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,
        VID_HARDWARE_MYRADIO,
        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.

The VID_HARDWARE_ types are unique to each device. Numbers are assigned by <alan@redhat.com> when device drivers are going to be released. Until then you can pull a suitably large number out of your hat and use it. 10000 should be safe for a very long time even allowing for the huge number of vendors making new and different radio cards at the moment.

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(check_region(io, MY_IO_SIZE))
        {
                printk(KERN_ERR
                    "myradio: port 0x%03X is in use.\n", io);
                return -EBUSY;
        }

        if(video_device_register(my_radio, VFL_TYPE_RADIO)==-1)
                return -EINVAL;
        request_region(io, MY_IO_SIZE, "myradio");
        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

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(stuct video_device *dev, int flags)
{
        if(users)
                return -EBUSY;
        users++;
        MOD_INC_USE_COUNT;
        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--;
        MOD_DEC_USE_COUNT;
}
```

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 *arg
{
        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

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

Table 2-4. struct video_tuner flags

Table 2-5. struct video_tuner modes

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