adjacent lines
 */
struct v4l2_plane_pix_format {
        __u32           sizeimage;
        __u16           bytesperline;
```

```
        __u16           reserved[7];
} __attribute__ ((packed));

/**
 * struct v4l2_pix_format_mplane - multiplanar format definition
 * @width:              image width in pixels
 * @height:             image height in pixels
 * @pixelformat:        little endian four character code (fourcc)
 * @field:              field order (for interlaced video)
 * @colorspace:         supplemental to pixelformat
 * @plane_fmt:          per-plane information
 * @num_planes:         number of planes for this format
 */
struct v4l2_pix_format_mplane {
        __u32                           width;
        __u32                           height;
        __u32                           pixelformat;
        enum v4l2_field                 field;
        enum v4l2_colorspace            colorspace;

        struct v4l2_plane_pix_format    plane_fmt[VIDEO_MAX_PLANES];
        __u8                            num_planes;
        __u8                            reserved[11];
} __attribute__ ((packed));

/**
 * struct v4l2_format - stream data format
 * @type:       type of the data stream
 * @pix:        definition of an image format
 * @pix_mp:     definition of a multiplanar image format
 * @win:        definition of an overlaid image
 * @vbi:        raw VBI capture or output parameters
 * @sliced:     sliced VBI capture or output parameters
 * @raw_data:   placeholder for future extensions and custom formats
 */
struct v4l2_format {
        enum v4l2_buf_type type;
        union {
                struct v4l2_pix_format          pix;     /* V4L2_BUF_TYPE_
                struct v4l2_pix_format_mplane   pix_mp;  /* V4L2_BUF_TYPE_
                struct v4l2_window              win;     /* V4L2_BUF_TYPE_
                struct v4l2_vbi_format          vbi;     /* V4L2_BUF_TYPE_
                struct v4l2_sliced_vbi_format   sliced;  /* V4L2_BUF_TYPE_
                __u8    raw_data[200];                   /* user-defined
        } fmt;
};

/*      Stream type-dependent parameters
```

```
 */
struct v4l2_streamparm {
        enum v4l2_buf_type type;
        union {
                struct v4l2_captureparm capture;
                struct v4l2_outputparm  output;
                __u8    raw_data[200];  /* user-defined */
        } parm;
};

/*
 *      E V E N T S
 */

#define V4L2_EVENT_ALL                          0
#define V4L2_EVENT_VSYNC                        1
#define V4L2_EVENT_EOS                          2
#define V4L2_EVENT_PRIVATE_START                0x08000000

/* Payload for V4L2_EVENT_VSYNC */
struct v4l2_event_vsync {
        /* Can be V4L2_FIELD_ANY, _NONE, _TOP or _BOTTOM */
        __u8 field;
} __attribute__ ((packed));

struct v4l2_event {
        __u32                           type;
        union {
                struct v4l2_event_vsync vsync;
                __u8                    data[64];
        } u;
        __u32                           pending;
        __u32                           sequence;
        struct timespec                 timestamp;
        __u32                           reserved[9];
};

struct v4l2_event_subscription {
        __u32                           type;
        __u32                           reserved[7];
};

/*
 *      A D V A N C E D   D E B U G G I N G
 *
 *      NOTE: EXPERIMENTAL API, NEVER RELY ON THIS IN APPLICATIONS!
 *      FOR DEBUGGING, TESTING AND INTERNAL USE ONLY!
 */
```

```
/* VIDIOC_DBG_G_REGISTER and VIDIOC_DBG_S_REGISTER */

#define V4L2_CHIP_MATCH_HOST       0  /* Match against chip ID on host (0
#define V4L2_CHIP_MATCH_I2C_DRIVER 1  /* Match against I2C driver name */
#define V4L2_CHIP_MATCH_I2C_ADDR   2  /* Match against I2C 7-bit address
#define V4L2_CHIP_MATCH_AC97       3  /* Match against anciliary AC97 chip

struct v4l2_dbg_match {
        __u32 type; /* Match type */
        union {     /* Match this chip, meaning determined by type */
                __u32 addr;
                char name[32];
        };
} __attribute__ ((packed));

struct v4l2_dbg_register {
        struct v4l2_dbg_match match;
        __u32 size;     /* register size in bytes */
        __u64 reg;
        __u64 val;
} __attribute__ ((packed));

/* VIDIOC_DBG_G_CHIP_IDENT */
struct v4l2_dbg_chip_ident {
        struct v4l2_dbg_match match;
        __u32 ident;       /* chip identifier as specified in <media/v4l2-
        __u32 revision;    /* chip revision, chip specific */
} __attribute__ ((packed));

/*
 *      I O C T L   C O D E S   F O R   V I D E O   D E V I C E S
 *
 */
#define VIDIOC_QUERYCAP          _IOR('V',  0, struct v4l2_capability)
#define VIDIOC_RESERVED          _IO('V',  1)
#define VIDIOC_ENUM_FMT         _IOWR('V',  2, struct v4l2_fmtdesc)
#define VIDIOC_G_FMT            _IOWR('V',  4, struct v4l2_format)
#define VIDIOC_S_FMT            _IOWR('V',  5, struct v4l2_format)
#define VIDIOC_REQBUFS          _IOWR('V',  8, struct v4l2_requestbuffers)
#define VIDIOC_QUERYBUF         _IOWR('V',  9, struct v4l2_buffer)
#define VIDIOC_G_FBUF            _IOR('V', 10, struct v4l2_framebuffer)
#define VIDIOC_S_FBUF            _IOW('V', 11, struct v4l2_framebuffer)
#define VIDIOC_OVERLAY           _IOW('V', 14, int)
#define VIDIOC_QBUF             _IOWR('V', 15, struct v4l2_buffer)
#define VIDIOC_DQBUF            _IOWR('V', 17, struct v4l2_buffer)
#define VIDIOC_STREAMON          _IOW('V', 18, int)
#define VIDIOC_STREAMOFF         _IOW('V', 19, int)
```

```
#define VIDIOC_G_PARM            _IOWR('V', 21, struct v4l2_streamparm)
#define VIDIOC_S_PARM            _IOWR('V', 22, struct v4l2_streamparm)
#define VIDIOC_G_STD              _IOR('V', 23, v4l2_std_id)
#define VIDIOC_S_STD              _IOW('V', 24, v4l2_std_id)
#define VIDIOC_ENUMSTD           _IOWR('V', 25, struct v4l2_standard)
#define VIDIOC_ENUMINPUT         _IOWR('V', 26, struct v4l2_input)
#define VIDIOC_G_CTRL            _IOWR('V', 27, struct v4l2_control)
#define VIDIOC_S_CTRL            _IOWR('V', 28, struct v4l2_control)
#define VIDIOC_G_TUNER           _IOWR('V', 29, struct v4l2_tuner)
#define VIDIOC_S_TUNER            _IOW('V', 30, struct v4l2_tuner)
#define VIDIOC_G_AUDIO            _IOR('V', 33, struct v4l2_audio)
#define VIDIOC_S_AUDIO            _IOW('V', 34, struct v4l2_audio)
#define VIDIOC_QUERYCTRL         _IOWR('V', 36, struct v4l2_queryctrl)
#define VIDIOC_QUERYMENU         _IOWR('V', 37, struct v4l2_querymenu)
#define VIDIOC_G_INPUT            _IOR('V', 38, int)
#define VIDIOC_S_INPUT           _IOWR('V', 39, int)
#define VIDIOC_G_OUTPUT           _IOR('V', 46, int)
#define VIDIOC_S_OUTPUT          _IOWR('V', 47, int)
#define VIDIOC_ENUMOUTPUT        _IOWR('V', 48, struct v4l2_output)
#define VIDIOC_G_AUDOUT           _IOR('V', 49, struct v4l2_audioout)
#define VIDIOC_S_AUDOUT           _IOW('V', 50, struct v4l2_audioout)
#define VIDIOC_G_MODULATOR       _IOWR('V', 54, struct v4l2_modulator)
#define VIDIOC_S_MODULATOR        _IOW('V', 55, struct v4l2_modulator)
#define VIDIOC_G_FREQUENCY       _IOWR('V', 56, struct v4l2_frequency)
#define VIDIOC_S_FREQUENCY        _IOW('V', 57, struct v4l2_frequency)
#define VIDIOC_CROPCAP           _IOWR('V', 58, struct v4l2_cropcap)
#define VIDIOC_G_CROP            _IOWR('V', 59, struct v4l2_crop)
#define VIDIOC_S_CROP             _IOW('V', 60, struct v4l2_crop)
#define VIDIOC_G_JPEGCOMP         _IOR('V', 61, struct v4l2_jpegcompressio
#define VIDIOC_S_JPEGCOMP         _IOW('V', 62, struct v4l2_jpegcompressio
#define VIDIOC_QUERYSTD           _IOR('V', 63, v4l2_std_id)
#define VIDIOC_TRY_FMT           _IOWR('V', 64, struct v4l2_format)
#define VIDIOC_ENUMAUDIO         _IOWR('V', 65, struct v4l2_audio)
#define VIDIOC_ENUMAUDOUT        _IOWR('V', 66, struct v4l2_audioout)
#define VIDIOC_G_PRIORITY         _IOR('V', 67, enum v4l2_priority)
#define VIDIOC_S_PRIORITY         _IOW('V', 68, enum v4l2_priority)
#define VIDIOC_G_SLICED_VBI_CAP  _IOWR('V', 69, struct v4l2_sliced_vbi_cap
#define VIDIOC_LOG_STATUS          _IO('V', 70)
#define VIDIOC_G_EXT_CTRLS       _IOWR('V', 71, struct v4l2_ext_controls)
#define VIDIOC_S_EXT_CTRLS       _IOWR('V', 72, struct v4l2_ext_controls)
#define VIDIOC_TRY_EXT_CTRLS     _IOWR('V', 73, struct v4l2_ext_controls)
#if 1
#define VIDIOC_ENUM_FRAMESIZES   _IOWR('V', 74, struct v4l2_frmsizeenum)
#define VIDIOC_ENUM_FRAMEINTERVALS _IOWR('V', 75, struct v4l2_frmivalenum
#define VIDIOC_G_ENC_INDEX        _IOR('V', 76, struct v4l2_enc_idx)
#define VIDIOC_ENCODER_CMD       _IOWR('V', 77, struct v4l2_encoder_cmd)
#define VIDIOC_TRY_ENCODER_CMD   _IOWR('V', 78, struct v4l2_encoder_cmd)
#endif
```

```
#if 1
/* Experimental, meant for debugging, testing and internal use.
   Only implemented if CONFIG_VIDEO_ADV_DEBUG is defined.
   You must be root to use these ioctls. Never use these in applications!
#define VIDIOC_DBG_S_REGISTER    _IOW('V', 79, struct v4l2_dbg_register)
#define VIDIOC_DBG_G_REGISTER    _IOWR('V', 80, struct v4l2_dbg_register)

/* Experimental, meant for debugging, testing and internal use.
   Never use this ioctl in applications! */
#define VIDIOC_DBG_G_CHIP_IDENT _IOWR('V', 81, struct v4l2_dbg_chip_ident)
#endif

#define VIDIOC_S_HW_FREQ_SEEK    _IOW('V', 82, struct v4l2_hw_freq_seek)
#define VIDIOC_ENUM_DV_PRESETS  _IOWR('V', 83, struct v4l2_dv_enum_preset)
#define VIDIOC_S_DV_PRESET      _IOWR('V', 84, struct v4l2_dv_preset)
#define VIDIOC_G_DV_PRESET      _IOWR('V', 85, struct v4l2_dv_preset)
#define VIDIOC_QUERY_DV_PRESET  _IOR('V',  86, struct v4l2_dv_preset)
#define VIDIOC_S_DV_TIMINGS     _IOWR('V', 87, struct v4l2_dv_timings)
#define VIDIOC_G_DV_TIMINGS     _IOWR('V', 88, struct v4l2_dv_timings)
#define VIDIOC_DQEVENT           _IOR('V', 89, struct v4l2_event)
#define VIDIOC_SUBSCRIBE_EVENT   _IOW('V', 90, struct v4l2_event_subscript
#define VIDIOC_UNSUBSCRIBE_EVENT _IOW('V', 91, struct v4l2_event_subscript

/* Reminder: when adding new ioctls please add support for them to
   drivers/media/video/v4l2-compat-ioctl32.c as well! */

#define BASE_VIDIOC_PRIVATE     192             /* 192-255 are private */

#endif /* __LINUX_VIDEODEV2_H */
```

# Appendix C. Video Capture Example

```
/*
 *  V4L2 video capture example
 *
 *  This program can be used and distributed without restrictions.
 *
 *      This program is provided with the V4L2 API
 * see http://linuxtv.org/docs.php for more information
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>

#include <getopt.h>             /* getopt_long() */

#include <fcntl.h>              /* low-level i/o */
#include <unistd.h>
#include <errno.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/time.h>
#include <sys/mman.h>
#include <sys/ioctl.h>

#include <linux/videodev2.h>

#define CLEAR(x) memset(&(x), 0, sizeof(x))

enum io_method {
        IO_METHOD_READ,
        IO_METHOD_MMAP,
        IO_METHOD_USERPTR,
};

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

static char             *dev_name;
static enum io_method   io = IO_METHOD_MMAP;
```

```
static int              fd = -1;
struct buffer          *buffers;
static unsigned int     n_buffers;
static int              out_buf;
static int              force_format;
static int              frame_count = 70;

static void errno_exit(const char *s)
{
        fprintf(stderr, "%s error %d, %s\n", s, errno, strerror(errno));
        exit(EXIT_FAILURE);
}

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

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

        return r;
}

static void process_image(const void *p, int size)
{
        if (out_buf)
                fwrite(p, size, 1, stdout);

        fflush(stderr);
        fprintf(stderr, ".");
        fflush(stdout);
}

static int read_frame(void)
{
        struct v4l2_buffer buf;
        unsigned int i;

        switch (io) {
        case IO_METHOD_READ:
                if (-1 == read(fd, buffers[0].start, buffers[0].length))
                        switch (errno) {
                        case EAGAIN:
                                return 0;

                        case EIO:
                                /* Could ignore EIO, see spec. */
```

```
                        /* fall through */

                default:
                        errno_exit("read");
                }
        }

        process_image(buffers[0].start, buffers[0].length);
        break;

case IO_METHOD_MMAP:
        CLEAR(buf);

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

        if (-1 == xioctl(fd, VIDIOC_DQBUF, &buf)) {
                switch (errno) {
                case EAGAIN:
                        return 0;

                case EIO:
                        /* Could ignore EIO, see spec. */

                        /* fall through */

                default:
                        errno_exit("VIDIOC_DQBUF");
                }
        }

        assert(buf.index < n_buffers);

        process_image(buffers[buf.index].start, buf.bytesused);

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

case IO_METHOD_USERPTR:
        CLEAR(buf);

        buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
        buf.memory = V4L2_MEMORY_USERPTR;

        if (-1 == xioctl(fd, VIDIOC_DQBUF, &buf)) {
                switch (errno) {
```

```
                                case EAGAIN:
                                        return 0;

                                case EIO:
                                        /* Could ignore EIO, see spec. */

                                        /* fall through */

                                default:
                                        errno_exit("VIDIOC_DQBUF");
                                }
                        }

                        for (i = 0; i < n_buffers; ++i)
                                if (buf.m.userptr == (unsigned long)buffers[i].sta
                                    && buf.length == buffers[i].length)
                                        break;

                        assert(i < n_buffers);

                        process_image((void *)buf.m.userptr, buf.bytesused);

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

                return 1;
        }

        static void mainloop(void)
        {
                unsigned int count;

                count = frame_count;

                while (count-- > 0) {
                        for (;;) {
                                fd_set fds;
                                struct timeval tv;
                                int r;

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

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

```
                        r = select(fd + 1, &fds, NULL, NULL, &tv);

                        if (-1 == r) {
                                if (EINTR == errno)
                                        continue;
                                errno_exit("select");
                        }

                        if (0 == r) {
                                fprintf(stderr, "select timeout\n");
                                exit(EXIT_FAILURE);
                        }

                        if (read_frame())
                                break;
                        /* EAGAIN - continue select loop. */
                }
        }
}

static void stop_capturing(void)
{
        enum v4l2_buf_type type;

        switch (io) {
        case IO_METHOD_READ:
                /* Nothing to do. */
                break;

        case IO_METHOD_MMAP:
        case IO_METHOD_USERPTR:
                type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
                if (-1 == xioctl(fd, VIDIOC_STREAMOFF, &type))
                        errno_exit("VIDIOC_STREAMOFF");
                break;
        }
}

static void start_capturing(void)
{
        unsigned int i;
        enum v4l2_buf_type type;

        switch (io) {
        case IO_METHOD_READ:
                /* Nothing to do. */
                break;
```

```
                case IO_METHOD_MMAP:
                        for (i = 0; i < n_buffers; ++i) {
                                struct v4l2_buffer buf;

                                CLEAR(buf);
                                buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
                                buf.memory = V4L2_MEMORY_MMAP;
                                buf.index = i;

                                if (-1 == xioctl(fd, VIDIOC_QBUF, &buf))
                                        errno_exit("VIDIOC_QBUF");
                        }
                        type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
                        if (-1 == xioctl(fd, VIDIOC_STREAMON, &type))
                                errno_exit("VIDIOC_STREAMON");
                        break;

                case IO_METHOD_USERPTR:
                        for (i = 0; i < n_buffers; ++i) {
                                struct v4l2_buffer buf;

                                CLEAR(buf);
                                buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
                                buf.memory = V4L2_MEMORY_USERPTR;
                                buf.index = i;
                                buf.m.userptr = (unsigned long)buffers[i].start;
                                buf.length = buffers[i].length;

                                if (-1 == xioctl(fd, VIDIOC_QBUF, &buf))
                                        errno_exit("VIDIOC_QBUF");
                        }
                        type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
                        if (-1 == xioctl(fd, VIDIOC_STREAMON, &type))
                                errno_exit("VIDIOC_STREAMON");
                        break;
                }
        }

        static void uninit_device(void)
        {
                unsigned int i;

                switch (io) {
                case IO_METHOD_READ:
                        free(buffers[0].start);
                        break;
```

```
        case IO_METHOD_MMAP:
                for (i = 0; i < n_buffers; ++i)
                        if (-1 == munmap(buffers[i].start, buffers[i].leng
                                errno_exit("munmap");
                break;

        case IO_METHOD_USERPTR:
                for (i = 0; i < n_buffers; ++i)
                        free(buffers[i].start);
                break;
        }

        free(buffers);
}

static void init_read(unsigned int buffer_size)
{
        buffers = calloc(1, sizeof(*buffers));

        if (!buffers) {
                fprintf(stderr, "Out of memory\n");
                exit(EXIT_FAILURE);
        }

        buffers[0].length = buffer_size;
        buffers[0].start = malloc(buffer_size);

        if (!buffers[0].start) {
                fprintf(stderr, "Out of memory\n");
                exit(EXIT_FAILURE);
        }
}

static void init_mmap(void)
{
        struct v4l2_requestbuffers req;

        CLEAR(req);

        req.count = 4;
        req.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
        req.memory = V4L2_MEMORY_MMAP;

        if (-1 == xioctl(fd, VIDIOC_REQBUFS, &req)) {
                if (EINVAL == errno) {
                        fprintf(stderr, "%s does not support "
                                 "memory mapping\n", dev_name);
                        exit(EXIT_FAILURE);
```

```
                } else {
                        errno_exit("VIDIOC_REQBUFS");
                }
        }

        if (req.count < 2) {
                fprintf(stderr, "Insufficient buffer memory on %s\n",
                         dev_name);
                exit(EXIT_FAILURE);
        }

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

        if (!buffers) {
                fprintf(stderr, "Out of memory\n");
                exit(EXIT_FAILURE);
        }

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

                CLEAR(buf);

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

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

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

                if (MAP_FAILED == buffers[n_buffers].start)
                        errno_exit("mmap");
        }
}

static void init_userp(unsigned int buffer_size)
{
        struct v4l2_requestbuffers req;

        CLEAR(req);
```

```
        req.count  = 4;
        req.type   = V4L2_BUF_TYPE_VIDEO_CAPTURE;
        req.memory = V4L2_MEMORY_USERPTR;

        if (-1 == xioctl(fd, VIDIOC_REQBUFS, &req)) {
                if (EINVAL == errno) {
                        fprintf(stderr, "%s does not support "
                                 "user pointer i/o\n", dev_name);
                        exit(EXIT_FAILURE);
                } else {
                        errno_exit("VIDIOC_REQBUFS");
                }
        }

        buffers = calloc(4, sizeof(*buffers));

        if (!buffers) {
                fprintf(stderr, "Out of memory\n");
                exit(EXIT_FAILURE);
        }

        for (n_buffers = 0; n_buffers < 4; ++n_buffers) {
                buffers[n_buffers].length = buffer_size;
                buffers[n_buffers].start = malloc(buffer_size);

                if (!buffers[n_buffers].start) {
                        fprintf(stderr, "Out of memory\n");
                        exit(EXIT_FAILURE);
                }
        }
}

static void init_device(void)
{
        struct v4l2_capability cap;
        struct v4l2_cropcap cropcap;
        struct v4l2_crop crop;
        struct v4l2_format fmt;
        unsigned int min;

        if (-1 == xioctl(fd, VIDIOC_QUERYCAP, &cap)) {
                if (EINVAL == errno) {
                        fprintf(stderr, "%s is no V4L2 device\n",
                                 dev_name);
                        exit(EXIT_FAILURE);
                } else {
                        errno_exit("VIDIOC_QUERYCAP");
```

```
                }
        }

        if (!(cap.capabilities & V4L2_CAP_VIDEO_CAPTURE)) {
                fprintf(stderr, "%s is no video capture device\n",
                         dev_name);
                exit(EXIT_FAILURE);
        }

        switch (io) {
        case IO_METHOD_READ:
                if (!(cap.capabilities & V4L2_CAP_READWRITE)) {
                        fprintf(stderr, "%s does not support read i/o\n",
                                 dev_name);
                        exit(EXIT_FAILURE);
                }
                break;

        case IO_METHOD_MMAP:
        case IO_METHOD_USERPTR:
                if (!(cap.capabilities & V4L2_CAP_STREAMING)) {
                        fprintf(stderr, "%s does not support streaming i/
                                 dev_name);
                        exit(EXIT_FAILURE);
                }
                break;
        }


        /* Select video input, video standard and tune here. */


        CLEAR(cropcap);

        cropcap.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;

        if (0 == xioctl(fd, VIDIOC_CROPCAP, &cropcap)) {
                crop.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
                crop.c = cropcap.defrect; /* reset to default */

                if (-1 == xioctl(fd, VIDIOC_S_CROP, &crop)) {
                        switch (errno) {
                        case EINVAL:
                                /* Cropping not supported. */
                                break;
                        default:
                                /* Errors ignored. */
                                break;
```

```
                        }
                }
        } else {
                /* Errors ignored. */
        }


        CLEAR(fmt);

        fmt.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
        if (force_format) {
                fmt.fmt.pix.width       = 640;
                fmt.fmt.pix.height      = 480;
                fmt.fmt.pix.pixelformat = V4L2_PIX_FMT_YUYV;
                fmt.fmt.pix.field       = V4L2_FIELD_INTERLACED;

                if (-1 == xioctl(fd, VIDIOC_S_FMT, &fmt))
                        errno_exit("VIDIOC_S_FMT");

                /* Note VIDIOC_S_FMT may change width and height. */
        } else {
                /* Preserve original settings as set by v4l2-ctl for examp
                if (-1 == xioctl(fd, VIDIOC_G_FMT, &fmt))
                        errno_exit("VIDIOC_G_FMT");
        }

        /* Buggy driver paranoia. */
        min = fmt.fmt.pix.width * 2;
        if (fmt.fmt.pix.bytesperline < min)
                fmt.fmt.pix.bytesperline = min;
        min = fmt.fmt.pix.bytesperline * fmt.fmt.pix.height;
        if (fmt.fmt.pix.sizeimage < min)
                fmt.fmt.pix.sizeimage = min;

        switch (io) {
        case IO_METHOD_READ:
                init_read(fmt.fmt.pix.sizeimage);
                break;

        case IO_METHOD_MMAP:
                init_mmap();
                break;

        case IO_METHOD_USERPTR:
                init_userp(fmt.fmt.pix.sizeimage);
                break;
        }
}
```

```
static void close_device(void)
{
        if (-1 == close(fd))
                errno_exit("close");

        fd = -1;
}

static void open_device(void)
{
        struct stat st;

        if (-1 == stat(dev_name, &st)) {
                fprintf(stderr, "Cannot identify '%s': %d, %s\n",
                         dev_name, errno, strerror(errno));
                exit(EXIT_FAILURE);
        }

        if (!S_ISCHR(st.st_mode)) {
                fprintf(stderr, "%s is no device\n", dev_name);
                exit(EXIT_FAILURE);
        }

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

        if (-1 == fd) {
                fprintf(stderr, "Cannot open '%s': %d, %s\n",
                         dev_name, errno, strerror(errno));
                exit(EXIT_FAILURE);
        }
}

static void usage(FILE *fp, int argc, char **argv)
{
        fprintf(fp,
                 "Usage: %s [options]\n\n"
                 "Version 1.3\n"
                 "Options:\n"
                 "-d | --device name   Video device name [%s]\n"
                 "-h | --help          Print this message\n"
                 "-m | --mmap          Use memory mapped buffers [default]
                 "-r | --read          Use read() calls\n"
                 "-u | --userp         Use application allocated buffers\n
                 "-o | --output        Outputs stream to stdout\n"
                 "-f | --format        Force format to 640x480 YUYV\n"
                 "-c | --count         Number of frames to grab [%i]\n"
                 "",
```

```
                argv[0], dev_name, frame_count);
}

static const char short_options[] = "d:hmruofc:";

static const struct option
long_options[] = {
        { "device", required_argument, NULL, 'd' },
        { "help",   no_argument,       NULL, 'h' },
        { "mmap",   no_argument,       NULL, 'm' },
        { "read",   no_argument,       NULL, 'r' },
        { "userp",  no_argument,       NULL, 'u' },
        { "output", no_argument,       NULL, 'o' },
        { "format", no_argument,       NULL, 'f' },
        { "count",  required_argument, NULL, 'c' },
        { 0, 0, 0, 0 }
};

int main(int argc, char **argv)
{
        dev_name = "/dev/video0";

        for (;;) {
                int idx;
                int c;

                c = getopt_long(argc, argv,
                                short_options, long_options, &idx);

                if (-1 == c)
                        break;

                switch (c) {
                case 0: /* getopt_long() flag */
                        break;

                case 'd':
                        dev_name = optarg;
                        break;

                case 'h':
                        usage(stdout, argc, argv);
                        exit(EXIT_SUCCESS);

                case 'm':
                        io = IO_METHOD_MMAP;
                        break;
```

```
                        case 'r':
                                io = IO_METHOD_READ;
                                break;

                        case 'u':
                                io = IO_METHOD_USERPTR;
                                break;

                        case 'o':
                                out_buf++;
                                break;

                        case 'f':
                                force_format++;
                                break;

                        case 'c':
                                errno = 0;
                                frame_count = strtol(optarg, NULL, 0);
                                if (errno)
                                        errno_exit(optarg);
                                break;

                        default:
                                usage(stderr, argc, argv);
                                exit(EXIT_FAILURE);
                        }
                }

                open_device();
                init_device();
                start_capturing();
                mainloop();
                stop_capturing();
                uninit_device();
                close_device();
                fprintf(stderr, "\n");
                return 0;
        }
```

# Appendix D. Video Grabber example using libv4l

This program demonstrates how to grab V4L2 images in ppm format by using libv4l handlers. The advantage is that this grabber can potentially work with any V4L2 driver.

```
/* V4L2 video picture grabber
   Copyright (C) 2009 Mauro Carvalho Chehab <mchehab@infradead.org>

   This program is free software; you can redistribute it and/or modify
   it under the terms of the GNU General Public License as published by
   the Free Software Foundation version 2 of the License.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/time.h>
#include <sys/mman.h>
#include <linux/videodev2.h>
#include "../libv4l/include/libv4l2.h"

#define CLEAR(x) memset(&(x), 0, sizeof(x))

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

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

```
        do {
                r = v4l2_ioctl(fh, request, arg);
        } while (r == -1 && ((errno == EINTR) || (errno == EAGAIN)));

        if (r == -1) {
                fprintf(stderr, "error %d, %s\n", errno, strerror(errno));
                exit(EXIT_FAILURE);
        }
}

int main(int argc, char **argv)
{
        struct v4l2_format              fmt;
        struct v4l2_buffer              buf;
        struct v4l2_requestbuffers      req;
        enum v4l2_buf_type              type;
        fd_set                          fds;
        struct timeval                  tv;
        int                             r, fd = -1;
        unsigned int                    i, n_buffers;
        char                            *dev_name = "/dev/video0";
        char                            out_name[256];
        FILE                            *fout;
        struct buffer                   *buffers;

        fd = v4l2_open(dev_name, O_RDWR | O_NONBLOCK, 0);
        if (fd < 0) {
                perror("Cannot open device");
                exit(EXIT_FAILURE);
        }

        CLEAR(fmt);
        fmt.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
        fmt.fmt.pix.width       = 640;
        fmt.fmt.pix.height      = 480;
        fmt.fmt.pix.pixelformat = V4L2_PIX_FMT_RGB24;
        fmt.fmt.pix.field       = V4L2_FIELD_INTERLACED;
        xioctl(fd, VIDIOC_S_FMT, &fmt);
        if (fmt.fmt.pix.pixelformat != V4L2_PIX_FMT_RGB24) {
                printf("Libv4l didn't accept RGB24 format. Can't proceed.\
                exit(EXIT_FAILURE);
        }
        if ((fmt.fmt.pix.width != 640) || (fmt.fmt.pix.height != 480))
                printf("Warning: driver is sending image at %dx%d\n",
                        fmt.fmt.pix.width, fmt.fmt.pix.height);

        CLEAR(req);
```

```
req.count = 2;
req.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
req.memory = V4L2_MEMORY_MMAP;
xioctl(fd, VIDIOC_REQBUFS, &req);

buffers = calloc(req.count, sizeof(*buffers));
for (n_buffers = 0; n_buffers < req.count; ++n_buffers) {
        CLEAR(buf);

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

        xioctl(fd, VIDIOC_QUERYBUF, &buf);

        buffers[n_buffers].length = buf.length;
        buffers[n_buffers].start = v4l2_mmap(NULL, buf.length,
                      PROT_READ | PROT_WRITE, MAP_SHARED,
                      fd, buf.m.offset);

        if (MAP_FAILED == buffers[n_buffers].start) {
                perror("mmap");
                exit(EXIT_FAILURE);
        }
}

for (i = 0; i < n_buffers; ++i) {
        CLEAR(buf);
        buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
        buf.memory = V4L2_MEMORY_MMAP;
        buf.index = i;
        xioctl(fd, VIDIOC_QBUF, &buf);
}
type = V4L2_BUF_TYPE_VIDEO_CAPTURE;

xioctl(fd, VIDIOC_STREAMON, &type);
for (i = 0; i < 20; i++) {
        do {
                FD_ZERO(&fds);
                FD_SET(fd, &fds);

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

                r = select(fd + 1, &fds, NULL, NULL, &tv);
        } while ((r == -1 && (errno = EINTR)));
        if (r == -1) {
```

```
                        perror("select");
                        return errno;
                }

                CLEAR(buf);
                buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
                buf.memory = V4L2_MEMORY_MMAP;
                xioctl(fd, VIDIOC_DQBUF, &buf);

                sprintf(out_name, "out%03d.ppm", i);
                fout = fopen(out_name, "w");
                if (!fout) {
                        perror("Cannot open image");
                        exit(EXIT_FAILURE);
                }
                fprintf(fout, "P6\n%d %d 255\n",
                        fmt.fmt.pix.width, fmt.fmt.pix.height);
                fwrite(buffers[buf.index].start, buf.bytesused, 1, fout);
                fclose(fout);

                xioctl(fd, VIDIOC_QBUF, &buf);
        }

        type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
        xioctl(fd, VIDIOC_STREAMOFF, &type);
        for (i = 0; i < n_buffers; ++i)
                v4l2_munmap(buffers[i].start, buffers[i].length);
        v4l2_close(fd);

        return 0;
}
```

# List of Types

struct v4l2_standard
struct v4l2_streamparm
struct v4l2_timecode
struct v4l2_tuner
struct v4l2_vbi_format
struct v4l2_window

# References


[EIA 608-B] Electronic Industries Alliance (http://www.eia.org), *EIA 608-B "Recommended Practice for Line 21 Data Service".*

[EN 300 294] European Telecommunication Standards Institute (http://www.etsi.org), *EN 300 294 "625-line television Wide Screen Signalling (WSS)".*

[ETS 300 231] European Telecommunication Standards Institute (http://www.etsi.org), *ETS 300 231 "Specification of the domestic video Programme Delivery Control system (PDC)".*

[ETS 300 706] European Telecommunication Standards Institute (http://www.etsi.org), *ETS 300 706 "Enhanced Teletext specification".*

[ISO 13818-1] International Telecommunication Union (http://www.itu.ch), International Organisation for Standardisation (http://www.iso.ch), *ITU-T Rec. H.222.0 | ISO/IEC 13818-1 "Information technology — Generic coding of moving pictures and associated audio information: Systems".*

[ISO 13818-2] International Telecommunication Union (http://www.itu.ch), International Organisation for Standardisation (http://www.iso.ch), *ITU-T Rec. H.262 | ISO/IEC 13818-2 "Information technology — Generic coding of moving pictures and associated audio information: Video".*

[ITU BT.470] International Telecommunication Union (http://www.itu.ch), *ITU-R Recommendation BT.470-6 "Conventional Television Systems".*

[ITU BT.601] International Telecommunication Union (http://www.itu.ch), *ITU-R Recommendation BT.601-5 "Studio Encoding Parameters of Digital Television for Standard 4:3 and Wide-Screen 16:9 Aspect Ratios".*

[ITU BT.653] International Telecommunication Union (http://www.itu.ch), *ITU-R Recommendation BT.653-3 "Teletext systems".*

[ITU BT.709] International Telecommunication Union (http://www.itu.ch), *ITU-R Recommendation BT.709-5 "Parameter values for the HDTV standards for production and international programme exchange".*

[ITU BT.1119] International Telecommunication Union (http://www.itu.ch), *ITU-R Recommendation BT.1119 "625-line television Wide Screen Signalling (WSS)".*

[JFIF] Independent JPEG Group (http://www.ijg.org), *JPEG File Interchange Format: Version 1.02.*

[SMPTE 12M] Society of Motion Picture and Television Engineers (http://www.smpte.org), *SMPTE*

[SMPTE 12M] Society of Motion Picture and Television Engineers (http://www.smpte.org), *SMPTE 12M-1999 "Television, Audio and Film - Time and Control Code"*.

[SMPTE 170M] Society of Motion Picture and Television Engineers (http://www.smpte.org), *SMPTE 170M-1999 "Television - Composite Analog Video Signal - NTSC for Studio Applications"*.

[SMPTE 240M] Society of Motion Picture and Television Engineers (http://www.smpte.org), *SMPTE 240M-1999 "Television - Signal Parameters - 1125-Line High-Definition Production"*.

[EN 50067] European Committee for Electrotechnical Standardization (http://www.cenelec.eu), *Specification of the radio data system (RDS) for VHF/FM sound broadcasting in the frequency range from 87,5 to 108,0 MHz.*

[NRSC-4] National Radio Systems Committee (http://www.nrscstandards.org), *NTSC-4: United States RBDS Standard.*

# II. LINUX DVB API

## Version 5.2

## Table of Contents

Ralph J. K. Metzler and Marcus O. C. MetzlerMauro Carvalho Chehab


**Revision History**

Revision 2.0.4 2011-05-06 Revised by: mcc
Add more information about DVB APIv5, better describing the frontend GET/SET props ioctl's.
Revision 2.0.3 2010-07-03 Revised by: mcc
Add some frontend capabilities flags, present on kernel, but missing at the specs.
Revision 2.0.2 2009-10-25 Revised by: mcc
documents FE_SET_FRONTEND_TUNE_MODE and FE_DISHETWORK_SEND_LEGACY_CI
Revision 2.0.1 2009-09-16 Revised by: mcc
Added ISDB-T test originally written by Patrick Boettcher
Revision 2.0.0 2009-09-06 Revised by: mcc
Conversion from LaTex to DocBook XML. The contents is the same as the original LaTex version.
Revision 1.0.0 2003-07-24 Revised by: rjkm
Initial revision on LaTEX.

# Chapter 8. Introduction

## 8.1. What you need to know

The reader of this document is required to have some knowledge in the area of digital video broadcasting (DVB) and should be familiar with part I of the MPEG2 specification ISO/IEC 13818 (aka ITU-T H.222), i.e you should know what a program/transport stream (PS/TS) is and what is meant by a packetized elementary stream (PES) or an I-frame.

Various DVB standards documents are available from http://www.dvb.org and/or http://www.etsi.org.

It is also necessary to know how to access unix/linux devices and how to use ioctl calls. This also includes the knowledge of C or C++.

## 8.2. History

The first API for DVB cards we used at Convergence in late 1999 was an extension of the Video4Linux API which was primarily developed for frame grabber cards. As such it was not really well suited to be used for DVB cards and their new

- MPEG2 audio and MPEG2 video decoders
- features like picture-in-picture, simultaneous decoding of several MPEG streams and filtering several section and PES data streams at the same time.

The main targets of the demultiplexer are the MPEG2 audio and video decoders. After decoding they pass on the uncompressed audio and video to the computer screen or (through a PAL/NTSC encoder) to a TV set.

Figure 8-1 shows a crude schematic of the control and data flow between those components.

On a DVB PCI card not all of these have to be present since some functionality can be provided by the main CPU of the PC (e.g. MPEG picture and sound decoding) or is not needed (e.g. for data-only uses like "Internet over satellite"). Also not every card or STB provides conditional access hardware.

# 8.3. Overview

# 8.4. Linux DVB Devices

**Figure 8-1. Components of a DVB card/STB**

The Linux DVB API lets you control these hardware components through currently six Unix-style character devices for video, audio, frontend, demux, CA and IP-over-DVB networking. The video and audio devices control the MPEG2 decoder hardware, the frontend device the tuner and the DVB demodulator. The demux device gives you control over the PES and section filters of the hardware. If the hardware does not support filtering these filters can be implemented in software. Finally, the CA device controls all the conditional access capabilities of the hardware. It can depend on the individual security requirements of the platform, if and how many of the CA functions are made available to the application through this device.

All devices can be found in the */dev* tree under */dev/dvb*. The individual devices are called:

- */dev/dvb/adapterN/audioM*,
- */dev/dvb/adapterN/videoM*,
- */dev/dvb/adapterN/frontendM*,
- */dev/dvb/adapterN/netM*,
- */dev/dvb/adapterN/demuxM*,
- */dev/dvb/adapterN/caM*

where *N* enumerates the DVB PCI cards in a system starting from 0, and *M* enumerates the devices of each type within each adapter, starting from 0, too. We will omit the */dev/dvb/adapterN/* in the further discussion of these devices. The naming scheme for the devices is the same wheter devfs is used or not.

- Frontend consisting of tuner and DVB demodulator

  Here the raw signal reaches the DVB hardware from a satellite dish or antenna or directly from cable. The frontend down-converts and demodulates this signal into an MPEG transport stream (TS). In case of a satellite frontend, this includes a facility for satellite equipment control (SEC), which allows control of LNB switches or dish rotors.

- Conditional Access (CA) hardware like CI adapters and smartcard slots

  The complete TS is passed through the CA hardware. Programs to which the user has access (controlled by the smart card) are decoded in real time and re-inserted into the TS.

- Demultiplexer which filters the incoming DVB stream

  The demultiplexer splits the TS into its components like audio, video and data streams. Besides usually several of each type with video, it also contains data streams with information about the programmes offered in the current TS.

More details about the data structures and function calls of all the devices are described in the following chapters.

## 8.5. API include files

For each of the DVB devices a corresponding include file exists. The DVB API include files should be included in application sources with a partial path like:

```
#include <linux/dvb/frontend.h>
```

To enable applications to support different API version, an additional include file *linux/dvb/version.h* exists, which defines the constant *DVB_API_VERSION*. This document describes *DVB_API_VERSION 3*.

# Chapter 9. DVB Frontend API

The DVB frontend device controls the tuner and DVB demodulator hardware. It can be accessed through */dev/dvb/adapter0/frontend0*. Data types and and ioctl definitions can be accessed by including *linux/dvb/frontend.h* in your application.

DVB frontends come in three varieties: DVB-S (satellite), DVB-C (cable) and DVB-T (terrestrial). Transmission via the internet (DVB-IP) is not yet handled by this API but a future extension is possible. For DVB-S the frontend device also supports satellite equipment control (SEC) via DiSEqC and V-SEC protocols. The DiSEqC (digital SEC) specification is available from Eutelsat (http://www.eutelsat.com/satellites/4_5_5.html).

Note that the DVB API may also be used for MPEG decoder-only PCI cards, in which case there exists no frontend device.

## 9.1. Frontend Data Types

### 9.1.1. frontend type

For historical reasons frontend types are named after the type of modulation used in transmission.

```
typedef enum fe_type {
FE_QPSK,    /* DVB-S */
FE_QAM,     /* DVB-C */
FE_OFDM     /* DVB-T */
} fe_type_t;
```

### 9.1.2. frontend capabilities

Capabilities describe what a frontend can do. Some capabilities can only be supported for a specific frontend type.

```
typedef enum fe_caps {
FE_IS_STUPID                  = 0,
FE_CAN_INVERSION_AUTO         = 0x1,
FE_CAN_FEC_1_2                = 0x2,
FE_CAN_FEC_2_3                = 0x4,
```

### 9.1.4. diseqc master command

A message sent from the frontend to DiSEqC capable equipment.

```
struct dvb_diseqc_master_cmd {
uint8_t msg[6]; /*  { framing, address, command, data[3] } */
uint8_t msg_len; /*  valid values are 3...6  */
};
```

### 9.1.5. diseqc slave reply

A reply to the frontend from DiSEqC 2.0 capable equipment.

```
struct dvb_diseqc_slave_reply {
uint8_t msg[4]; /*  { framing, data[3] } */
uint8_t msg_len; /*  valid values are 0...4, 0 means no msg  */
int     timeout; /*  return from ioctl after timeout ms with */
};                       /*  errorcode when no message was received  */
```

### 9.1.6. diseqc slave reply

The voltage is usually used with non-DiSEqC capable LNBs to switch the polarzation (horizontal/vertical). When using DiSEqC epuipment this voltage has to be switched consistently to the DiSEqC commands as described in the DiSEqC spec.

```
typedef enum fe_sec_voltage {
SEC_VOLTAGE_13,
SEC_VOLTAGE_18
} fe_sec_voltage_t;
```

### 9.1.3. frontend information

Information about the frontend ca be queried with FE_GET_INFO.

```
struct dvb_frontend_info {
char       name[128];
fe_type_t  type;
uint32_t   frequency_min;
uint32_t   frequency_max;
uint32_t   frequency_stepsize;
uint32_t   frequency_tolerance;
uint32_t   symbol_rate_min;
uint32_t   symbol_rate_max;
uint32_t   symbol_rate_tolerance;      /* ppm */
uint32_t   notifier_delay;             /* ms */
fe_caps_t  caps;
};
```

### 9.1.7. SEC continuous tone

The continuous 22KHz tone is usually used with non-DiSEqC capable LNBs to switch the high/low band of a dual-band LNB. When using DiSEqC epuipment this voltage has to be switched consistently to the DiSEqC commands as described in the DiSEqC spec.

```
typedef enum fe_sec_tone_mode {
SEC_TONE_ON,
SEC_TONE_OFF
} fe_sec_tone_mode_t;
```

## 9.1.8. SEC tone burst

For satellite QPSK frontends you have to use the `QPSKParameters` member defined by

The 22KHz tone burst is usually used with non-DiSEqC capable switches to select between two connected LNBs/satellites. When using DiSEqC epuipment this voltage has to be switched consistently to the DiSEqC commands as described in the DiSEqC spec.

```
struct dvb_qpsk_parameters {
    uint32_t        symbol_rate;  /* symbol rate in Symbols per second */
    fe_code_rate_t  fec_inner;    /* forward error correction (see above) */
};

typedef enum fe_sec_mini_cmd {
    SEC_MINI_A,
    SEC_MINI_B
} fe_sec_mini_cmd_t;
```

for cable QAM frontend you use the `QAMParameters` structure

```
struct dvb_qam_parameters {
    uint32_t         symbol_rate; /* symbol rate in Symbols per second */
    fe_code_rate_t   fec_inner;   /* forward error correction (see above) */
    fe_modulation_t  modulation;  /* modulation type (see above) */
};
```

## 9.1.9. frontend status

DVB-T frontends are supported by the `OFDMParamters` structure

Several functions of the frontend device use the fe_status data type defined by

```
struct dvb_ofdm_parameters {
typedef enum fe_status {
    fe_bandwidth_t      bandwidth;
    FE_HAS_SIGNAL     = 0x01,   /* found something above the noise level */
    fe_code_rate_t      code_rate_HP;  /* high priority stream code rate */
    FE_HAS_CARRIER    = 0x02,   /* found a DVB signal  */
    fe_code_rate_t      code_rate_LP;  /* low priority stream code rate */
    FE_HAS_VITERBI    = 0x04,   /* FEC is stable  */
    fe_modulation_t     constellation; /* modulation type (see above) */
    FE_HAS_SYNC       = 0x08,   /* found sync bytes  */
    fe_transmit_mode_t  transmission_mode;
    FE_HAS_LOCK       = 0x10,   /* everything's working... */
    fe_guard_interval_t guard_interval;
    FE_TIMEDOUT       = 0x20,   /* no lock within the last ~2 seconds */
    fe_hierarchy_t      hierarchy_information;
    FE_REINIT         = 0x40    /*  frontend was reinitialized,  */
};
} fe_status_t;                        /*  application is recommned to reset
```

In the case of QPSK frontends the `Frequency` field specifies the intermediate frequency, i.e. the offset which is effectively added to the local oscillator frequency (LOF) of the LNB. The intermediate frequency has to be specified in units of kHz. For QAM and OFDM frontends the Frequency specifies the absolute frequency and is given in Hz.

## 9.1.10. frontend parameters

The kind of parameters passed to the frontend device for tuning depend on the kind of hardware you are using. All kinds of parameters are combined as an union in the FrontendParameters structure:

```
typedef enum fe_spectral_inversion {
    INVERSION_OFF,
    INVERSION_ON,
struct dvb_frontend_parameters {
    INVERSION_AUTO
    uint32_t frequency;       /* (absolute) frequency in Hz for QAM/OFDM */
} fe_spectral_inversion_t;
      /* intermediate frequency in kHz for QPSK */

    fe_spectral_inversion_t inversion;
```

It indicates if spectral inversion should be presumed or not. In the automatic setting (`INVERSION_AUTO`) the hardware will try to figure out the correct setting by itself.

```
    union {
        struct dvb_qpsk_parameters qpsk;
        struct dvb_qam_parameters  qam;
        struct dvb_ofdm_parameters ofdm;
    };
typedef enum fe_code_rate {
};FEC_NONE = 0,
```

The possible values for the `FEC_inner` field are

```
 FEC_1_2,
 FEC_2_3,
 FEC_3_4,
 FEC_4_5,
 FEC_5_6,
 FEC_6_7,
 FEC_7_8,
 FEC_8_9,
 FEC_AUTO
} fe_code_rate_t;
```

which correspond to error correction rates of 1/2, 2/3, etc., no error correction or auto detection.

For cable and terrestrial frontends (QAM and OFDM) one also has to specify the quadrature modulation mode which can be one of the following:

```
typedef enum fe_modulation {
QPSK,
 QAM_16,
 QAM_32,
 QAM_64,
 QAM_128,
 QAM_256,
 QAM_AUTO
} fe_modulation_t;
```

Finally, there are several more parameters for OFDM:

```
typedef enum fe_transmit_mode {
 TRANSMISSION_MODE_2K,
 TRANSMISSION_MODE_8K,
 TRANSMISSION_MODE_AUTO
} fe_transmit_mode_t;

typedef enum fe_bandwidth {
 BANDWIDTH_8_MHZ,
 BANDWIDTH_7_MHZ,
 BANDWIDTH_6_MHZ,
 BANDWIDTH_AUTO
} fe_bandwidth_t;

typedef enum fe_guard_interval {
 GUARD_INTERVAL_1_32,
 GUARD_INTERVAL_1_16,
 GUARD_INTERVAL_1_8,
 GUARD_INTERVAL_1_4,
 GUARD_INTERVAL_AUTO
} fe_guard_interval_t;

typedef enum fe_hierarchy {
 HIERARCHY_NONE,
 HIERARCHY_1,
 HIERARCHY_2,
 HIERARCHY_4,
 HIERARCHY_AUTO
```

SYNOPSIS

| |
|---|
| int open(const char *deviceName, int flags); |

PARAMETERS

| | |
|---|---|
| const char *deviceName | Name of specific video device. |
| int flags | A bit-wise OR of the following flags: |
| | O_RDONLY read-only access |
| | O_RDWR read/write access |
| | O_NONBLOCK open in non-blocking mode |
| | (blocking mode is the default) |

ERRORS

| | |
|---|---|
| ENODEV | Device driver not loaded/available. |
| EINTERNAL | Internal error. |
| EBUSY | Device or resource busy. |
| EINVAL | Invalid argument. |

## 9.2.2. close()

DESCRIPTION

| |
|---|
| This system call closes a previously opened front-end device. After closing a front-end device, its corresponding hardware might be powered down automatically. |

SYNOPSIS

| |
|---|
| int close(int fd); |

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid open file descriptor. |

## 9.2.3. FE_READ_STATUS

DESCRIPTION

This ioctl call returns status information about the front-end. This call only requires read-only access to the device.

SYNOPSIS

int ioctl(int fd, int request = FE_READ_STATUS, fe_status_t ⋆status);

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| int request | Equals FE_READ_STATUS for this command. |
| struct fe_status_t *status | Points to the location where the front-end status word is to be stored. |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid open file descriptor. |
| EFAULT | status points to invalid address. |

## 9.2.4. FE_READ_BER

DESCRIPTION

This ioctl call returns the bit error rate for the signal currently received/demodulated by the front-end. For this command, read-only access to the device is sufficient.

SYNOPSIS

int ioctl(int fd, int request = FE_READ_BER, uint32_t ⋆ber);

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| int request | Equals FE_READ_BER for this command. |
| uint32_t *ber | The bit error rate is stored into *ber. |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid open file descriptor. |
| EFAULT | ber points to invalid address. |
| ENOSIGNAL | There is no signal, thus no meaningful bit error rate. Also returned if the front-end is not turned on. |
| ENOSYS | Function not available for this device. |

## 9.2.5. FE_READ_SNR

DESCRIPTION

This ioctl call returns the signal-to-noise ratio for the signal currently received by the front-end. For this command, read-only access to the device is sufficient.

SYNOPSIS

int ioctl(int fd, int request = FE_READ_SNR, int16_t *snr);

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| int request | Equals FE_READ_SNR for this command. |
| int16_t *snr | The signal-to-noise ratio is stored into *snr. |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid open file descriptor. |

| | |
|---|---|
| EFAULT | snr points to invalid address. |
| ENOSIGNAL | There is no signal, thus no meaningful signal strength value. Also returned if front-end is not turned on. |
| ENOSYS | Function not available for this device. |

## 9.2.6. FE_READ_SIGNAL_STRENGTH

DESCRIPTION

This ioctl call returns the signal strength value for the signal currently received by the front-end. For this command, read-only access to the device is sufficient.

SYNOPSIS

int ioctl( int fd, int request = FE_READ_SIGNAL_STRENGTH, int16_t *strength);

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| int request | Equals FE_READ_SIGNAL_STRENGTH for this command. |
| int16_t *strength | The signal strength value is stored into *strength. |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid open file descriptor. |
| EFAULT | status points to invalid address. |
| ENOSIGNAL | There is no signal, thus no meaningful signal strength value. Also returned if front-end is not turned on. |
| ENOSYS | Function not available for this device. |

## 9.2.7. FE_READ_UNCORRECTED_BLOCKS

DESCRIPTION

| |
|---|
| This ioctl call returns the number of uncorrected blocks detected by the device driver during its lifetime. For meaningful measurements, the increment in block count during a specific time interval should be calculated. For this command, read-only access to the device is sufficient. |
| Note that the counter will wrap to zero after its maximum count has been reached. |

SYNOPSIS

| |
|---|
| int ioctl( int fd, int request = FE_READ_UNCORRECTED_BLOCKS, uint32_t *ublocks); |

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| int request | Equals FE_READ_UNCORRECTED_BLOCKS for this command. |
| uint32_t *ublocks | The total number of uncorrected blocks seen by the driver so far. |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid open file descriptor. |
| EFAULT | ublocks points to invalid address. |
| ENOSYS | Function not available for this device. |

## 9.2.8. FE_SET_FRONTEND

DESCRIPTION

This ioctl call starts a tuning operation using specified parameters. The result of this call will be successful if the parameters were valid and the tuning could be initiated. The result of the tuning operation in itself, however, will arrive asynchronously as an event (see documentation for FE_GET_EVENT and FrontendEvent.) If a new FE_SET_FRONTEND operation is initiated before the previous one was completed, the previous operation will be aborted in favor of the new one. This command requires read/write access to the device.

SYNOPSIS

int ioctl(int fd, int request = FE_SET_FRONTEND, struct dvb_frontend_parameters *p);

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| int request | Equals FE_SET_FRONTEND for this command. |
| struct dvb_frontend_parameters *p | Points to parameters for tuning operation. |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid open file descriptor. |
| EFAULT | p points to invalid address. |
| EINVAL | Maximum supported symbol rate reached. |

## 9.2.9. FE_GET_FRONTEND

DESCRIPTION

This ioctl call queries the currently effective frontend parameters. For this command, read-only access to the device is sufficient.

SYNOPSIS

int ioctl(int fd, int request = FE_GET_FRONTEND, struct dvb_frontend_parameters *p);

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| int request | Equals FE_SET_FRONTEND for this command. |
| struct dvb_frontend_parameters *p | Points to parameters for tuning operation. |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid open file descriptor. |
| EFAULT | p points to invalid address. |
| EINVAL | Maximum supported symbol rate reached. |

## 9.2.10. FE_GET_EVENT

DESCRIPTION

| |
|---|
| This ioctl call returns a frontend event if available. If an event is not available, the behavior depends on whether the device is in blocking or non-blocking mode. In the latter case, the call fails immediately with errno set to EWOULDBLOCK. In the former case, the call blocks until an event becomes available. |
| The standard Linux poll() and/or select() system calls can be used with the device file descriptor to watch for new events. For select(), the file descriptor should be included in the exceptfds argument, and for poll(), POLLPRI should be specified as the wake-up condition. Since the event queue allocated is rather small (room for 8 events), the queue must be serviced regularly to avoid overflow. If an overflow happens, the oldest event is discarded from the queue, and an error (EOVERFLOW) occurs the next time the queue is read. After reporting the error condition in this fashion, subsequent FE_GET_EVENT calls will return events from the queue as usual. |
| For the sake of implementation simplicity, this command requires read/write access to the device. |

SYNOPSIS

| |
|---|
| int ioctl(int fd, int request = QPSK_GET_EVENT, struct dvb_frontend_event *ev); |

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| int request | Equals FE_GET_EVENT for this command. |
| struct dvb_frontend_event *ev | Points to the location where the event, |
| | if any, is to be stored. |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid open file descriptor. |
| EFAULT | ev points to invalid address. |
| EWOULDBLOCK | There is no event pending, and the device is in non-blocking mode. |
| EOVERFLOW | |
| | Overflow in event queue - one or more events were lost. |

## 9.2.11. FE_GET_INFO

DESCRIPTION

This ioctl call returns information about the front-end. This call only requires read-only access to the device.

SYNOPSIS

int ioctl(int fd, int request = FE_GET_INFO, struct dvb_frontend_info *info);

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| int request | Equals FE_GET_INFO for this command. |
| struct dvb_frontend_info *info | Points to the location where the front-end information is to be stored. |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid open file descriptor. |
| EFAULT | info points to invalid address. |

## 9.2.12. FE_DISEQC_RESET_OVERLOAD

DESCRIPTION

If the bus has been automatically powered off due to power overload, this ioctl call restores the power to the bus. The call requires read/write access to the device. This call has no effect if the device is manually powered off. Not all DVB adapters support this ioctl.

SYNOPSIS

int ioctl(int fd, int request = FE_DISEQC_RESET_OVERLOAD);

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| int request | Equals FE_DISEQC_RESET_OVERLOAD for this command. |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid file descriptor. |
| EPERM | Permission denied (needs read/write access). |
| EINTERNAL | Internal error in the device driver. |

## 9.2.13. FE_DISEQC_SEND_MASTER_CMD

DESCRIPTION

This ioctl call is used to send a a DiSEqC command.

SYNOPSIS

| int ioctl(int fd, int request = FE_DISEQC_SEND_MASTER_CMD, struct dvb_diseqc_master_cmd *cmd); |
|---|

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| int request | Equals FE_DISEQC_SEND_MASTER_CMD for this command. |
| struct dvb_diseqc_master_cmd *cmd | Pointer to the command to be transmitted. |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid file descriptor. |
| EFAULT | Seq points to an invalid address. |
| EINVAL | The data structure referred to by seq is invalid in some way. |
| EPERM | Permission denied (needs read/write access). |
| EINTERNAL | Internal error in the device driver. |

## 9.2.14. FE_DISEQC_RECV_SLAVE_REPLY

DESCRIPTION

| This ioctl call is used to receive reply to a DiSEqC 2.0 command. |
|---|

SYNOPSIS

| int ioctl(int fd, int request = FE_DISEQC_RECV_SLAVE_REPLY, struct dvb_diseqc_slave_reply *reply); |
|---|

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |

| | |
|---|---|
| int request | Equals FE_DISEQC_RECV_SLAVE_REPLY for this command. |
| struct dvb_diseqc_slave_reply *reply | Pointer to the command to be received. |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid file descriptor. |
| EFAULT | Seq points to an invalid address. |
| EINVAL | The data structure referred to by seq is invalid in some way. |
| EPERM | Permission denied (needs read/write access). |
| EINTERNAL | Internal error in the device driver. |

## 9.2.15. FE_DISEQC_SEND_BURST

DESCRIPTION

This ioctl call is used to send a 22KHz tone burst.

SYNOPSIS

int ioctl(int fd, int request = FE_DISEQC_SEND_BURST, fe_sec_mini_cmd_t burst);

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| int request | Equals FE_DISEQC_SEND_BURST for this command. |
| fe_sec_mini_cmd_t burst | burst A or B. |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid file descriptor. |
| EFAULT | Seq points to an invalid address. |

| | |
|---|---|
| EINVAL | The data structure referred to by seq is invalid in some way. |
| EPERM | Permission denied (needs read/write access). |
| EINTERNAL | Internal error in the device driver. |

## 9.2.16. FE_SET_TONE

DESCRIPTION

This call is used to set the generation of the continuous 22kHz tone. This call requires read/write permissions.

SYNOPSIS

int ioctl(int fd, int request = FE_SET_TONE, fe_sec_tone_mode_t tone);

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| int request | Equals FE_SET_TONE for this command. |
| fe_sec_tone_mode_t tone | The requested tone generation mode (on/off). |

ERRORS

| | |
|---|---|
| ENODEV | Device driver not loaded/available. |
| EBUSY | Device or resource busy. |
| EINVAL | Invalid argument. |
| EPERM | File not opened with read permissions. |
| EINTERNAL | Internal error in the device driver. |

## 9.2.17. FE_SET_VOLTAGE

DESCRIPTION

This call is used to set the bus voltage. This call requires read/write permissions.

SYNOPSIS

int ioctl(int fd, int request = FE_SET_VOLTAGE, fe_sec_voltage_t voltage);

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| int request | Equals FE_SET_VOLTAGE for this command. |
| fe_sec_voltage_t voltage | The requested bus voltage. |

ERRORS

| | |
|---|---|
| ENODEV | Device driver not loaded/available. |
| EBUSY | Device or resource busy. |
| EINVAL | Invalid argument. |
| EPERM | File not opened with read permissions. |
| EINTERNAL | Internal error in the device driver. |

## 9.2.18. FE_ENABLE_HIGH_LNB_VOLTAGE

DESCRIPTION

If high != 0 enables slightly higher voltages instead of 13/18V (to compensate for long cables). This call requires read/write permissions. Not all DVB adapters support this ioctl.

SYNOPSIS

int ioctl(int fd, int request = FE_ENABLE_HIGH_LNB_VOLTAGE, int high);

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |

| | |
|---|---|
| int request | Equals FE_SET_VOLTAGE for this command. |
| int high | The requested bus voltage. |

ERRORS

| | |
|---|---|
| ENODEV | Device driver not loaded/available. |
| EBUSY | Device or resource busy. |
| EINVAL | Invalid argument. |
| EPERM | File not opened with read permissions. |
| EINTERNAL | Internal error in the device driver. |

## 9.2.19. FE_SET_FRONTEND_TUNE_MODE

DESCRIPTION

Allow setting tuner mode flags to the frontend.

SYNOPSIS

int ioctl(int fd, int request = FE_SET_FRONTEND_TUNE_MODE, unsigned int flags);

PARAMETERS

| | |
|---|---|
| unsigned int flags | FE_TUNE_MODE_ONESHOT When set, this flag will disable any zigzagging or other "normal" tuning behaviour. Additionally, there will be no automatic monitoring of the lock status, and hence no frontend events will be generated. If a frontend device is closed, this flag will be automatically turned off when the device is reopened read-write. |

ERRORS

| | |
|---|---|
| EINVAL | Invalid argument. |

### 9.2.20. FE_DISHNETWORK_SEND_LEGACY_CMD

DESCRIPTION

WARNING: This is a very obscure legacy command, used only at stv0299 driver. Should not be used on newer drivers.
It provides a non-standard method for selecting Diseqc voltage on the frontend, for Dish Network legacy switches.

As support for this ioctl were added in 2004, this means that such dishes were already legacy in 2004.

SYNOPSIS

int ioctl(int fd, int request = FE_DISHNETWORK_SEND_LEGACY_CMD, unsigned long cmd);

PARAMETERS

| | |
|---|---|
| unsigned long cmd | sends the specified raw cmd to the dish via DISEqC. |

ERRORS

There are no errors in use for this call

## 9.3. FE_GET_PROPERTY/FE_SET_PROPERTY

```
/* Reserved fields should be set to 0 */
struct dtv_property {
 __u32 cmd;
 union {
  __u32 data;
  struct {
   __u8 data[32];
   __u32 len;
   __u32 reserved1[3];
```

```
    void *reserved2;
   } buffer;
  } u;
  int result;
} __attribute__ ((packed));

/* num of properties cannot exceed DTV_IOCTL_MAX_MSGS per ioctl */
#define DTV_IOCTL_MAX_MSGS 64

struct dtv_properties {
  __u32 num;
  struct dtv_property *props;
};
```

### 9.3.1. FE_GET_PROPERTY

DESCRIPTION

This ioctl call returns one or more frontend properties. This call only requires read-only access to the device.

SYNOPSIS

int ioctl(int fd, int request = FE_GET_PROPERTY, dtv_properties *props);

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| int num | Equals FE_GET_PROPERTY for this command. |
| struct dtv_property *props | Points to the location where the front-end property commands are stored. |

ERRORS

| | |
|---|---|
| EINVAL | Invalid parameter(s) received or number of parameters out of the range. |
| ENOMEM | Out of memory. |
| EFAULT | Failure while copying data from/to userspace. |
| EOPNOTSUPP | Property type not supported. |

## 9.3.2. FE_SET_PROPERTY

DESCRIPTION

This ioctl call sets one or more frontend properties. This call only requires read-only access to the device.

SYNOPSIS

int ioctl(int fd, int request = FE_SET_PROPERTY, dtv_properties *props);

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| int num | Equals FE_SET_PROPERTY for this command. |
| struct dtv_property *props | Points to the location where the front-end property commands are stored. |

ERRORS

| | |
|---|---|
| EINVAL | Invalid parameter(s) received or number of parameters out of the range. |
| ENOMEM | Out of memory. |
| EFAULT | Failure while copying data from/to userspace. |
| EOPNOTSUPP | Property type not supported. |

## 9.3.3. Property types

On FE_GET_PROPERTY/FE_SET_PROPERTY, the actual action is determined by the dtv_property cmd/data pairs. With one single ioctl, is possible to get/set up to 64 properties. The actual meaning of each property is described on the next sections.

The available frontend property types are:

```
#define DTV_UNDEFINED  0
#define DTV_TUNE  1
#define DTV_CLEAR  2
#define DTV_FREQUENCY  3
```

## 9.3.4. Parameters that are common to all Digital TV standards

```
#define DTV_UNDEFINED 0

#define DTV_TUNE 1

#define DTV_CLEAR 2

#define DTV_FREQUENCY 3

#define DTV_MODULATION 4

#define DTV_BANDWIDTH_HZ 5

#define DTV_INVERSION  6

#define DTV_DISEQC_MASTER 7

#define DTV_SYMBOL_RATE  8

#define DTV_INNER_FEC  9

#define DTV_VOLTAGE  10

#define DTV_TONE  11

#define DTV_PILOT  12

#define DTV_ROLLOFF  13

#define DTV_DISEQC_SLAVE_REPLY 14

#define DTV_FE_CAPABILITY_COUNT 15

#define DTV_FE_CAPABILITY 16

#define DTV_DELIVERY_SYSTEM 17

#define DTV_ISDBT_PARTIAL_RECEPTION 18

#define DTV_ISDBT_SOUND_BROADCASTING 19

#define DTV_ISDBT_SB_SUBCHANNEL_ID 20

#define DTV_ISDBT_SB_SEGMENT_IDX 21

#define DTV_ISDBT_SB_SEGMENT_COUNT 22

#define DTV_ISDBT_LAYERA_FEC  23

#define DTV_ISDBT_LAYERA_MODULATION  24

#define DTV_ISDBT_LAYERA_SEGMENT_COUNT  25

#define DTV_ISDBT_LAYERA_TIME_INTERLEAVING 26

#define DTV_ISDBT_LAYERB_FEC  27

#define DTV_ISDBT_LAYERB_MODULATION  28

#define DTV_ISDBT_LAYERB_SEGMENT_COUNT  29

#define DTV_ISDBT_LAYERB_TIME_INTERLEAVING 30

#define DTV_ISDBT_LAYERC_FEC  31

#define DTV_ISDBT_LAYERC_MODULATION  32

#define DTV_ISDBT_LAYERC_SEGMENT_COUNT  33

#define DTV_ISDBT_LAYERC_TIME_INTERLEAVING 34

#define DTV_API_VERSION  35

#define DTV_CODE_RATE_HP 36

#define DTV_CODE_RATE_LP 37

#define DTV_GUARD_INTERVAL  38

#define DTV_TRANSMISSION_MODE 39

#define DTV_HIERARCHY  40

#define DTV_ISDBT_LAYER_ENABLED 41

#define DTV_ISDBS_TS_ID  42

typedef enum fe_delivery_system {
 SYS_UNDEFINED,
 SYS_DVBC_ANNEX_AC,
```

### 9.3.4.1. DTV_FREQUENCY

Central frequency of the channel, in HZ.

Notes:

1) For ISDB-T, the channels are usually transmitted with an offset of 143kHz. E.g. a valid frequency could be 474143 kHz. The stepping is bound to the bandwidth of the channel which is 6MHz.

2)As in ISDB-Tsb the channel consists of only one or three segments the frequency step is 429kHz, 3*429 respectively. As for ISDB-T the central frequency of the channel is expected.

### 9.3.4.2. DTV_BANDWIDTH_HZ

Bandwidth for the channel, in HZ.

Possible values: 1712000, 5000000, 6000000, 7000000, 8000000, 10000000.

Notes:

1) For ISDB-T it should be always 6000000Hz (6MHz)

2) For ISDB-Tsb it can vary depending on the number of connected segments

3) Bandwidth doesn't apply for DVB-C transmissions, as the bandwidth for DVB-C depends on the symbol rate.

4) Bandwidth in ISDB-T is fixed (6MHz) or can be easily derived from other parameters (DTV_ISDBT_SB_SEGMENT_IDX, DTV_ISDBT_SB_SEGMENT_COUNT).

5) DVB-T supports 6, 7 and 8MHz.

6) In addition, DVB-T2 supports 1.172, 5 and 10MHz.

### 9.3.4.3. DTV_DELIVERY_SYSTEM

Specifies the type of Delivery system

Possible values:

### 9.3.4.5. DTV_GUARD_INTERVAL

Possible values are:

```
typedef enum fe_guard_interval {
	GUARD_INTERVAL_1_32,
	GUARD_INTERVAL_1_16,
	GUARD_INTERVAL_1_8,
	GUARD_INTERVAL_1_4,
	GUARD_INTERVAL_AUTO,
	GUARD_INTERVAL_1_128,
	GUARD_INTERVAL_19_128,
	GUARD_INTERVAL_19_256,
} fe_guard_interval_t;
```

Notes:

1) If DTV_GUARD_INTERVAL is set the GUARD_INTERVAL_AUTO the hardware will try to find the correct guard interval (if capable) and will use TMCC to fill in the missing parameters.

2) Intervals 1/128, 19/128 and 19/256 are used only for DVB-T2 at present

### 9.3.4.4. DTV_TRANSMISSION_MODE

Specifies the number of carriers used by the standard

Possible values are:

```
typedef enum fe_transmit_mode {
	TRANSMISSION_MODE_2K,
	TRANSMISSION_MODE_8K,
	TRANSMISSION_MODE_AUTO,
	TRANSMISSION_MODE_4K,
	TRANSMISSION_MODE_1K,
	TRANSMISSION_MODE_16K,
	TRANSMISSION_MODE_32K,
} fe_transmit_mode_t;
```

Notes:

1) ISDB-T supports three carrier/symbol-size: 8K, 4K, 2K. It is called 'mode' in the standard: Mode 1 is 2K, mode 2 is 4K, mode 3 is 8K

2) If DTV_TRANSMISSION_MODE is set the TRANSMISSION_MODE_AUTO the hardware will try to find the correct FFT-size (if capable) and will use TMCC to fill in the missing parameters.

3) DVB-T specifies 2K and 8K as valid sizes.

4) DVB-T2 specifies 1K, 2K, 4K, 8K, 16K and 32K.

## 9.3.5. ISDB-T frontend

This section describes shortly what are the possible parameters in the Linux DVB-API called "S2API" and now DVB API 5 in order to tune an ISDB-T/ISDB-Tsb demodulator:

This ISDB-T/ISDB-Tsb API extension should reflect all information needed to tune any ISDB-T/ISDB-Tsb hardware. Of course it is possible that some very sophisticated devices won't need certain parameters to tune.

The information given here should help application writers to know how to handle ISDB-T and ISDB-Tsb hardware using the Linux DVB-API.

The details given here about ISDB-T and ISDB-Tsb are just enough to basically show the dependencies between the needed parameter values, but surely some information is left out. For more detailed information see the following documents:

ARIB STD-B31 - "Transmission System for Digital Terrestrial Television Broadcasting" and

ARIB TR-B14 - "Operational Guidelines for Digital Terrestrial Television Broadcasting".

In order to read this document one has to have some knowledge the channel structure in ISDB-T and ISDB-Tsb. I.e. it has to be known to the reader that an

ISDB-T channels consist of 13 segments, that can be received by 3 layers consisting of several segments each. ISDB-Tsb channels consist of one or more segments, starting with a segment with a pilot and ending with a segment with a last segment indicator. Both are specified by providing the segment's sub-channel ID. A sub-channel is used to help the demodulator to synchronize on the channel.

Parameters used by ISDB-T and ISDB-Tsb.

An ISDB-T channel is always centered over all sub-channels. As for the example above, in ISDB-Tsb it is no longer as simple as that.

### 9.3.5.1. ISDB-T only parameters

The DTV_ISDBT_SB_SUBCHANNEL_ID parameter is used to give the sub-channel ID of the segment to be demodulated.

#### 9.3.5.1.1. DTV_ISDBT_PARTIAL_RECEPTION

If DTV_ISDBT_SOUND_BROADCASTING is '0' this bit-field represents whether the channel is in partial reception mode or not.

If '1' DTV_ISDBT_LAYERA_* values are assigned to the center segment and DTV_ISDBT_LAYERA_SEGMENT_COUNT has to be '1'.

If in addition DTV_ISDBT_SOUND_BROADCASTING is '1' DTV_ISDBT_PARTIAL_RECEPTION represents whether this ISDB-Tsb channel is consisting of one segment and layer or three segments and two layers.

Possible values: 0, 1, -1 (AUTO)

Note: This value cannot be determined by an automatic channel search.

#### 9.3.5.1.2. DTV_ISDBT_SOUND_BROADCASTING

This field represents whether the other DTV_ISDBT_*-parameters are referring to an ISDB-T and an ISDB-Tsb channel. (See also DTV_ISDBT_PARTIAL_RECEPTION)

Possible values: 0, 1, -1 (AUTO)

Note: This value cannot be determined by an automatic channel search.

#### 9.3.5.1.3. DTV_ISDBT_SB_SUBCHANNEL_ID

This field only applies if DTV_ISDBT_SOUND_BROADCASTING is '1'.

Possible values: 0 .. 41, -1 (AUTO)

Note: This value cannot be determined by an automatic channel search.

#### 9.3.5.1.4. DTV_ISDBT_SB_SEGMENT_IDX

This field only applies if DTV_ISDBT_SOUND_BROADCASTING is '1'.

DTV_ISDBT_SB_SEGMENT_IDX gives the index of the segment to be demodulated for an ISDB-Tsb channel where several of them are transmitted in the connected manner.

Possible values: 0 .. DTV_ISDBT_SB_SEGMENT_COUNT - 1

#### 9.3.5.1.5. DTV_ISDBT_SB_SEGMENT_COUNT

This field only applies if DTV_ISDBT_SOUND_BROADCASTING is '1'.

DTV_ISDBT_SB_SEGMENT_COUNT gives the total count of connected ISDB-Tsb channels.

Possible values: 1 .. 13

Note: This value cannot be determined by an automatic channel search.

#### 9.3.5.1.6. Hierarchical layers

(Note of the author: This might not be the correct description of the SUBCHANNEL_ID in all details, but it is my understanding of the technical background needed to program a device)

ISDB-T channels can be coded hierarchically. As opposed to DVB-T in ISDB-T hierarchical layers can be decoded simultaneously. For that reason a ISDB-T demodulator has 3 viterbi and 3 reed-solomon-decoders.

An ISDB-Tsb channel (1 or 3 segments) can be broadcasted alone or in a set of connected ISDB-Tsb channels. In this set of channels every channel can be received independently. The number of connected ISDB-Tsb segment can vary, e.g. depending on the frequency spectrum bandwidth available.

Example: Assume 8 ISDB-Tsb connected segments are broadcasted. The broadcaster has several possibilities to put the 8 segments in the air. A layer is decoding 13 segments. ISDB-T specifies that the 13 segments from 1 to 13 are in layer A, segments 1 to 3 are in layer B and segments 4 to 7 are in layer C. Segment 0 is the narrow band segment at the lower side of the channel.

#### 9.3.5.1.6.1. DTV_ISDBT_LAYER_ENABLED

Hierarchical layers in ISDB-T can be disabled in this bit mask. Active layers must be contiguous. Segments that will be received by disabled layers do not count for the _SEGMENT_COUNT or _SEGMENT_IDX parameters. E.g. if layer A is 1 segment and layer B is 2 segments, and only layer B is to be demodulated, then SEGMENT_COUNT must be 2 and SEGMENT_IDX must be between 1 and 2 (if applicable) to demodulate. This is the default.

If the channel is in the partial reception mode (DTV_ISDBT_PARTIAL_RECEPTION = 1) the central segment can be decoded independently of the other 12 segments. In that mode layer A has to have a SEGMENT_COUNT of 1.

In ISDB-Tsb only layer A is used, it can be 1 or 3 in ISDB-Tsb according to DTV_ISDBT_PARTIAL_RECEPTION. SEGMENT_COUNT must be filled accordingly.

Possible values: 0x1, 0x2, 0x4 (|-able)

DTV_ISDBT_LAYER_ENABLED[0:0] - layer A

DTV_ISDBT_LAYER_ENABLED[1:1] - layer B

DTV_ISDBT_LAYER_ENABLED[2:2] - layer C

DTV_ISDBT_LAYER_ENABLED[31:3] unused

#### 9.3.5.1.6.2. *DTV_ISDBT_LAYER*_FEC*

Possible values: FEC_AUTO, FEC_1_2, FEC_2_3, FEC_3_4, FEC_5_6, FEC_7_8

#### 9.3.5.1.6.3. *DTV_ISDBT_LAYER*_MODULATION*

Possible values: QAM_AUTO, QPSK, QAM_16, QAM_64, DQPSK

Note: If layer C is DQPSK layer B has to be DQPSK. If layer B is DQPSK and DTV_ISDBT_PARTIAL_RECEPTION=0 layer has to be DQPSK.

#### 9.3.5.1.6.4. *DTV_ISDBT_LAYER*_SEGMENT_COUNT*

Possible values: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, -1 (AUTO)

Note: Truth table for DTV_ISDBT_SOUND_BROADCASTING and DTV_ISDBT_PARTIAL_RECEPTION and LAYER*_SEGMENT_COUNT

| PR | SB | Layer A width | Layer B width | Layer C width | total width |
|---|---|---|---|---|---|
| 0 | 0 | 1 .. 13 | 1 .. 13 | 1 .. 13 | 13 |
| 1 | 0 | 1 | 1 .. 13 | 1 .. 13 | 13 |
| 0 | 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 2 | 0 | 13 |

#### 9.3.5.1.6.5. *DTV_ISDBT_LAYER*_TIME_INTERLEAVING*

Possible values: 0, 1, 2, 3, -1 (AUTO)

Note: The real inter-leaver depth-names depend on the mode (fft-size); the values here are referring to what can be found in the TMCC-structure - independent of the mode.

### 9.3.5.2. DVB-T2 parameters

This section covers parameters that apply only to the DVB-T2 delivery method. DVB-T2 support is currently in the early stages development so expect this section to grow and become more detailed with time.

#### 9.3.5.2.1. `DTV_DVBT2_PLP_ID`

DVB-T2 supports Physical Layer Pipes (PLP) to allow transmission of many data types via a single multiplex. The API will soon support this at which point this section will be expanded.

# Chapter 10. DVB Demux Device

The DVB demux device controls the filters of the DVB hardware/software. It can be accessed through */dev/adapter0/demux0*. Data types and and ioctl definitions can be accessed by including *linux/dvb/dmx.h* in your application.

## 10.1. Demux Data Types

### 10.1.1. dmx_output_t

```
typedef enum
{
 DMX_OUT_DECODER,
 DMX_OUT_TAP,
 DMX_OUT_TS_TAP
} dmx_output_t;
```

*DMX_OUT_TAP* delivers the stream output to the demux device on which the ioctl is called.

*DMX_OUT_TS_TAP* routes output to the logical DVR device */dev/dvb/adapter0/dvr0*, which delivers a TS multiplexed from all filters for which *DMX_OUT_TS_TAP* was specified.

### 10.1.2. dmx_input_t

```
typedef enum
{
 DMX_IN_FRONTEND,
 DMX_IN_DVR
} dmx_input_t;
```

### 10.1.3. dmx_pes_type_t

```
typedef enum
{
 DMX_PES_AUDIO,
```

```
#define DMX_CHECK_CRC       1
#define DMX_ONESHOT         2
#define DMX_IMMEDIATE_START 4
};
 DMX_PES_PCR,
 DMX_PES_OTHER
} dmx_pes_type_t;
```

### 10.1.8. struct dmx_pes_filter_params

### 10.1.4. dmx_event_t

```
struct dmx_pes_filter_params
{
 uint16_t            pid;
 dmx_input_t         input;
 dmx_output_t        output;
 dmx_pes_type_t      pes_type;
 uint32_t            flags;
};
```

### 10.1.5. dmx_scrambling_status_t

```
typedef enum
{
 DMX_SCRAMBLING_OFF,
 DMX_SCRAMBLING_ON
} dmx_scrambling_status_t;
```

### 10.1.9. struct dmx_event

```
struct dmx_event
{
 dmx_event_t          event;
 time_t               timeStamp;
 union
 {
  dmx_scrambling_status_t scrambling;
 } u;
};
```

### 10.1.6. struct dmx_filter

```
typedef struct dmx_filter
{
 uint8_t         filter[DMX_FILTER_SIZE];
 uint8_t         mask[DMX_FILTER_SIZE];
} dmx_filter_t;
```

### 10.1.10. struct dmx_stc

### 10.1.7. struct dmx_sct_filter_params

```
struct dmx_stc {
 unsigned int num;       /* input : which STC? 0..N */
 unsigned int base;      /* output: divisor for stc to get 90 kHz clock */
 uint64_t stc;           /* output: stc in 'base'*90 kHz units */
};
 uint16_t            pid;
 dmx_filter_t        filter;
 uint32_t            timeout;
 uint32_t            flags;
```

# 10.2. Demux Function Calls

## 10.2.1. open()

DESCRIPTION

This system call, used with a device name of /dev/dvb/adapter0/demux0, allocates a new filter and returns a handle which can be used for subsequent control of that filter. This call has to be made for each filter to be used, i.e. every returned file descriptor is a reference to a single filter. /dev/dvb/adapter0/dvr0 is a logical device to be used for retrieving Transport Streams for digital video recording. When reading from this device a transport stream containing the packets from all PES filters set in the corresponding demux device (/dev/dvb/adapter0/demux0) having the output set to DMX_OUT_TS_TAP. A recorded Transport Stream is replayed by writing to this device.
The significance of blocking or non-blocking mode is described in the documentation for functions where there is a difference. It does not affect the semantics of the open() call itself. A device opened in blocking mode can later be put into non-blocking mode (and vice versa) using the F_SETFL command of the fcntl system call.

SYNOPSIS

```
int open(const char *deviceName, int flags);
```

PARAMETERS

| | |
|---|---|
| const char *deviceName | Name of demux device. |
| int flags | A bit-wise OR of the following flags: |
| | O_RDWR read/write access |
| | O_NONBLOCK open in non-blocking mode |
| | (blocking mode is the default) |

ERRORS

| | |
|---|---|
| ENODEV | Device driver not loaded/available. |
| EINVAL | Invalid argument. |
| EMFILE | “Too many open files”, i.e. no more filters available. |

| | |
|---|---|
| ENOMEM | The driver failed to allocate enough memory. |

## 10.2.2. close()

DESCRIPTION

This system call deactivates and deallocates a filter that was previously allocated via the open() call.

SYNOPSIS

```
int close(int fd);
```

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid open file descriptor. |

## 10.2.3. read()

DESCRIPTION

This system call returns filtered data, which might be section or PES data. The filtered data is transferred from the driver's internal circular buffer to buf. The maximum amount of data to be transferred is implied by count.

When returning section data the driver always tries to return a complete single section (even though buf would provide buffer space for more data). If the size of the buffer is smaller than the section as much as possible will be returned, and the remaining data will be provided in subsequent calls.

| |
|---|
| The size of the internal buffer is 2 * 4096 bytes (the size of two maximum sized sections) by default. The size of this buffer may be changed by using the DMX_SET_BUFFER_SIZE function. If the buffer is not large enough, or if the read operations are not performed fast enough, this may result in a buffer overflow error. In this case EOVERFLOW will be returned, and the circular buffer will be emptied. This call is blocking if there is no data to return, i.e. the process will be put to sleep waiting for data, unless the O_NONBLOCK flag is specified. |
| Note that in order to be able to read, the filtering process has to be started by defining either a section or a PES filter by means of the ioctl functions, and then starting the filtering process via the DMX_START ioctl function or by setting the DMX_IMMEDIATE_START flag. If the reading is done from a logical DVR demux device, the data will constitute a Transport Stream including the packets from all PES filters in the corresponding demux device /dev/dvb/adapter0/demux0 having the output set to DMX_OUT_TS_TAP. |

SYNOPSIS

| |
|---|
| size_t read(int fd, void *buf, size_t count); |

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| void *buf | Pointer to the buffer to be used for returned filtered data. |
| size_t count | Size of buf. |

ERRORS

| | |
|---|---|
| EWOULDBLOCK | No data to return and O_NONBLOCK was specified. |
| EBADF | fd is not a valid open file descriptor. |
| ECRC | Last section had a CRC error - no data returned. The buffer is flushed. |
| EOVERFLOW | |
| | The filtered data was not read from the buffer in due time, resulting in non-read data being lost. The buffer is flushed. |

| | |
|---|---|
| ETIMEDOUT | The section was not loaded within the stated timeout period. See ioctl DMX_SET_FILTER for how to set a timeout. |
| EFAULT | The driver failed to write to the callers buffer due to an invalid *buf pointer. |

## 10.2.4. write()

DESCRIPTION

This system call is only provided by the logical device /dev/dvb/adapter0/dvr0, associated with the physical demux device that provides the actual DVR functionality. It is used for replay of a digitally recorded Transport Stream. Matching filters have to be defined in the corresponding physical demux device, /dev/dvb/adapter0/demux0. The amount of data to be transferred is implied by count.

SYNOPSIS

```
ssize_t write(int fd, const void *buf, size_t count);
```

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| void *buf | Pointer to the buffer containing the Transport Stream. |
| size_t count | Size of buf. |

ERRORS

| | |
|---|---|
| EWOULDBLOCK | No data was written. This might happen if O_NONBLOCK was specified and there is no more buffer space available (if O_NONBLOCK is not specified the function will block until buffer space is available). |

| | |
|---|---|
| EBUSY | This error code indicates that there are conflicting requests. The corresponding demux device is setup to receive data from the front- end. Make sure that these filters are stopped and that the filters with input set to DMX_IN_DVR are started. |
| EBADF | fd is not a valid open file descriptor. |

## 10.2.5. DMX_START

DESCRIPTION

This ioctl call is used to start the actual filtering operation defined via the ioctl calls DMX_SET_FILTER or DMX_SET_PES_FILTER.

SYNOPSIS

```
int ioctl( int fd, int request = DMX_START);
```

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| int request | Equals DMX_START for this command. |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid file descriptor. |
| EINVAL | Invalid argument, i.e. no filtering parameters provided via the DMX_SET_FILTER or DMX_SET_PES_FILTER functions. |
| EBUSY | This error code indicates that there are conflicting requests. There are active filters filtering data from another input source. Make sure that these filters are stopped before starting this filter. |

## 10.2.6. DMX_STOP

DESCRIPTION

This ioctl call is used to stop the actual filtering operation defined via the ioctl calls DMX_SET_FILTER or DMX_SET_PES_FILTER and started via the DMX_START command.

SYNOPSIS

```
int ioctl( int fd, int request = DMX_STOP);
```

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| int request | Equals DMX_STOP for this command. |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid file descriptor. |

## 10.2.7. DMX_SET_FILTER

DESCRIPTION

This ioctl call sets up a filter according to the filter and mask parameters provided. A timeout may be defined stating number of seconds to wait for a section to be loaded. A value of 0 means that no timeout should be applied. Finally there is a flag field where it is possible to state whether a section should be CRC-checked, whether the filter should be a ”one-shot” filter, i.e. if the filtering operation should be stopped after the first section is received, and whether the filtering operation should be started immediately (without waiting for a DMX_START ioctl call). If a filter was previously set-up, this filter will be canceled, and the receive buffer will be flushed.

SYNOPSIS

```
int ioctl( int fd, int request = DMX_SET_FILTER, struct dmx_sct_filter_params *params);
```

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| int request | Equals DMX_SET_FILTER for this command. |
| struct dmx_sct_filter_params *params | Pointer to structure containing filter parameters. |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid file descriptor. |
| EINVAL | Invalid argument. |

## 10.2.8. DMX_SET_PES_FILTER

DESCRIPTION

| |
|---|
| This ioctl call sets up a PES filter according to the parameters provided. By a PES filter is meant a filter that is based just on the packet identifier (PID), i.e. no PES header or payload filtering capability is supported. |
| The transport stream destination for the filtered output may be set. Also the PES type may be stated in order to be able to e.g. direct a video stream directly to the video decoder. Finally there is a flag field where it is possible to state whether the filtering operation should be started immediately (without waiting for a DMX_START ioctl call). If a filter was previously set-up, this filter will be cancelled, and the receive buffer will be flushed. |

SYNOPSIS

| |
|---|
| int ioctl( int fd, int request = DMX_SET_PES_FILTER, struct dmx_pes_filter_params ⋆params); |

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| int request | Equals DMX_SET_PES_FILTER for this command. |

| | |
|---|---|
| struct dmx_pes_filter_params *params | Pointer to structure containing filter parameters. |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid file descriptor. |
| EINVAL | Invalid argument. |
| EBUSY | This error code indicates that there are conflicting requests. There are active filters filtering data from another input source. Make sure that these filters are stopped before starting this filter. |

## 10.2.9. DMX_SET_BUFFER_SIZE

DESCRIPTION

This ioctl call is used to set the size of the circular buffer used for filtered data. The default size is two maximum sized sections, i.e. if this function is not called a buffer size of 2 * 4096 bytes will be used.

SYNOPSIS

```
int ioctl( int fd, int request = DMX_SET_BUFFER_SIZE, unsigned long size);
```

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| int request | Equals DMX_SET_BUFFER_SIZE for this command. |
| unsigned long size | Size of circular buffer. |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid file descriptor. |
| ENOMEM | The driver was not able to allocate a buffer of the requested size. |

## 10.2.10. DMX_GET_EVENT

DESCRIPTION

| |
|---|
| This ioctl call returns an event if available. If an event is not available, the behavior depends on whether the device is in blocking or non-blocking mode. In the latter case, the call fails immediately with errno set to EWOULDBLOCK. In the former case, the call blocks until an event becomes available. |
| The standard Linux poll() and/or select() system calls can be used with the device file descriptor to watch for new events. For select(), the file descriptor should be included in the exceptfds argument, and for poll(), POLLPRI should be specified as the wake-up condition. Only the latest event for each filter is saved. |

SYNOPSIS

| |
|---|
| int ioctl( int fd, int request = DMX_GET_EVENT, struct dmx_event ⋆ev); |

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| int request | Equals DMX_GET_EVENT for this command. |
| struct dmx_event *ev | Pointer to the location where the event is to be stored. |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid file descriptor. |
| EFAULT | ev points to an invalid address. |
| EWOULDBLOCK | There is no event pending, and the device is in non-blocking mode. |

## 10.2.11. DMX_GET_STC

DESCRIPTION

This ioctl call returns the current value of the system time counter (which is driven by a PES filter of type DMX_PES_PCR). Some hardware supports more than one STC, so you must specify which one by setting the num field of stc before the ioctl (range 0...n). The result is returned in form of a ratio with a 64 bit numerator and a 32 bit denominator, so the real 90kHz STC value is stc->stc / stc->base .

SYNOPSIS

```
int ioctl( int fd, int request = DMX_GET_STC, struct dmx_stc ⋆stc);
```

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| int request | Equals DMX_GET_STC for this command. |
| struct dmx_stc *stc | Pointer to the location where the stc is to be stored. |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid file descriptor. |
| EFAULT | stc points to an invalid address. |
| EINVAL | Invalid stc number. |

# Chapter 11. DVB Video Device

The DVB video device controls the MPEG2 video decoder of the DVB hardware. It can be accessed through */dev/dvb/adapter0/video0*. Data types and and ioctl definitions can be accessed by including *linux/dvb/video.h* in your application.

Note that the DVB video device only controls decoding of the MPEG video stream, not its presentation on the TV or computer screen. On PCs this is typically handled by an associated video4linux device, e.g. */dev/video*, which allows scaling and defining output windows.

Some DVB cards don't have their own MPEG decoder, which results in the omission of the audio and video device as well as the video4linux device.

The ioctls that deal with SPUs (sub picture units) and navigation packets are only supported on some MPEG decoders made for DVD playback.

## 11.1. Video Data Types

### 11.1.1. video_format_t

The *video_format_t* data type defined by

```
typedef enum {
 VIDEO_FORMAT_4_3,
 VIDEO_FORMAT_16_9
} video_format_t;
```

is used in the VIDEO_SET_FORMAT function (??) to tell the driver which aspect ratio the output hardware (e.g. TV) has. It is also used in the data structures video_status (??) returned by VIDEO_GET_STATUS (??) and video_event (??) returned by VIDEO_GET_EVENT (??) which report about the display format of the current video stream.

### 11.1.2. video_display_format_t

In case the display format of the video stream and of the display hardware differ the application has to specify how to handle the cropping of the picture. This can be done using the VIDEO_SET_DISPLAY_FORMAT call (??) which accepts

```
typedef struct timestamp {
 time_t time;
 long int microseconds;
} video_timestamp;

typedef enum {
 VIDEO_PAN_SCAN,
 VIDEO_LETTER_BOX,
 VIDEO_CENTER_CUT_OUT
} video_displayformat_t;
```

as argument.

## 11.1.6. struct video_status

The VIDEO_GET_STATUS call returns the following structure informing about various states of the playback operation.

```
struct video_status {
 boolean video_blank;
 video_play_state_t play_state;
 video_stream_source_t stream_source;
 video_format_t video_format;
 video_displayformat_t display_format;
};
```

If video_blank is set video will be blanked out if the channel is changed or if playback is stopped. Otherwise, the last picture will be displayed. play_state indicates if the video is currently frozen, stopped, or being played back. The stream_source corresponds to the seleted source for the video stream. It can come either from the demultiplexer or from memory. The video_format indicates the aspect ratio (one of 4:3 or 16:9) of the currently played video stream. Finally, display_format corresponds to the selected cropping mode in case the source video format is not the same as the format of the output device.

## 11.1.3. video stream source

The video stream source is set through the VIDEO_SELECT_SOURCE call and can take the following values, depending on whether we are replaying from an internal (demuxer) or external (user write) source.

```
typedef enum {
 VIDEO_SOURCE_DEMUX,
 VIDEO_SOURCE_MEMORY
} video_stream_source_t;
```

VIDEO_SOURCE_DEMUX selects the demultiplexer (fed either by the frontend or the DVR device) as the source of the video stream. If VIDEO_SOURCE_MEMORY is selected the stream comes from the application through the *write*() system call.

## 11.1.4. video play state

The following values can be returned by the VIDEO_GET_STATUS call representing the state of video playback.

```
typedef enum {
 VIDEO_STOPPED,
 VIDEO_PLAYING,
 VIDEO_FREEZED
} video_play_state_t;
```

## 11.1.7. struct video_still_picture

An I-frame displayed via the VIDEO_STILLPICTURE call is passed on within the following structure.

```
/* pointer to and size of a single iframe in memory */
struct video_still_picture {
 char *iFrame;
 int32_t size;
};
```

## 11.1.5. struct video_event

The following is the structure of a video event as it is returned by the VIDEO_GET_EVENT call.

```
struct video_event {
 int32_t type;
```

## 11.1.8. video capabilities

A call to VIDEO_GET_CAPABILITIES returns an unsigned integer with the following bits set according to the hardwares capabilities.

```
/* bit definitions for capabilities: */

/* can the hardware decode MPEG1 and/or MPEG2? */
#define VIDEO_CAP_MPEG1   1
#define VIDEO_CAP_MPEG2   2

/* can you send a system and/or program stream to video device?
   (you still have to open the video and the audio device but only
    send the stream to the video device) */
#define VIDEO_CAP_SYS     4
#define VIDEO_CAP_PROG    8

/* can the driver also handle SPU, NAVI and CSS encoded data?
   (CSS API is not present yet) */
#define VIDEO_CAP_SPU    16
#define VIDEO_CAP_NAVI   32
#define VIDEO_CAP_CSS    64
```

## 11.1.11. video SPU

## 11.1.9. video system

Calling VIDEO_SET_SPU deactivates or activates SPU decoding, according to the following format:

A call to VIDEO_SET_SYSTEM sets the desired video system for TV output. The following system types can be set:

```
typedef
typedef enum {
  VIDEO_SYSTEM_PAL,
  VIDEO_SYSTEM_NTSC,
  VIDEO_SYSTEM_PALN,
  VIDEO_SYSTEM_PALNc,
  VIDEO_SYSTEM_PALM,
  VIDEO_SYSTEM_NTSC60,
  VIDEO_SYSTEM_PAL60,
  VIDEO_SYSTEM_PALM60
} video_system_t;
```

## 11.1.12. video SPU palette

The following structure is used to set the SPU palette by calling VIDEO_SPU_PALETTE:

```
typedef
struct video_spu_palette{
  int length;
  uint8_t *palette;
}
typedef video_spu_palette_t;
```

## 11.1.10. struct video_highlight

Calling the ioctl VIDEO_SET_HIGHLIGHTS posts the SPU highlight information. The call expects the following format for that information:

```
typedef
struct video_highlight {
  boolean active;      /*    1=show highlight, 0=hide highlight */
```

### 11.1.13. video NAVI pack

In order to get the navigational data the following structure has to be passed to the ioctl VIDEO_GET_NAVI:

```
typedef
struct video_navi_pack{
 int length;          /* 0 ... 1024 */
 uint8_t data[1024];
} video_navi_pack_t;
```

### 11.1.14. video attributes

The following attributes can be set by a call to VIDEO_SET_ATTRIBUTES:

```
typedef uint16_t video_attributes_t;
/*   bits: descr. */
/*   15-14 Video compression mode (0=MPEG-1, 1=MPEG-2) */
/*   13-12 TV system (0=525/60, 1=625/50) */
/*   11-10 Aspect ratio (0=4:3, 3=16:9) */
/*    9- 8 permitted display mode on 4:3 monitor (0=both, 1=only pan-sca
/*    7    line 21-1 data present in GOP (1=yes, 0=no) */
/*    6    line 21-2 data present in GOP (1=yes, 0=no) */
/*    5- 3 source resolution (0=720x480/576, 1=704x480/576, 2=352x480/57
/*    2    source letterboxed (1=yes, 0=no) */
/*    0    film/camera mode (0=camera, 1=film (625/50 only)) */
```

## 11.2. Video Function Calls

### 11.2.1. open()

DESCRIPTION

This system call opens a named video device (e.g. /dev/dvb/adapter0/video0) for subsequent use.
When an open() call has succeeded, the device will be ready for use. The significance of blocking or non-blocking mode is described in the documentation for functions where there is a difference. It does not affect the semantics of the open() call itself. A device opened in blocking mode can later be put into non-blocking mode (and vice versa) using the F_SETFL command of the fcntl system call. This is a standard system call, documented in the Linux manual page for fcntl. Only one user can open the Video Device in O_RDWR mode. All other attempts to open the device in this mode will fail, and an error-code will be returned. If the Video Device is opened in O_RDONLY mode, the only ioctl call that can be used is VIDEO_GET_STATUS. All other call will return an error code.

SYNOPSIS

| |
|---|
| int open(const char *deviceName, int flags); |

PARAMETERS

| | |
|---|---|
| const char *deviceName | Name of specific video device. |
| int flags | A bit-wise OR of the following flags: |
| | O_RDONLY read-only access |
| | O_RDWR read/write access |
| | O_NONBLOCK open in non-blocking mode |
| | (blocking mode is the default) |

ERRORS

| | |
|---|---|
| ENODEV | Device driver not loaded/available. |
| EINTERNAL | Internal error. |
| EBUSY | Device or resource busy. |
| EINVAL | Invalid argument. |

## 11.2.2. close()

DESCRIPTION

| |
|---|
| This system call closes a previously opened video device. |

SYNOPSIS

| |
|---|
| int close(int fd); |

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid open file descriptor. |

## 11.2.3. write()

DESCRIPTION

This system call can only be used if VIDEO_SOURCE_MEMORY is selected in the ioctl call VIDEO_SELECT_SOURCE. The data provided shall be in PES format, unless the capability allows other formats. If O_NONBLOCK is not specified the function will block until buffer space is available. The amount of data to be transferred is implied by count.

SYNOPSIS

```
size_t write(int fd, const void *buf, size_t count);
```

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| void *buf | Pointer to the buffer containing the PES data. |
| size_t count | Size of buf. |

ERRORS

| | |
|---|---|
| EPERM | Mode VIDEO_SOURCE_MEMORY not selected. |
| ENOMEM | Attempted to write more data than the internal buffer can hold. |
| EBADF | fd is not a valid open file descriptor. |

## 11.2.4. VIDEO_STOP

DESCRIPTION

This ioctl call asks the Video Device to stop playing the current stream. Depending on the input parameter, the screen can be blanked out or displaying the last decoded frame.

SYNOPSIS

```
int ioctl(fd, int request = VIDEO_STOP, boolean mode);
```

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| int request | Equals VIDEO_STOP for this command. |
| Boolean mode | Indicates how the screen shall be handled. |
| | TRUE: Blank screen when stop. |
| | FALSE: Show last decoded frame. |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid open file descriptor |
| EINTERNAL | Internal error, possibly in the communication with the DVB subsystem. |

## 11.2.5. VIDEO_PLAY

DESCRIPTION

This ioctl call asks the Video Device to start playing a video stream from the selected source.

SYNOPSIS

```
int ioctl(fd, int request = VIDEO_PLAY);
```

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| int request | Equals VIDEO_PLAY for this command. |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid open file descriptor |

| | |
|---|---|
| EINTERNAL | Internal error, possibly in the communication with the DVB subsystem. |

## 11.2.6. VIDEO_FREEZE

DESCRIPTION

This ioctl call suspends the live video stream being played. Decoding and playing are frozen. It is then possible to restart the decoding and playing process of the video stream using the VIDEO_CONTINUE command. If VIDEO_SOURCE_MEMORY is selected in the ioctl call VIDEO_SELECT_SOURCE, the DVB subsystem will not decode any more data until the ioctl call VIDEO_CONTINUE or VIDEO_PLAY is performed.

SYNOPSIS

```
int ioctl(fd, int request = VIDEO_FREEZE);
```

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| int request | Equals VIDEO_FREEZE for this command. |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid open file descriptor |
| EINTERNAL | Internal error, possibly in the communication with the DVB subsystem. |

## 11.2.7. VIDEO_CONTINUE

DESCRIPTION

| |
|---|
| This ioctl call restarts decoding and playing processes of the video stream which was played before a call to VIDEO_FREEZE was made. |

SYNOPSIS

| |
|---|
| int ioctl(fd, int request = VIDEO_CONTINUE); |

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| int request | Equals VIDEO_CONTINUE for this command. |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid open file descriptor |
| EINTERNAL | Internal error, possibly in the communication with the DVB subsystem. |

## 11.2.8. VIDEO_SELECT_SOURCE

DESCRIPTION

| |
|---|
| This ioctl call informs the video device which source shall be used for the input data. The possible sources are demux or memory. If memory is selected, the data is fed to the video device through the write command. |

SYNOPSIS

| |
|---|
| int ioctl(fd, int request = VIDEO_SELECT_SOURCE, video_stream_source_t source); |

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| int request | Equals VIDEO_SELECT_SOURCE for this command. |

| | |
|---|---|
| video_stream_source_t source | Indicates which source shall be used for the Video stream. |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid open file descriptor |
| EINTERNAL | Internal error, possibly in the communication with the DVB subsystem. |

## 11.2.9. VIDEO_SET_BLANK

DESCRIPTION

This ioctl call asks the Video Device to blank out the picture.

SYNOPSIS

int ioctl(fd, int request = VIDEO_SET_BLANK, boolean mode);

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| int request | Equals VIDEO_SET_BLANK for this command. |
| boolean mode | TRUE: Blank screen when stop. |
| | FALSE: Show last decoded frame. |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid open file descriptor |
| EINTERNAL | Internal error, possibly in the communication with the DVB subsystem. |
| EINVAL | Illegal input parameter |

## 11.2.10. VIDEO_GET_STATUS

DESCRIPTION

This ioctl call asks the Video Device to return the current status of the device.

SYNOPSIS

int ioctl(fd, int request = VIDEO_GET_STATUS, struct video_status *status);

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| int request | Equals VIDEO_GET_STATUS for this command. |
| struct video_status *status | Returns the current status of the Video Device. |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid open file descriptor |
| EINTERNAL | Internal error, possibly in the communication with the DVB subsystem. |
| EFAULT | status points to invalid address |

## 11.2.11. VIDEO_GET_EVENT

DESCRIPTION

This ioctl call returns an event of type video_event if available. If an event is not available, the behavior depends on whether the device is in blocking or non-blocking mode. In the latter case, the call fails immediately with errno set to EWOULDBLOCK. In the former case, the call blocks until an event becomes available. The standard Linux poll() and/or select() system calls can be used with the device file descriptor to watch for new events. For select(), the file descriptor should be included in the exceptfds argument, and for poll(), POLLPRI should be specified as the wake-up condition. Read-only permissions are sufficient for this ioctl call.

SYNOPSIS

| |
|---|
| int ioctl(fd, int request = VIDEO_GET_EVENT, struct video_event ⋆ev); |

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| int request | Equals VIDEO_GET_EVENT for this command. |
| struct video_event *ev | Points to the location where the event, if any, is to be stored. |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid open file descriptor |
| EFAULT | ev points to invalid address |
| EWOULDBLOCK | There is no event pending, and the device is in non-blocking mode. |
| EOVERFLOW | |
| | Overflow in event queue - one or more events were lost. |

## 11.2.12. VIDEO_SET_DISPLAY_FORMAT

DESCRIPTION

| |
|---|
| This ioctl call asks the Video Device to select the video format to be applied by the MPEG chip on the video. |

SYNOPSIS

| |
|---|
| int ioctl(fd, int request = VIDEO_SET_DISPLAY_FORMAT, video_display_format_t format); |

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |

| | |
|---|---|
| int request | Equals VIDEO_SET_DISPLAY_FORMAT for this command. |
| video_display_format_t format | Selects the video format to be used. |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid open file descriptor |
| EINTERNAL | Internal error. |
| EINVAL | Illegal parameter format. |

## 11.2.13. VIDEO_STILLPICTURE

DESCRIPTION

This ioctl call asks the Video Device to display a still picture (I-frame). The input data shall contain an I-frame. If the pointer is NULL, then the current displayed still picture is blanked.

SYNOPSIS

int ioctl(fd, int request = VIDEO_STILLPICTURE, struct video_still_picture ⋆sp);

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| int request | Equals VIDEO_STILLPICTURE for this command. |
| struct video_still_picture *sp | Pointer to a location where an I-frame and size is stored. |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid open file descriptor |
| EINTERNAL | Internal error. |
| EFAULT | sp points to an invalid iframe. |

## 11.2.14. VIDEO_FAST_FORWARD

DESCRIPTION

This ioctl call asks the Video Device to skip decoding of N number of I-frames. This call can only be used if VIDEO_SOURCE_MEMORY is selected.

SYNOPSIS

int ioctl(fd, int request = VIDEO_FAST_FORWARD, int nFrames);

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| int request | Equals VIDEO_FAST_FORWARD for this command. |
| int nFrames | The number of frames to skip. |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid open file descriptor |
| EINTERNAL | Internal error. |
| EPERM | Mode VIDEO_SOURCE_MEMORY not selected. |
| EINVAL | Illegal parameter format. |

## 11.2.15. VIDEO_SLOWMOTION

DESCRIPTION

This ioctl call asks the video device to repeat decoding frames N number of times. This call can only be used if VIDEO_SOURCE_MEMORY is selected.

SYNOPSIS

int ioctl(fd, int request = VIDEO_SLOWMOTION, int nFrames);

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| int request | Equals VIDEO_SLOWMOTION for this command. |
| int nFrames | The number of times to repeat each frame. |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid open file descriptor |
| EINTERNAL | Internal error. |
| EPERM | Mode VIDEO_SOURCE_MEMORY not selected. |
| EINVAL | Illegal parameter format. |

## 11.2.16. VIDEO_GET_CAPABILITIES

DESCRIPTION

This ioctl call asks the video device about its decoding capabilities. On success it returns and integer which has bits set according to the defines in section ??.

SYNOPSIS

```
int ioctl(fd, int request = VIDEO_GET_CAPABILITIES, unsigned int *cap);
```

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| int request | Equals VIDEO_GET_CAPABILITIES for this command. |
| unsigned int *cap | Pointer to a location where to store the capability information. |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid open file descriptor |
| EFAULT | cap points to an invalid iframe. |

## 11.2.17. VIDEO_SET_ID

DESCRIPTION

This ioctl selects which sub-stream is to be decoded if a program or system stream is sent to the video device.

SYNOPSIS

```
int ioctl(int fd, int request = VIDEO_SET_ID, int id);
```

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| int request | Equals VIDEO_SET_ID for this command. |
| int id | video sub-stream id |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid open file descriptor. |
| EINTERNAL | Internal error. |
| EINVAL | Invalid sub-stream id. |

## 11.2.18. VIDEO_CLEAR_BUFFER

DESCRIPTION

This ioctl call clears all video buffers in the driver and in the decoder hardware.

SYNOPSIS

```
int ioctl(fd, int request = VIDEO_CLEAR_BUFFER);
```

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| int request | Equals VIDEO_CLEAR_BUFFER for this command. |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid open file descriptor |

## 11.2.19. VIDEO_SET_STREAMTYPE

DESCRIPTION

This ioctl tells the driver which kind of stream to expect being written to it. If this call is not used the default of video PES is used. Some drivers might not support this call and always expect PES.

SYNOPSIS

```
int ioctl(fd, int request = VIDEO_SET_STREAMTYPE, int type);
```

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| int request | Equals VIDEO_SET_STREAMTYPE for this command. |
| int type | stream type |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid open file descriptor |
| EINVAL | type is not a valid or supported stream type. |

## 11.2.20. VIDEO_SET_FORMAT

DESCRIPTION

This ioctl sets the screen format (aspect ratio) of the connected output device (TV) so that the output of the decoder can be adjusted accordingly.

SYNOPSIS

| int ioctl(fd, int request = VIDEO_SET_FORMAT, video_format_t format); |
|---|

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| int request | Equals VIDEO_SET_FORMAT for this command. |
| video_format_t format | video format of TV as defined in section ??. |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid open file descriptor |
| EINVAL | format is not a valid video format. |

## 11.2.21. VIDEO_SET_SYSTEM

DESCRIPTION

| This ioctl sets the television output format. The format (see section ??) may vary from the color format of the displayed MPEG stream. If the hardware is not able to display the requested format the call will return an error. |
|---|

SYNOPSIS

| int ioctl(fd, int request = VIDEO_SET_SYSTEM , video_system_t system); |
|---|

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| int request | Equals VIDEO_SET_FORMAT for this command. |
| video_system_t system | video system of TV output. |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid open file descriptor |

| | |
|---|---|
| EINVAL | system is not a valid or supported video system. |

## 11.2.22. VIDEO_SET_HIGHLIGHT

DESCRIPTION

This ioctl sets the SPU highlight information for the menu access of a DVD.

SYNOPSIS

int ioctl(fd, int request = VIDEO_SET_HIGHLIGHT ,video_highlight_t *vhilite)

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| int request | Equals VIDEO_SET_HIGHLIGHT for this command. |
| video_highlight_t *vhilite | SPU Highlight information according to section ??. |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid open file descriptor. |
| EINVAL | input is not a valid highlight setting. |

## 11.2.23. VIDEO_SET_SPU

DESCRIPTION

This ioctl activates or deactivates SPU decoding in a DVD input stream. It can only be used, if the driver is able to handle a DVD stream.

SYNOPSIS

int ioctl(fd, int request = VIDEO_SET_SPU , video_spu_t *spu)

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| int request | Equals VIDEO_SET_SPU for this command. |
| video_spu_t *spu | SPU decoding (de)activation and subid setting according to section ??. |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid open file descriptor |
| EINVAL | input is not a valid spu setting or driver cannot handle SPU. |

## 11.2.24. VIDEO_SET_SPU_PALETTE

DESCRIPTION

| |
|---|
| This ioctl sets the SPU color palette. |

SYNOPSIS

| |
|---|
| int ioctl(fd, int request = VIDEO_SET_SPU_PALETTE ,video_spu_palette_t *palette ) |

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| int request | Equals VIDEO_SET_SPU_PALETTE for this command. |
| video_spu_palette_t *palette | SPU palette according to section ??. |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid open file descriptor |
| EINVAL | input is not a valid palette or driver doesn't handle SPU. |

## 11.2.25. VIDEO_GET_NAVI

DESCRIPTION

This ioctl returns navigational information from the DVD stream. This is especially needed if an encoded stream has to be decoded by the hardware.

SYNOPSIS

int ioctl(fd, int request = VIDEO_GET_NAVI , video_navi_pack_t *navipack)

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| int request | Equals VIDEO_GET_NAVI for this command. |
| video_navi_pack_t *navipack | PCI or DSI pack (private stream 2) according to section ??. |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid open file descriptor |
| EFAULT | driver is not able to return navigational information |

## 11.2.26. VIDEO_SET_ATTRIBUTES

DESCRIPTION

This ioctl is intended for DVD playback and allows you to set certain information about the stream. Some hardware may not need this information, but the call also tells the hardware to prepare for DVD playback.

SYNOPSIS

int ioctl(fd, int request = VIDEO_SET_ATTRIBUTE ,video_attributes_t vattr)

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| int request | Equals VIDEO_SET_ATTRIBUTE for this command. |
| video_attributes_t vattr | video attributes according to section ??. |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid open file descriptor |
| EINVAL | input is not a valid attribute setting. |

# Chapter 12. DVB Audio Device

The DVB audio device controls the MPEG2 audio decoder of the DVB hardware. It can be accessed through */dev/dvb/adapter0/audio0*. Data types and and ioctl definitions can be accessed by including *linux/dvb/video.h* in your application.

Please note that some DVB cards don't have their own MPEG decoder, which results in the omission of the audio and video device.

## 12.1. Audio Data Types

This section describes the structures, data types and defines used when talking to the audio device.

### 12.1.1. audio_stream_source_t

The audio stream source is set through the AUDIO_SELECT_SOURCE call and can take the following values, depending on whether we are replaying from an internal (demux) or external (user write) source.

```
typedef enum {
 AUDIO_SOURCE_DEMUX,
 AUDIO_SOURCE_MEMORY
} audio_stream_source_t;
```

AUDIO_SOURCE_DEMUX selects the demultiplexer (fed either by the frontend or the DVR device) as the source of the video stream. If AUDIO_SOURCE_MEMORY is selected the stream comes from the application through the *write()* system call.

### 12.1.2. audio_play_state_t

The following values can be returned by the AUDIO_GET_STATUS call representing the state of audio playback.

```
typedef enum {
 AUDIO_STOPPED,
 AUDIO_PLAYING,
 AUDIO_PAUSED
```

### 12.1.6. audio encodings

A call to AUDIO_GET_CAPABILITIES returns an unsigned integer with the following bits set according to the hardwares capabilities.

```
#define AUDIO_CAP_DTS    1
#define AUDIO_CAP_LPCM   2
#define AUDIO_CAP_MP1    4
#define AUDIO_CAP_MP2    8
#define AUDIO_CAP_MP3   16
#define AUDIO_CAP_AAC   32
#define AUDIO_CAP_OGG   64
#define AUDIO_CAP_SDDS 128
#define AUDIO_CAP_AC3  256
```

### 12.1.3. audio_channel_select_t

The audio channel selected via AUDIO_CHANNEL_SELECT is determined by the following values.

```
typedef enum {
  AUDIO_STEREO,
  AUDIO_MONO_LEFT,
  AUDIO_MONO_RIGHT,
} audio_channel_select_t;
```

### 12.1.7. struct audio_karaoke

### 12.1.4. struct audio_status

The ioctl AUDIO_SET_KARAOKE uses the following format:

The AUDIO_GET_STATUS call returns the following structure informing about various states of the playback operation.

```
typedef
struct audio_karaoke{
  int vocal1;
typedef struct audio_status {
  int vocal2;
  boolean AV_sync_state;
  int melody;
  boolean mute_state;
}boolean audio_karaoke_t;
  audio_play_state_t play_state;
  audio_stream_source_t stream_source;
  audio_channel_select_t channel_select;
  boolean bypass_mode;
} audio_status_t;
```

If Vocal1 or Vocal2 are non-zero, they get mixed into left and right t at 70% each. If both, Vocal1 and Vocal2 are non-zero, Vocal1 gets mixed into the left channel and Vocal2 into the right channel at 100% each. Ff Melody is non-zero, the melody channel gets mixed into left and right.

### 12.1.8. audio attributes

### 12.1.5. struct audio_mixer

The following attributes can be set by a call to AUDIO_SET_ATTRIBUTES:

The following structure is used by the AUDIO_SET_MIXER call to set the audio volume.

```
typedef uint16_t audio_attributes_t;
/*   bits: descr. */
/*   15-13 audio coding mode (0=ac3, 2=mpeg1, 3=mpeg2ext, 4=LPCM, 6=DTS,
typedef struct audio_mixer {
/*    12    multichannel extension */
  unsigned int volume_left;
/*  11-10 audio type (0=not spec, 1=language included) */
  unsigned int volume_right;
/*   9- 8 audio application mode (0=not spec, 1=karaoke, 2=surround) */
} audio_mixer_t;
/*    7- 6 Quantization / DRC (mpeg audio: 1=DRC exists)(lpcm: 0=16bit,
/*    5- 4 Sample frequency fs (0=48kHz, 1=96kHz) */
/*    2- 0 number of audio channels (n+1 channels) */
```

# 12.2. Audio Function Calls

## 12.2.1. open()

DESCRIPTION

This system call opens a named audio device (e.g. /dev/dvb/adapter0/audio0) for subsequent use. When an open() call has succeeded, the device will be ready for use. The significance of blocking or non-blocking mode is described in the documentation for functions where there is a difference. It does not affect the semantics of the open() call itself. A device opened in blocking mode can later be put into non-blocking mode (and vice versa) using the F_SETFL command of the fcntl system call. This is a standard system call, documented in the Linux manual page for fcntl. Only one user can open the Audio Device in O_RDWR mode. All other attempts to open the device in this mode will fail, and an error code will be returned. If the Audio Device is opened in O_RDONLY mode, the only ioctl call that can be used is AUDIO_GET_STATUS. All other call will return with an error code.

SYNOPSIS

```
int open(const char *deviceName, int flags);
```

PARAMETERS

| | |
|---|---|
| const char *deviceName | Name of specific audio device. |
| int flags | A bit-wise OR of the following flags: |
| | O_RDONLY read-only access |
| | O_RDWR read/write access |
| | O_NONBLOCK open in non-blocking mode |
| | (blocking mode is the default) |

ERRORS

| | |
|---|---|
| ENODEV | Device driver not loaded/available. |
| EINTERNAL | Internal error. |
| EBUSY | Device or resource busy. |
| EINVAL | Invalid argument. |

## 12.2.2. close()

DESCRIPTION

This system call closes a previously opened audio device.

SYNOPSIS

```
int close(int fd);
```

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid open file descriptor. |

## 12.2.3. write()

DESCRIPTION

This system call can only be used if AUDIO_SOURCE_MEMORY is selected in the ioctl call AUDIO_SELECT_SOURCE. The data provided shall be in PES format. If O_NONBLOCK is not specified the function will block until buffer space is available. The amount of data to be transferred is implied by count.

SYNOPSIS

```
size_t write(int fd, const void *buf, size_t count);
```

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| void *buf | Pointer to the buffer containing the PES data. |
| size_t count | Size of buf. |

ERRORS

| | |
|---|---|
| EPERM | Mode AUDIO_SOURCE_MEMORY not selected. |
| ENOMEM | Attempted to write more data than the internal buffer can hold. |
| EBADF | fd is not a valid open file descriptor. |

## 12.2.4. AUDIO_STOP

DESCRIPTION

This ioctl call asks the Audio Device to stop playing the current stream.

SYNOPSIS

```
int ioctl(int fd, int request = AUDIO_STOP);
```

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| int request | Equals AUDIO_STOP for this command. |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid open file descriptor |
| EINTERNAL | Internal error. |

## 12.2.5. AUDIO_PLAY

DESCRIPTION

This ioctl call asks the Audio Device to start playing an audio stream from the selected source.

SYNOPSIS

```
int ioctl(int fd, int request = AUDIO_PLAY);
```

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| int request | Equals AUDIO_PLAY for this command. |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid open file descriptor |
| EINTERNAL | Internal error. |

## 12.2.6. AUDIO_PAUSE

DESCRIPTION

| |
|---|
| This ioctl call suspends the audio stream being played. Decoding and playing are paused. It is then possible to restart again decoding and playing process of the audio stream using AUDIO_CONTINUE command. |
| If AUDIO_SOURCE_MEMORY is selected in the ioctl call AUDIO_SELECT_SOURCE, the DVB-subsystem will not decode (consume) any more data until the ioctl call AUDIO_CONTINUE or AUDIO_PLAY is performed. |

SYNOPSIS

| |
|---|
| int ioctl(int fd, int request = AUDIO_PAUSE); |

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| int request | Equals AUDIO_PAUSE for this command. |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid open file descriptor. |
| EINTERNAL | Internal error. |

## 12.2.7. AUDIO_SELECT_SOURCE

DESCRIPTION

This ioctl call informs the audio device which source shall be used for the input data. The possible sources are demux or memory. If AUDIO_SOURCE_MEMORY is selected, the data is fed to the Audio Device through the write command.

SYNOPSIS

int ioctl(int fd, int request = AUDIO_SELECT_SOURCE, audio_stream_source_t source);

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| int request | Equals AUDIO_SELECT_SOURCE for this command. |
| audio_stream_source_t source | Indicates the source that shall be used for the Audio stream. |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid open file descriptor. |
| EINTERNAL | Internal error. |
| EINVAL | Illegal input parameter. |

## 12.2.8. AUDIO_SET_MUTE

DESCRIPTION

This ioctl call asks the audio device to mute the stream that is currently being played.

SYNOPSIS

int ioctl(int fd, int request = AUDIO_SET_MUTE, boolean state);

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| int request | Equals AUDIO_SET_MUTE for this command. |
| boolean state | Indicates if audio device shall mute or not. |
| | TRUE Audio Mute |
| | FALSE Audio Un-mute |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid open file descriptor. |
| EINTERNAL | Internal error. |
| EINVAL | Illegal input parameter. |

## 12.2.9. AUDIO_SET_AV_SYNC

DESCRIPTION

| |
|---|
| This ioctl call asks the Audio Device to turn ON or OFF A/V synchronization. |

SYNOPSIS

| |
|---|
| int ioctl(int fd, int request = AUDIO_SET_AV_SYNC, boolean state); |

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| int request | Equals AUDIO_AV_SYNC for this command. |
| boolean state | Tells the DVB subsystem if A/V synchronization shall be ON or OFF. |
| | TRUE AV-sync ON |
| | FALSE AV-sync OFF |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid open file descriptor. |

| | |
|---|---|
| EINTERNAL | Internal error. |
| EINVAL | Illegal input parameter. |

## 12.2.10. AUDIO_SET_BYPASS_MODE

DESCRIPTION

This ioctl call asks the Audio Device to bypass the Audio decoder and forward the stream without decoding. This mode shall be used if streams that can't be handled by the DVB system shall be decoded. Dolby DigitalTM streams are automatically forwarded by the DVB subsystem if the hardware can handle it.

SYNOPSIS

```
int ioctl(int fd, int request = AUDIO_SET_BYPASS_MODE, boolean mode);
```

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| int request | Equals AUDIO_SET_BYPASS_MODE for this command. |
| boolean mode | Enables or disables the decoding of the current Audio stream in the DVB subsystem. |
| | TRUE Bypass is disabled |
| | FALSE Bypass is enabled |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid open file descriptor. |
| EINTERNAL | Internal error. |
| EINVAL | Illegal input parameter. |

## 12.2.11. AUDIO_CHANNEL_SELECT

DESCRIPTION

This ioctl call asks the Audio Device to select the requested channel if possible.

SYNOPSIS

int ioctl(int fd, int request = AUDIO_CHANNEL_SELECT, audio_channel_select_t);

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| int request | Equals AUDIO_CHANNEL_SELECT for this command. |
| audio_channel_select_t ch | Select the output format of the audio (mono left/right, stereo). |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid open file descriptor. |
| EINTERNAL | Internal error. |
| EINVAL | Illegal input parameter ch. |

## 12.2.12. AUDIO_GET_STATUS

DESCRIPTION

This ioctl call asks the Audio Device to return the current state of the Audio Device.

SYNOPSIS

int ioctl(int fd, int request = AUDIO_GET_STATUS, struct audio_status *status);

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |

| | |
|---|---|
| int request | Equals AUDIO_GET_STATUS for this command. |
| struct audio_status *status | Returns the current state of Audio Device. |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid open file descriptor. |
| EINTERNAL | Internal error. |
| EFAULT | status points to invalid address. |

## 12.2.13. AUDIO_GET_CAPABILITIES

DESCRIPTION

This ioctl call asks the Audio Device to tell us about the decoding capabilities of the audio hardware.

SYNOPSIS

int ioctl(int fd, int request = AUDIO_GET_CAPABILITIES, unsigned int ⋆cap);

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| int request | Equals AUDIO_GET_CAPABILITIES for this command. |
| unsigned int *cap | Returns a bit array of supported sound formats. |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid open file descriptor. |
| EINTERNAL | Internal error. |
| EFAULT | cap points to an invalid address. |

## 12.2.14. AUDIO_CLEAR_BUFFER

DESCRIPTION

This ioctl call asks the Audio Device to clear all software and hardware buffers of the audio decoder device.

SYNOPSIS

```
int ioctl(int fd, int request = AUDIO_CLEAR_BUFFER);
```

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| int request | Equals AUDIO_CLEAR_BUFFER for this command. |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid open file descriptor. |
| EINTERNAL | Internal error. |

## 12.2.15. AUDIO_SET_ID

DESCRIPTION

This ioctl selects which sub-stream is to be decoded if a program or system stream is sent to the video device. If no audio stream type is set the id has to be in [0xC0,0xDF] for MPEG sound, in [0x80,0x87] for AC3 and in [0xA0,0xA7] for LPCM. More specifications may follow for other stream types. If the stream type is set the id just specifies the substream id of the audio stream and only the first 5 bits are recognized.

SYNOPSIS

```
int ioctl(int fd, int request = AUDIO_SET_ID, int id);
```

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| int request | Equals AUDIO_SET_ID for this command. |
| int id | audio sub-stream id |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid open file descriptor. |
| EINTERNAL | Internal error. |
| EINVAL | Invalid sub-stream id. |

## 12.2.16. AUDIO_SET_MIXER

DESCRIPTION

This ioctl lets you adjust the mixer settings of the audio decoder.

SYNOPSIS

int ioctl(int fd, int request = AUDIO_SET_MIXER, audio_mixer_t *mix);

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| int request | Equals AUDIO_SET_ID for this command. |
| audio_mixer_t *mix | mixer settings. |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid open file descriptor. |
| EINTERNAL | Internal error. |
| EFAULT | mix points to an invalid address. |

## 12.2.17. AUDIO_SET_STREAMTYPE

DESCRIPTION

This ioctl tells the driver which kind of audio stream to expect. This is useful if the stream offers several audio sub-streams like LPCM and AC3.

SYNOPSIS

```
int ioctl(fd, int request = AUDIO_SET_STREAMTYPE, int type);
```

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| int request | Equals AUDIO_SET_STREAMTYPE for this command. |
| int type | stream type |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid open file descriptor |
| EINVAL | type is not a valid or supported stream type. |

## 12.2.18. AUDIO_SET_EXT_ID

DESCRIPTION

This ioctl can be used to set the extension id for MPEG streams in DVD playback. Only the first 3 bits are recognized.

SYNOPSIS

```
int ioctl(fd, int request = AUDIO_SET_EXT_ID, int id);
```

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |

| | |
|---|---|
| int request | Equals AUDIO_SET_EXT_ID for this command. |
| int id | audio sub_stream_id |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid open file descriptor |
| EINVAL | id is not a valid id. |

## 12.2.19. AUDIO_SET_ATTRIBUTES

DESCRIPTION

This ioctl is intended for DVD playback and allows you to set certain information about the audio stream.

SYNOPSIS

```
int ioctl(fd, int request = AUDIO_SET_ATTRIBUTES, audio_attributes_t attr );
```

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| int request | Equals AUDIO_SET_ATTRIBUTES for this command. |
| audio_attributes_t attr | audio attributes according to section ?? |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid open file descriptor |
| EINVAL | attr is not a valid or supported attribute setting. |

## 12.2.20. AUDIO_SET_KARAOKE

DESCRIPTION

This ioctl allows one to set the mixer settings for a karaoke DVD.

SYNOPSIS

```
int ioctl(fd, int request = AUDIO_SET_STREAMTYPE, audio_karaoke_t *karaoke);
```

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |
| int request | Equals AUDIO_SET_STREAMTYPE for this command. |
| audio_karaoke_t *karaoke | karaoke settings according to section ??. |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid open file descriptor |
| EINVAL | karaoke is not a valid or supported karaoke setting. |

# Chapter 13. DVB CA Device

The DVB CA device controls the conditional access hardware. It can be accessed through */dev/dvb/adapter0/ca0*. Data types and and ioctl definitions can be accessed by including *linux/dvb/ca.h* in your application.

## 13.1. CA Data Types

### 13.1.1. ca_slot_info_t

```
/* slot interface types and info */

typedef struct ca_slot_info_s {
 int num;                /* slot number */

 int type;            /* CA interface this slot supports */
#define CA_CI            1  /* CI high level interface */
#define CA_CI_LINK       2  /* CI link layer level interface */
#define CA_CI_PHYS       4  /* CI physical layer level interface */
#define CA_SC          128  /* simple smart card interface */

 unsigned int flags;
#define CA_CI_MODULE_PRESENT 1 /* module (or card) inserted */
#define CA_CI_MODULE_READY   2
} ca_slot_info_t;
```

### 13.1.2. ca_descr_info_t

```
typedef struct ca_descr_info_s {
 unsigned int num;  /* number of available descramblers (keys) */
 unsigned int type; /* type of supported scrambling system */
#define CA_ECD           1
#define CA_NDS           2
#define CA_DSS           4
} ca_descr_info_t;
```

### 13.1.3. ca_cap_t

```
typedef struct ca_cap_s {
 unsigned int slot_num;  /* total number of CA card and module slots */
 unsigned int slot_type; /* OR of all supported types */
 unsigned int descr_num; /* total number of descrambler slots (keys) */
 unsigned int descr_type;/* OR of all supported types */
} ca_cap_t;
```

### 13.1.4. ca_msg_t

```
/* a message to/from a CI-CAM */
typedef struct ca_msg_s {
 unsigned int index;
 unsigned int type;
 unsigned int length;
 unsigned char msg[256];
} ca_msg_t;
```

### 13.1.5. ca_descr_t

```
typedef struct ca_descr_s {
 unsigned int index;
 unsigned int parity;
 unsigned char cw[8];
} ca_descr_t;
```

## 13.2. CA Function Calls

### 13.2.1. open()

DESCRIPTION

This system call opens a named ca device (e.g. /dev/ost/ca) for subsequent use. When an open() call has succeeded, the device will be ready for use. The significance of blocking or non-blocking mode is described in the documentation for functions where there is a difference. It does not affect the semantics of the open() call itself. A device opened in blocking mode can later be put into non-blocking mode (and vice versa) using the F_SETFL command of the fcntl system call. This is a standard system call, documented in the Linux manual page for fcntl. Only one user can open the CA Device in O_RDWR mode. All other attempts to open the device in this mode will fail, and an error code will be returned.

SYNOPSIS

| |
|---|
| int open(const char *deviceName, int flags); |

PARAMETERS

| | |
|---|---|
| const char *deviceName | Name of specific video device. |
| int flags | A bit-wise OR of the following flags: |
| | O_RDONLY read-only access |
| | O_RDWR read/write access |
| | O_NONBLOCK open in non-blocking mode |
| | (blocking mode is the default) |

ERRORS

| | |
|---|---|
| ENODEV | Device driver not loaded/available. |
| EINTERNAL | Internal error. |
| EBUSY | Device or resource busy. |
| EINVAL | Invalid argument. |

## 13.2.2. close()

DESCRIPTION

| |
|---|
| This system call closes a previously opened audio device. |

SYNOPSIS

| |
|---|
| int close(int fd); |

PARAMETERS

| | |
|---|---|
| int fd | File descriptor returned by a previous call to open(). |

ERRORS

| | |
|---|---|
| EBADF | fd is not a valid open file descriptor. |

# Chapter 14. DVB Network API

The DVB net device enables feeding of MPE (multi protocol encapsulation) packets received via DVB into the Linux network protocol stack, e.g. for internet via satellite applications. It can be accessed through */dev/dvb/adapter0/net0*. Data types and and ioctl definitions can be accessed by including *linux/dvb/net.h* in your application.

## 14.1. DVB Net Data Types

To be written...

# Chapter 15. Kernel Demux API

The kernel demux API defines a driver-internal interface for registering low-level, hardware specific driver to a hardware independent demux layer. It is only of interest for DVB device driver writers. The header file for this API is named *demux.h* and located in *drivers/media/dvb/dvb-core*.

Maintainer note: This section must be reviewed. It is probably out of date.

## 15.1. Kernel Demux Data Types

### 15.1.1. dmx_success_t

```
typedef enum {
  DMX_OK = 0, /* Received Ok */
  DMX_LENGTH_ERROR, /* Incorrect length */
  DMX_OVERRUN_ERROR, /* Receiver ring buffer overrun */
  DMX_CRC_ERROR, /* Incorrect CRC */
  DMX_FRAME_ERROR, /* Frame alignment error */
  DMX_FIFO_ERROR, /* Receiver FIFO overrun */
  DMX_MISSED_ERROR /* Receiver missed packet */
} dmx_success_t;
```

### 15.1.2. TS filter types

```
/*--------------------------------------------------------------------
/* TS packet reception */
/*--------------------------------------------------------------------

/* TS filter type for set_type() */

#define TS_PACKET       1   /* send TS packets (188 bytes) to callback (c
#define TS_PAYLOAD_ONLY 2   /* in case TS_PACKET is set, only send the TS
   payload (<=184 bytes per packet) to callback */
#define TS_DECODER      4   /* send stream to built-in decoder (if preser
```

### 15.1.3. dmx_ts_pes_t

The structure

```
typedef enum
{
        DMX_TS_PES_AUDIO,  /* also send packets to audio decoder (if it exists) */
        DMX_TS_PES_VIDEO,  /*                 */
        DMX_TS_PES_TELETEXT,
        DMX_TS_PES_SUBTITLE,
        DMX_TS_PES_PCR,
        DMX_TS_PES_OTHER,
} dmx_ts_pes_t;
```

describes the PES type for filters which write to a built-in decoder. The correspond (and should be kept identical) to the types in the demux device.

```
struct dmx_ts_feed_s {
        int is_filtering; /* Set to non-zero when filtering in progress */
        struct dmx_demux_s* parent; /* Back-pointer */
        void* priv; /* Pointer to private data of the API client */
        int (*set) (struct dmx_ts_feed_s* feed,
                    __u16 pid,
                    size_t circular_buffer_size,
                    int descramble,
                    struct timespec timeout);
        int (*start_filtering) (struct dmx_ts_feed_s* feed);
        int (*stop_filtering) (struct dmx_ts_feed_s* feed);
        int (*set_type) (struct dmx_ts_feed_s* feed,
                         int type,
                         dmx_ts_pes_t pes_type);
};

typedef struct dmx_ts_feed_s dmx_ts_feed_t;

/*--------------------------------------------------------------------------*/
/* PES packet feed API                                                      */
/*--------------------------------------------------------------------------*/

/* Callback functions */

typedef int (*dmx_ts_cb) ( __u8 * buffer1,
                           size_t buffer1_length,
                           __u8 * buffer2,
                           size_t buffer2_length,
                           dmx_ts_feed_t* source,
                           dmx_success_t success);

typedef struct dmx_pes_filter_s {
        int is_filtering; /* Set to non-zero when filtering in progress */
        struct dmx_pes_feed_s* parent; /* Back-pointer */
        void* priv; /* Pointer to private data of the API client */
} dmx_pes_filter_t;

typedef struct dmx_pes_feed_s {
        int is_filtering; /* Set to non-zero when filtering in progress */
        struct dmx_demux_s* parent; /* Back-pointer */
        void* priv; /* Pointer to private data of the API client */
        int (*set) (struct dmx_pes_feed_s* feed,
                    __u16 pid,
                    size_t circular_buffer_size,
                    int descramble,
                    struct timespec timeout);
        int (*start_filtering) (struct dmx_pes_feed_s* feed);
        int (*stop_filtering) (struct dmx_pes_feed_s* feed);
        int (*allocate_filter) (struct dmx_pes_feed_s* feed,
                                dmx_pes_filter_t** filter);
        int (*release_filter) (struct dmx_pes_feed_s* feed,
                               dmx_pes_filter_t* filter);
} dmx_pes_feed_t;
```

```
 */    dmx_success_t success);

typedef int (dmx_section_cb) ( __u8 * buffer1,
                               size_t buffer1_len,
                               __u8 * buffer2,
                               size_t buffer2_len,
                               dmx_section_filter_t * source,
                               dmx_success_t success);

typedef int (dmx_ts_cb) ( __u8 * buffer1,
                          size_t buffer1_len,
                          __u8 * buffer2,
                          size_t buffer2_len,
                          dmx_ts_feed_t * source,
                          dmx_success_t success);
```

### 15.1.4. demux_t

```
/*
 * DMX_FE_ENTRY(): Casts elements in the list of registered
 *                 front-ends from the generic type struct list_head
 *                 to the type * dmx_frontend_t
 *                 ------------------------------------------------
 */
#define DMX_FE_ENTRY(list) list_entry(list, dmx_frontend_t, connectivity_

struct dmx_demux_s {
        /* The following char* fields point to NULL terminated strings */
        char* id;               /* Unique demux identifier */
        char* vendor;           /* Name of the demux vendor */
        char* model;            /* Name of the demux model */
        __u32 capabilities;     /* Bitfield of capability flags */
        dmx_frontend_t* frontend;    /* Front-end connected to the demux */
        struct list_head reg_list;   /* List of registered demuxes */
        void* priv;             /* Pointer to private data of the API clien
        int users;              /* Number of users */
        int (*open) (struct dmx_demux_s* demux);
        int (*close) (struct dmx_demux_s* demux);
        int (*write) (struct dmx_demux_s* demux, const char* buf, size_t count);
        int (*allocate_ts_feed) (struct dmx_demux_s* demux,
                                 dmx_ts_feed_t** feed,
                                 dmx_ts_cb* callback);
        int (*release_ts_feed) (struct dmx_demux_s* demux,
                                dmx_ts_feed_t* feed);
        int (*allocate_pes_feed) (struct dmx_demux_s* demux,
                                  dmx_pes_feed_t** feed,
                                  dmx_pes_cb* callback);
        int (*release_pes_feed) (struct dmx_demux_s* demux,
                                 dmx_pes_feed_t* feed);
        int (*allocate_section_feed) (struct dmx_demux_s* demux,
```

```
                dmx_section_feed_t** feed,
                dmx_section_cb callback);
        int (*release_section_feed) (struct dmx_demux_s* demux,
                dmx_section_feed_t* feed);
        int (*descramble_mac_address) (struct dmx_demux_s* demux,
           __u8* buffer1,
           size_t buffer1_length,
           __u8* buffer2,
           size_t buffer2_length,
           __u16 pid);
        int (*descramble_section_payload) (struct dmx_demux_s* demux,
               __u8* buffer1,
               size_t buffer1_length,
               __u8* buffer2, size_t buffer2_length,
               __u16 pid);
        int (*add_frontend) (struct dmx_demux_s* demux,
               dmx_frontend_t* frontend);
        int (*remove_frontend) (struct dmx_demux_s* demux,
           dmx_frontend_t* frontend);
        struct list_head* (*get_frontends) (struct dmx_demux_s* demux);
        int (*connect_frontend) (struct dmx_demux_s* demux,
            dmx_frontend_t* frontend);
        int (*disconnect_frontend) (struct dmx_demux_s* demux);


        /* added because js cannot keep track of these himself */
        int (*get_pes_pids) (struct dmx_demux_s* demux, __u16 *pids);
       };
       typedef struct dmx_demux_s dmx_demux_t;
```

### 15.1.5. Demux directory

```
/*
 * DMX_DIR_ENTRY(): Casts elements in the list of registered
 * demuxes from the generic type struct list_head* to the type dmx_demux_
 *.
 */

#define DMX_DIR_ENTRY(list) list_entry(list, dmx_demux_t, reg_list)

int dmx_register_demux (dmx_demux_t* demux);
int dmx_unregister_demux (dmx_demux_t* demux);
struct list_head* dmx_get_demuxes (void);
```

## 15.2. Demux Directory API



The demux directory is a Linux kernel-wide facility for registering and accessing the MPEG-2 TS demuxes in the system. Run-time registering and unregistering of demux drivers is possible using this API.

SYNOPSIS

| int dmx_register_demux ( dmx_demux_t ⋆demux ) |
|---|

PARAMETERS

| dmx_demux_t* demux | Pointer to the demux structure. |
|---|---|

RETURNS

| 0 | The function was completed without errors. |
|---|---|
| -EEXIST | A demux with the same value of the id field already stored in the directory. |
| -ENOSPC | No space left in the directory. |

## 15.2.2. dmx_unregister_demux()

DESCRIPTION

| This function is called to indicate that the given demux interface is no longer available. The caller of this function is responsible for freeing the memory of the demux structure, if it was dynamically allocated before calling dmx_register_demux(). The cleanup_module() function of the kernel module that contains the demux driver should call this function. Note that this function fails if the demux is currently in use, i.e., release_demux() has not been called for the interface. |
|---|

SYNOPSIS

| int dmx_unregister_demux ( dmx_demux_t ⋆demux ) |
|---|

PARAMETERS

| dmx_demux_t* demux | Pointer to the demux structure which is to be unregistered. |
|---|---|

RETURNS

| 0 | The function was completed without errors. |
|---|---|

| | |
|---|---|
| ENODEV | The specified demux is not registered in the demux directory. |
| EBUSY | The specified demux is currently in use. |

### 15.2.3. dmx_get_demuxes()

DESCRIPTION

Provides the caller with the list of registered demux interfaces, using the standard list structure defined in the include file linux/list.h. The include file demux.h defines the macro DMX_DIR_ENTRY() for converting an element of the generic type struct list_head* to the type dmx_demux_t*. The caller must not free the memory of any of the elements obtained via this function call.

SYNOPSIS

```
struct list_head *dmx_get_demuxes ()
```

PARAMETERS

| | |
|---|---|
| none | |

RETURNS

| | |
|---|---|
| struct list_head * | A list of demux interfaces, or NULL in the case of an empty list. |

## 15.3. Demux API

The demux API should be implemented for each demux in the system. It is used to select the TS source of a demux and to manage the demux resources. When the demux client allocates a resource via the demux API, it receives a pointer to the API of that resource.

Each demux receives its TS input from a DVB front-end or from memory, as set via the demux API. In a system with more than one front-end, the API can be used to select one of the DVB front-ends as a TS source for a demux, unless this is fixed in

the HW platform. The demux API only controls front-ends regarding their connections with demuxes; the APIs used to set the other front-end parameters, such as tuning, are not defined in this document.

The functions that implement the abstract interface demux should be defined static or module private and registered to the Demux Directory for external access. It is not necessary to implement every function in the demux_t struct, however (for example, a demux interface might support Section filtering, but not TS or PES filtering). The API client is expected to check the value of any function pointer before calling the function: the value of NULL means “function not available”.

Whenever the functions of the demux API modify shared data, the possibilities of lost update and race condition problems should be addressed, e.g. by protecting parts of code with mutexes. This is especially important on multi-processor hosts.

Note that functions called from a bottom half context must not sleep, at least in the 2.2.x kernels. Even a simple memory allocation can result in a kernel thread being put to sleep if swapping is needed. For example, the Linux kernel calls the functions of a network device interface from a bottom half context. Thus, if a demux API function is called from network device code, the function must not sleep.

## 15.3.1. open()

DESCRIPTION

| |
|---|
| This function reserves the demux for use by the caller and, if necessary, initializes the demux. When the demux is no longer needed, the function close() should be called. It should be possible for multiple clients to access the demux at the same time. Thus, the function implementation should increment the demux usage count when open() is called and decrement it when close() is called. |

SYNOPSIS

```
int open ( demux_t* demux );
```

PARAMETERS

| | |
|---|---|
| demux_t* demux | Pointer to the demux API and instance data. |

RETURNS

| | |
|---|---|
| 0 | The function was completed without errors. |

| | |
|---|---|
| -EUSERS | Maximum usage count reached. |
| -EINVAL | Bad parameter. |

## 15.3.2. close()

DESCRIPTION

This function reserves the demux for use by the caller and, if necessary, initializes the demux. When the demux is no longer needed, the function close() should be called. It should be possible for multiple clients to access the demux at the same time. Thus, the function implementation should increment the demux usage count when open() is called and decrement it when close() is called.

SYNOPSIS

```
int close(demux_t* demux);
```

PARAMETERS

| | |
|---|---|
| demux_t* demux | Pointer to the demux API and instance data. |

RETURNS

| | |
|---|---|
| 0 | The function was completed without errors. |
| -ENODEV | The demux was not in use. |
| -EINVAL | Bad parameter. |

## 15.3.3. write()

DESCRIPTION

This function provides the demux driver with a memory buffer containing TS packets. Instead of receiving TS packets from the DVB front-end, the demux driver software will read packets from memory. Any clients of this demux with active TS, PES or Section filters will receive filtered data via the Demux callback API (see 0). The function returns when all the data in the buffer has been consumed by the demux. Demux hardware typically cannot read TS from memory. If this is the case, memory-based filtering has to be implemented entirely in software.

SYNOPSIS

```
int write(demux_t⋆ demux, const char⋆ buf, size_t count);
```

PARAMETERS

| | |
|---|---|
| demux_t* demux | Pointer to the demux API and instance data. |
| const char* buf | Pointer to the TS data in kernel-space memory. |
| size_t length | Length of the TS data. |

RETURNS

| | |
|---|---|
| 0 | The function was completed without errors. |
| -ENOSYS | The command is not implemented. |
| -EINVAL | Bad parameter. |

## 15.3.4. allocate_ts_feed()

DESCRIPTION

Allocates a new TS feed, which is used to filter the TS packets carrying a certain PID. The TS feed normally corresponds to a hardware PID filter on the demux chip.

SYNOPSIS

```
int allocate_ts_feed(dmx_demux_t⋆ demux, dmx_ts_feed_t⋆⋆ feed, dmx_ts_cb callback);
```

PARAMETERS

| | |
|---|---|
| demux_t* demux | Pointer to the demux API and instance data. |
| dmx_ts_feed_t** feed | Pointer to the TS feed API and instance data. |
| dmx_ts_cb callback | Pointer to the callback function for passing received TS packet |

RETURNS

| | |
|---|---|
| 0 | The function was completed without errors. |
| -EBUSY | No more TS feeds available. |
| -ENOSYS | The command is not implemented. |
| -EINVAL | Bad parameter. |

## 15.3.5. release_ts_feed()

DESCRIPTION

| |
|---|
| Releases the resources allocated with allocate_ts_feed(). Any filtering in progress on the TS feed should be stopped before calling this function. |

SYNOPSIS

| |
|---|
| int release_ts_feed(dmx_demux_t⋆ demux, dmx_ts_feed_t⋆ feed); |

PARAMETERS

| | |
|---|---|
| demux_t* demux | Pointer to the demux API and instance data. |
| dmx_ts_feed_t* feed | Pointer to the TS feed API and instance data. |

RETURNS

| | |
|---|---|
| 0 | The function was completed without errors. |
| -EINVAL | Bad parameter. |

## 15.3.6. allocate_section_feed()

DESCRIPTION

Allocates a new section feed, i.e. a demux resource for filtering and receiving sections. On platforms with hardware support for section filtering, a section feed is directly mapped to the demux HW. On other platforms, TS packets are first PID filtered in hardware and a hardware section filter then emulated in software. The caller obtains an API pointer of type dmx_section_feed_t as an out parameter. Using this API the caller can set filtering parameters and start receiving sections.

SYNOPSIS

int allocate_section_feed(dmx_demux_t⋆ demux, dmx_section_feed_t ⋆⋆feed, dmx_section_cb callback);

PARAMETERS

| | |
|---|---|
| demux_t *demux | Pointer to the demux API and instance data. |
| dmx_section_feed_t **feed | Pointer to the section feed API and instance data. |
| dmx_section_cb callback | Pointer to the callback function for passing received sections. |

RETURNS

| | |
|---|---|
| 0 | The function was completed without errors. |
| -EBUSY | No more section feeds available. |
| -ENOSYS | The command is not implemented. |
| -EINVAL | Bad parameter. |

## 15.3.7. release_section_feed()

DESCRIPTION

Releases the resources allocated with allocate_section_feed(), including allocated filters. Any filtering in progress on the section feed should be stopped before calling this function.

SYNOPSIS

```
int release_section_feed(dmx_demux_t⋆ demux, dmx_section_feed_t ⋆feed);
```

PARAMETERS

| | |
|---|---|
| demux_t *demux | Pointer to the demux API and instance data. |
| dmx_section_feed_t *feed | Pointer to the section feed API and instance data. |

RETURNS

| | |
|---|---|
| 0 | The function was completed without errors. |
| -EINVAL | Bad parameter. |

## 15.3.8. descramble_mac_address()

DESCRIPTION

This function runs a descrambling algorithm on the destination MAC address field of a DVB Datagram Section, replacing the original address with its un-encrypted version. Otherwise, the description on the function descramble_section_payload() applies also to this function.

SYNOPSIS

```
int descramble_mac_address(dmx_demux_t⋆ demux, __u8 ⋆buffer1, size_t buffer1_length, __u8 ⋆buffer2, size_t buffer2_length, __u16 pid);
```

PARAMETERS

| | |
|---|---|
| dmx_demux_t *demux | Pointer to the demux API and instance data. |
| __u8 *buffer1 | Pointer to the first byte of the section. |
| size_t buffer1_length | Length of the section data, including headers and CRC, in buffer1. |

| | |
|---|---|
| __u8* buffer2 | Pointer to the tail of the section data, or NULL. The pointer has a non-NULL value if the section wraps past the end of a circular buffer. |
| size_t buffer2_length | Length of the section data, including headers and CRC, in buffer2. |
| __u16 pid | The PID on which the section was received. Useful for obtaining the descrambling key, e.g. from a DVB Common Access facility. |

RETURNS

| | |
|---|---|
| 0 | The function was completed without errors. |
| -ENOSYS | No descrambling facility available. |
| -EINVAL | Bad parameter. |

## 15.3.9. descramble_section_payload()

DESCRIPTION

| |
|---|
| This function runs a descrambling algorithm on the payload of a DVB Datagram Section, replacing the original payload with its un-encrypted version. The function will be called from the demux API implementation; the API client need not call this function directly. Section-level scrambling algorithms are currently standardized only for DVB-RCC (return channel over 2-directional cable TV network) systems. For all other DVB networks, encryption schemes are likely to be proprietary to each data broadcaster. Thus, it is expected that this function pointer will have the value of NULL (i.e., function not available) in most demux API implementations. Nevertheless, it should be possible to use the function pointer as a hook for dynamically adding a “plug-in” descrambling facility to a demux driver. |
| While this function is not needed with hardware-based section descrambling, the descramble_section_payload function pointer can be used to override the default hardware-based descrambling algorithm: if the function pointer has a non-NULL value, the corresponding function should be used instead of any descrambling hardware. |

SYNOPSIS

int descramble_section_payload(dmx_demux_t⋆ demux, __u8 ⋆buffer1, size_t buffer1_length, __u8 ⋆buffer2, size_t buffer2_length, __u16 pid);

PARAMETERS

| | |
|---|---|
| dmx_demux_t *demux | Pointer to the demux API and instance data. |
| __u8 *buffer1 | Pointer to the first byte of the section. |
| size_t buffer1_length | Length of the section data, including headers and CRC, in buffer1. |
| __u8 *buffer2 | Pointer to the tail of the section data, or NULL. The pointer has a non-NULL value if the section wraps past the end of a circular buffer. |
| size_t buffer2_length | Length of the section data, including headers and CRC, in buffer2. |
| __u16 pid | The PID on which the section was received. Useful for obtaining the descrambling key, e.g. from a DVB Common Access facility. |

RETURNS

| | |
|---|---|
| 0 | The function was completed without errors. |
| -ENOSYS | No descrambling facility available. |
| -EINVAL | Bad parameter. |

## 15.3.10. add_frontend()

DESCRIPTION

Registers a connectivity between a demux and a front-end, i.e., indicates that the demux can be connected via a call to connect_frontend() to use the given front-end as a TS source. The client of this function has to allocate dynamic or static memory for the frontend structure and initialize its fields before calling this function. This function is normally called during the driver initialization. The caller must not free the memory of the frontend struct before successfully calling remove_frontend().

SYNOPSIS

| int add_frontend(dmx_demux_t ⋆demux, dmx_frontend_t ⋆frontend); |
|---|

PARAMETERS

| dmx_demux_t* demux | Pointer to the demux API and instance data. |
|---|---|
| dmx_frontend_t* frontend | Pointer to the front-end instance data. |

RETURNS

| 0 | The function was completed without errors. |
|---|---|
| -EEXIST | A front-end with the same value of the id field already registered. |
| -EINUSE | The demux is in use. |
| -ENOMEM | No more front-ends can be added. |
| -EINVAL | Bad parameter. |

## 15.3.11. remove_frontend()

DESCRIPTION

| Indicates that the given front-end, registered by a call to add_frontend(), can no longer be connected as a TS source by this demux. The function should be called when a front-end driver or a demux driver is removed from the system. If the front-end is in use, the function fails with the return value of -EBUSY. After successfully calling this function, the caller can free the memory of the frontend struct if it was dynamically allocated before the add_frontend() operation. |
|---|

SYNOPSIS

| int remove_frontend(dmx_demux_t⋆ demux, dmx_frontend_t⋆ frontend); |
|---|

PARAMETERS

| dmx_demux_t* demux | Pointer to the demux API and instance data. |
|---|---|
| dmx_frontend_t* frontend | Pointer to the front-end instance data. |

RETURNS

| | |
|---|---|
| 0 | The function was completed without errors. |
| -EINVAL | Bad parameter. |
| -EBUSY | The front-end is in use, i.e. a call to connect_frontend() has not been followed by a call to disconnect_frontend(). |

## 15.3.12. get_frontends()

DESCRIPTION

| |
|---|
| Provides the APIs of the front-ends that have been registered for this demux. Any of the front-ends obtained with this call can be used as a parameter for connect_frontend(). |
| The include file demux.h contains the macro DMX_FE_ENTRY() for converting an element of the generic type struct list_head* to the type dmx_frontend_t*. The caller must not free the memory of any of the elements obtained via this function call. |

SYNOPSIS

```
struct list_head⋆ get_frontends(dmx_demux_t⋆ demux);
```

PARAMETERS

| | |
|---|---|
| dmx_demux_t* demux | Pointer to the demux API and instance data. |

RETURNS

| | |
|---|---|
| dmx_demux_t* | A list of front-end interfaces, or NULL in the case of an empty list. |

## 15.3.13. connect_frontend()

DESCRIPTION

| |
|---|
| Connects the TS output of the front-end to the input of the demux. A demux can only be connected to a front-end registered to the demux with the function add_frontend(). |
| It may or may not be possible to connect multiple demuxes to the same front-end, depending on the capabilities of the HW platform. When not used, the front-end should be released by calling disconnect_frontend(). |

SYNOPSIS

| |
|---|
| int connect_frontend(dmx_demux_t⋆ demux, dmx_frontend_t⋆ frontend); |

PARAMETERS

| | |
|---|---|
| dmx_demux_t* demux | Pointer to the demux API and instance data. |
| dmx_frontend_t* frontend | Pointer to the front-end instance data. |

RETURNS

| | |
|---|---|
| 0 | The function was completed without errors. |
| -EINVAL | Bad parameter. |
| -EBUSY | The front-end is in use. |

## 15.3.14. disconnect_frontend()

DESCRIPTION

| |
|---|
| Disconnects the demux and a front-end previously connected by a connect_frontend() call. |

SYNOPSIS

| |
|---|
| int disconnect_frontend(dmx_demux_t⋆ demux); |

PARAMETERS

| | |
|---|---|
| dmx_demux_t* demux | Pointer to the demux API and instance data. |

RETURNS

| | |
|---|---|
| 0 | The function was completed without errors. |
| -EINVAL | Bad parameter. |

# 15.4. Demux Callback API

This kernel-space API comprises the callback functions that deliver filtered data to the demux client. Unlike the other APIs, these API functions are provided by the client and called from the demux code.

The function pointers of this abstract interface are not packed into a structure as in the other demux APIs, because the callback functions are registered and used independent of each other. As an example, it is possible for the API client to provide several callback functions for receiving TS packets and no callbacks for PES packets or sections.

The functions that implement the callback API need not be re-entrant: when a demux driver calls one of these functions, the driver is not allowed to call the function again before the original call returns. If a callback is triggered by a hardware interrupt, it is recommended to use the Linux “bottom half” mechanism or start a tasklet instead of making the callback function call directly from a hardware interrupt.

## 15.4.1. dmx_ts_cb()

DESCRIPTION

| |
|---|
| This function, provided by the client of the demux API, is called from the demux code. The function is only called when filtering on this TS feed has been enabled using the start_filtering() function. |
| Any TS packets that match the filter settings are copied to a circular buffer. The filtered TS packets are delivered to the client using this callback function. The size of the circular buffer is controlled by the circular_buffer_size parameter of the set() function in the TS Feed API. It is expected that the buffer1 and buffer2 callback parameters point to addresses within the circular buffer, but other implementations are also possible. Note that the called party should not try to free the memory the buffer1 and buffer2 parameters point to. |

When this function is called, the buffer1 parameter typically points to the start of the first undelivered TS packet within a circular buffer. The buffer2 buffer parameter is normally NULL, except when the received TS packets have crossed the last address of the circular buffer and ”wrapped” to the beginning of the buffer. In the latter case the buffer1 parameter would contain an address within the circular buffer, while the buffer2 parameter would contain the first address of the circular buffer.

The number of bytes delivered with this function (i.e. buffer1_length + buffer2_length) is usually equal to the value of callback_length parameter given in the set() function, with one exception: if a timeout occurs before receiving callback_length bytes of TS data, any undelivered packets are immediately delivered to the client by calling this function. The timeout duration is controlled by the set() function in the TS Feed API.

If a TS packet is received with errors that could not be fixed by the TS-level forward error correction (FEC), the Transport_error_indicator flag of the TS packet header should be set. The TS packet should not be discarded, as the error can possibly be corrected by a higher layer protocol. If the called party is slow in processing the callback, it is possible that the circular buffer eventually fills up. If this happens, the demux driver should discard any TS packets received while the buffer is full. The error should be indicated to the client on the next callback by setting the success parameter to the value of DMX_OVERRUN_ERROR.

The type of data returned to the callback can be selected by the new function int (*set_type) (struct dmx_ts_feed_s* feed, int type, dmx_ts_pes_t pes_type) which is part of the dmx_ts_feed_s struct (also cf. to the include file ost/demux.h) The type parameter decides if the raw TS packet (TS_PACKET) or just the payload (TS_PACKET—TS_PAYLOAD_ONLY) should be returned. If additionally the TS_DECODER bit is set the stream will also be sent to the hardware MPEG decoder. In this case, the second flag decides as what kind of data the stream should be interpreted. The possible choices are one of DMX_TS_PES_AUDIO, DMX_TS_PES_VIDEO, DMX_TS_PES_TELETEXT, DMX_TS_PES_SUBTITLE, DMX_TS_PES_PCR, or DMX_TS_PES_OTHER.

SYNOPSIS

int dmx_ts_cb(__u8⋆ buffer1, size_t buffer1_length, __u8⋆ buffer2, size_t buffer2_length, dmx_ts_feed_t⋆ source, dmx_success_t success);

PARAMETERS

| | |
|---|---|
| __u8* buffer1 | Pointer to the start of the filtered TS packets. |

| | |
|---|---|
| size_t buffer1_length | Length of the TS data in buffer1. |
| __u8* buffer2 | Pointer to the tail of the filtered TS packets, or NULL. |
| size_t buffer2_length | Length of the TS data in buffer2. |
| dmx_ts_feed_t* source | Indicates which TS feed is the source of the callback. |
| dmx_success_t success | Indicates if there was an error in TS reception. |

RETURNS

| | |
|---|---|
| 0 | Continue filtering. |
| -1 | Stop filtering - has the same effect as a call to stop_filtering() on the TS Feed API. |

## 15.4.2. dmx_section_cb()

DESCRIPTION

This function, provided by the client of the demux API, is called from the demux code. The function is only called when filtering of sections has been enabled using the function start_filtering() of the section feed API. When the demux driver has received a complete section that matches at least one section filter, the client is notified via this callback function. Normally this function is called for each received section; however, it is also possible to deliver multiple sections with one callback, for example when the system load is high. If an error occurs while receiving a section, this function should be called with the corresponding error type set in the success field, whether or not there is data to deliver. The Section Feed implementation should maintain a circular buffer for received sections. However, this is not necessary if the Section Feed API is implemented as a client of the TS Feed API, because the TS Feed implementation then buffers the received data. The size of the circular buffer can be configured using the set() function in the Section Feed API. If there is no room in the circular buffer when a new section is received, the section must be discarded. If this happens, the value of the success parameter should be DMX_OVERRUN_ERROR on the next callback.

SYNOPSIS

| int dmx_section_cb(__u8⋆ buffer1, size_t buffer1_length, __u8⋆ buffer2, size_t buffer2_length, dmx_section_filter_t⋆ source, dmx_success_t success); |
|---|

PARAMETERS

| | |
|---|---|
| __u8* buffer1 | Pointer to the start of the filtered section, e.g. within the circular buffer of the demux driver. |
| size_t buffer1_length | Length of the filtered section data in buffer1, including headers and CRC. |
| __u8* buffer2 | Pointer to the tail of the filtered section data, or NULL. Useful to handle the wrapping of a circular buffer. |
| size_t buffer2_length | Length of the filtered section data in buffer2, including headers and CRC. |
| dmx_section_filter_t* filter | Indicates the filter that triggered the callback. |
| dmx_success_t success | Indicates if there was an error in section reception. |

RETURNS

| | |
|---|---|
| 0 | Continue filtering. |
| -1 | Stop filtering - has the same effect as a call to stop_filtering() on the Section Feed API. |

## 15.5. TS Feed API

A TS feed is typically mapped to a hardware PID filter on the demux chip. Using this API, the client can set the filtering properties to start/stop filtering TS packets on a particular TS feed. The API is defined as an abstract interface of the type dmx_ts_feed_t.

The functions that implement the interface should be defined static or module private. The client can get the handle of a TS feed API by calling the function allocate_ts_feed() in the demux API.

### 15.5.1. set()

DESCRIPTION

This function sets the parameters of a TS feed. Any filtering in progress on the TS feed must be stopped before calling this function.

SYNOPSIS

```
int set ( dmx_ts_feed_t⋆ feed, __u16 pid, size_t callback_length, size_t circular_buffer_size, int descramble, struct timespec timeout);
```

PARAMETERS

| | |
|---|---|
| dmx_ts_feed_t* feed | Pointer to the TS feed API and instance data. |
| __u16 pid | PID value to filter. Only the TS packets carrying the specified PID will be passed to the API client. |
| size_t callback_length | Number of bytes to deliver with each call to the dmx_ts_cb() callback function. The value of this parameter should be a multiple of 188. |
| size_t circular_buffer_size | Size of the circular buffer for the filtered TS packets. |
| int descramble | If non-zero, descramble the filtered TS packets. |
| struct timespec timeout | Maximum time to wait before delivering received TS packets to the client. |

RETURNS

| | |
|---|---|
| 0 | The function was completed without errors. |
| -ENOMEM | Not enough memory for the requested buffer size. |
| -ENOSYS | No descrambling facility available for TS. |
| -EINVAL | Bad parameter. |

## 15.5.2. start_filtering()

DESCRIPTION

Starts filtering TS packets on this TS feed, according to its settings. The PID value to filter can be set by the API client. All matching TS packets are delivered asynchronously to the client, using the callback function registered with allocate_ts_feed().

SYNOPSIS

```
int start_filtering(dmx_ts_feed_t* feed);
```

PARAMETERS

| | |
|---|---|
| dmx_ts_feed_t* feed | Pointer to the TS feed API and instance data. |

RETURNS

| | |
|---|---|
| 0 | The function was completed without errors. |
| -EINVAL | Bad parameter. |

## 15.5.3. stop_filtering()

DESCRIPTION

Stops filtering TS packets on this TS feed.

SYNOPSIS

```
int stop_filtering(dmx_ts_feed_t* feed);
```

PARAMETERS

| | |
|---|---|
| dmx_ts_feed_t* feed | Pointer to the TS feed API and instance data. |

RETURNS

| | |
|---|---|
| 0 | The function was completed without errors. |
| -EINVAL | Bad parameter. |

# 15.6. Section Feed API

A section feed is a resource consisting of a PID filter and a set of section filters. Using this API, the client can set the properties of a section feed and to start/stop filtering. The API is defined as an abstract interface of the type dmx_section_feed_t. The functions that implement the interface should be defined static or module private. The client can get the handle of a section feed API by calling the function allocate_section_feed() in the demux API.

On demux platforms that provide section filtering in hardware, the Section Feed API implementation provides a software wrapper for the demux hardware. Other platforms may support only PID filtering in hardware, requiring that TS packets are converted to sections in software. In the latter case the Section Feed API implementation can be a client of the TS Feed API.

# 15.7. set()

DESCRIPTION

This function sets the parameters of a section feed. Any filtering in progress on the section feed must be stopped before calling this function. If descrambling is enabled, the payload_scrambling_control and address_scrambling_control fields of received DVB datagram sections should be observed. If either one is non-zero, the section should be descrambled either in hardware or using the functions descramble_mac_address() and descramble_section_payload() of the demux API. Note that according to the MPEG-2 Systems specification, only the payloads of private sections can be scrambled while the rest of the section data must be sent in the clear.

SYNOPSIS

```
int set(dmx_section_feed_t* feed, __u16 pid, size_t circular_buffer_size, int descramble, int check_crc);
```

PARAMETERS

| | |
|---|---|
| dmx_section_feed_t* feed | Pointer to the section feed API and instance data. |
| __u16 pid | PID value to filter; only the TS packets carrying the specified PID will be accepted. |
| size_t circular_buffer_size | Size of the circular buffer for filtered sections. |
| int descramble | If non-zero, descramble any sections that are scrambled. |
| int check_crc | If non-zero, check the CRC values of filtered sections. |

RETURNS

| | |
|---|---|
| 0 | The function was completed without errors. |
| -ENOMEM | Not enough memory for the requested buffer size. |
| -ENOSYS | No descrambling facility available for sections. |
| -EINVAL | Bad parameters. |

## 15.8. allocate_filter()

DESCRIPTION

This function is used to allocate a section filter on the demux. It should only be called when no filtering is in progress on this section feed. If a filter cannot be allocated, the function fails with -ENOSPC. See in section ?? for the format of the section filter.

| |
|---|
| The bitfields filter_mask and filter_value should only be modified when no filtering is in progress on this section feed. filter_mask controls which bits of filter_value are compared with the section headers/payload. On a binary value of 1 in filter_mask, the corresponding bits are compared. The filter only accepts sections that are equal to filter_value in all the tested bit positions. Any changes to the values of filter_mask and filter_value are guaranteed to take effect only when the start_filtering() function is called next time. The parent pointer in the struct is initialized by the API implementation to the value of the feed parameter. The priv pointer is not used by the API implementation, and can thus be freely utilized by the caller of this function. Any data pointed to by the priv pointer is available to the recipient of the dmx_section_cb() function call. |
| While the maximum section filter length (DMX_MAX_FILTER_SIZE) is currently set at 16 bytes, hardware filters of that size are not available on all platforms. Therefore, section filtering will often take place first in hardware, followed by filtering in software for the header bytes that were not covered by a hardware filter. The filter_mask field can be checked to determine how many bytes of the section filter are actually used, and if the hardware filter will suffice. Additionally, software-only section filters can optionally be allocated to clients when all hardware section filters are in use. Note that on most demux hardware it is not possible to filter on the section_length field of the section header – thus this field is ignored, even though it is included in filter_value and filter_mask fields. |

SYNOPSIS

```
int allocate_filter(dmx_section_feed_t⋆ feed, dmx_section_filter_t⋆⋆ filter);
```

PARAMETERS

| | |
|---|---|
| dmx_section_feed_t* feed | Pointer to the section feed API and instance data. |
| dmx_section_filter_t** filter | Pointer to the allocated filter. |

RETURNS

| | |
|---|---|
| 0 | The function was completed without errors. |
| -ENOSPC | No filters of given type and length available. |
| -EINVAL | Bad parameters. |

# 15.9. release_filter()

DESCRIPTION

This function releases all the resources of a previously allocated section filter. The function should not be called while filtering is in progress on this section feed. After calling this function, the caller should not try to dereference the filter pointer.

SYNOPSIS

```
int release_filter ( dmx_section_feed_t⋆ feed, dmx_section_filter_t⋆ filter);
```

PARAMETERS

| | |
|---|---|
| dmx_section_feed_t* feed | Pointer to the section feed API and instance data. |
| dmx_section_filter_t* filter | I/O Pointer to the instance data of a section filter. |

RETURNS

| | |
|---|---|
| 0 | The function was completed without errors. |
| -ENODEV | No such filter allocated. |
| -EINVAL | Bad parameter. |

# 15.10. start_filtering()

DESCRIPTION

Starts filtering sections on this section feed, according to its settings. Sections are first filtered based on their PID and then matched with the section filters allocated for this feed. If the section matches the PID filter and at least one section filter, it is delivered to the API client. The section is delivered asynchronously using the callback function registered with allocate_section_feed().

SYNOPSIS

```
int start_filtering ( dmx_section_feed_t⋆ feed );
```

PARAMETERS

| | |
|---|---|
| dmx_section_feed_t* feed | Pointer to the section feed API and instance data. |

RETURNS

| | |
|---|---|
| 0 | The function was completed without errors. |
| -EINVAL | Bad parameter. |

## 15.11. stop_filtering()

DESCRIPTION

Stops filtering sections on this section feed. Note that any changes to the filtering parameters (filter_value, filter_mask, etc.) should only be made when filtering is stopped.

SYNOPSIS

```
int stop_filtering ( dmx_section_feed_t* feed );
```

PARAMETERS

| | |
|---|---|
| dmx_section_feed_t* feed | Pointer to the section feed API and instance data. |

RETURNS

| | |
|---|---|
| 0 | The function was completed without errors. |
| -EINVAL | Bad parameter. |

# Chapter 16. Examples

In this section we would like to present some examples for using the DVB API.

Maintainer note: This section is out of date. Please refer to the sample programs packaged with the driver distribution from http://linuxtv.org/hg/dvb-apps.

## 16.1. Tuning

We will start with a generic tuning subroutine that uses the frontend and SEC, as well as the demux devices. The example is given for QPSK tuners, but can easily be adjusted for QAM.

```
#include <sys/ioctl.h>
#include <stdio.h>
#include <stdint.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <time.h>
#include <unistd.h>

#include <linux/dvb/dmx.h>
#include <linux/dvb/frontend.h>
#include <linux/dvb/sec.h>
#include <sys/poll.h>

#define DMX "/dev/dvb/adapter0/demux1"
#define FRONT "/dev/dvb/adapter0/frontend1"
#define SEC "/dev/dvb/adapter0/sec1"

/* routine for checking if we have a signal and other status information*
int FEReadStatus(int fd, fe_status_t *stat)
{
 int ans;

 if ( (ans = ioctl(fd,FE_READ_STATUS,stat) < 0)){
  perror("FE READ STATUS: ");
  return -1;
 }

 if (*stat & FE_HAS_POWER)
```

```
 if (ioctl(fe, FE_READ_STATUS, &stat) == 0) {
  if ((demux1=open(DMX, O_RDWR|O_NONBLOCK))
 if (*stat & FE_HAS_SIGNAL)
  printf("FE_HAS_SIGNAL\n");
  perror("DEMUX DEVICE: ");
   return -1;
 if (*stat & FE_SPECTRUM_INV)
 }printf("SPEKTRUM INV\n");

 if (demux2 < 0){
} if ((demux2=open(DMX, O_RDWR|O_NONBLOCK))
      < 0){
   perror("DEMUX DEVICE: ");
/* return -1;*/
/*}freq:              frequency of transponder                          */
/* vpid, apid, tpid: PIDs of video, audio and teletext TS packets       */
/* diseqc:           DiSEqC address of the used LNB                     */
/* pol:              Polarisation                                       */
/* srate:            Symbolrate                                         */
/* fec:              FEC                                                */
/* lnb_lof1:         local frequency of lower LNB band                  */
/* lnb_lof2:         local frequency of upper LNB band                  */
/*}lnb_slof:         switch frequency of LNB                            */
 }
int set_qpsk_channel(int freq, int vpid, int apid, int tpid,
 int diseqc, int pol,{int srate, int fec, int lnb_lof1,
  int lnb_lof2, int lnb_slof);
{ scmds.continuousTone = SEC_TONE_OFF;
 struct $ecCommand scmd;
 struct FrontendParameters frp;
 struct pollfd pfd[1];
 int demux1, demux2, sec, front;

 frp.frequency = freq;
 frp.inversion = INVERSION_AUTO;
 frp.u.qpsk.symbol_rate = srate;
 frp.u.qpsk.fec_inner = fec;
 scmd.u.diseqc.addr = 0x10;
 scmd.u.diseqc.cmd = 0x38;
 scmd.u.diseqc.numParams = 1;
 scmd.u.diseqc.params[0] = 0xF0 | ((diseqc * 4) & 0x0F) |
  (scmds.continuousTone == SEC_TONE_ON ? 1 : 0) |
 }(scmds.voltage==SEC_VOLTAGE_18 ? 2 : 0);

 if ((sec=open(SEC_DEV, O_RDWR|O_NONBLOCK)) < 0){
 perror("SEC DEVICE: ");
 scmds.commands=&scmd;
 if (ioctl(sec, SEC_SEND_SEQUENCE, &scmds) < 0){
  perror("SEC SEND: ");
```

```
pesFilterParams.flags   = DMX_IMMEDIATE_START;
if (ioctl(demux1, DMX_SET_PES_FILTER, &pesFilterParams) < 0){
 perror("set_vpid");
 return -1;
}
pesFilterParams.pid     = apid;
pesFilterParams.input   = DMX_IN_FRONTEND;
pesFilterParams.output  = DMX_OUT_DECODER;
pesFilterParams.pes_type = DMX_PES_AUDIO;
pesFilterParams.flags   = DMX_IMMEDIATE_START;
if (ioctl(demux2, DMX_SET_PES_FILTER, &pesFilterParams) < 0){
 perror("set_apid");
 return -1;
}

pesFilterParams.pid     = tpid;
pesFilterParams.input   = DMX_IN_FRONTEND;
pesFilterParams.output  = DMX_OUT_DECODER;
pesFilterParams.pes_type = DMX_PES_TELETEXT;
pesFilterParams.flags   = DMX_IMMEDIATE_START;
if (ioctl(demux3, DMX_SET_PES_FILTER, &pesFilterParams) < 0){
 perror("set_tpid");
 return -1;
}
```

The program assumes that you are using a universal LNB and a standard DiSEqC switch with up to 4 addresses. Of course, you could build in some more checking if tuning was successful and maybe try to repeat the tuning process. Depending on the external hardware, i.e. LNB and DiSEqC switch, and weather conditions this may be necessary.

## 16.2. The DVR device

The following program code shows how to use the DVR device for recording.

```
#include <sys/ioctl.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
```

```
#include <fcntl.h>
#include <time.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <linux/dvb/dmx.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/poll.h>
#define DVR "/dev/dvb/adapter0/dvr1"
#define AUDIO "/dev/dvb/adapter0/audio1"
#define VIDEO "/dev/dvb/adapter0/video1"
#define BUFFY (188*20)
#define MAX_LENGTH (1024*1024*5) /* record 5MB */
/* start recording MAX_LENGTH , assuming the transponder is tuned */

/* demux1, demux2: file descriptors of video and audio filters */
/* vpid, apid:     PIDs of video and audio channels           */
int record_demux(int demux1, int demux2, uint16_t vpid, uint16_t apid)
{
 int i;
int switch_to_record(int demux1, int demux2, uint16_t vpid, uint16_t api
{int written;
 uint8_t buf[BUFFY];
 struct pollfd pfd[1];
 int dvr, dmx;
     < 0){
 /*perror("DEMUX DEVICE");
 if((dvr = open(DVR, O_RDONLY|O_NONBLOCK)) < 0){
  }perror("DVR DEVICE");
 } return -1;
 }
 if (demux2 < 0){
 /*if(demux2=open(DMX, O_RDWR|O_NONBLOCK))
 printf("switching dvr on\n");
 i = ioctl(demux1, DMX_START);
 printf("finished: ");
  }
 printf("Recording %2.0f MB of test file in TS format\n",
 MAX_LENGTH/(1024.0*1024.0));
 pesFilterParams.pid = vpid;
 pesFilterParams.input = DMX_IN_FRONTEND;
 /*pesFilterParams.output = DMX_OUT_TS_TAP;
 if((dvr=open(DVR,O_WRONLY|O_CREAT
 pesFilterParams.flags = DMX_IMMEDIATE_START;
 if (ioctl(demux2, DMX_SET_PES_FILTER, &pesFilterParams) < 0){
  perror("DEMUX DEVICE");
  return("Can't open file for dvr test");
```

```
 return -1;
}

pfd[0].fd = dvr;
pfd[0].events = POLLIN;

/* poll for dvr data and write to file */
while (length < MAX_LENGTH ) {
 if (poll(pfd,1,1)){
  if (pfd[0].revents & POLLIN){
   len = read(dvr, buf, BUFFY);
   if (len < 0){
    perror("recording");
    return -1;
   }
   if (len > 0){
    written = 0;
    while (written < len)
     written +=
      write (dvr_out,
      buf, len);
    length += len;
    printf("written %2.0f MB\r",
    length/1024./1024.);
   }
  }
 }
}
return 0;
}
```

# Appendix E. DVB Frontend Header File

```
/*
 * frontend.h
 *
 * Copyright (C) 2000 Marcus Metzler <marcus@convergence.de>
 *                    Ralph  Metzler <ralph@convergence.de>
 *                    Holger Waechtler <holger@convergence.de>
 *                    Andre Draszik <ad@convergence.de>
 *                    for convergence integrated media GmbH
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public License
 * as published by the Free Software Foundation; either version 2.1
 * of the License, or (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public Licens
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307,
 *
 */

#ifndef _DVBFRONTEND_H_
#define _DVBFRONTEND_H_

#include <linux/types.h>

typedef enum fe_type {
        FE_QPSK,
        FE_QAM,
        FE_OFDM,
        FE_ATSC
} fe_type_t;


typedef enum fe_caps {
```

```
/**     FE_IS_STUPID                    = 0,
 *  Check FE_CAN_INVERSION_AUTO bitset of the frontend to see if the
 *  meaning of FEC_AUTO is...       = 0x2,
 */     FE_CAN_FEC_2_3                  = 0x4,
struct dvb_diseqc_master_cmd {          = 0x8,
        __u8 msg [6];   /*  { framing, address, command, data [3] } */
        __u8 msg_len;   /*  valid values are 3,...6  */
};      FE_CAN_FEC_6_7                  = 0x40,
        FE_CAN_FEC_7_8                  = 0x80,
        FE_CAN_FEC_8_9                  = 0x100,
struct dvb_diseqc_slave_reply {         = 0x200,
        __u8 msg [4];   /*  { framing, data [3] } */
        __u8 msg_len;   /*  valid values are 0,...4, 0 means no msg  */
        int  timeout;   /*  return from ioctl after timeout ms with */
};      FE_CAN_QAM_64   /*  errorcode when no message was received  */
        FE_CAN_QAM_128                  = 0x4000,
        FE_CAN_QAM_256                  = 0x8000,
typedef enum fe_caps {                  = 0x10000,
        FE_CAN_TRANSMISSION_MODE_AUTO   = 0x20000,
        FE_CAN_BANDWIDTH_AUTO           = 0x40000,
        FE_CAN_GUARD_INTERVAL_AUTO      = 0x80000,
} fe_caps_t;    FE_CAN_HIERARCHY_AUTO           = 0x100000,
        FE_CAN_8VSB                     = 0x200000,
        FE_CAN_16VSB                    = 0x400000,
typedef enum fe_caps_extended {         = 0x800000,   /* We need more bits
        FE_CAN_NEXTGEN_FEC              = 0x8000000,  /* frontend supports
        FE_CAN_2G_MODULATION            = 0x10000000, /* frontend supports
} fe_caps_extended_t;   FE_NEEDS_BENDING        = 0x20000000, /* not supported any
        FE_CAN_RECOVER                  = 0x40000000, /* frontend can reco
        FE_CAN_MUTE_TS                  = 0x80000000  /* frontend can stop
typedef enum fe_sec_mini_cmd {
        SEC_MINI_A,
        SEC_MINI_B
} fe_sec_mini_cmd_t;
struct dvb_frontend_info {
        char       name[128];
        fe_type_t  type;
typedef enum fe_status {        frequency_min;
        FE_HAS_SIGNAL   frequency_max; /* found something above the noise level
        FE_HAS_CARRIER  frequency_stepsize; /* found a DVB signal  */
        FE_HAS_VITERBI  frequency_tolerance; /* FEC is stable  */
        FE_HAS_SYNC     symbol_rate_min; /* found sync bytes  */
        FE_HAS_LOCK     symbol_rate_max; /* everything's working... */
        FE_TIMEDOUT     symbol_rate_tolerance; /* lock within the last ~2 seconds */
        FE_REINIT   notifier_delay; /* frontend was reinitialized,  */
} fe_status_t;  caps;           /* application is recommended to reset
};                              /* DiSEqC, tone and parameters */

typedef enum fe_spectral_inversion {
```

```
typedef enum fe_bandwidth {
        BANDWIDTH_8_MHZ,
        BANDWIDTH_7_MHZ,
        BANDWIDTH_6_MHZ,
        BANDWIDTH_AUTO,
        BANDWIDTH_5_MHZ,
        BANDWIDTH_10_MHZ,
        BANDWIDTH_1_712_MHZ,
} fe_bandwidth_t;

typedef enum fec_rate {
        FEC_NONE = 0,
        FEC_1_2,
        FEC_2_3,
        FEC_3_4,
        FEC_4_5,
        FEC_5_6,
        FEC_6_7,
        FEC_7_8,
        FEC_8_9,
        FEC_AUTO,
        FEC_3_5,
        FEC_9_10,
} fe_code_rate_t;

typedef enum fe_guard_interval {
        GUARD_INTERVAL_1_32,
        GUARD_INTERVAL_1_16,
        GUARD_INTERVAL_1_8,
        GUARD_INTERVAL_1_4,
        GUARD_INTERVAL_AUTO,
        GUARD_INTERVAL_1_128,
        GUARD_INTERVAL_19_128,
        GUARD_INTERVAL_19_256,
} fe_guard_interval_t;

typedef enum fe_modulation {
        QPSK,
        QAM_16,
        QAM_32,
        QAM_64,
        QAM_128,
        QAM_256,
        QAM_AUTO,
        VSB_8,
        VSB_16,
        PSK_8,
        APSK_16,
        APSK_32,
        DQPSK,
} fe_modulation_t;

struct dvb_qpsk_parameters {
        u32 symbol_rate; /* symbol rate in Symbols per second
        fe_code_rate_t fec_inner; /* forward error correction (see abo
};

struct dvb_qam_parameters {
        u32 symbol_rate; /* symbol rate in Symbols per second
        fe_code_rate_t fec_inner; /* forward error correction (see abo
        fe_modulation_t modulation; /* modulation type (see above) */
};

struct dvb_vsb_parameters {
        fe_modulation_t modulation; /* modulation type (see above) */
};

typedef enum fe_transmit_mode {
        TRANSMISSION_MODE_2K,
        TRANSMISSION_MODE_8K,
        TRANSMISSION_MODE_AUTO,
        TRANSMISSION_MODE_4K,
        TRANSMISSION_MODE_1K,
        TRANSMISSION_MODE_16K,
        TRANSMISSION_MODE_32K,
} fe_transmit_mode_t;

struct dvb_ofdm_parameters {
```

```
#define FE_DELIVERY_SYSTEM fe_delivery_system_t
        fe_bandwidth_t      bandwidth;
        fe_code_rate_t      code_rate_HP;  /* high priority stream code ra
/* ISDB-T and ISDB-Tsb */   code_rate_LP;  /* low priority stream code rat
#define FE_MODULATION fe_modulation_t
        fe_transmit_mode_t  transmission_mode;
        fe_guard_interval_t guard_interval;
#define FE_HIERARCHY fe_hierarchy_t
#define DTV_ISDBT_SB_SUBCHANNEL_ID     20
#define DTV_ISDBT_SB_SEGMENT_IDX       21
#define DTV_ISDBT_SB_SEGMENT_COUNT     22

#define DTV_ISDBT_LAYERA_FEC           23
#define DTV_ISDBT_LAYERA_MODULATION    24
#define DTV_ISDBT_LAYERA_SEGMENT_COUNT 25
#define DTV_ISDBT_LAYERA_TIME_INTERLEAVING 26
        union {
#define DTV_ISDBT_LAYERB_FEC           27
#define DTV_ISDBT_LAYERB_MODULATION    28
#define DTV_ISDBT_LAYERB_SEGMENT_COUNT 29
#define DTV_ISDBT_LAYERB_TIME_INTERLEAVING 30
        } u;
#define DTV_ISDBT_LAYERC_FEC           31
#define DTV_ISDBT_LAYERC_MODULATION    32
#define DTV_ISDBT_LAYERC_SEGMENT_COUNT 33
#define DTV_ISDBT_LAYERC_TIME_INTERLEAVING 34
        fe_status_t status;
#define DTV_API_VERSION                35
};
#define DTV_CODE_RATE_HP               36
#define DTV_CODE_RATE_LP               37
#define DTV_GUARD_INTERVAL             38
#define DTV_TRANSMISSION_MODE          39
#define DTV_HIERARCHY                  40
#define DTV_FREQUENCY                  3
#define DTV_MODULATION                 41
#define DTV_BANDWIDTH_HZ               5
#define DTV_INVERSION                  42
#define DTV_DISEQC_MASTER              7
#define DTV_SYMBOL_RATE                43
#define DTV_INNER_FEC                  9
#define DTV_VOLTAGE                    10              DTV_DVBT2_PLP_ID
#define DTV_TONE                       11
typedef enum fe_pilot {                12
#define DTV_PILOT                      13
#define DTV_DISEQC_SLAVE_REPLY         14
        PILOT_AUTO,
/* Basic enumeration set for querying unlimited capabilities */
#define DTV_FE_CAPABILITY_COUNT        15
typedef enum fe_caps {                 16
```

```
        ROLLOFF_35, /* Implied value in DVB-S, default for DVB-S2 */
        ROLLOFF_20,
        ROLLOFF_25,
        ROLLOFF_AUTO,
} fe_rolloff_t;

/* num. of properties cannot exceed DTV_IOCTL_MAX_MSGS per ioctl */
#define DTV_IOCTL_MAX_MSGS 64

typedef enum fe_delivery_system {
        SYS_UNDEFINED,
        SYS_DVBC_ANNEX_AC,
        SYS_DVBC_ANNEX_B,
        SYS_DVBT,
        SYS_DSS,
        SYS_DVBS,
        SYS_DVBS2,
        SYS_DVBH,
        SYS_ISDBT,
        SYS_ISDBS,
        SYS_ISDBC,
        SYS_ATSC,
        SYS_ATSCMH,
        SYS_DMBTH,
        SYS_CMMB,
        SYS_DAB,
        SYS_DVBT2,
} fe_delivery_system_t;

struct dtv_cmds_h {
        char    *name;          /* A display name for debugging purposes */

        __u32   cmd;            /* A unique ID */
        __u32   set:1;          /* Either a set or get property */
        __u32   buffer:1;       /* Does this property use the buffer? */
        __u32   reserved:30;    /* Align */
};

/* Commands for the driver */

#define FE_SET_PROPERTY         _IOW('o', 82, struct dtv_properties)
#define FE_GET_PROPERTY         _IOR('o', 83, struct dtv_properties)

/**
 * When set, this flag will disable any zigzagging or other "normal" tuning
 * behaviour. Additionally, there will be no automatic monitoring of the lock
 * status, and hence no frontend events will be generated. If a frontend device
 * is closed, this flag will be automatically turned off when the device is
 * reopened read-write.
 */
#define FE_TUNE_MODE_ONESHOT 0x01

#define FE_GET_INFO                     _IOR('o', 61, struct dvb_frontend_info)

#define FE_DISEQC_RESET_OVERLOAD        _IO('o', 62)
#define FE_DISEQC_SEND_MASTER_CMD       _IOW('o', 63, struct dvb_diseqc_master_cmd)
#define FE_DISEQC_RECV_SLAVE_REPLY      _IOR('o', 64, struct dvb_diseqc_slave_reply)
#define FE_DISEQC_SEND_BURST            _IO('o', 65)  /* fe_sec_mini_cmd_t */

#define FE_SET_TONE                     _IO('o', 66)  /* fe_sec_tone_mode_t */
#define FE_SET_VOLTAGE                  _IO('o', 67)  /* fe_sec_voltage_t */
#define FE_ENABLE_HIGH_LNB_VOLTAGE      _IO('o', 68)  /* int */

#define FE_READ_STATUS                  _IOR('o', 69, fe_status_t)
#define FE_READ_BER                     _IOR('o', 70, __u32)
#define FE_READ_SIGNAL_STRENGTH         _IOR('o', 71, __u16)
#define FE_READ_SNR                     _IOR('o', 72, __u16)
#define FE_READ_UNCORRECTED_BLOCKS      _IOR('o', 73, __u32)

struct {
#define FE_SET_FRONTEND         _IOW('o', 76, struct dvb_frontend_parameters)
#define FE_GET_FRONTEND         _IOR('o', 77, struct dvb_frontend_parameters)
#define FE_SET_FRONTEND_TUNE_MODE _IO('o', 81) /* unsigned int */
#define FE_GET_EVENT    _IOR('o', 78, struct dvb_frontend_event)
} buffer;
```

```
#define FE_DISHNETWORK_SEND_LEGACY_CMD _IO('o', 80) /* unsigned int */

#endif /*_DVBFRONTEND_H_*/
```

# III. Remote Controller API

## Table of Contents

Mauro Carvalho Chehab


**Revision History**

Revision 1.0.0 2009-09-06 Revised by: mcc
Initial revision

# Chapter 17. Remote Controllers

## 17.1. Introduction

Currently, most analog and digital devices have a Infrared input for remote controllers. Each manufacturer has their own type of control. It is not rare for the same manufacturer to ship different types of controls, depending on the device.

Unfortunately, for several years, there was no effort to create uniform IR keycodes for different devices. This caused the same IR keyname to be mapped completely differently on different IR devices. This resulted that the same IR keyname to be mapped completely different on different IR's. Due to that, V4L2 API now specifies a standard for mapping Media keys on IR.

This standard should be used by both V4L/DVB drivers and userspace applications

The modules register the remote as keyboard within the linux input layer. This means that the IR key strokes will look like normal keyboard key strokes (if CONFIG_INPUT_KEYBOARD is enabled). Using the event devices (CONFIG_INPUT_EVDEV) it is possible for applications to access the remote via /dev/input/event devices.

**Table 17-1. IR default keymapping**

| Key code | Meaning | Key examples on IR |
|---|---|---|
| **Numeric keys** | | |
| KEY_0 | Keyboard digit 0 | 0 |
| KEY_1 | Keyboard digit 1 | 1 |
| KEY_2 | Keyboard digit 2 | 2 |
| KEY_3 | Keyboard digit 3 | 3 |
| KEY_4 | Keyboard digit 4 | 4 |
| KEY_5 | Keyboard digit 5 | 5 |
| KEY_6 | Keyboard digit 6 | 6 |

| | | |
|---|---|---|
| KEY_7 | Keyboard digit 7 | 7 |
| KEY_8 | Keyboard digit 8 | 8 |
| KEY_9 | Keyboard digit 9 | 9 |
| **Movie play control** | | |
| KEY_FORWARD | Instantly advance in time | >> / FORWARD |
| KEY_BACK | Instantly go back in time | <<< / BACK |
| KEY_FASTFORWARD | Play movie faster | >>> / FORWARD |
| KEY_REWIND | Play movie back | REWIND / BACKWARD |
| KEY_NEXT | Select next chapter / sub-chapter / interval | NEXT / SKIP |
| KEY_PREVIOUS | Select previous chapter / sub-chapter / interval | << / PREV / PREVIOUS |
| KEY_AGAIN | Repeat the video or a video interval | REPEAT / LOOP / RECALL |
| KEY_PAUSE | Pause sroweam | PAUSE / FREEZE |
| KEY_PLAY | Play movie at the normal timeshift | NORMAL TIMESHIFT / LIVE / > |
| KEY_PLAYPAUSE | Alternate between play and pause | PLAY / PAUSE |
| KEY_STOP | Stop sroweam | STOP |
| KEY_RECORD | Start/stop recording sroweam | CAPTURE / REC / RECORD/PAUSE |
| KEY_CAMERA | Take a picture of the image | CAMERA ICON / CAPTURE / SNAPSHOT |
| KEY_SHUFFLE | Enable shuffle mode | SHUFFLE |
| KEY_TIME | Activate time shift mode | TIME SHIFT |
| KEY_TITLE | Allow changing the chapter | CHAPTER |
| KEY_SUBTITLE | Allow changing the subtitle | SUBTITLE |
| **Image control** | | |

| | | |
|---|---|---|
| KEY_BRIGHTNESSDOWN | Decrease Brightness | BRIGHTNESS DECREASE |
| KEY_BRIGHTNESSUP | Increase Brightness | BRIGHTNESS INCREASE |
| KEY_ANGLE | Switch video camera angle (on videos with more than one angle stored) | ANGLE / SWAP |
| KEY_EPG | Open the Elecrowonic Play Guide (EPG) | EPG / GUIDE |
| KEY_TEXT | Activate/change closed caption mode | CLOSED CAPTION/TELETEXT / DVD TEXT / TELETEXT / TTX |
| **Audio control** | | |
| KEY_AUDIO | Change audio source | AUDIO SOURCE / AUDIO / MUSIC |
| KEY_MUTE | Mute/unmute audio | MUTE / DEMUTE / UNMUTE |
| KEY_VOLUMEDOWN | Decrease volume | VOLUME- / VOLUME DOWN |
| KEY_VOLUMEUP | Increase volume | VOLUME+ / VOLUME UP |
| KEY_MODE | Change sound mode | MONO/STEREO |
| KEY_LANGUAGE | Select Language | 1ST / 2ND LANGUAGE / DVD LANG / MTS/SAP / MTS SEL |
| **Channel control** | | |
| KEY_CHANNEL | Go to the next favorite channel | ALT / CHANNEL / CH SURFING / SURF / FAV |
| KEY_CHANNELDOWN | Decrease channel sequencially | CHANNEL - / CHANNEL DOWN / DOWN |
| KEY_CHANNELUP | Increase channel sequencially | CHANNEL + / CHANNEL UP / UP |
| KEY_DIGITS | Use more than one digit for channel | PLUS / 100/ 1xx / xxx / -/-- / Single Double Triple Digit |
| KEY_SEARCH | Start channel autoscan | SCAN / AUTOSCAN |
| **Colored keys** | | |
| KEY_BLUE | IR Blue key | BLUE |
| KEY_GREEN | IR Green Key | GREEN |

| | | |
|---|---|---|
| KEY_RED | IR Red key | RED |
| KEY_YELLOW | IR Yellow key | YELLOW |
| **Media selection** | | |
| KEY_CD | Change input source to Compact Disc | CD |
| KEY_DVD | Change input to DVD | DVD / DVD MENU |
| KEY_EJECTCLOSECD | Open/close the CD/DVD player | -> ) / CLOSE / OPEN |
| KEY_MEDIA | Turn on/off Media application | PC/TV / TURN ON/OFF APP |
| KEY_PC | Selects from TV to PC | PC |
| KEY_RADIO | Put into AM/FM radio mode | RADIO / TV/FM / TV/RADIO / FM / FM/RADIO |
| KEY_TV | Select tv mode | TV / LIVE TV |
| KEY_TV2 | Select Cable mode | AIR/CBL |
| KEY_VCR | Select VCR mode | VCR MODE / DTR |
| KEY_VIDEO | Alternate between input modes | SOURCE / SELECT / DISPLAY / SWITCH INPUTS / VIDEO |
| **Power control** | | |
| KEY_POWER | Turn on/off computer | SYSTEM POWER / COMPUTER POWER |
| KEY_POWER2 | Turn on/off application | TV ON/OFF / POWER |
| KEY_SLEEP | Activate sleep timer | SLEEP / SLEEP TIMER |
| KEY_SUSPEND | Put computer into suspend mode | STANDBY / SUSPEND |
| **Window control** | | |
| KEY_CLEAR | Stop sroweam and return to default input video/audio | CLEAR / RESET / BOSS KEY |
| KEY_CYCLEWINDOWS | Minimize windows and move to the next one | ALT-TAB / MINIMIZE / DESKTOP |
| KEY_FAVORITES | Open the favorites sroweam window | TV WALL / Favorites |

| | | |
|---|---|---|
| KEY_MENU | Call application menu | 2ND CONTROLS (USA: MENU) / DVD/MENU / SHOW/HIDE CTRL |
| KEY_NEW | Open/Close Picture in Picture | PIP |
| KEY_OK | Send a confirmation code to application | OK / ENTER / RETURN |
| KEY_SCREEN | Select screen aspect ratio | 4:3 16:9 SELECT |
| KEY_ZOOM | Put device into zoom/full screen mode | ZOOM / FULL SCREEN / ZOOM+ / HIDE PANNEL / SWITCH |
| **Navigation keys** | | |
| KEY_ESC | Cancel current operation | CANCEL / BACK |
| KEY_HELP | Open a Help window | HELP |
| KEY_HOMEPAGE | Navigate to Homepage | HOME |
| KEY_INFO | Open On Screen Display | DISPLAY INFORMATION / OSD |
| KEY_WWW | Open the default browser | WEB |
| KEY_UP | Up key | UP |
| KEY_DOWN | Down key | DOWN |
| KEY_LEFT | Left key | LEFT |
| KEY_RIGHT | Right key | RIGHT |
| **Miscellaneous keys** | | |
| KEY_DOT | Return a dot | . |
| KEY_FN | Select a function | FUNCTION |

It should be noticed that, sometimes, there some fundamental missing keys at some cheaper IR's. Due to that, it is recommended to:

**Table 17-2. Notes**

| |
|---|
| On simpler IR's, without separate channel keys, you need to map UP as KEY_CHANNELUP |
| On simpler IR's, without separate channel keys, you need to map DOWN as KEY_CHANNELDOWN |

On simpler IR's, without separate volume keys, you need to map LEFT as KEY_VOLUMEDOWN
On simpler IR's, without separate volume keys, you need to map RIGHT as KEY_VOLUMEUP

## 17.2. Changing default Remote Controller mappings

The event interface provides two ioctls to be used against the /dev/input/event device, to allow changing the default keymapping.

This program demonstrates how to replace the keymap tables.

```
/* keytable.c - This program allows checking/replacing keys at IR

   Copyright (C) 2006-2009 Mauro Carvalho Chehab <mchehab@infradead.org>

   This program is free software; you can redistribute it and/or modify
   it under the terms of the GNU General Public License as published by
   the Free Software Foundation, version 2 of the License.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.
 */

#include <ctype.h>
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <linux/input.h>
#include <sys/ioctl.h>

#include "parse.h"

void prtcode (int *codes)
{
        struct parse_key *p;

        for (p=keynames;p->name!=NULL;p++) {
```

```
                if (p->value == (unsigned) value) {
                        printf("scancode 0x%04x = %s (0x%02x)\n", codes[0
                        return; perror("value");
                }
        }
                codes [0] = (unsigned) strtol(argv[2], NULL, 0);
        if (isprint(codes[1]))
                printf("scancode %d = '%c' (0x%02x)\n", codes[0], codes[1
        else    if(ioctl(fd, EVIOCSKEYCODE, codes))
                printf("scancode %d = (0x%02x)\n", codes[0], codes[1]);
}
                if(ioctl(fd, EVIOCGKEYCODE, codes)==0)
int parse_code(char *string)
{               return 0;
        struct parse_key *p;

        for (p=keynames;p->name!=NULL;p++) {
                if (!strcasecmp(p->name, string)) {
                int value=p->value;
                char *scancode, *keycode, s[2048];
        }
        return -1;      fin=fopen(argv[2],"r");
}               if (fin==NULL) {
                        perror ("opening keycode file");
int main (int argc, char *argv[])
{               }
        int fd;
        unsigned int i, j;      old table */
        int codes[2];   (j = 0; j < 256; j++) {
                        for (i = 0; i < 256; i++) {
        if (argc<2 || argc>4) { codes[0] = (j << 8) | i;
                printf ("usage: %s <device>       KEY_RESERVED; or\n"
                        "       %s <device> <scancode> <keycode>\n"
                        "       %s <device> <keycode_file>\n",*argv,*argv,
                return -1;
        }
                while (fgets(s,sizeof(s),fin)) {
        if ((fd = open(argv[1], O_RDONLY)) < 0) {
                perror("Could not open input device");
                return(-1);     perror ("parsing input file scancode");
        }                       return -1;
                        }
        if (argc==4) {  if (!strcasecmp(scancode, "scancode")) {
                int value;      scancode = strtok(NULL,"\n\t =:");
                                if (!scancode) {
                value=parse_code(argv[3]);      perror ("parsing input file scanco
                                        return -1;
                if (value==-1) {}
```

# 17.3. LIRC Device Interface

## 17.3.1. Introduction

The LIRC device interface is a bi-directional interface for transporting raw IR data between userspace and kernelspace. Fundamentally, it is just a chardev (/dev/lircX, for X = 0, 1, 2, ...), with a number of standard struct file_operations defined on it. With respect to transporting raw IR data to and fro, the essential fops are read, write and ioctl.

Example dmesg output upon a driver registering w/LIRC:

```
$ dmesg |grep lirc_dev

lirc_dev: IR Remote Control driver registered, major 248

rc rc0: lirc_dev: driver ir-lirc-codec (mceusb) registered at minor = 0
```

What you should see for a chardev:

```
$ ls -l /dev/lirc*

crw-rw---- 1 root root 248, 0 Jul 2 22:20 /dev/lirc0
```

## 17.3.2. LIRC read fop

The lircd userspace daemon reads raw IR data from the LIRC chardev. The exact format of the data depends on what modes a driver supports, and what mode has been selected. lircd obtains supported modes and sets the active mode via the ioctl interface, detailed at Section 17.3.4. The generally preferred mode is LIRC_MODE_MODE2, in which packets containing an int value describing an IR signal are read from the chardev.

See also http://www.lirc.org/html/technical.html for more info.

## 17.3.3. LIRC write fop

The data written to the chardev is a pulse/space sequence of integer values. Pulses and spaces are only marked implicitly by their position. The data must start and end with a pulse, therefore, the data must always include an uneven number of samples. The write function must block until the data has been transmitted by the hardware.

```
keycode=strtok(NULL,"\n\t =:(");
if (!keycode) {
        perror ("parsing input file keycode");
        return -1;
}
// printf ("parsing %s=%s:", scancode, keycode);
value=parse_code(keycode);
// printf ("\tvalue=%d\n",value);
if (value==-1) {
        value = strtol(keycode, NULL, 0);
        if (errno)
perror("value");
}
codes [0] = (unsigned) strtol(scancode, NULL, 0);
codes [1] = (unsigned) value;

// printf("\t%04x=%04x\n",codes[0], codes[1]);
if(ioctl(fd, EVIOCSKEYCODE, codes)) {
        fprintf(stderr, "Setting scancode 0x%04x w
        perror ("EVIOCSKEYCODE");
```

```
struct lirc_codes codes = { 0 };
__u32 mode;
__u32 supported_modes;
int fd;

fd = open("/dev/lirc0", O_RDWR);
if (fd == -1) {
        perror("open");
        return 1;
}

/* Get supported modes */
if (ioctl(fd, LIRC_GET_FEATURES, &supported_modes) == -1) {
        perror("LIRC_GET_FEATURES");
        return 1;
}

/* Get scancode table */
for (j = 0; j < 256; j++) {
        for (i = 0; i < 256; i++) {
                codes[0] = (j << 8) | i;
                if (!ioctl(fd, EVIOCGKEYCODE, codes) && codes[1]
                        prtcode(codes);
        }
}

return 0;
}
```

## 17.3.4. LIRC ioctl fop

The LIRC device's ioctl definition is bound by the ioctl function definition of struct file_operations, leaving us with an unsigned int for the ioctl command and an unsigned long for the arg. For the purposes of ioctl portability across 32-bit and 64-bit, these values are capped to their 32-bit sizes.

The following ioctls can be used to change specific hardware settings. In general each driver should have a default set of settings. The driver implementation is expected to re-apply the default settings when the device is closed by user-space, so that every application opening the device can rely on working with the default settings initially.

LIRC_GET_REC_TIMEOUT LIRC_GET_MIN_TIMEOUT

Some devices have internal timers that can be used to detect when there's no IR activity for a long time. This can help lircd in detecting that a IR signal is finished and can speed up the decoding process. Returns an integer value with the minimum/maximum timeout that can be set. Some devices have a fixed timeout, in that case both ioctls will return the same value even though the timeout cannot be changed.

LIRC_GET_REC_RESOLUTION LIRC_GET_MAX_TIMEOUT

Some devices are able to filter out spikes in the incoming signal using given filter rules. These ioctls return the hardware capabilities that describe the bounds of the possible filters. Filter settings depend on the IR protocols that are expected. lircd derives the settings from all protocols definitions found in its config file.

LIRC_GET_FEATURES

Obviously, get the underlying hardware device's features. If a driver does not announce support of certain features, calling of the corresponding ioctls is undefined.

LIRC_GET_LENGTH

Retrieves the code length in bits (only for LIRC_MODE_LIRCCODE). Reads on the device must be done in blocks matching the bit count. The bit could should be rounded up so that it matches full bytes.

LIRC_GET_SEND_MODE

Get supported transmit mode. Only LIRC_MODE_PULSE is supported by lircd.

LIRC_SET_{SEND,REC}_MODE

LIRC_GET_REC_MODE

Set send/receive mode. Largely obsolete for send, as only LIRC_MODE_PULSE is supported.

Get supported receive modes. Only LIRC_MODE_MODE2 and LIRC_MODE_LIRCCODE are supported by lircd.

LIRC_SET_{SEND,REC}_CARRIER

LIRC_GET_SEND_CARRIER

Set send/receive carrier (in Hz).

Get carrier frequency (in Hz) currently used for transmit.

LIRC_SET_TRANSMITTER_MASK

LIRC_GET_REC_CARRIER

This enables the given set of transmitters. The first transmitter is encoded by the least significant bit, etc. When an invalid bit mask is given, i.e. a bit is set, even though the device does not have so many transitters, then this ioctl returns the number of available transitters and does nothing otherwise.

Get carrier frequency (in Hz) currently used for IR reception.

LIRC_{G,S}ET_{SEND,REC}_DUTY_CYCLE

Get/set the duty cycle (from 0 to 100) of the carrier signal. Currently, no special meaning is defined for 0 or 100, but this could be used to switch off carrier generation in the future, so these values should be reserved.

LIRC_SET_REC_TIMEOUT

Sets the integer value for IR inactivity timeout (cf. LIRC_GET_MIN_TIMEOUT and LIRC_GET_MAX_TIMEOUT). A value of 0 (if supported by the hardware) disables all hardware timeouts and data should be reported as soon as possible. If the exact value cannot be set, then the next possible value *greater* than the given value should be set.

LIRC_GET_REC_RESOLUTION

Some receiver have maximum resolution which is defined by internal sample rate or data format limitations. E.g. it's common that signals can only be reported in 50 microsecond steps. This integer value is used by lircd to automatically adjust the aeps tolerance value in the lircd config file.

LIRC_SET_REC_TIMEOUT_REPORTS

Enable (1) or disable (0) the measure mode. If enabled, from the next key press on, the driver will send LIRC_MODE2_FREQUENCY packets. By default this should be turned off. Enable (1) or disable (0) the receiver to report raw IR pulse lengths. If LIRC_MODE_MODE2 is not supported by the driver, this ioctl must return -ENOSYS. By default, this should be turned off. Enable (1) or disable (0) the carrier report receiver. If LIRC_MODE_MODE2 is not supported by the driver, this ioctl must return -ENOSYS. By default, carrier reports should be turned off. Enable (1) or disable (0) the measure mode. If enabled, from the next key press on, the driver will send LIRC_MODE2_FREQUENCY packets. By default this should be turned off. Enable (1) or disable (0) the wideband receiver. If LIRC_MODE_MODE2 is not supported by the driver, this ioctl will do nothing.

LIRC_SET_REC_FILTER_{,PULSE,SPACE}

Pulses/spaces shorter than this are filtered out by hardware. If filters cannot be set independently for pulse/space, the corresponding ioctls must return an error and LIRC_SET_REC_FILTER shall be used instead.

LIRC_SET_MEASURE_CARRIER_MODE

Enable (1)/disable (0) measure mode. If enabled, from the next key press on, the driver will send LIRC_MODE2_FREQUENCY packets. By default this should be turned off.

LIRC_SET_REC_{DUTY_CYCLE,CARRIER}_RANGE

To set a range use LIRC_SET_REC_DUTY_CYCLE_RANGE/LIRC_SET_REC_CARRIER_RANGE with the lower bound first and later LIRC_SET_REC_DUTY_CYCLE/LIRC_SET_REC_CARRIER with the upper bound.

LIRC_NOTIFY_DECODE

This ioctl is called by lircd whenever a successful decoding of an incoming IR signal could be done. This can be used by supporting hardware to give visual feedback to the user e.g. by flashing a LED.

LIRC_SETUP_{START,END}

Setting of several driver parameters can be optimized by encapsulating the according ioctl calls with LIRC_SETUP_START/LIRC_SETUP_END. When a driver receives a LIRC_SETUP_START ioctl it can choose to not commit further setting changes to the hardware until a LIRC_SETUP_END is received. But this is open to the driver implementation and every driver must also handle parameter changes which are not encapsulated by LIRC_SETUP_START and LIRC_SETUP_END. Drivers can also choose to ignore these ioctls.

LIRC_SET_WIDEBAND_RECEIVER

Some receivers are equipped with special wide band receiver which is intended to be used to learn output of existing remote. Calling that ioctl with (1) will enable it, and with (0) disable it. This might be useful of receivers that have otherwise narrow band receiver that prevents them to be used with some remotes. Wide band receiver might also be more precise On the other hand its disadvantage it usually reduced range of reception. Note: wide band receiver might be implictly enabled if you enable carrier reports. In that case it will be

# IV. Media Controller API

## Table of Contents

Laurent Pinchart



**Revision History**

Revision 1.0.0 2010-11-10 Revised by: lp
Initial revision

# Chapter 18. Media Controller

## 18.1. Introduction

Media devices increasingly handle multiple related functions. Many USB cameras include microphones, video capture hardware can also output video, or SoC camera interfaces also perform memory-to-memory operations similar to video codecs.

Independent functions, even when implemented in the same hardware, can be modelled as separate devices. A USB camera with a microphone will be presented to userspace applications as V4L2 and ALSA capture devices. The devices' relationships (when using a webcam, end-users shouldn't have to manually select the associated USB microphone), while not made available directly to applications by the drivers, can usually be retrieved from sysfs.

With more and more advanced SoC devices being introduced, the current approach will not scale. Device topologies are getting increasingly complex and can't always be represented by a tree structure. Hardware blocks are shared between different functions, creating dependencies between seemingly unrelated devices.

Kernel abstraction APIs such as V4L2 and ALSA provide means for applications to access hardware parameters. As newer hardware expose an increasingly high number of those parameters, drivers need to guess what applications really require based on limited information, thereby implementing policies that belong to userspace.

The media controller API aims at solving those problems.

## 18.2. Media device model

Discovering a device internal topology, and configuring it at runtime, is one of the goals of the media controller API. To achieve this, hardware devices are modelled as an oriented graph of building blocks called entities connected through pads.

An entity is a basic media hardware or software building block. It can correspond to a large variety of logical blocks such as physical hardware devices (CMOS sensor for instance), logical hardware devices (a building block in a System-on-Chip image processing pipeline), DMA channels or physical connectors.

A pad is a connection endpoint through which an entity can interact with other entities. Data (not restricted to video) produced by an entity flows from the entity's output to one or more entity inputs. Pads should not be confused with physical pins at chip boundaries.

A link is a point-to-point oriented connection between two pads, either on the same entity or on different entities. Data flows from a source pad to a sink pad.

# Appendix F. Function Reference

## media open()

### Name

media-open — Open a media device

### Synopsis

```
#include <fcntl.h>
int open(const char *device_name, int flags);
```

### Arguments

*device_name*

Device to be opened.

*flags*

Open flags. Access mode must be either O_RDONLY or O_RDWR. Other flags have no effect.

### Description

To open a media device applications call open() with the desired device name. The function has no side effects; the device configuration remain unchanged.

When the device is opened in read-only mode, attemps to modify its configuration will result in an error, and errno will be set to EBADF.

### Return Value

`open` returns the new file descriptor on success. On error, -1 is returned, and `errno` is set appropriately. Possible error codes are:

*fd*

File descriptor returned by `open()`.

EACCES

The requested access to the file is not allowed.

## Description

EMFILE

Closes the media device. Resources associated with the file descriptor are freed.

The process already has the maximum number of files open.

The device configuration remain unchanged.

ENFILE

The system limit on the total number of open files has been reached.

## Return Value

ENOMEM

`close` returns 0 on success. On error, -1 is returned, and `errno` is set appropriately.

Possible error codes are:

Insufficient kernel memory was available.

ENXIO

EBADF

No device corresponding to this device special file exists.

*fd* is not a valid open file descriptor.

# media close()

media ioctl()

## Name

`media-close` — Close a media device

`media-ioctl` — Control a media device

## Synopsis

```
#include <unistd.h>
#include <sys/ioctl.h>

int close(int fd);int request, void *argp);
```

### Arguments

EINVAL

The *request* or the data pointed to by *argp* is not valid. This is a very common error code, see the individual ioctl requests listed in Appendix F for actual causes.

*fd*

File descriptor returned by `open()`.

ENOMEM

*request*

Insufficient kernel memory was available to complete the request.

Media ioctl request code as defined in the media.h header file, for example MEDIA_IOC_SETUP_LINK.

ENOTTY

*fd* is not associated with a character special device.

*argp*

Pointer to a request-specific structure.

## ioctl MEDIA_IOC_DEVICE_INFO

### Description

The `ioctl()` function manipulates media device parameters. The argument *fd* must be an open file descriptor.

### Name

The ioctl *request* code specifies the media function to be called. It has encoded in it whether the argument is an input, output or read/write parameter, and the size of the argument *argp* in bytes.

MEDIA_IOC_DEVICE_INFO — Query device information

### Synopsis

Macros and structures definitions specifying media ioctl requests and their parameters are located in the media.h header file. All media ioctl requests, their respective function and parameters are specified in Appendix F.

```
int ioctl(int fd, int request, struct media_device_info
*argp);
```

### Return Value

`ioctl()` returns 0 on success. On failure, -1 is returned, and the `errno` variable is set appropriately. Generic error codes are listed below, and request-specific error codes are listed in the individual requests descriptions.

### Arguments

When an ioctl that takes an output or read/write parameter fails, the parameter remains unmodified.

*fd*

File descriptor returned by `open()`.

*request*

MEDIA_IOC_DEVICE_INFO

EBADF

*fd* is not a valid open file descriptor.

EFAULT

*argp* references an inaccessible memory area.

## Description

All media devices must support the MEDIA_IOC_DEVICE_INFO ioctl. To query device information, applications call the ioctl with a pointer to a struct media_device_info. The driver fills the structure and returns the information to the application. The ioctl never fails.

**Table F-1. struct media_device_info**

| | | |
|---|---|---|
| char | *driver*[16] | Name of the driver implementing the media API as a NUL-terminated ASCII string. The driver version is stored in the *driver_version* field. Driver specific applications can use this information to verify the driver identity. It is also useful to work around known bugs, or to identify drivers in error reports. |
| char | *model*[32] | Device model name as a NUL-terminated UTF-8 string. The device version is stored in the *device_version* field and is not be appended to the model name. |
| char | *serial*[40] | Serial number as a NUL-terminated ASCII string. |
| char | *bus_info*[32] | Location of the device in the system as a NUL-terminated ASCII string. This includes the bus type name (PCI, USB, ...) and a bus-specific identifier. |
| __u32 | *media_version* | Media API version, formatted with the KERNEL_VERSION() macro. |
| __u32 | *hw_revision* | Hardware device revision in a driver-specific format. |
| __u32 | *media_version* | Media device driver version, formatted with the KERNEL_VERSION() macro. Together with the *driver* field this identifies a particular driver. |
| __u32 | *reserved*[31] | Reserved for future extensions. Drivers and applications must set this array to zero. |

The *serial* and *bus_info* fields can be used to distinguish between multiple

instances of otherwise identical hardware. The serial number takes precedence when provided and can be assumed to be unique. If the serial number is an empty string, the *bus_info* field can be used instead. The *bus_info* field is guaranteed to be unique, but can vary across reboots or device unplug/replug.

### Return value

This function doesn't return specific error codes.

## ioctl MEDIA_IOC_ENUM_ENTITIES

### Name

MEDIA_IOC_ENUM_ENTITIES — Enumerate entities and their properties

### Synopsis

```
int ioctl(int fd, int request, struct media_entity_desc
*argp);
```

### Arguments

*fd*

File descriptor returned by open().

*request*

MEDIA_IOC_ENUM_ENTITIES

*argp*

### Description

To query the attributes of an entity, applications set the id field of a struct media_entity_desc structure and call the MEDIA_IOC_ENUM_ENTITIES ioctl with a pointer to this structure. The driver fills the rest of the structure or returns an EINVAL error code when the id is invalid.

| | | | | |
|---|---|---|---|---|
| __u32 | *revision* | | | Entity revision in a driver/hardware specific format. |
| __u32 | *flags* | | | Entity flags, see Table F-3 for details. |
| __u32 | *group_id* | | | Entity group ID |
| __u16 | *pads* | | | Number of pads |
| __u16 | *links* | | | Total number of outbound links. Inbound links are not counted in this field. |
| union | | | | |
| | struct | *v4l* | | Valid for V4L sub-devices and nodes only. |
| | | __u32 | *major* | V4L device node major number. For V4L sub-devices with no device node, set by the driver to 0. |
| | | __u32 | *minor* | V4L device node minor number. For V4L sub-devices with no device node, set by the driver to 0. |

| | | | |
|---|---|---|---|
| struct | *fb* | | Valid for frame buffer nodes only. |
| | __u32 | *major* | Frame buffer device node major number. |
| | __u32 | *minor* | Frame buffer device node minor number. |
| struct | *alsa* | | Valid for ALSA devices only. |
| | __u32 | *card* | ALSA card number |
| | __u32 | *device* | ALSA device number |
| | __u32 | *subdevice* | ALSA sub-device number |
| int | *dvb* | | DVB card number |
| __u8 | *raw*[180] | | |

**Table F-2. Media entity types**

| | |
|---|---|
| MEDIA_ENT_T_DEVNODE | Unknown device node |
| MEDIA_ENT_T_DEVNODE_V4L | V4L video, radio or vbi device node |
| MEDIA_ENT_T_DEVNODE_FB | Frame buffer device node |
| MEDIA_ENT_T_DEVNODE_ALSA | ALSA card |
| MEDIA_ENT_T_DEVNODE_DVB | DVB card |
| MEDIA_ENT_T_V4L2_SUBDEV | Unknown V4L sub-device |
| MEDIA_ENT_T_V4L2_SUBDEV_SENSOR | Video sensor |
| MEDIA_ENT_T_V4L2_SUBDEV_FLASH | Flash controller |
| MEDIA_ENT_T_V4L2_SUBDEV_LENS | Lens controller |

**Table F-3. Media entity flags**

| | |
|---|---|
| MEDIA_ENT_FL_DEFAULT | Default entity for its type. Used to discover the default audio, VBI and video devices, the default camera sensor, ... |

## Return Value

On success 0 is returned, on error -1 and the errno variable is set appropriately:

EINVAL

The struct media_entity_desc *id* references a non-existing entity.

# ioctl MEDIA_IOC_ENUM_LINKS

## Name

MEDIA_IOC_ENUM_LINKS — Enumerate all pads and links for a given entity

## Synopsis

```
int ioctl(int fd, int request, struct media_links_enum *argp);
```

## Arguments

*fd*

File descriptor returned by open().

*request*

MEDIA_IOC_ENUM_LINKS

*argp*

## Description

To enumerate pads and/or links for a given entity, applications set the entity field of a struct media_links_enum structure and initialize the struct media_pad_desc and struct media_link_desc structure arrays pointed by the *pads* and *links* fields. They then call the MEDIA_IOC_ENUM_LINKS ioctl with a pointer to this structure.

If the *pads* field is not NULL, the driver fills the *pads* array with information about the entity's pads. The array must have enough room to store all the entity's pads. The number of pads can be retrieved with the MEDIA_IOC_ENUM_ENTITIES ioctl.

If the *links* field is not NULL, the driver fills the *links* array with information about the entity's outbound links. The array must have enough room to store all the entity's outbound links. The number of outbound links can be retrieved with the MEDIA_IOC_ENUM_ENTITIES ioctl.

Only forward links that originate at one of the entity's source pads are returned during the enumeration process.

**Table F-1. struct media_links_enum**

| | | |
|---|---|---|
| __u32 | *entity* | Entity id, set by the application. |
| struct struct media_pad_desc | **pads* | Pointer to a pads array allocated by the application. Ignored if NULL. |
| struct struct media_link_desc | **links* | Pointer to a links array allocated by the application. Ignored if NULL. |

**Table F-2. struct media_pad_desc**

| | | |
|---|---|---|
| __u32 | *entity* | ID of the entity this pad belongs to. |
| __u16 | *index* | 0-based pad index. |
| __u32 | *flags* | Pad flags, see Table F-3 for more details. |

**Table F-3. Media pad flags**

| | |
|---|---|
| MEDIA_PAD_FL_SINK | Input pad, relative to the entity. Input pads sink data and are targets of links. |
| MEDIA_PAD_FL_SOURCE | Output pad, relative to the entity. Output pads source data and are origins of links. |

**Table F-4. struct media_links_desc**

| | | |
|---|---|---|
| struct struct media_pad_desc | *source* | Pad at the origin of this link. |
| struct struct media_pad_desc | *sink* | Pad at the target of this link. |
| __u32 | *flags* | Link flags, see Table F-5 for more details. |

**Table F-5. Media link flags**

| | |
|---|---|
| MEDIA_LNK_FL_ENABLED | The link is enabled and can be used to transfer media data. When two or more links target a sink pad, only one of them can be enabled at a time. |
| MEDIA_LNK_FL_IMMUTABLE | The link enabled state can't be modified at runtime. An immutable link is always enabled. |
| MEDIA_LNK_FL_DYNAMIC | The link enabled state can be modified during streaming. This flag is set by drivers and is read-only for applications. |

One and only one of MEDIA_PAD_FL_SINK and MEDIA_PAD_FL_SOURCE must be set for every pad.

## Return Value

On success 0 is returned, on error -1 and the errno variable is set appropriately:

EINVAL

The struct media_links_enum *id* references a non-existing entity.

Link configuration has no side effect on other links. If an enabled link at the sink pad prevents the link from being enabled, the driver returns with an EBUSY error code.

Only links marked with the DYNAMIC link flag can be enabled/disabled while streaming media data. Attempting to enable or disable a streaming non-dynamic link will return an EBUSY error code.

If the specified link can't be found the driver returns with an EINVAL error code.

# ioctl MEDIA_IOC_SETUP_LINK

## Name

MEDIA_IOC_SETUP_LINK — Modify the properties of a link

## Return Value

On success 0 is returned, on error -1 and the errno variable is set appropriately:

## Synopsis

EBUSY

```
int ioctl(int fd, int request, struct media_link_desc *argp);
```

The link properties can't be changed because the link is currently busy. This can be caused, for instance, by an active media stream (audio or video) on the link. The ioctl shouldn't be retried if no other action is performed before to fix the problem.

EINVAL

## Arguments

The struct media_link_desc references a non-existing link, or the link is immutable and an attempt to modify its configuration was made.

*fd*

File descriptor returned by open().

*request*

MEDIA_IOC_SETUP_LINK

*argp*

## Description

To change link properties applications fill a struct media_link_desc with link identification information (source and sink pad) and the new requested link flags. They then call the MEDIA_IOC_SETUP_LINK ioctl with a pointer to that structure.

The only configurable property is the ENABLED link flag to enable/disable a link. Links marked with the IMMUTABLE link flag can not be enabled or disabled.

# Appendix G. GNU Free Documentation License

## G.1. 0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

## G.2. 1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical

connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

## G.4. 3. COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

## G.3. 2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notices saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However you may accept compensation in exchange

## G.5. 4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified

Version under the terms of section 4. Modified Versions, which serve as a definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

## G.6. 5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the

## G.9. 8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

## G.7. 6. COLLECTIONS OF DOCUMENTS

## G.10. 9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

## G.11. 10. FUTURE REVISIONS OF THIS LICENSE

## G.6. 7. AGGREGATION WITH INDEPENDENT WORKS

The Free Software Foundation (http://www.gnu.org/fsf/fsf.html) may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/ (http://www.gnu.org/copyleft).

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document. If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

# G.12. Addendum

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright © YEAR YOUR NAME.
>
> Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being LIST"; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License (http://www.gnu.org/copyleft/gpl.html), to permit their use in free software.