LINUX DVB API

Convergence integrated media GmbH

Dr. Ralph J.K. Metzler
<rjkm@convergence.de>

Dr. Marcus O.C. Metzler
<mocm@convergence.de>

08/17/2001
V 0.9.2

Contents

Chapter 1

Introduction

1.1 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 ISO/IEC 13818, 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.

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++.

1.2 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 features like recording MPEG streams and filtering several section and PES data streams at the same time.

In early 2000, we were approached by Nokia with a proposal for a new standard Linux DVB API. As a commitment to the development of terminals based on open standards, Nokia and Convergence made it available to all Linux developers and published it on `http://www.linuxtv.org/` in September 2000. Convergence is the maintainer of the Linux DVB API. Together with the LinuxTV community (i.e. you, the reader of this document), the Linux DVB API will be constantly reviewed and improved upon. With the Linux driver for the Siemens/Hauppauge DVB PCI card Convergence provides a first implementation of the Linux DVB API.

1.3 Overview

A DVB PCI card or DVB set-top-box (STB) usually consists of the following main hardware components:

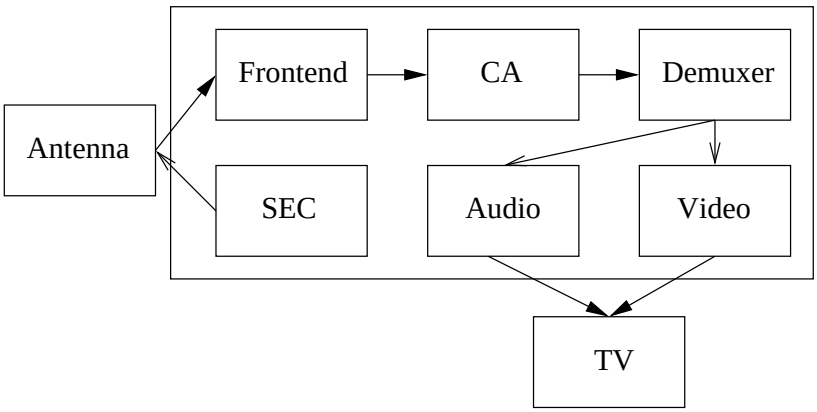

Figure 1.1: Components of a DVB card/STB

- 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).

- SEC for controlling external hardware like LNBs and antennas

  This part of the hardware can send signals back through the satellite cable to control the polarization of the LNB, to switch between different LNBs or even to control the movements of a dish rotor.

- 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 and video streams. Besides usually several of such audio and video streams it also contains data strams with information about the programs offered in this or other streams of the same provider.

- MPEG2 audio and video decoder

  The main targets of the demultiplexer are the MPEG2 audio and video decoders. After decoding the pass on the uncompressed audio and video to the computer screen or (through a PAL/NTSC encoder) to a TV set.

Figure 1.1 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.

1.4 Linux DVB Devices

The Linux DVB API lets you control these hardware components through currently six Unix-style character devices for video, audio, frontend, SEC, demux and CA. The video and audio devices control the MPEG2 decoder hardware, the frontend device the tuner and the DVB demodulator. External hardware like DiSEqC switches and rotors can be controlled through the SEC device. 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/ost`, where OST stands for Open Standards Terminal. The individual devices are called

- `/dev/ost/audio`,
- `/dev/ost/video`,
- `/dev/ost/qpskfe`,
- `/dev/ost/qamfe`,
- `/dev/ost/sec`,
- `/dev/ost/demux`,
- `/dev/ost/ca`,

but we will omit the “`/dev/ost/`” in the further dicussion of these devices.

If more than one card is present in the system the other cards can be accessed through the corresponding devices with the card’s number appended. `/dev/ost/demux0` (which is identical to `/dev/ost/demux`) would, e.g., control the demultiplexer of the first card, while `/dev/ost/demux1` would control the demultiplexer of the second card, and so on.

More details about the data structures and function calls of all the devices are described in the following chapters.

1.5 DVB Devices with Devfs

Recent Linux kernel versions support a special file system called *devfs* which is a replacement for the traditional device directory. With devfs a Linux DVB driver will only

create those device file entries which actually exist. It also makes dealing with more complex DVB hardware much easier. The device structure described above is not well suited to deal with multiple DVB cards with more than one frontend or demultiplexer. Consider, e.g., two DVB cards, one with two frontends and one demultiplexer, the other with one frontend and two demultiplexers. If we just assign them consecutive numbers, there would be a demultiplexer and a frontend which do notbelong to the same card but have the same number.

With *devfs* we propose a different scheme for the device names. The root directory for all DVB cards will be `/dev/dvb`. Each card gets assigned a sub-directory with the name `/dev/card0`, `/dev/card1`, etc. The files created in these sub-directories will correspond directly to the hardware actually present on the card. Thus, if the first card has one QAM frontend, one demultiplexer and otherwise no other hardware, only `/dev/dvb/card0/qamfe0` and `/dev/dvb/card0/demux0` will be created. When a second DVB-S card with one frontend (including SEC) device, two demultiplexers and an MPEG2 audio/video decoder is added, the complete `/dev/dvb` tree will look like this:

```
/dev/dvb/card0/qam0
               demux0

/dev/dvb/card1/video0
               audio0
               demux0
               demux1
               qpskfe0
               sec0
```

1.6 Using the Devices

...

Chapter 2

DVB Video Device

The DVB video device controls the MPEG2 video decoder of the DVB hardware. It can be accessed through `/dev/ost/video`. The include file `ost/video.h` defines the data types and lists all I/O calls.

2.1 Video Data Types

2.1.1 videoFormat_t

The `videoFormat_t` data type defined by

```
typedef enum {
        VIDEO_FORMAT_4_3,
        VIDEO_FORMAT_16_9
} videoFormat_t;
```

is used in the VIDEO_SET_FORMAT function (2.2.20) to tell the driver which aspect ratio the output hardware (e.g. TV) has. It is also used in the data structures videoStatus (2.1.6) returned by VIDEO_GET_STATUS (2.2.10) and videoEvent (2.1.5) returned by VIDEO_GET_EVENT (2.2.11) which report about the display format of the current video stream.

2.1.2 videoDisplayFormat_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 (2.2.12) which accepts

```
typedef enum {
        VIDEO_PAN_SCAN,
        VIDEO_LETTER_BOX,
        VIDEO_CENTER_CUT_OUT
} videoDisplayFormat_t;
```

as argument.

2.1.3 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
} videoStreamSource_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.

2.1.4 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
} videoPlayState_t;
```

2.1.5 video event

The following is the structure of a video event as it is returned by the VIDEO_GET_EVENT call.

```
struct videoEvent {
        int32_t type;
        time_t timestamp;
        union {
                videoFormat_t videoFormat;
        } u;
};
```

2.1.6 video status

The VIDEO_GET_STATUS call returns the following structure informing about various states of the playback operation.

```
struct videoStatus {
        boolean videoBlank;
        videoPlayState_t playState;
        videoStreamSource_t streamSource;
        videoFormat_t videoFormat;
        videoDisplayFormat_t displayFormat;
};
```

If videoBlank is set video will be blanked out if the channel is changed or if playback is stopped. Otherwise, the last picture will be displayed. playState indicates if the video is currently frozen, stopped, or being played back. The streamSource corresponds to the seleted source for the video stream. It can come either from the demultiplexer or from memory. The videoFormat indicates the aspect ratio (one of 4:3 or 16:9) of the currently played video stream. Finally, displayFormat corresponds to the selected cropping mode in case the source video format is not the same as the format of the output device.

2.1.7 video display 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 videoDisplayStillPicture {
        char *iFrame;
        int32_t size;
};
```

2.1.8 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
```

2.1.9 video system

A call to VIDEO_SET_SYSTEM sets the desired video system for TV output. The following system types can be set:

```
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
} videoSystem_t;
```

2.1.10 video highlights

Calling the ioctl VIDEO_SET_HIGHLIGHTS posts the SPU highlight information. The call expects the following format for that information:

```
typedef
struct videoHighlight {
        boolean active;      /*    1=show highlight, 0=hide highlight */
        uint8_t contrast1;   /*    7- 4  Pattern pixel contrast */
                             /*    3- 0  Background pixel contrast */
        uint8_t contrast2;   /*    7- 4  Emphasis pixel-2 contrast */
                             /*    3- 0  Emphasis pixel-1 contrast */
        uint8_t color1;      /*    7- 4  Pattern pixel color */
                             /*    3- 0  Background pixel color */
        uint8_t color2;      /*    7- 4  Emphasis pixel-2 color */
                             /*    3- 0  Emphasis pixel-1 color */
        uint32_t ypos;       /*   23-22  auto action mode */
                             /*   21-12  start y */
                             /*    9- 0  end y */
        uint32_t xpos;       /*   23-22  button color number */
                             /*   21-12  start x */
                             /*    9- 0  end x */
} videoHighlight_t;
```

2.1.11 video SPU

Calling VIDEO_SET_SPU deactivates or activates SPU decoding, according to the following format:

```
typedef
struct videoSPU {
```

```
        boolean active;
        int streamID;
} videoSPU_t;
```

2.1.12 video SPU palette

The following structure is used to set the SPU palette by calling VIDEO_SPU_PALETTE:

```
typedef
struct videoSPUPalette{      /* SPU Palette information */
        int length;
        uint8_t *palette;
} videoSPUPalette_t;
```

2.1.13 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 videoNaviPack{
        int length;          /* 0 ... 1024 */
        uint8_t data[1024];
} videoNaviPack_t;
```

2.1.14 video attributes

The following attributes can be set by a call to VIDEO_SET_ATTRIBUTES:

```
typedef uint16_t videoAttributes_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)) */
```

2.2 Video Function Calls

2.2.1 open()

DESCRIPTION

This system call opens a named video device (e.g. /dev/ost/video) 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.

2.2.2 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.

2.2.3 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.

2.2.4 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.

2.2.5 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.

2.2.6 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.

2.2.7 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.

2.2.8 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,
videoStreamSource_t source);
```

PARAMETERS

int fd	File descriptor returned by a previous call to open().
int request	Equals VIDEO_SELECT_SOURCE for this command.
videoStreamSource_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.

2.2.9 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

2.2.10 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
videoStatus *status);
```

PARAMETERS

int fd	File descriptor returned by a previous call to open().
int request	Equals VIDEO_GET_STATUS for this command.
struct videoStatus *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

2.2.11 VIDEO_GET_EVENT

DESCRIPTION

This ioctl call returns an event of type videoEvent 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
videoEvent *ev);
```

PARAMETERS

int fd	File descriptor returned by a previous call to open().
int request	Equals VIDEO_GET_EVENT for this command.
struct videoEvent *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.
EBUFFEROVERFLOW	Overflow in event queue - one or more events were lost.

2.2.12 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,
videoDisplayFormat_t format);
```

PARAMETERS

int fd	File descriptor returned by a previous call to open().
int request	Equals VIDEO_SET_DISPLAY_FORMAT for this command.
videoDisplayFormat_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.

2.2.13 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 videoDisplayStillPicture *sp);
```

PARAMETERS

int fd	File descriptor returned by a previous call to open().
int request	Equals VIDEO_STILLPICTURE for this command.
struct videoDisplayStillPicture *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.

2.2.14 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.

2.2.15 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.

2.2.16 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 2.1.8.

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.

2.2.17 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.

2.2.18 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

2.2.19 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.

2.2.20 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,
videoFormat_t format);
```

PARAMETERS

int fd	File descriptor returned by a previous call to open().
int request	Equals VIDEO_SET_FORMAT for this command.
videoFormat_t format	video format of TV as defined in section 2.1.1.

ERRORS

EBADF	fd is not a valid open file descriptor
EINVAL	format is not a valid video format.

2.2.21 VIDEO_SET_SYSTEM

DESCRIPTION

This ioctl sets the television output format. The format (see section 2.1.9) 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 ,
videoSystem_t system);
```

PARAMETERS

int fd	File descriptor returned by a previous call to open().
int request	Equals VIDEO_SET_FORMAT for this command.
videoSystem_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.

2.2.22 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
,videoHighlight_t *vhilite)
```

PARAMETERS

int fd	File descriptor returned by a previous call to open().
int request	Equals VIDEO_SET_HIGHLIGHT for this command.
videoHighlight_t *vhilite	SPU Highlight information according to section 2.1.10.

ERRORS

EBADF	fd is not a valid open file descriptor.
EINVAL	input is not a valid highlight setting.

2.2.23 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 ,
videoSPU_t *spu)
```

PARAMETERS

int fd	File descriptor returned by a previous call to open().
int request	Equals VIDEO_SET_SPU for this command.
videoSPU_t *spu	SPU decoding (de)activation and subid setting according to section 2.1.11.

ERRORS

EBADF	fd is not a valid open file descriptor
EINVAL	input is not a valid spu setting or driver cannot handle SPU.

2.2.24 VIDEO_SET_SPU_PALETTE

DESCRIPTION

This ioctl sets the SPU color palette.

SYNOPSIS

```
int ioctl(fd, int request = VIDEO_SET_SPU_PALETTE
,videoSPUPalette_t *palette )
```

PARAMETERS

int fd	File descriptor returned by a previous call to open().
int request	Equals VIDEO_SET_SPU_PALETTE for this command.
videoSPUPalette_t *palette	SPU palette according to section 2.1.12.

ERRORS

EBADF	fd is not a valid open file descriptor
EINVAL	input is not a valid palette or driver doesn't handle SPU.

2.2.25 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 ,
videoNaviPack_t *navipack)
```

PARAMETERS

int fd	File descriptor returned by a previous call to open().
int request	Equals VIDEO_GET_NAVI for this command.
videoNaviPack_t *navipack	PCI or DSI pack (private stream 2) according to section 2.1.13.

ERRORS

EBADF	fd is not a valid open file descriptor
EFAULT	driver is not able to return navigational information

2.2.26 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
,videoAttributes_t vattr)
```

PARAMETERS

int fd	File descriptor returned by a previous call to open().
int request	Equals VIDEO_SET_ATTRIBUTE for this command.
videoAttributes_t vattr	video attributes according to section 2.1.14.

ERRORS

EBADF	fd is not a valid open file descriptor
EINVAL	input is not a valid attribute setting.

Chapter 3

DVB Audio Device

The DVB audio device controls the MPEG2 audio decoder of the DVB hardware. It can be accessed through `/dev/ost/audio`.

3.1 Audio Data Types

This section describes the structures, data types and defines used when talking to the audio device.

3.1.1 audioStreamSource_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 (demuxer) or external (user write) source.

```
typedef enum {
        AUDIO_SOURCE_DEMUX,
        AUDIO_SOURCE_MEMORY
} audioStreamSource_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.

3.1.2 audioPlayState_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
} audioPlayState_t;
```

3.1.3 audioChannelSelect_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,
} audioChannelSelect_t;
```

3.1.4 audioStatus_t

The AUDIO_GET_STATUS call returns the following structure informing about various states of the playback operation.

```
typedef struct audioStatus {
        boolean AVSyncState;
        boolean muteState;
        audioPlayState_t playState;
        audioStreamSource_t streamSource;
        audioChannelSelect_t channelSelect;
        boolean bypassMode;
} audioStatus_t;
```

3.1.5 audioMixer_t

The following structure is used by the AUDIO_SET_MIXER call to set the audio volume.

```
typedef struct audioMixer {
        unsigned int volume_left;
        unsigned int volume_right;
} audioMixer_t;
```

3.1.6 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
```

3.1.7 audio karaoke

The ioctl AUDIO_SET_KARAOKE uses the following format:

```
typedef
struct audioKaraoke{          /* if Vocal1 or Vocal2 are non-zero, they get mixed
        int vocal1;           /* into left and right t at 70% each */
        int vocal2;           /* if both, Vocal1 and Vocal2 are non-zero, Vocal1 ge
        int melody;           /* mixed into the left channel and */
                              /* Vocal2 into the right channel at 100% each. */
                              /* if Melody is non-zero, the melody channel gets mix
} audioKaraoke_t;
```

3.1.8 audio attributes

The following attributes can be set by a call to AUDIO_SET_ATTRIBUTES:

```
typedef uint16_t audioAttributes_t;
/*   bits: descr. */
/*   15-13 audio coding mode (0=ac3, 2=mpeg1, 3=mpeg2ext, 4=LPCM, 6=DTS, */
/*   12    multichannel extension */
/*   11-10 audio type (0=not spec, 1=language included) */
/*    9- 8 audio application mode (0=not spec, 1=karaoke, 2=surround) */
/*    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) */
```

3.2 Audio Function Calls

3.2.1 open()

DESCRIPTION

This system call opens a named audio device (e.g. /dev/ost/audio) 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.

3.2.2 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.

3.2.3 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.

3.2.4 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.

3.2.5 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.

3.2.6 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.

3.2.7 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,
audioStreamSource_t source);
```

PARAMETERS

int fd	File descriptor returned by a previous call to open().
int request	Equals AUDIO_SELECT_SOURCE for this command.
audioStreamSource_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.

3.2.8 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.

3.2.9 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.

3.2.10 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.

3.2.11 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, audioChannelSelect_t);
```

PARAMETERS

int fd	File descriptor returned by a previous call to open().
int request	Equals AUDIO_CHANNEL_SELECT for this command.
audioChannelSelect_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.

3.2.12 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 audioStatus *status);
```

PARAMETERS

int fd	File descriptor returned by a previous call to open().
int request	Equals AUDIO_GET_STATUS for this command.
struct audioStatus *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.

3.2.13 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.

3.2.14 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.

3.2.15 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.

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.

3.2.16 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,
audioMixer_t *mix);
```

PARAMETERS

int fd	File descriptor returned by a previous call to open().
int request	Equals AUDIO_SET_ID for this command.
audioMixer_t *mix	mixer settings.

ERRORS

EBADF	fd is not a valid open file descriptor.
EINTERNAL	Internal error.
EFAULT	mix points to an invalid address.

3.2.17 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 MPEG2 audio 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.

3.2.18 AUDIO_SET_EXT_ID

DESCRIPTION

This ioctl can be used to set the audio sub_stream_id for DVD playback

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.

3.2.19 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,
audioAttributes_t attr );
```

PARAMETERS

int fd	File descriptor returned by a previous call to open().
int request	Equals AUDIO_SET_ATTRIBUTES for this command.
iaudioAttributes_t attr	audio attributes according to section 3.1.8

ERRORS

EBADF	fd is not a valid open file descriptor
EINVAL	attr is not a valid or supported attribute setting.

3.2.20 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,
audioKaraoke_t *karaoke);
```

PARAMETERS

int fd	File descriptor returned by a previous call to open().
int request	Equals AUDIO_SET_STREAMTYPE for this command.
audioKaraoke_t *karaoke	karaoke settings according to section 3.1.7.

ERRORS

EBADF	fd is not a valid open file descriptor
EINVAL	karaoke is not a valid or supported karaoke setting.

Chapter 4

DVB Frontend API

The DVB frontend device controls the tuner and DVB demodulator hardware. It can be accessed through `/dev/ost/qpskfe` or `/dev/ost/qamfe`. If you are using `devfs` you can use `/dev/dvb/card0/qpskfe`, `/dev/dvb/card0/qamfe`, `/dev/dvb/card1/qpskfe`, etc. Only the actually present kind of frontend for each card will be made visible through `devfs`.

4.1 Frontend Data Types

4.1.1 frontend status

Several functions of the frontend device use the feStatus data type defined by

```
typedef uint32_t feStatus;
```

to indicate the current state and/or state changes of the frontend hardware.
It can take on the values

```
#define FE_HAS_POWER          1
#define FE_HAS_SIGNAL         2
#define QPSK_SPECTRUM_INV     4
#define FE_HAS_LOCK           8
#define TUNER_HAS_LOCK      128
```

which can be ORed together and have the following meaning:

FE_HAS_POWER	the frontend is powered up and is ready to be used
FE_HAS_SIGNAL	the frontend detects a signal above the normal noise level
QPSK_SPECTRUM_INV	spectrum inversion is enabled/was necessary for lock
FE_HAS_LOCK	frontend successfully locked to a DVB signal
TUNER_HAS_LOCK	the tuner has a frequency lock

4.1.2 frontend parameters

The kind of parameters passed to the frontend device for tuning depend on the kind of hardware you are using. For satellite QPSK frontend you have to use qpskParameters defined by:

```
struct qpskParameters {
        uint32_t iFrequency;
        uint32_t SymbolRate;
        uint8_t FEC_inner;
};
```

for cable QAM frontend you use qamParameters defined by:

```
struct qamParameters {
        uint32_t Frequency;
        uint32_t SymbolRate;
        uint8_t FEC_inner;
        uint8_t QAM;
};
```

In the case of QPSK frontends the iFrequency 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 frontends the Frequency specifies the absolute frequency and is given in Hz.
The possible values for the FEC_inner field are

```
enum {
        FEC_AUTO,
        FEC_1_2,
        FEC_2_3,
        FEC_3_4,
        FEC_5_6,
        FEC_7_8,
        FEC_NONE
};
```

which correspond to error correction rates of $\frac{1}{2}$, $\frac{2}{3}$, etc., no error correction or auto detection.
For cable frontends one also has to specify the quadrature modulation mode which can be one of the following:

```
typedef enum
{       QAM_16,
        QAM_32,
        QAM_64,
        QAM_128,
        QAM_256
} QAM_TYPE;
```

4.1.3 frontend events

```
enum {
        QPSK_UNEXPECTED_EV,
        QPSK_COMPLETION_EV,
        QPSK_FAILURE_EV
};

enum {
        FE_UNEXPECTED_EV,
        FE_COMPLETION_EV,
        FE_FAILURE_EV
};

struct qpskEvent {
        int32_t type;
        time_t timestamp;
        union {
                struct {
                        feStatus previousStatus;
                        feStatus currentStatus;
                } unexpectedEvent;
                struct qpskParameters completionEvent;
                feStatus failureEvent;
        } u;
};

struct qamEvent {
        int32_t type;
        time_t timestamp;
        union {
                struct {
                        feStatus previousStatus;
                        feStatus currentStatus;
                } unexpectedEvent;
                struct qamParameters completionEvent;
                feStatus failureEvent;
        } u;
};

struct qpskRegister {
        uint8_t chipId;
        uint8_t address;
        uint8_t value;
};

struct qamRegister {
```

```
        uint8_t chipId;
        uint8_t address;
        uint8_t value;
};

struct qpskFrontendInfo {
        uint32_t minFrequency;
        uint32_t maxFrequency;
        uint32_t maxSymbolRate;
        uint32_t minSymbolRate;
        uint32_t hwType;
        uint32_t hwVersion;
};

struct qamFrontendInfo {
        uint32_t minFrequency;
        uint32_t maxFrequency;
        uint32_t maxSymbolRate;
        uint32_t minSymbolRate;
        uint32_t hwType;
        uint32_t hwVersion;
};

typedef enum {
        OST_POWER_ON,
        OST_POWER_STANDBY,
        OST_POWER_SUSPEND,
        OST_POWER_OFF
} powerState_t;
```

4.2 Frontend Function Calls

4.2.1 open()

DESCRIPTION

This system call opens a named frontend device (e.g. /dev/ost/qpskfe for a satellite frontend or /dev/ost/qamfe for a cable frontend) for subsequent use.
The device can be opened in read-only mode, which only allows monitoring of device status and statistics, or read/write mode, which allows any kind of use (e.g. performing tuning operations.)
In a system with multiple front-ends, it is usually the case that multiple devices cannot be open in read/write mode simultaneously. As long as a front-end device is opened in read/write mode, other open() calls in read/write mode will either fail or block, depending on whether non-blocking or blocking mode was specified. A front-end 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. When an open() call has succeeded, the device will be ready for use in the specified mode. This implies that the corresponding hardware is powered up, and that other front-ends may have been powered down to make that possible.

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.

4.2.2 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, but only when this is needed to open another front-end device. To affect an unconditional power down, it should be done explicitly using the OST_SET_POWER_STATE ioctl. 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.

4.2.3 OST_SELFTEST

DESCRIPTION

This ioctl call initiates an automatic self-test of the front-end hardware. This call requires read/write access to the device.

SYNOPSIS

```
int ioctl(int fd, int request = OST_SELFTEST);
```

PARAMETERS

int fd	File descriptor returned by a previous call to open().
int request	Equals OST_SELFTEST for this command.

ERRORS

-1	Self test failure.

4.2.4 OST_SET_POWER_STATE

DESCRIPTION

This ioctl call, implemented in many OST device drivers, enables direct control over the power state of the hardware device, which may be on, off, standby, or suspend. The latter two are low-power modes, which disable all functionality of the device until turned on again. In contrast to the off state, however, the standby and suspend states resume operation in the same state as when the device was active. The only difference between the standby and suspend states is a different tradeoff between resume time and power consumption. Power consumption may be lower in the suspend state at the cost of a longer resume time.

A device that implements this call does not necessarily support two low-power modes. If it only supports one low-power state, or none at all, the OST_SET_POWER_STATE operation for the missing states will still succeed, but it will be mapped to an existing state as per this table:

number of low-power states supported	requested state	resulting state
1	standby	suspend
1	suspend	suspend
0	standby	on
0	suspend	on

For other cases where a required state is missing, an error code will be returned. This can happen if a device does not support the power-off state, but nevertheless implements this ioctl operation for control of low-power states. When opening a device in read/write mode, the driver ensures that the corresponding hardware device is turned on initially. If the device is later turned off or put in suspend mode, it has to be explicitly turned on again.

This call requires read/write access to the device. (Note that the power management driver can affect the power state of devices without using this ioctl operation, so having exclusive read/write access to a device does not imply total control over the power state.)

SYNOPSIS

```
int ioctl(int fd, int request = OST_SET_POWER_STATE,
uint32_t state);
```

PARAMETERS

int fd	File descriptor returned by a previous call to open().	
int request	Equals OST_SET_POWER_STATE for this command.	
uint32_t state	Requested power state. One of:	
	OST_POWER_ON	turn power on
	OST_POWER_STANDBY	set device in standby mode
	OST_POWER_SUSPEND	set device in suspend mode
	OST_POWER_OFF	turn power off

ERRORS

EBADF	fd is not a valid open file descriptor.
EINVAL	Illegal state, or not available on this device.
EPERM	Permission denied (needs read/write access).
ENOSYS	Function not available for this device.

4.2.5 OST_GET_POWER_STATE

DESCRIPTION

This ioctl call, implemented in many OST device drivers, obtains the power state of the hardware device, which may be on, off, standby, or suspend. A device that implements this call does not necessarily support all four states. If there is only one low-power state, the suspend state will be returned for that state. If there is no low-power state, the on state will be reported standby and suspend states will be equivalent to the on state. For this command, read-only access to the device is sufficient.

SYNOPSIS

```
int ioctl(int fd, int request = OST_GET_POWER_STATE,
uint32_t *state);
```

PARAMETERS

int fd	File descriptor returned by a previous call to open().	
int request	Equals OST_GET_POWER_STATE for this command.	
uint32_t *state	Requested power state. One of:	
	OST_POWER_ON	power is on
	OST_POWER_STANDBY	device in standby mode
	OST_POWER_SUSPEND	device in suspend mode
	OST_POWER_OFF	power is off

ERRORS

EBADF	fd is not a valid open file descriptor.
EINVAL	Illegal state, or not available on this device.
EFAULT	state points to invalid address.
ENOSYS	Function not available for this device.

4.2.6 QPSK_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 = QPSK_READ_STATUS,
feStatus *status);
```

PARAMETERS

int fd	File descriptor returned by a previous call to open().
int request	Equals QPSK_READ_STATUS for this command.
struct feStatus *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.

4.2.7 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, as a multiple of 10^{-9}, is stored into *ber. Example: a value of 2500 corresponds to a bit error rate of $2.5 \cdot 10^{-6}$, or 1 error in 400000 bits.

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.

4.2.8 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, int32_t
*snr);
```

PARAMETERS

int fd	File descriptor returned by a previous call to open().
int request	Equals FE_READ_SNR for this command.
int32_t *snr	The signal-to-noise ratio, as a multiple of 10^{-6} dB, is stored into *snr. Example: a value of 12,300,000 corresponds to a signal-to-noise ratio of 12.3 dB.

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.

4.2.9 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, int32_t *strength);
```

PARAMETERS

int fd	File descriptor returned by a previous call to open().
int request	Equals FE_READ_SIGNAL_STRENGTH for this command.
int32_t *strength	The signal strength value, as a multiple of 10^{-6} dBm, is stored into *strength. Example: a value of -12,500,000 corresponds to a signal strength value of -12.5 dBm.

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.

4.2.10 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.

4.2.11 FE_GET_NEXT_FREQUENCY

DESCRIPTION

When scanning a frequency range, it is desirable to use a scanning step size that is as large as possible, yet small enough to be able to lock to any signal within the range. This ioctl operation does just that - it increments a given frequency by a step size suitable for efficient scanning. The step size used by this function may be a quite complex function of the given frequency, hardware capabilities, and parameter settings of the device. Thus, a returned result is only valid for the current state of the device. For this command, read-only access to the device is sufficient.
Note that scanning may still be excruciatingly slow on some hardware, for other reasons than a non-optimal scanning step size.

SYNOPSIS

```
int ioctl( int fd, int request =
FE_GET_NEXT_FREQUENCY, uint32_t *freq);
```

PARAMETERS

int fd	File descriptor returned by a previous call to open().
int request	Equals FE_GET_NEXT_FREQUENCY for this command.
uint32_t *freq	Input: a given frequency Output: the frequency corresponding to the next higher frequency setting.

ERRORS

EBADF	fd is not a valid open file descriptor.
EFAULT	freq points to invalid address.
EINVAL	Maximum supported frequency reached.
ENOSYS	Function not available for this device.

4.2.12 FE_GET_NEXT_SYMBOL_RATE

DESCRIPTION

When scanning a range of symbol rates (e.g. for ”blind acquisition”) it is desirable to use a scanning step size that is as large as possible, yet small enough to detect any valid signal within the range. This ioctl operation does just that - it increments a given symbol rate by a step size suitable for efficient scanning. The step size used by this function may be a quite complex function of the given symbol rate, hardware capabilities, and parameter settings of the device. Thus, a returned result is only valid for the current state of the device. For this command, read-only access to the device is sufficient.

SYNOPSIS

```
int ioctl( int fd, int request =
FE_GET_NEXT_SYMBOL_RATE, uint32_t *symbolRate);
```

PARAMETERS

int fd	File descriptor returned by a previous call to open().
int request	Equals FE_GET_NEXT_SYMBOL_RATE for this command.
uint32_t *symbolRate	Input: a given symbol rate Output: the symbol rate corresponding to the next higher symbol rate.

ERRORS

EBADF	fd is not a valid open file descriptor.
EFAULT	symbolRate points to invalid address.
EINVAL	Maximum supported symbol rate reached.
ENOSYS	Function not available for this device.

4.2.13 QPSK_TUNE

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 QPSK_GET_EVENT and qpskEvent.) If a new QPSK_TUNE 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 = QPSK_TUNE, struct
qpskParameters *p);
```

PARAMETERS

int fd	File descriptor returned by a previous call to open().
int request	Equals QPSK_TUNE for this command.
struct qpskParameters *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.

4.2.14 QPSK_GET_EVENT

DESCRIPTION

This ioctl call returns an event of type qpskEvent 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 (EBUFFEROVERFLOW) occurs the next time the queue is read. After reporting the error condition in this fashion, subsequent QPSK_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 qpskEvent *ev);
```

PARAMETERS

int fd	File descriptor returned by a previous call to open().
int request	Equals QPSK_GET_EVENT for this command.
struct qpskEvent *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.
EBUFFEROVERFLOW	Overflow in event queue - one or more events were lost.

4.2.15 QPSK_FE_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 = QPSK_FE_INFO, struct
qpskFrontendInfo *info);
```

PARAMETERS

int fd	File descriptor returned by a previous call to open().
int request	Equals QPSK_FE_INFO for this command.
struct qpskFrontend-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.

4.2.16 QPSK_WRITE_REGISTER

DESCRIPTION

This ioctl call is intended for hardware-specific diagnostics. It writes an 8-bit value at an 8-bit address of a register in a chip identified by an 8-bit index.

SYNOPSIS

```
int ioctl(int fd, int request = QPSK_WRITE_REGISTER,
struct qpskRegister *reg);
```

PARAMETERS

int fd	File descriptor returned by a previous call to open().
int request	Equals QPSK_WRITE_REGISTER for this command.
struct qpskRegister *reg	Specifies a value that should be written into a specified register in a specified chip.

ERRORS

EBADF	fd is not a valid open file descriptor.
EFAULT	reg points to invalid address.
EINVAL	Register specification invalid.

4.2.17 QPSK_READ_REGISTER

DESCRIPTION

This ioctl call is intended for hardware-specific diagnostics. It reads an 8-bit value at an 8-bit address of a register in a chip identified by an 8-bit index.

SYNOPSIS

```
int ioctl(int fd, int request = QPSK_READ_REGISTER,
struct qpskRegister *reg);
```

PARAMETERS

int fd	File descriptor returned by a previous call to open().
int request	Equals QPSK_READ_REGISTER for this command.
struct qpskRegister *reg	I: specifies a register in a specified chip from which a value should be read. O: the value is read into the qpskRegister structure.

ERRORS

EBADF	fd is not a valid open file descriptor.
EFAULT	reg points to invalid address.
EINVAL	Register specification invalid.

4.2.18 QAM_TUNE

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 QAM_GET_EVENT and qamEvent.) If a new QAM_TUNE 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 = QAM_TUNE, struct
qamParameters *p);
```

PARAMETERS

int fd	File descriptor returned by a previous call to open().
int request	Equals QAM_TUNE for this command.
struct qamParameters *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.

4.2.19 QAM_GET_EVENT

DESCRIPTION

This ioctl call returns an event of type qamEvent 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 (EBUFFEROVERFLOW) occurs the next time the queue is read. After reporting the error condition in this fashion, subsequent QAM_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 = QAM_GET_EVENT,
struct qamEvent *ev);
```

PARAMETERS

int fd	File descriptor returned by a previous call to open().
int request	Equals QAM_GET_EVENT for this command.
struct qamEvent *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.
EBUFFEROVERFLOW	Overflow in event queue - one or more events were lost.

4.2.20 QAM_FE_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 = QAM_FE_INFO, struct
qamFrontendInfo *info);
```

PARAMETERS

int fd	File descriptor returned by a previous call to open().
int request	Equals QAM_FE_INFO for this command.
struct qamFrontend-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.

4.2.21 QAM_WRITE_REGISTER

DESCRIPTION

This ioctl call is intended for hardware-specific diagnostics. It writes an 8-bit value at an 8-bit address of a register in a chip identified by an 8-bit index.

SYNOPSIS

```
int ioctl(int fd, int request = QAM_WRITE_REGISTER,
struct qamRegister *reg);
```

PARAMETERS

int fd	File descriptor returned by a previous call to open().
int request	Equals QAM_WRITE_REGISTER for this command.
struct qamRegister *reg	Specifies a value that should be written into a specified register in a specified chip.

ERRORS

EBADF	fd is not a valid open file descriptor.
EFAULT	reg points to invalid address.
EINVAL	Register specification invalid.

4.2.22 QAM_READ_REGISTER

DESCRIPTION

This ioctl call is intended for hardware-specific diagnostics. It reads an 8-bit value at an 8-bit address of a register in a chip identified by an 8-bit index.

SYNOPSIS

```
int ioctl(int fd, int request = QAM_READ_REGISTER,
struct qamRegister *reg);
```

PARAMETERS

int fd	File descriptor returned by a previous call to open().
int request	Equals QAM_READ_REGISTER for this command.
struct qamRegister *reg	I: specifies a register in a specified chip from which a value should be read. O: the value is read into the qamRegister structure.

ERRORS

EBADF	fd is not a valid open file descriptor.
EFAULT	reg points to invalid address.
EINVAL	Register specification invalid.

Chapter 5

DVB SEC API

The DVB SEC device controls the Satellite Equipment Control of the DVB hardware, i.e. DiSEqC and V-SEC. It is accessed through `/dev/ost/sec`.

5.1 SEC Data Types

5.1.1 secDiseqcCmd

```
struct secDiseqcCmd {
        uint8_t addr;
        uint8_t cmd;
        uint8_t numParams;
        uint8_t params[SEC_MAX_DISEQC_PARAMS];
};
```

5.1.2 secVoltage

```
typedef uint32_t secVoltage;

enum {
        SEC_VOLTAGE_OFF,
        SEC_VOLTAGE_LT,
        SEC_VOLTAGE_13,
        SEC_VOLTAGE_13_5,
        SEC_VOLTAGE_18,
        SEC_VOLTAGE_18_5
};
```

5.1.3 secToneMode

```
typedef uint32_t secToneMode;
```

```
typedef enum {
        SEC_TONE_ON,
        SEC_TONE_OFF
} secToneMode_t;
```

5.1.4 secMiniCmd

```
typedef uint32_t secMiniCmd;
```

```
typedef enum {
        SEC_MINI_NONE,
        SEC_MINI_A,
        SEC_MINI_B
} secMiniCmd_t;
```

```
struct secStatus {
        int32_t     busMode;
        secVoltage  selVolt;
        secToneMode contTone;
};
```

```
enum {
        SEC_BUS_IDLE,
        SEC_BUS_BUSY,
        SEC_BUS_OFF,
        SEC_BUS_OVERLOAD
};
```

5.1.5 secCommand

```
struct secCommand {
        int32_t type;
        union {
                struct secDiseqcCmd diseqc;
                uint8_t vsec;
                uint32_t pause;
        } u;
};
```

5.1.6 secCmdSequence

```
struct secCmdSequence {
        secVoltage voltage;
        secMiniCmd miniCommand;
        secToneMode continuousTone;

        uint32_t numCommands;
```

```
        struct secCommand* commands;
};

enum {
        SEC_CMDTYPE_DISEQC,
        SEC_CMDTYPE_VSEC,
        SEC_CMDTYPE_PAUSE
};

typedef enum {
        SEC_DISEQC_SENT,
        SEC_VSEC_SENT,
        SEC_PAUSE_COMPLETE,
        SEC_CALLBACK_ERROR
} secCallback_t;
```

5.2 SEC Function Calls

5.2.1 open()

DESCRIPTION

This system call opens a named SEC device for subsequent use. If the device is opened in read-only mode, only status and statistics monitoring is allowed. If the device is opened in read/write mode, all types of operations can be performed. Any number of applications can have simultaneous access to the device.

SYNOPSIS

```
int open(const char *deviceName, int flags);
```

PARAMETERS

const char *deviceName	Name of specific SEC device.
int flags	A bit-wise OR of the following flags: O_RDONLY read-only access O_RDWR read/write access The optional flag O_NONBLOCK is not supported. If O_NONBLOCK is set, open() and most other subsequent calls to the device will return -1 and set errno to EWOULDBLOCK. The communication with the peripheral devices is sequential by nature, so it is probably preferable to use the device in synchronous mode. This is the motivation for not going through the extra effort of implementing asynchronous operation of the device.

ERRORS

ENODEV	Device driver not loaded/available.
EFAULT	deviceName does not refer to a valid memory area.
EBUSY	Device or resource busy.
EINVAL	Invalid argument.

5.2.2 close()

DESCRIPTION

This system call closes a previously opened SEC 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.

5.2.3 SEC_GET_STATUS

DESCRIPTION

This call gets the status of the device.

SYNOPSIS

```
int ioctl(int fd, int request = SEC_GET_STATUS,
struct secStatus* status);
```

PARAMETERS

int fd	File descriptor returned by a previous call to open().
int request	Equals SEC_GET_STATUS for this command.
struct secStatus* status	The status of the device.

ERRORS

ENODEV	Device driver not loaded/available.
EFAULT	status is an invalid pointer.
EBUSY	Device or resource busy.
EINVAL	Invalid argument.
EPERM	File not opened with read permissions.
EINTERNAL	Internal error in the device driver.

5.2.4 SEC_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.

SYNOPSIS

```
int ioctl(int fd, int request = SEC_RESET_OVERLOAD);
```

PARAMETERS

int fd	File descriptor returned by a previous call to open().
int request	Equals SEC_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.

5.2.5 SEC_RESET_OVERLOAD

DESCRIPTION

This ioctl call is used to send a sequence of DiSEqCTM and/or V-SEC commands. The first version of the SEC device does not support V-SEC signaling and it aborts the operation with an error code if a V-SEC command is detected in the input data.

- The call will fail with errno set to EBUSOVERLOAD if the bus is overloaded. If the bus is overloaded, SEC_RESET_OVERLOAD can be called and the operation can be retried.
- If seq.numCommands equals 0 and seq.miniCommand equals SEC_MINI_NONE, the bus voltage will be switched and the continuous 22kHz tone generation enabled/disabled immediately.

This operation is atomic. If several processes calls this ioctl simultaneously, the operations will be serialized so a complete sequence is sent at a time.

SYNOPSIS

```
int ioctl(int fd, int request = SEC_SEND_SEQUENCE,
struct secCmdSequence *seq);
```

PARAMETERS

int fd	File descriptor returned by a previous call to open().
int request	Equals SEC_SEND_SEQUENCE for this command.
struct secCmdSequence *seq	Pointer to the command sequence 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.
EBUSMODE	The device is not prepared for transmission (e.g. it might be manually powered off).
EBUSOVERLOAD	Bus overload has occurred.

5.2.6 SEC_SET_TONE

DESCRIPTION

This call is used to set the generation of the continuous 22kHz tone. The possibility to just change the tone mode is already provided by ioctl SEC_SEND_SEQUENCE, but SEC_SET_TONE is an easier to use interface. To keep the transmission of a command sequence as an atomic operation, SEC_SET_TONE will block if a transmission is in progress. This call requires read/write permissions.

SYNOPSIS

```
int ioctl(int fd, int request = SEC_SET_TONE,
secToneMode tone);
```

PARAMETERS

int fd	File descriptor returned by a previous call to open().
int request	Equals SEC_SET_TONE for this command.
secToneMode 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.

5.2.7 SEC_SET_VOLTAGE

DESCRIPTION

This call is used to set the bus voltage. The possibility to just change the bus voltage is already provided by ioctl SEC_SEND_SEQUENCE, but SEC_SET_VOLTAGE is an easier to use interface. To keep the transmission of a command sequence as an atomic operation, SEC_SET_VOLTAGE will block if a transmission is in progress. This call requires read/write permissions.

SYNOPSIS

```
int ioctl(int fd, int request = SEC_SET_VOLTAGE,
secVoltage voltage);
```

PARAMETERS

int fd	File descriptor returned by a previous call to open().
int request	Equals SEC_SET_VOLTAGE for this command.
secVoltage 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.

Chapter 6

DVB Demux Device

The DVB demux device controls the filters of the DVB hardware/software. It can be accessed through `/dev/ost/demux`.

6.1 Demux Data Types

```
typedef uint16_t dvb_pid_t;
```

6.1.1 dmxOutput_t

```
typedef enum
{
        DMX_OUT_DECODER,
        DMX_OUT_TAP,
        DMX_OUT_TS_TAP
} dmxOutput_t;
```

/* Output multiplexed into a new TS */ /* (to be retrieved by reading from the */ /* logical DVR device). */

6.1.2 dmxInput_t

```
typedef enum
{
        DMX_IN_FRONTEND, /* Input from a front-end device.  */
        DMX_IN_DVR       /* Input from the logical DVR device.  */
} dmxInput_t;
```

6.1.3 dmxPesType_t

```
typedef enum
{
```

```
        DMX_PES_AUDIO,
        DMX_PES_VIDEO,
        DMX_PES_TELETEXT,
        DMX_PES_SUBTITLE,
        DMX_PES_PCR,
        DMX_PES_OTHER
} dmxPesType_t;
```

6.1.4 dmxEvent_t

```
typedef enum
{
        DMX_SCRAMBLING_EV,
        DMX_FRONTEND_EV
} dmxEvent_t;
```

6.1.5 dmxScramblingStatus_t

```
typedef enum
{
        DMX_SCRAMBLING_OFF,
        DMX_SCRAMBLING_ON
} dmxScramblingStatus_t;
```

6.1.6 dmxFilter_t

```
typedef struct dmxFilter
{
        uint8_t         filter[DMX_FILTER_SIZE];
        uint8_t         mask[DMX_FILTER_SIZE];
} dmxFilter_t;
```

6.1.7 dmxSctFilterParams

```
struct dmxSctFilterParams
{
        dvb_pid_t                   pid;
        dmxFilter_t                 filter;
        uint32_t                    timeout;
        uint32_t                    flags;
#define DMX_CHECK_CRC       1
#define DMX_ONESHOT         2
#define DMX_IMMEDIATE_START 4
};
```

6.1.8 dmxPesFilterParams

```
struct dmxPesFilterParams
{
        dvb_pid_t                       pid;
        dmxInput_t                      input;
        dmxOutput_t                     output;
        dmxPesType_t                    pesType;
        uint32_t                        flags;
};
```

6.1.9 dmxEvent

```
struct dmxEvent
{
        dmxEvent_t                  event;
        time_t                      timeStamp;
        union
        {
                dmxScramblingStatus_t scrambling;
        } u;
};
```

6.2 Demux Function Calls

6.2.1 open()

DESCRIPTION

This system call, used with a device name of /dev/ost/demuxn, where n denotes the specific demux device to be opened, 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/ost/dvrn is a logical device to be used for retrieving Transport Streams for digital video recording. n identifies the physical demux device that provides the actual DVR functionality. When reading from this device a transport stream containing the packets from all PES filters set in the corresponding demux device (/dev/ost/demuxn) 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.

6.2.2 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.

6.2.3 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 EBUFFEROVERFLOW 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/ost/demuxn 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.
EBUFFEROVERFLOW	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.

6.2.4 write()

DESCRIPTION

This system call is only provided by the logical device /dev/ost/dvrn, where n identifies 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/ost/demuxn. 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.

6.2.5 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.

6.2.6 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.

6.2.7 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 dmxSctFilterParams *params);
```

PARAMETERS

int fd	File descriptor returned by a previous call to open().
int request	Equals DMX_SET_FILTER for this command.
struct dmxSctFilterParams *params	Pointer to structure containing filter parameters.

ERRORS

EBADF	fd is not a valid file descriptor.
EINVAL	Invalid argument.

6.2.8 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 dmxPesFilterParams *params);
```

PARAMETERS

int fd	File descriptor returned by a previous call to open().
int request	Equals DMX_SET_PES_FILTER for this command.
struct dmxPesFilterParams *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.

6.2.9 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.

6.2.10 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 dmxEvent *ev);
```

PARAMETERS

int fd	File descriptor returned by a previous call to open().
int request	Equals DMX_GET_EVENT for this command.
struct dmxEvent *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.

Chapter 7

DVB CA Device

The DVB CA device controls the conditional access hardware. It can be accessed through `/dev/ost/ca`.

7.1 CA Data Types

7.1.1 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;
```

7.1.2 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;
```

7.1.3 ca_cap_t

```
typedef struct ca_cap_s {
        unsigned int slot_num;  /* total number of CA card and module slo
        unsigned int slot_type; /* OR of all supported types */
        unsigned int descr_num; /* total number of descrambler slots (key
        unsigned int descr_type;/* OR of all supported types */
} ca_cap_t;
```

7.1.4 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;
```

7.1.5 ca_descr_t

```
typedef struct ca_descr_s {
        unsigned int index;
        unsigned int parity;
        unsigned char cw[8];
} ca_descr_t;
```

7.2 CA Function Calls

7.2.1 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.

7.2.2 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 8

Kernel Demux API

The kernel demux API

8.1 Kernel Demux Data Types

8.1.1 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;
```

8.1.2 TS filter types

```
/*--------------------------------------------------------------------------*/
/* TS packet reception */
/*--------------------------------------------------------------------------*/

/* TS filter type for set_type() */

#define TS_PACKET       1   /* send TS packets (188 bytes) to callback (default) *
#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 present) */
```

8.1.3 dmx_ts_pes_t

The structure /

```
typedef enum
{
        DMX_TS_PES_AUDIO,   /* also send packets to audio decoder (if it
        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 callback_length,
                    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 reception (not supported yet) */
/*--------------------------------------------------------------------------

typedef struct dmx_pes_filter_s {
        struct dmx_pes_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;

\label{sectionfilter}
typedef struct {
        __u8 filter_value [DMX_MAX_FILTER_SIZE];
        __u8 filter_mask [DMX_MAX_FILTER_SIZE];
        struct dmx_section_feed_s* parent; /* Back-pointer */
        void* priv; /* Pointer to private data of the API client */
} dmx_section_filter_t;

struct dmx_section_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_section_feed_s* feed,
                    __u16 pid,
                    size_t circular_buffer_size,
                    int descramble,
                    int check_crc);
        int (*allocate_filter) (struct dmx_section_feed_s* feed,
                                dmx_section_filter_t** filter);
        int (*release_filter) (struct dmx_section_feed_s* feed,
                               dmx_section_filter_t* filter);
        int (*start_filtering) (struct dmx_section_feed_s* feed);
        int (*stop_filtering) (struct dmx_section_feed_s* feed);
};
typedef struct dmx_section_feed_s dmx_section_feed_t;

/*--------------------------------------------------------------------------*/
/* 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 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_pes_cb) ( __u8 * buffer1,
                            size_t buffer1_len,
                            __u8 * buffer2,
                            size_t buffer2_len,
                            dmx_pes_filter_t* source,
                            dmx_success_t success);

/*--------------------------------------------------------------------------
/* DVB Front-End */
/*--------------------------------------------------------------------------

typedef enum {
        DMX_OTHER_FE = 0,
        DMX_SATELLITE_FE,
        DMX_CABLE_FE,
        DMX_TERRESTRIAL_FE,
        DMX_LVDS_FE,
        DMX_ASI_FE, /* DVB-ASI interface */
        DMX_MEMORY_FE
} dmx_frontend_source_t;

typedef struct {
        /* The following char* fields point to NULL terminated strings */
        char* id;                   /* Unique front-end identifier */
        char* vendor;               /* Name of the front-end vendor */
        char* model;                /* Name of the front-end model */
        struct list_head connectivity_list; /* List of front-ends that ca
                                               be connected to a particul
                                               demux */
        void* priv;     /* Pointer to private data of the API client */
        dmx_frontend_source_t source;
} dmx_frontend_t;

/*--------------------------------------------------------------------------
```

```
/* MPEG-2 TS Demux */
/*------------------------------------------------------------------------*/

/*
 * Flags OR'ed in the capabilites field of struct dmx_demux_s.
 */

#define DMX_TS_FILTERING                        1
#define DMX_PES_FILTERING                       2
#define DMX_SECTION_FILTERING                   4
#define DMX_MEMORY_BASED_FILTERING              8    /* write() available */
#define DMX_CRC_CHECKING                        16
#define DMX_TS_DESCRAMBLING                     32
#define DMX_SECTION_PAYLOAD_DESCRAMBLING        64
#define DMX_MAC_ADDRESS_DESCRAMBLING            128
```

8.1.4 demux_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_list)

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 client
        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_
                                           __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;
```

8.1.5 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);
```

8.2 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.

All demux drivers in the directory implement the abstract interface dmx_demux_t.

8.2.1 dmx_register_demux()

DESCRIPTION

This function makes a demux driver interface available to the Linux kernel. It is usually called by the init_module() function of the kernel module that contains the demux driver. The caller of this function is responsible for allocating dynamic or static memory for the demux structure and for initializing its fields before calling this function. The memory allocated for the demux structure must not be freed before calling dmx_unregister_demux(),

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.

8.2.2 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.

8.2.3 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.

8.3 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.

8.3.1 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.

8.3.2 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.

8.3.3 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.

8.3.4 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.

8.3.5 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.

8.3.6 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.

8.3.7 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.

8.3.8 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.

8.3.9 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.

8.3.10 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.

8.3.11 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().

8.3.12 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.

8.3.13 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.

8.3.14 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.

8.4 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.

8.4.1 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.

8.4.2 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.

8.5 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.

8.5.1 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.

8.5.2 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.

8.5.3 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.

8.6 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.

8.6.1 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.

8.6.2 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.

8.6.3 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.

8.6.4 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.

8.6.5 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 9

Examples

In this section we would like to present some examples for using the DVB API.

9.1 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 <ost/dmx.h>
#include <ost/frontend.h>
#include <ost/sec.h>
#include <sys/poll.h>

#define DMX "/dev/ost/demux"
#define QPSK "/dev/ost/qpskfe"
#define SEC "/dev/ost/sec"

/* routine for checking if we have a signal */
int has_signal(int front)
{
        feStatus stat;

        if ( front < 0 ){
                if((front = open(QPSK,O_RDWR)) < 0){
```

```
                        perror("FRONTEND DEVICE: ");
                        return -1;
                }
        }

        FEReadStatus(front, &stat);
        if (stat & FE_HAS_SIGNAL)
                return 0;
        else {
                printf("Tuning failed\n");
                return -1;
        }
}

/* tune qpsk */
/* 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:             Symbol Rate                                   */
/* 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)
{
        struct secCommand scmd;
        struct secCmdSequence scmds;
        struct dmxPesFilterParams pesFilterParams;
        struct qpskParameters qpsk;
        struct pollfd pfd[1];
        struct qpskEvent event;
        int front, sec, demux1, demux2, demux3;

        /* Open all the necessary the devices */

        if((front = open(QPSK,O_RDWR)) < 0){
                perror("FRONTEND DEVICE: ");
                return -1;
        }

        if((sec = open(SEC,O_RDWR)) < 0){
                perror("SEC DEVICE: ");
                return -1;
        }

        if ((demux1 = open(DMX, O_RDWR|O_NONBLOCK))  < 0){
```

```
                perror("DEMUX DEVICE: ");
                return -1;
        }

        if ((demux2 = open(DMX, O_RDWR|O_NONBLOCK))  < 0){
                perror("DEMUX DEVICE: ");
                return -1;
        }

        if ((demux3 = open(DMX, O_RDWR|O_NONBLOCK))  < 0){
                perror("DEMUX DEVICE: ");
                return -1;
        }

      /* Set the frequency of the transponder, taking into account the
         local frequencies of the LNB */

        if (freq < lnb_slof) {
                qpsk.iFrequency = (freq - lnb_lof1);
                scmds.continuousTone = SEC_TONE_OFF;
        } else {
                qpsk.iFrequency = (freq - lnb_lof2);
                scmds.continuousTone = SEC_TONE_ON;
        }

      /* Set the polarity of the transponder by setting the correct
         voltage on the universal LNB */

        if (pol) scmds.voltage = SEC_VOLTAGE_18;
        else scmds.voltage = SEC_VOLTAGE_13;

      /* In case we have a DiSEqC, set it to the correct address */

        scmd.type=0;
        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);

        scmds.miniCommand=SEC_MINI_NONE;
        scmds.numCommands=1;
        scmds.commands=&scmd;

      /* Send the data to the SEC device to prepare the LNB for tuning  */
        if (ioctl(sec, SEC_SEND_SEQUENCE, &scmds) < 0){
                perror("SEC SEND: ");
                return -1;
        }
```

```
/* Set symbol rate and FEC */

  qpsk.SymbolRate = srate;
  qpsk.FEC_inner = fec;

/* Now send it all to the frontend device */
  if (ioctl(front, QPSK_TUNE, &qpsk) < 0){
          perror("QPSK TUNE: ");
          return -1;
  }

/* poll for QPSK event to check if tuning worked */
  pfd[0].fd = front;
  pfd[0].events = POLLIN;

  if (poll(pfd,1,3000)){
          if (pfd[0].revents & POLLIN){
                  printf("Getting QPSK event\n");
                  if ( ioctl(front, QPSK_GET_EVENT, &event)

                       == -EBUFFEROVERFLOW){
                          perror("qpsk get event");
                          return -1;
                  }
                  printf("Received ");
                  switch(event.type){
                  case FE_UNEXPECTED_EV:
                          printf("unexpected event\n");
                          return -1;
                  case FE_FAILURE_EV:
                          printf("failure event\n");
                          return -1;
                 case FE_COMPLETION_EV:
                          printf("completion event\n");
                  }
          }
  }


/* Set the filters for video, audio and teletext demuxing */

  pesFilterParams.pid     = vpid;
  pesFilterParams.input   = DMX_IN_FRONTEND;
  pesFilterParams.output  = DMX_OUT_DECODER;
  pesFilterParams.pesType = DMX_PES_VIDEO;
  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.pesType = 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.pesType = DMX_PES_TELETEXT;
        pesFilterParams.flags   = DMX_IMMEDIATE_START;
        if (ioctl(demux3, DMX_SET_PES_FILTER, &pesFilterParams) < 0){
                perror("set_tpid");
                return -1;
        }

        /* check if we have a signal */
        return has_signal(front);
}
```

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.

9.2 The DVR device

The following program code shows how to use the DVR device for recording.

```
#include <sys/ioctl.h>
#include <stdio.h>
#include <stdint.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <time.h>
#include <unistd.h>

#include <ost/dmx.h>
#include <ost/video.h>
#include <sys/poll.h>
```

```
#define DVR "/dev/ost/dvr"
#define AUDIO "/dev/ost/audio"
#define VIDEO "/dev/ost/video"

#define BUFFY (188*20)
#define MAX_LENGTH (1024*1024*5) /* record 5MB */


/* switch the demuxes to recording, assuming the transponder is tuned */

/* demux1, demux2: file descriptor of video and audio filters */
/* vpid, apid:     PIDs of video and audio channels           */

int switch_to_record(int demux1, int demux2, uint16_t vpid, uint16_t apid)
{
        struct dmxPesFilterParams pesFilterParams;

        if (demux1 < 0){
                if ((demux1=open(DMX, O_RDWR|O_NONBLOCK))
                    < 0){
                        perror("DEMUX DEVICE: ");
                        return -1;
                }
        }

        if (demux2 < 0){
                if ((demux2=open(DMXdemuxs, O_RDWR|O_NONBLOCK))
                    < 0){
                        perror("DEMUX DEVICE: ");
                        return -1;
                }
        }

        pesFilterParams.pid = vpid;
        pesFilterParams.input = DMX_IN_FRONTEND;
        pesFilterParams.output = DMX_OUT_TS_TAP;
        pesFilterParams.pesType = DMX_PES_VIDEO;
        pesFilterParams.flags = DMX_IMMEDIATE_START;
        if (ioctl(demux1, DMX_SET_PES_FILTER, &pesFilterParams) < 0){
                perror("DEMUX DEVICE");
                return -1;
        }
        pesFilterParams.pid = apid;
        pesFilterParams.input = DMX_IN_FRONTEND;
        pesFilterParams.output = DMX_OUT_TS_TAP;
        pesFilterParams.pesType = DMX_PES_AUDIO;
        pesFilterParams.flags = DMX_IMMEDIATE_START;
        if (ioctl(demux2, DMX_SET_PES_FILTER, &pesFilterParams) < 0){
                perror("DEMUX DEVICE");
                return -1;
```

```
        }
        return 0;
}

/* start recording MAX_LENGTH , assuming the transponder is tuned */

/* demux1, demux2: file descriptor of video and audio filters */
/* vpid, apid:     PIDs of video and audio channels           */
int record_dvr(int demux1, int demux2, uint16_t vpid, uint16_t apid)
{
        int i;
        int len;
        int written;
        uint8_t buf[BUFFY];
        uint64_t length;
        struct pollfd pfd[1];
        int dvr, dvr_out;

        /* open dvr device */
        if ((dvr = open(DVR, O_RDONLY|O_NONBLOCK)) < 0){
                        perror("DVR DEVICE");
                        return -1;
        }

        /* switch video and audio demuxes to dvr */
        printf ("Switching dvr on\n");
        i = switch_to_record(demux1, demux2, vpid, apid);
        printf("finished: ");

        printf("Recording %2.0f MB of test file in TS format\n",
               MAX_LENGTH/(1024.0*1024.0));
        length = 0;

        /* open output file */
        if ((dvr_out = open(DVR_FILE,O_WRONLY|O_CREAT
                                |O_TRUNC, S_IRUSR|S_IWUSR
                                |S_IRGRP|S_IWGRP|S_IROTH|
                                S_IWOTH)) < 0){
                perror("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;
}
```