Video4Linux Programming

Alan Cox

alan@redhat.com

Video4Linux Programming

by Alan Cox

Table of Contents

List of Tables

Chapter 1. Introduction

Parts of this document first appeared in Linux Magazine under a ninety day exclusivity.

Video4Linux is intended to provide a common programming interface for the many TV and capture cards now on the market, as well as parallel port and USB video cameras. Radio, teletext decoders and vertical blanking data interfaces are also provided.

Chapter 2. Radio Devices

There are a wide variety of radio interfaces available for PC's, and these are generally very simple to program. The biggest problem with supporting such devices is normally extracting documentation from the vendor.

The radio interface supports a simple set of control ioctls standardised across all radio and tv interfaces. It does not support read or write, which are used for video streams. The reason radio cards do not allow you to read the audio stream into an application is that without exception they provide a connection on to a soundcard. Soundcards can be used to read the radio data just fine.

2.1. Registering Radio Devices

The Video4linux core provides an interface for registering devices. The first step in writing our radio card driver is to register it.

```
static struct video_device my_radio
{
        "My radio",
        VID_TYPE_TUNER,
        radio_open.
        radio_close,
        NULL,                   /* no read */
        NULL,                    /* no write */
        NULL,                   /* no poll */
        radio_ioctl,
        NULL,                   /* no special init function */
        NULL                   /* no private data */
};
```

This declares our video4linux device driver interface. The VID_TYPE_ value defines what kind of an interface we are, and defines basic capabilities.

The only defined value relevant for a radio card is VID_TYPE_TUNER which indicates that the device can be tuned. Clearly our radio is going to have some way to change channel so it is tuneable.

We declare an open and close routine, but we do not need read or write, which are used to read and write video data to or from the card itself. As we have no read or write there is no poll function.

The private initialise function is run when the device is registered. In this driver we've already done all the work needed. The final pointer is a private data pointer that can be used by the device driver to attach and retrieve private data structures. We set this field "priv" to NULL for the moment.

Having the structure defined is all very well but we now need to register it with the kernel.

```
static int io = 0x320;

int __init myradio_init(struct video_init *v)
{
        if(!request_region(io, MY_IO_SIZE, "myradio"))
        {
                printk(KERN_ERR
                    "myradio: port 0x%03X is in use.\n", io);
                return -EBUSY;
        }

        if(video_device_register(&my_radio, VFL_TYPE_RADIO)==-1) {
                release_region(io, MY_IO_SIZE);
                return -EINVAL;
        }
        return 0;
}
```

The first stage of the initialisation, as is normally the case, is to check that the I/O space we are about to fiddle with doesn't belong to some other driver. If it is we leave well alone. If the user gives the address of the wrong device then we will spot this. These policies will generally avoid crashing the machine.

Now we ask the Video4Linux layer to register the device for us. We hand it our carefully designed video_device structure and also tell it which group of devices we want it registered with. In this case VFL_TYPE_RADIO.

The types available are

Table 2-1. Device Types

VFL_TYPE_RADIO	/dev/radio{n}	Radio devices are assigned in this block. As with all of these selections the actual number assignment is done by the video layer accordijng to what is free.
VFL_TYPE_GRABBER	/dev/video{n}	Video capture devices and also -- counter-intuitively for the name -- hardware video playback devices such as MPEG2 cards.
VFL_TYPE_VBI	/dev/vbi{n}	The VBI devices capture the hidden lines on a television picture that carry further information like closed caption data, teletext (primarily in Europe) and now Intercast and the ATVEC internet television encodings.
VFL_TYPE_VTX	/dev/vtx[n}	VTX is 'Videotext' also known as 'Teletext'. This is a system for sending numbered, 40x25, mostly textual page images over the hidden lines. Unlike the /dev/vbi interfaces, this is for 'smart' decoder chips. (The use of the word smart here has to be taken in context, the smartest teletext chips are fairly dumb pieces of technology).

We are most definitely a radio.

Finally we allocate our I/O space so that nobody treads on us and return 0 to signify general happiness with the state of the universe.

2.2. Opening And Closing The Radio

The functions we declared in our video_device are mostly very simple. Firstly we can drop in what is basically standard code for open and close.

```
static int users = 0;

static int radio_open(struct video_device *dev, int flags)
{
        if(users)
                return -EBUSY;
        users++;
        return 0;
}
```

At open time we need to do nothing but check if someone else is also using the radio card. If nobody is using it we make a note that we are using it, then we ensure that nobody unloads our driver on us.

```
static int radio_close(struct video_device *dev)
{
        users--;
}
```

At close time we simply need to reduce the user count and allow the module to become unloadable.

If you are sharp you will have noticed neither the open nor the close routines attempt to reset or change the radio settings. This is intentional. It allows an application to set up the radio and exit. It avoids a user having to leave an application running all the time just to listen to the radio.

2.3. The Ioctl Interface

This leaves the ioctl routine, without which the driver will not be terribly useful to anyone.

```
static int radio_ioctl(struct video_device *dev, unsigned int cmd, void *a
{
        switch(cmd)
        {
                case VIDIOCGCAP:
                {
                        struct video_capability v;
                        v.type = VID_TYPE_TUNER;
                        v.channels = 1;
                        v.audios = 1;
                        v.maxwidth = 0;
                        v.minwidth = 0;
                        v.maxheight = 0;
                        v.minheight = 0;
                        strcpy(v.name, "My Radio");
                        if(copy_to_user(arg, &v, sizeof(v)))
                                return -EFAULT;
                        return 0;
                }
```

VIDIOCGCAP is the first ioctl all video4linux devices must support. It allows the applications to find out what sort of a card they have found and to figure out what they want to do about it. The fields in the structure are

Table 2-2. struct video_capability fields

name	The device text name. This is intended for the user.
channels	The number of different channels you can tune on this card. It could even by zero for a card that has no tuning capability. For our simple FM radio it is 1. An AM/FM radio would report 2.
audios	The number of audio inputs on this device. For our radio there is only one audio input.
minwidth,minheight	The smallest size the card is capable of capturing images in. We set these to zero. Radios do not capture pictures

maxwidth,maxheight	The largest image size the card is capable of capturing. For our radio we report 0.
type	This reports the capabilities of the device, and matches the field we filled in in the struct video_device when registering.

Having filled in the fields, we use copy_to_user to copy the structure into the users buffer. If the copy fails we return an EFAULT to the application so that it knows it tried to feed us garbage.

The next pair of ioctl operations select which tuner is to be used and let the application find the tuner properties. We have only a single FM band tuner in our example device.

```
case VIDIOCGTUNER:
{
        struct video_tuner v;
        if(copy_from_user(&v, arg, sizeof(v))!=0)
                return -EFAULT;
        if(v.tuner)
                return -EINVAL;
        v.rangelow=(87*16000);
        v.rangehigh=(108*16000);
        v.flags = VIDEO_TUNER_LOW;
        v.mode = VIDEO_MODE_AUTO;
        v.signal = 0xFFFF;
        strcpy(v.name, "FM");
        if(copy_to_user(&v, arg, sizeof(v))!=0)
                return -EFAULT;
        return 0;
}
```

The VIDIOCGTUNER ioctl allows applications to query a tuner. The application sets the tuner field to the tuner number it wishes to query. The query does not change the tuner that is being used, it merely enquires about the tuner in question.

We have exactly one tuner so after copying the user buffer to our temporary structure we complain if they asked for a tuner other than tuner 0.

The video_tuner structure has the following fields

Table 2-3. struct video_tuner fields

int tuner	The number of the tuner in question
char name[32]	A text description of this tuner. "FM" will do fine. This is intended for the application.
u32 flags	Tuner capability flags
u16 mode	The current reception mode
u16 signal	The signal strength scaled between 0 and 65535. If a device cannot tell the signal strength it should report 65535. Many simple cards contain only a signal/no signal bit. Such cards will report either 0 or 65535.
u32 rangelow, rangehigh	The range of frequencies supported by the radio or TV. It is scaled according to the VIDEO_TUNER_LOW flag.

Table 2-4. struct video_tuner flags

VIDEO_TUNER_PAL	A PAL TV tuner
VIDEO_TUNER_NTSC	An NTSC (US) TV tuner
VIDEO_TUNER_SECAM	A SECAM (French) TV tuner
VIDEO_TUNER_LOW	The tuner frequency is scaled in 1/16th of a KHz steps. If not it is in 1/16th of a MHz steps
VIDEO_TUNER_NORM	The tuner can set its format
VIDEO_TUNER_STEREO_ON	The tuner is currently receiving a stereo signal

Table 2-5. struct video_tuner modes

VIDEO_MODE_PAL	PAL Format
VIDEO_MODE_NTSC	NTSC Format (USA)
VIDEO_MODE_SECAM	French Format
VIDEO_MODE_AUTO	A device that does not need to do TV format switching

The settings for the radio card are thus fairly simple. We report that we are a tuner called "FM" for FM radio. In order to get the best tuning resolution we report VIDEO_TUNER_LOW and select tuning to 1/16th of KHz. Its unlikely our card

can do that resolution but it is a fair bet the card can do better than 1/16th of a MHz. VIDEO_TUNER_LOW is appropriate to almost all radio usage.

We report that the tuner automatically handles deciding what format it is receiving - true enough as it only handles FM radio. Our example card is also incapable of detecting stereo or signal strengths so it reports a strength of 0xFFFF (maximum) and no stereo detected.

To finish off we set the range that can be tuned to be 87-108Mhz, the normal FM broadcast radio range. It is important to find out what the card is actually capable of tuning. It is easy enough to simply use the FM broadcast range. Unfortunately if you do this you will discover the FM broadcast ranges in the USA, Europe and Japan are all subtly different and some users cannot receive all the stations they wish.

The application also needs to be able to set the tuner it wishes to use. In our case, with a single tuner this is rather simple to arrange.

```
case VIDIOCSTUNER:
{
        struct video_tuner v;
        if(copy_from_user(&v, arg, sizeof(v)))
                return -EFAULT;
        if(v.tuner != 0)
                return -EINVAL;
        return 0;
}
```

We copy the user supplied structure into kernel memory so we can examine it. If the user has selected a tuner other than zero we reject the request. If they wanted tuner 0 then, surprisingly enough, that is the current tuner already.

The next two ioctls we need to provide are to get and set the frequency of the radio. These both use an unsigned long argument which is the frequency. The scale of the frequency depends on the VIDEO_TUNER_LOW flag as I mentioned earlier on. Since we have VIDEO_TUNER_LOW set this will be in 1/16ths of a KHz.

```
static unsigned long current_freq;
```

```
case VIDIOCGFREQ:
        if(copy_to_user(arg, &current_freq,
                sizeof(unsigned long))
                return -EFAULT;
```

```
                return 0;
```

Querying the frequency in our case is relatively simple. Our radio card is too dumb to let us query the signal strength so we remember our setting if we know it. All we have to do is copy it to the user.

```
        case VIDIOCSFREQ:
        {
                u32 freq;
                if(copy_from_user(arg, &freq,
                        sizeof(unsigned long))!=0)
                        return -EFAULT;
                if(hardware_set_freq(freq)<0)
                        return -EINVAL;
                current_freq = freq;
                return 0;
        }
```

Setting the frequency is a little more complex. We begin by copying the desired frequency into kernel space. Next we call a hardware specific routine to set the radio up. This might be as simple as some scaling and a few writes to an I/O port. For most radio cards it turns out a good deal more complicated and may involve programming things like a phase locked loop on the card. This is what documentation is for.

The final set of operations we need to provide for our radio are the volume controls. Not all radio cards can even do volume control. After all there is a perfectly good volume control on the sound card. We will assume our radio card has a simple 4 step volume control.

There are two ioctls with audio we need to support

```
static int current_volume=0;

        case VIDIOCGAUDIO:
        {
                struct video_audio v;
                if(copy_from_user(&v, arg, sizeof(v)))
                        return -EFAULT;
                if(v.audio != 0)
                        return -EINVAL;
```

```
                v.volume = 16384*current_volume;
                v.step = 16384;
                strcpy(v.name, "Radio");
                v.mode = VIDEO_SOUND_MONO;
                v.balance = 0;
                v.base = 0;
                v.treble = 0;

                if(copy_to_user(arg. &v, sizeof(v)))
                        return -EFAULT;
                return 0;
        }
```

Much like the tuner we start by copying the user structure into kernel space. Again we check if the user has asked for a valid audio input. We have only input 0 and we punt if they ask for another input.

Then we fill in the video_audio structure. This has the following format

Table 2-6. struct video_audio fields

audio	The input the user wishes to query
volume	The volume setting on a scale of 0-65535
base	The base level on a scale of 0-65535
treble	The treble level on a scale of 0-65535
flags	The features this audio device supports
name	A text name to display to the user. We picked "Radio" as it explains things quite nicely.
mode	The current reception mode for the audio We report MONO because our card is too stupid to know if it is in mono or stereo.
balance	The stereo balance on a scale of 0-65535, 32768 is middle.
step	The step by which the volume control jumps. This is used to help make it easy for applications to set slider behaviour.

Table 2-7. struct video_audio flags

VIDEO_AUDIO_MUTE	The audio is currently muted. We could fake this in our driver but we choose not to bother.
VIDEO_AUDIO_MUTABLE	The input has a mute option
VIDEO_AUDIO_TREBLE	The input has a treble control
VIDEO_AUDIO_BASS	The input has a base control

Table 2-8. struct video_audio modes

VIDEO_SOUND_MONO	Mono sound
VIDEO_SOUND_STEREO	Stereo sound
VIDEO_SOUND_LANG1	Alternative language 1 (TV specific)
VIDEO_SOUND_LANG2	Alternative language 2 (TV specific)

Having filled in the structure we copy it back to user space.

The VIDIOCSAUDIO ioctl allows the user to set the audio parameters in the video_audio structure. The driver does its best to honour the request.

```
case VIDIOCSAUDIO:
{
        struct video_audio v;
        if(copy_from_user(&v, arg, sizeof(v)))
                return -EFAULT;
        if(v.audio)
                return -EINVAL;
        current_volume = v/16384;
        hardware_set_volume(current_volume);
        return 0;
}
```

In our case there is very little that the user can set. The volume is basically the limit. Note that we could pretend to have a mute feature by rewriting this to

```
case VIDIOCSAUDIO:
{
        struct video_audio v;
        if(copy_from_user(&v, arg, sizeof(v)))
                return -EFAULT;
        if(v.audio)
```

```
                        return -EINVAL;
                current_volume = v/16384;
                if(v.flags&VIDEO_AUDIO_MUTE)
                        hardware_set_volume(0);
                else
                        hardware_set_volume(current_volume);
                current_muted = v.flags &
                                        VIDEO_AUDIO_MUTE;
                return 0;
        }
```

This with the corresponding changes to the VIDIOCGAUDIO code to report the state of the mute flag we save and to report the card has a mute function, will allow applications to use a mute facility with this card. It is questionable whether this is a good idea however. User applications can already fake this themselves and kernel space is precious.

We now have a working radio ioctl handler. So we just wrap up the function

```
        }
        return -ENOIOCTLCMD;
}
```

and pass the Video4Linux layer back an error so that it knows we did not understand the request we got passed.

2.4. Module Wrapper

Finally we add in the usual module wrapping and the driver is done.

```
#ifndef MODULE

static int io = 0x300;

#else

static int io = -1;

#endif
```

```
MODULE_AUTHOR("Alan Cox");
MODULE_DESCRIPTION("A driver for an imaginary radio card.");
module_param(io, int, 0444);
MODULE_PARM_DESC(io, "I/O address of the card.");

static int __init init(void)
{
        if(io==-1)
        {
                printk(KERN_ERR
         "You must set an I/O address with io=0x???\n");
                return -EINVAL;
        }
        return myradio_init(NULL);
}

static void __exit cleanup(void)
{
        video_unregister_device(&my_radio);
        release_region(io, MY_IO_SIZE);
}

module_init(init);
module_exit(cleanup);
```

In this example we set the IO base by default if the driver is compiled into the kernel: you can still set it using "my_radio.irq" if this file is called `my_radio.c`. For the module we require the user sets the parameter. We set io to a nonsense port (-1) so that we can tell if the user supplied an io parameter or not.

We use MODULE_ defines to give an author for the card driver and a description. We also use them to declare that io is an integer and it is the address of the card, and can be read by anyone from sysfs.

The clean-up routine unregisters the video_device we registered, and frees up the I/O space. Note that the unregister takes the actual video_device structure as its argument. Unlike the file operations structure which can be shared by all instances of a device a video_device structure as an actual instance of the device. If you are registering multiple radio devices you need to fill in one structure per device (most likely by setting up a template and copying it to each of the actual device structures).

Chapter 3. Video Capture Devices

3.1. Video Capture Device Types

The video capture devices share the same interfaces as radio devices. In order to explain the video capture interface I will use the example of a camera that has no tuners or audio input. This keeps the example relatively clean. To get both combine the two driver examples.

Video capture devices divide into four categories. A little technology backgrounder. Full motion video even at television resolution (which is actually fairly low) is pretty resource-intensive. You are continually passing megabytes of data every second from the capture card to the display. several alternative approaches have emerged because copying this through the processor and the user program is a particularly bad idea .

The first is to add the television image onto the video output directly. This is also how some 3D cards work. These basic cards can generally drop the video into any chosen rectangle of the display. Cards like this, which include most mpeg1 cards that used the feature connector, aren't very friendly in a windowing environment. They don't understand windows or clipping. The video window is always on the top of the display.

Chroma keying is a technique used by cards to get around this. It is an old television mixing trick where you mark all the areas you wish to replace with a single clear colour that isn't used in the image - TV people use an incredibly bright blue while computing people often use a particularly virulent purple. Bright blue occurs on the desktop. Anyone with virulent purple windows has another problem besides their TV overlay.

The third approach is to copy the data from the capture card to the video card, but to do it directly across the PCI bus. This relieves the processor from doing the work but does require some smartness on the part of the video capture chip, as well as a suitable video card. Programming this kind of card and more so debugging it can be extremely tricky. There are some quite complicated interactions with the display and you may also have to cope with various chipset bugs that show up when PCI cards start talking to each other.

To keep our example fairly simple we will assume a card that supports overlaying a flat rectangular image onto the frame buffer output, and which can also capture stuff into processor memory.

3.2. Registering Video Capture Devices

This time we need to add more functions for our camera device.

```
static struct video_device my_camera
{
        "My Camera",
        VID_TYPE_OVERLAY|VID_TYPE_SCALES|\
        VID_TYPE_CAPTURE|VID_TYPE_CHROMAKEY,
        camera_open.
        camera_close,
        camera_read,      /* no read */
        NULL,             /* no write */
        camera_poll,      /* no poll */
        camera_ioctl,
        NULL,             /* no special init function */
        NULL              /* no private data */
};
```

We need a read() function which is used for capturing data from the card, and we need a poll function so that a driver can wait for the next frame to be captured.

We use the extra video capability flags that did not apply to the radio interface. The video related flags are

Table 3-1. Capture Capabilities

VID_TYPE_CAPTURE	We support image capture
VID_TYPE_TELETEXT	A teletext capture device (vbi{n])
VID_TYPE_OVERLAY	The image can be directly overlaid onto the frame buffer
VID_TYPE_CHROMAKEY	Chromakey can be used to select which parts of the image to display
VID_TYPE_CLIPPING	It is possible to give the board a list of rectangles to draw around.
VID_TYPE_FRAMERAM	The video capture goes into the video memory and actually changes it. Applications need to know this so they can clean up after the card
VID_TYPE_SCALES	The image can be scaled to various sizes, rather than being a single fixed size.

VID_TYPE_MONOCHROME	The capture will be monochrome. This isn't a complete answer to the question since a mono camera on a colour capture card will still produce mono output.
VID_TYPE_SUBCAPTURE	The card allows only part of its field of view to be captured. This enables applications to avoid copying all of a large image into memory when only some section is relevant.

We set VID_TYPE_CAPTURE so that we are seen as a capture card, VID_TYPE_CHROMAKEY so the application knows it is time to draw in virulent purple, and VID_TYPE_SCALES because we can be resized.

Our setup is fairly similar. This time we also want an interrupt line for the 'frame captured' signal. Not all cards have this so some of them cannot handle poll().

```
static int io = 0x320;
static int irq = 11;

int __init mycamera_init(struct video_init *v)
{
        if(!request_region(io, MY_IO_SIZE, "mycamera"))
        {
                printk(KERN_ERR
                       "mycamera: port 0x%03X is in use.\n", io);
                return -EBUSY;
        }

        if(video_device_register(&my_camera,
            VFL_TYPE_GRABBER)==-1) {
                release_region(io, MY_IO_SIZE);
                return -EINVAL;
        }
        return 0;
}
```

This is little changed from the needs of the radio card. We specify VFL_TYPE_GRABBER this time as we want to be allocated a /dev/video name.

3.3. Opening And Closing The Capture Device

```
static int users = 0;

static int camera_open(struct video_device *dev, int flags)
{
        if(users)
                return -EBUSY;
        if(request_irq(irq, camera_irq, 0, "camera", dev)<0)
                return -EBUSY;
        users++;
        return 0;
}


static int camera_close(struct video_device *dev)
{
        users--;
        free_irq(irq, dev);
}
```

The open and close routines are also quite similar. The only real change is that we now request an interrupt for the camera device interrupt line. If we cannot get the interrupt we report EBUSY to the application and give up.

3.4. Interrupt Handling

Our example handler is for an ISA bus device. If it was PCI you would be able to share the interrupt and would have set IRQF_SHARED to indicate a shared IRQ. We pass the device pointer as the interrupt routine argument. We don't need to since we only support one card but doing this will make it easier to upgrade the driver for multiple devices in the future.

Our interrupt routine needs to do little if we assume the card can simply queue one frame to be read after it captures it.

```
static struct wait_queue *capture_wait;
static int capture_ready = 0;

static void camera_irq(int irq, void *dev_id,
```

```
                                struct pt_regs *regs)
{
        capture_ready=1;
        wake_up_interruptible(&capture_wait);
}
```

The interrupt handler is nice and simple for this card as we are assuming the card is buffering the frame for us. This means we have little to do but wake up anybody interested. We also set a capture_ready flag, as we may capture a frame before an application needs it. In this case we need to know that a frame is ready. If we had to collect the frame on the interrupt life would be more complex.

The two new routines we need to supply are camera_read which returns a frame, and camera_poll which waits for a frame to become ready.

```
static int camera_poll(struct video_device *dev,
 struct file *file, struct poll_table *wait)
{
        poll_wait(file, &capture_wait, wait);
        if(capture_read)
                return POLLIN|POLLRDNORM;
        return 0;
}
```

Our wait queue for polling is the capture_wait queue. This will cause the task to be woken up by our camera_irq routine. We check capture_read to see if there is an image present and if so report that it is readable.

3.5. Reading The Video Image

```
static long camera_read(struct video_device *dev, char *buf,
                                  unsigned long count)
{
        struct wait_queue wait = { current, NULL };
        u8 *ptr;
        int len;
        int i;

        add_wait_queue(&capture_wait, &wait);
```

```
while(!capture_ready)
{
        if(file->flags&O_NDELAY)
        {
                remove_wait_queue(&capture_wait, &wait);
                current->state = TASK_RUNNING;
                return -EWOULDBLOCK;
        }
        if(signal_pending(current))
        {
                remove_wait_queue(&capture_wait, &wait);
                current->state = TASK_RUNNING;
                return -ERESTARTSYS;
        }
        schedule();
        current->state = TASK_INTERRUPTIBLE;
}
remove_wait_queue(&capture_wait, &wait);
current->state = TASK_RUNNING;
```

The first thing we have to do is to ensure that the application waits until the next frame is ready. The code here is almost identical to the mouse code we used earlier in this chapter. It is one of the common building blocks of Linux device driver code and probably one which you will find occurs in any drivers you write.

We wait for a frame to be ready, or for a signal to interrupt our waiting. If a signal occurs we need to return from the system call so that the signal can be sent to the application itself. We also check to see if the user actually wanted to avoid waiting - ie if they are using non-blocking I/O and have other things to get on with.

Next we copy the data from the card to the user application. This is rarely as easy as our example makes out. We will add capture_w, and capture_h here to hold the width and height of the captured image. We assume the card only supports 24bit RGB for now.

```
capture_ready = 0;

ptr=(u8 *)buf;
len = capture_w * 3 * capture_h; /* 24bit RGB */

if(len>count)
        len=count;  /* Doesn't all fit */
```

```
        for(i=0; i<len; i++)
        {
                put_user(inb(io+IMAGE_DATA), ptr);
                ptr++;
        }

        hardware_restart_capture();

        return i;
}
```

For a real hardware device you would try to avoid the loop with put_user(). Each call to put_user() has a time overhead checking whether the accesses to user space are allowed. It would be better to read a line into a temporary buffer then copy this to user space in one go.

Having captured the image and put it into user space we can kick the card to get the next frame acquired.

3.6. Video Ioctl Handling

As with the radio driver the major control interface is via the ioctl() function. Video capture devices support the same tuner calls as a radio device and also support additional calls to control how the video functions are handled. In this simple example the card has no tuners to avoid making the code complex.

```
static int camera_ioctl(struct video_device *dev, unsigned int cmd, void *
{
        switch(cmd)
        {
                case VIDIOCGCAP:
                {
                        struct video_capability v;
                        v.type = VID_TYPE_CAPTURE|\
                                 VID_TYPE_CHROMAKEY|\
                                 VID_TYPE_SCALES|\
                                 VID_TYPE_OVERLAY;
                        v.channels = 1;
                        v.audios = 0;
```

```
                v.maxwidth = 640;
                v.minwidth = 16;
                v.maxheight = 480;
                v.minheight = 16;
                strcpy(v.name, "My Camera");
                if(copy_to_user(arg, &v, sizeof(v)))
                        return -EFAULT;
                return 0;
        }
```

The first ioctl we must support and which all video capture and radio devices are required to support is VIDIOCGCAP. This behaves exactly the same as with a radio device. This time, however, we report the extra capabilities we outlined earlier on when defining our video_dev structure.

We now set the video flags saying that we support overlay, capture, scaling and chromakey. We also report size limits - our smallest image is 16x16 pixels, our largest is 640x480.

To keep things simple we report no audio and no tuning capabilities at all.

```
        case VIDIOCGCHAN:
        {
                struct video_channel v;
                if(copy_from_user(&v, arg, sizeof(v)))
                        return -EFAULT;
                if(v.channel != 0)
                        return -EINVAL;
                v.flags = 0;
                v.tuners = 0;
                v.type = VIDEO_TYPE_CAMERA;
                v.norm = VIDEO_MODE_AUTO;
                strcpy(v.name, "Camera Input");break;
                if(copy_to_user(&v, arg, sizeof(v)))
                        return -EFAULT;
                return 0;
        }
```

This follows what is very much the standard way an ioctl handler looks in Linux. We copy the data into a kernel space variable and we check that the request is valid (in this case that the input is 0). Finally we copy the camera info back to the user.

The VIDIOCGCHAN ioctl allows a user to ask about video channels (that is inputs to the video card). Our example card has a single camera input. The fields in the structure are

Table 3-2. struct video_channel fields

channel	The channel number we are selecting
name	The name for this channel. This is intended to describe the port to the user. Appropriate names are therefore things like "Camera" "SCART input"
flags	Channel properties
type	Input type
norm	The current television encoding being used if relevant for this channel.

Table 3-3. struct video_channel flags

VIDEO_VC_TUNER	Channel has a tuner.
VIDEO_VC_AUDIO	Channel has audio.

Table 3-4. struct video_channel types

VIDEO_TYPE_TV	Television input.
VIDEO_TYPE_CAMERA	Fixed camera input.
0	Type is unknown.

Table 3-5. struct video_channel norms

VIDEO_MODE_PAL	PAL encoded Television
VIDEO_MODE_NTSC	NTSC (US) encoded Television
VIDEO_MODE_SECAM	SECAM (French) Television
VIDEO_MODE_AUTO	Automatic switching, or format does not matter

The corresponding VIDIOCSCHAN ioctl allows a user to change channel and to request the norm is changed - for example to switch between a PAL or an NTSC

format camera.

```
case VIDIOCSCHAN:
{
        struct video_channel v;
        if(copy_from_user(&v, arg, sizeof(v)))
                return -EFAULT;
        if(v.channel != 0)
                return -EINVAL;
        if(v.norm != VIDEO_MODE_AUTO)
                return -EINVAL;
        return 0;
}
```

The implementation of this call in our driver is remarkably easy. Because we are assuming fixed format hardware we need only check that the user has not tried to change anything.

The user also needs to be able to configure and adjust the picture they are seeing. This is much like adjusting a television set. A user application also needs to know the palette being used so that it knows how to display the image that has been captured. The VIDIOCGPICT and VIDIOCSPICT ioctl calls provide this information.

```
case VIDIOCGPICT
{
        struct video_picture v;
        v.brightness = hardware_brightness();
        v.hue = hardware_hue();
        v.colour = hardware_saturation();
        v.contrast = hardware_brightness();
        /* Not settable */
        v.whiteness = 32768;
        v.depth = 24;           /* 24bit */
        v.palette = VIDEO_PALETTE_RGB24;
        if(copy_to_user(&v, arg,
             sizeof(v)))
                return -EFAULT;
        return 0;
}
```

The brightness, hue, color, and contrast provide the picture controls that are akin to a conventional television. Whiteness provides additional control for greyscale images. All of these values are scaled between 0-65535 and have 32768 as the mid point setting. The scaling means that applications do not have to worry about the capability range of the hardware but can let it make a best effort attempt.

Our depth is 24, as this is in bits. We will be returning RGB24 format. This has one byte of red, then one of green, then one of blue. This then repeats for every other pixel in the image. The other common formats the interface defines are

Table 3-6. Framebuffer Encodings

GREY	Linear greyscale. This is for simple cameras and the like
RGB565	The top 5 bits hold 32 red levels, the next six bits hold green and the low 5 bits hold blue.
RGB555	The top bit is clear. The red green and blue levels each occupy five bits.

Additional modes are support for YUV capture formats. These are common for TV and video conferencing applications.

The VIDIOCSPICT ioctl allows a user to set some of the picture parameters. Exactly which ones are supported depends heavily on the card itself. It is possible to support many modes and effects in software. In general doing this in the kernel is a bad idea. Video capture is a performance-sensitive application and the programs can often do better if they aren't being 'helped' by an overkeen driver writer. Thus for our device we will report RGB24 only and refuse to allow a change.

```
case VIDIOCSPICT:
{
        struct video_picture v;
        if(copy_from_user(&v, arg, sizeof(v)))
                return -EFAULT;
        if(v.depth!=24 ||
           v.palette != VIDEO_PALETTE_RGB24)
                return -EINVAL;
        set_hardware_brightness(v.brightness);
        set_hardware_hue(v.hue);
        set_hardware_saturation(v.colour);
```

```
                        set_hardware_brightness(v.contrast);
                        return 0;
                }
```

We check the user has not tried to change the palette or the depth. We do not want to carry out some of the changes and then return an error. This may confuse the application which will be assuming no change occurred.

In much the same way as you need to be able to set the picture controls to get the right capture images, many cards need to know what they are displaying onto when generating overlay output. In some cases getting this wrong even makes a nasty mess or may crash the computer. For that reason the VIDIOCSBUF ioctl used to set up the frame buffer information may well only be usable by root.

We will assume our card is one of the old ISA devices with feature connector and only supports a couple of standard video modes. Very common for older cards although the PCI devices are way smarter than this.

```
static struct video_buffer capture_fb;

                case VIDIOCGFBUF:
                {
                        if(copy_to_user(arg, &capture_fb,
                             sizeof(capture_fb)))
                                return -EFAULT;
                        return 0;

                }
```

We keep the frame buffer information in the format the ioctl uses. This makes it nice and easy to work with in the ioctl calls.

```
                case VIDIOCSFBUF:
                {
                        struct video_buffer v;

                        if(!capable(CAP_SYS_ADMIN))
                                return -EPERM;
```

```
                if(copy_from_user(&v, arg, sizeof(v)))
                        return -EFAULT;
                if(v.width!=320 && v.width!=640)
                        return -EINVAL;
                if(v.height!=200 && v.height!=240
                        && v.height!=400
                        && v.height !=480)
                        return -EINVAL;
                memcpy(&capture_fb, &v, sizeof(v));
                hardware_set_fb(&v);
                return 0;
        }
```

The capable() function checks a user has the required capability. The Linux operating system has a set of about 30 capabilities indicating privileged access to services. The default set up gives the superuser (uid 0) all of them and nobody else has any.

We check that the user has the SYS_ADMIN capability, that is they are allowed to operate as the machine administrator. We don't want anyone but the administrator making a mess of the display.

Next we check for standard PC video modes (320 or 640 wide with either EGA or VGA depths). If the mode is not a standard video mode we reject it as not supported by our card. If the mode is acceptable we save it so that VIDIOCFBUF will give the right answer next time it is called. The hardware_set_fb() function is some undescribed card specific function to program the card for the desired mode.

Before the driver can display an overlay window it needs to know where the window should be placed, and also how large it should be. If the card supports clipping it needs to know which rectangles to omit from the display. The video_window structure is used to describe the way the image should be displayed.

Table 3-7. struct video_window fields

width	The width in pixels of the desired image. The card may use a smaller size if this size is not available
height	The height of the image. The card may use a smaller size if this size is not available.

x	The X position of the top left of the window. This is in pixels relative to the left hand edge of the picture. Not all cards can display images aligned on any pixel boundary. If the position is unsuitable the card adjusts the image right and reduces the width.
y	The Y position of the top left of the window. This is counted in pixels relative to the top edge of the picture. As with the width if the card cannot display starting on this line it will adjust the values.
chromakey	The colour (expressed in RGB32 format) for the chromakey colour if chroma keying is being used.
clips	An array of rectangles that must not be drawn over.
clipcount	The number of clips in this array.

Each clip is a struct video_clip which has the following fields

Table 3-8. video_clip fields

x, y	Co-ordinates relative to the display
width, height	Width and height in pixels
next	A spare field for the application to use

The driver is required to ensure it always draws in the area requested or a smaller area, and that it never draws in any of the areas that are clipped. This may well mean it has to leave alone. small areas the application wished to be drawn.

Our example card uses chromakey so does not have to address most of the clipping. We will add a video_window structure to our global variables to remember our parameters, as we did with the frame buffer.

```
case VIDIOCGWIN:
{
        if(copy_to_user(arg, &capture_win,
            sizeof(capture_win)))
                return -EFAULT;
```

```
        return 0;
}


case VIDIOCSWIN:
{
        struct video_window v;
        if(copy_from_user(&v, arg, sizeof(v)))
                return -EFAULT;
        if(v.width > 640 || v.height > 480)
                return -EINVAL;
        if(v.width < 16 || v.height < 16)
                return -EINVAL;
        hardware_set_key(v.chromakey);
        hardware_set_window(v);
        memcpy(&capture_win, &v, sizeof(v));
        capture_w = v.width;
        capture_h = v.height;
        return 0;
}
```

Because we are using Chromakey our setup is fairly simple. Mostly we have to check the values are sane and load them into the capture card.

With all the setup done we can now turn on the actual capture/overlay. This is done with the VIDIOCCAPTURE ioctl. This takes a single integer argument where 0 is on and 1 is off.

```
case VIDIOCCAPTURE:
{
        int v;
        if(get_user(v, (int *)arg))
                return -EFAULT;
        if(v==0)
                hardware_capture_off();
        else
        {
                if(capture_fb.width == 0
                    || capture_w == 0)
                        return -EINVAL;
                hardware_capture_on();
        }
        return 0;
```

```
                        }
```

We grab the flag from user space and either enable or disable according to its value. There is one small corner case we have to consider here. Suppose that the capture was requested before the video window or the frame buffer had been set up. In those cases there will be unconfigured fields in our card data, as well as unconfigured hardware settings. We check for this case and return an error if the frame buffer or the capture window width is zero.

```
                default:
                        return -ENOIOCTLCMD;
        }
}
```

We don't need to support any other ioctls, so if we get this far, it is time to tell the video layer that we don't now what the user is talking about.

3.7. Other Functionality

The Video4Linux layer supports additional features, including a high performance mmap() based capture mode and capturing part of the image. These features are out of the scope of the book. You should however have enough example code to implement most simple video4linux devices for radio and TV cards.

Chapter 4. Known Bugs And Assumptions

Multiple Opens

The driver assumes multiple opens should not be allowed. A driver can work around this but not cleanly.

API Deficiencies

The existing API poorly reflects compression capable devices. There are plans afoot to merge V4L, V4L2 and some other ideas into a better interface.

Chapter 5. Public Functions Provided

video_register_device_index

LINUX

Kernel Hackers ManualNovember 2009

Name

`video_register_device_index` — register video4linux devices

Synopsis

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

Arguments

vfd

video device structure we want to register

type

type of device to register

nr

which device number (0 == /dev/video0, 1 == /dev/video1, ... -1 == first free)

index

stream number based on parent device; -1 if auto assign, requested number otherwise

Description

The registration code assigns minor numbers based on the type requested. -ENFILE is returned in all the device slots for this category are full. If not then the minor field is set and the driver initialize function is called (if non `NULL`).

Zero is returned on success.

Valid types are

`VFL_TYPE_GRABBER` - A frame grabber

`VFL_TYPE_VTX` - A teletext device

`VFL_TYPE_VBI` - Vertical blank data (undecoded)

`VFL_TYPE_RADIO` - A radio card

video_unregister_device

LINUX

Kernel Hackers ManualNovember 2009

Name

`video_unregister_device` — unregister a video4linux device

Synopsis

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

Arguments

`vfd`

the device to unregister

Description

This unregisters the passed device and deassigns the minor number. Future open calls will be met with errors.