

Compiling a V4L USB DVB driver for FreeBSD

Tim Borgeaud

This paper describes the creation of a USB DVB-T driver for FreeBSD using Linux (V4L) DVB driver code and some Linux compatibility magic.

A possible approach to porting a driver from Linux to FreeBSD is to provide emulated Linux functionality in order to allow Linux driver source code to be compiled into a working FreeBSD driver (module). There exists some code, Luigi Rizzo's linux-kmod-compat port, which may be used to create USB device drivers in this manner. This paper describes the adoption of this approach and the extension of the compatibility code to enable the porting of a Linux DVB driver to FreeBSD. It includes some discussion of the basics of FreeBSD, and Linux, device drivers, the differences between, and similarities of, Linux and FreeBSD (kernel) functionality, and the resulting emulation code and DVB driver.

1. Introduction

1.1. Recording digital television broadcasting

The digital video recorder (DVR) has been around for a little while. However, it is perhaps only more recently that it has become attractive to use a general purpose computer to record and replay television. Large file storage capacity is cheaply available, typical desktop computers can be easily connected to large screens (including televisions) and, with the advent of digital video broadcasting (DVB), digital TV receiver devices are readily available.

However, as for other, inexpensive, hardware devices, drivers for DVB receivers for operating systems other than MS Windows are not so readily available. This presents some compatibility problems for anyone who may wish to enable TV recording on a Unix-like system. For a, preferred, BSD based system, there are particularly few available drivers.

For recording and playback of digital video broadcasts on a general purpose computer, the (obvious) operating system options were:

- MS Windows

This option has the advantage that hardware drivers, and related software, will generally be supplied with any TV tuner/recorder that is likely to be used.

The disadvantage would be that for many uses this operating system would be the least familiar and potentially the least flexible.

- OSX

Compatible DVB receivers are available for OSX. However, using OSX does present quite large restrictions on both the computer system and TV receiver hardware that can be used.

- Linux

Various device drivers are available, if hardware is carefully chosen there should be no large problems. There are even distributions specifically prepared for use for home theatre style installations (although such a dedicated system was not actually desired).

- FreeBSD (or any other BSD)

This option represents an operating system proven to be reliable and maintainable and, quite importantly, whose limitations were fairly well known. However, as mentioned above, device driver availability was likely to be a big problem.

For FreeBSD, initially, there existed a driver for just one type of DVB receiver device. The driver created by Raaf[1] was for a particular revision of the Freecom DVB-T USB stick.

However, since the video4linux (V4L) project appeared to contain drivers for various DVB devices[2], including the Freecom DVB-T USB and other more readily available USB sticks, it would be possible to attempt porting one of the V4L drivers to FreeBSD. In fact, a FreeBSD port of the V4L smslxxx driver has been created by Ganaël Laplanche[3]. Certainly, it would be worth examining the possibility of creating a new driver port before falling back to using another operating system.

1.2. linux-kmod-compat

Initial investigations into the porting of drivers began with consulting the FreeBSD Architecture Handbook, Raaf's (dvbusb) driver and some V4L driver source.

The first thing that became apparent was that the V4L DVB drivers existed in a larger V4L DVB driver framework (of sorts), and had quite a different structure to Raaf's dvbusb driver. A port of a V4L driver would involve some unravelling of a few layers of Linux driver code and/or the porting of the required parts of the common code.

However, as investigations into the general area of porting Linux video related drivers to FreeBSD were pursued and driver code was examined a bit more closely, another interesting possibility presented itself.

There exists a project[4], created by Luigi Rizzo, to enable the building of Linux device drivers on FreeBSD. This project consists of a layer of compatibility code, providing FreeBSD compatible implementations of Linux functions, structures, and macros, such that various Linux driver sources may be compiled, without modification, on FreeBSD. The code of this project is, in fact, available as a FreeBSD port (devel/linux-kmod-compat) and two video camera drivers that make use of this compatibility layer are also available as ports (multimedia/linux-gspca-kmod and multimedia/linux-ov511-kmod).

The possibility of being able to simply compile unmodified Linux DVB driver source code was certainly going to be worth pursuing. Even if, with the existing linux-kmod-compat port, it was not possible to compile all the required Linux source for a working DVB driver, it might be possible to follow the methodology, and extend the compatibility layer.

The advantage of the use of a compatibility layer would be that it should be straightforward to create ports of updated versions of a driver or, indeed, ports of other similar drivers. Such porting efforts should not require much understanding of any particular driver or piece of hardware. Perhaps, once a suitably extended compatibility layer was available, little understanding of drivers and kernel functionality in general would be required.

Therefore, the next steps taken were to take a look at the linux-kmod-compat port, try to figure out what bits of V4L source code would be required for a working DVB driver and attempt to compile, at least some, Linux DVB driver source.

Unsurprisingly, compilation of Linux DVB driver source didn't work with the unmodified linux-kmod-compat port. Therefore, work began on developing the compatibility layer for use with DVB driver source. Fortunately, the existing compatibility layer, which included some very useful notes and comments, proved to be a good starting point on the path into the world of FreeBSD (and Linux) device drivers that lay ahead.

2. The very basics of FreeBSD (and Linux) device drivers

Although the use of a layer of compatibility code as described in this paper is aimed, to a certain extent, at reducing the need to deal with all the technical complexities of creating a device driver, it is based upon some fundamentals of FreeBSD, and Linux, device drivers. In particular, it is based upon the (unsurprising) similarities between device drivers of such systems.

This section contains a description of those fundamentals of device drivers upon which the use of the layer of compatibility code rests from the viewpoint of the approach involving such a layer of code.

As previously mentioned, the FreeBSD Architecture Handbook, the linux-kmod-compat port, Raaf's dvbusb driver and various other FreeBSD driver sources, including example code found at `/usr/share/examples/`, were examined in order to get some idea about how Linux DVB driver source could be used to make a DVB driver kernel module for FreeBSD.

Although there may exist various ways in which a kernel may communicate with a device driver, what became apparent from early examination and development of the compatibility layer code, is that there are two basic methods of communication that are most relevant to the driver code concerned thus far.

Both of these methods involve the device driver supplying a particular set of functions that can be invoked by the kernel (or a sub-system such as a bus driver). The invoked driver function may then, in turn, call various kernel functions and/or return a value back to the invoker.

2.1. Communication of hardware related events

The first set of such functions are those functions that are invoked when certain hardware events take place. These functions are the basic skeleton of a driver. On FreeBSD, the most significant of these functions are:

- A function used to query a driver to see to what extent, if any, a driver may support a particular hardware device.

A hardware device driver will, generally, be associated with some sort of bus. When a new device appears on a bus, the drivers associated with the bus will be queried, by invoking their probe function, in turn in an attempt to find a driver that may be used for the device.

- A function used to initialize the driver and device.

When a device is attached, and a driver for that device found, an attach function will be called. It is within this function that the driver begins its work.

- A function to be called when an attached device is detached.

The kernel invokes a detach function to inform the driver when one of its hardware devices is detached.

Linux drivers use a very similar set of functions. Perhaps the only significant difference is that there is no analogue to the FreeBSD driver probe function. Instead, it seems that, a table of device ids, with which a driver may be used, are provided alongside the driver functions to be used by the parent bus (driver).

Therefore, wrappers around Linux driver functions can be used for the attach and detach functions required for a FreeBSD driver. The required probe function can make use of the table of device ids that is present within the Linux driver source.

2.2. Filesystem entries for devices

In the world of Unix, the file-like interface is used extensively. Unsurprisingly, communication with devices is often carried out using a file-like interface. One or more virtual filesystem entries (e.g. entries in `/dev`) are used to represent a device, and file operations may be performed upon these device representations.

It is, in general, the responsibility of drivers themselves to create such virtual filesystem entries. Typically, it is during device initialization that Linux and FreeBSD device drivers will create such character device (cdev) filesystem entries. This is achieved by invocations of the appropriate kernel function (`make_dev` on FreeBSD) with the relevant cdev parameters, including a second set of driver functions that will be associated with the cdev.

The virtual filesystem that handles cdevs (`devfs` on FreeBSD) can then invoke driver functions in order to respond to userland requests to a cdev.

During normal operation, most of the work performed by a (USB) device driver will take place as a result of userland requests to one or more cdevs that represent a device (or some part of a device).

3. Using a layer of compatibility code to compile Linux driver source code for a FreeBSD kernel

3.1. What the compatibility layer is not

The main aim of the compatibility layer is to enable Linux driver source to be used to create a FreeBSD driver. Ideally, the compatibility layer would be developed to the point at which Linux driver source could be used entirely unmodified.

However, there are some incompatibilities that are, simply, much more easily dealt with by minor modifications to Linux source code and a complete solution is not necessary in order to create a working FreeBSD driver from Linux source.

- The compatibility layer does not read or adapt Linux makefiles. These are not used at all.
- The compatibility layer has been developed to create a FreeBSD USB device driver. Support for other buses has not been added.
- The compatibility layer does not create modules of shared driver functionality. Generic DVB functionality was replaced by simplified functionality for use by a single USB DVB driver module.
- The compatibility layer is not a tool to turn Linux driver source code into FreeBSD source.
- The compatibility layer does not provide a general method by which a FreeBSD driver can be prepared from Linux source.
- The compatibility layer is not intended to completely avoid the need to modify Linux source.

3.2. Porting: the three directions of attack

The compatibility layer, as it has been developed thus far, constitutes just one direction of attack on the problem of porting a driver. The porting strategy of using the compatibility layer really involves attacking the porting from several directions:

- Create new code (native to FreeBSD).

A traditional port of a driver would involve the creation of what would be, essentially, new code based upon some other existing driver or drivers (from another system).

- Modify Linux driver code.

Rather than actually writing new source code, a port of a driver could conceivably be prepared only by modifying Linux driver source. However, in order to make Linux driver source compatible with FreeBSD, only by modifying source, it is likely that the extent of the modifications required would be more or less equivalent to writing new code.

- Adapt the FreeBSD environment (including the build environment) in order to be able to make use of Linux driver source.

This is what the compatibility layer is all about. Through the use of various structure, macro and function definitions the compatibility layer provides functionality and reinterpretation of Linux driver source so that it can work with a FreeBSD kernel.

3.3. The new FreeBSD USB stack and emulation of Linux USB functionality

There are various areas of Linux functionality that need to be emulated, various sets of Linux functions that need to be implemented, in order to compile V4L USB DVB driver source. It should come as no surprise that for a USB device driver, Linux USB functionality is, perhaps, the most significant of these areas. Indeed, much of the code of the original linux-kmod-compat port is concerned with emulating Linux USB functionality.

Therefore, one of the early considerations was the extent to which the linux-kmod-compat port supported the USB functionality that might be required by a USB DVB driver and how it might be possible to extend support should it be necessary. Initial investigation into this area revealed that a second USB stack had been developed for FreeBSD.

This newer stack, developed by Hans Petter Selasky, apparently addressed limitations of the, then current, FreeBSD stack that might have some bearing upon a DVB driver. Furthermore, the new stack was being integrated into the official FreeBSD kernel sources and included a similar set of emulated Linux USB functions and related definitions as present in the linux-kmod-compat port.

Therefore, the Linux compatibility code of Hans Petter Selasky's new USB stack became the basis upon which FreeBSD drivers utilising Linux driver source were to be built. This code provided the skeleton of a FreeBSD USB device driver and virtually all the necessary emulation of Linux USB functionality around which the remainder of the compatibility layer would function. It is also where this paper's examination of the compatibility layer code will begin.

3.4. The skeleton of a compatibility layer USB device driver

The functions that are required for a FreeBSD USB device driver work in concert with the compatibility layer's `usb_linux_register` and `usb_linux_deregister` functions. These two functions are invoked via emulated Linux functions and/or macros. They are responsible for maintaining a list of `usb_driver` structures, each of which contains Linux driver details, including pointers to the `probe` and `disconnect` functions.

The `usb_linux_attach` function, as shown in Example 1, is used when a compatible device is attached. This function retrieves the `usb_driver` corresponding to the Linux driver details to be used, initializes data used by the compatibility layer, invokes the Linux `probe` function from the driver details and, if the `probe` function succeeds, adds the Linux driver details to a list of attached drivers.

Example 1. The *attach* function from `usb_compat_linux.c`

```
static int
usb_linux_attach(device_t dev)
{
    struct usb_attach_arg *uaa = device_get_ivars(dev);
    struct usb_linux_softc *sc = device_get_softc(dev);
    struct usb_driver *udrv;
    const struct usb_device_id *id = NULL;

    mtx_lock(&Giant);
    LIST_FOREACH(udrv, &usb_linux_driver_list, linux_driver_list) {
        id = usb_linux_lookup_id(udrv->id_table, uaa);
        if (id)
            break;
    }
    mtx_unlock(&Giant);

    if (id == NULL) {
        return (ENXIO);
    }
    if (usb_linux_create_usb_device(uaa->device, dev) != 0)
```

```

        return (ENOMEM);
device_set_usb_desc(dev);

sc->sc_fbsd_udev = uaa->device;
sc->sc_fbsd_dev = dev;
sc->sc_udrv = udrv;
sc->sc_ui = usb_ifnum_to_if(uaa->device, uaa->info.bIfaceNum);
if (sc->sc_ui == NULL) {
    return (EINVAL);
}
if (udrv->probe) {
    if ((udrv->probe) (sc->sc_ui, id)) {
        return (ENXIO);
    }
}
mtx_lock(&Giant);
LIST_INSERT_HEAD(&usb_linux_attached_list, sc, sc_attached_list);
mtx_unlock(&Giant);

/* success */
return (0);
}

```

The FreeBSD probe and detach functions, `usb_linux_probe` and `usb_linux_detach`, work with compatibility data in a similar manner. The `usb_linux_probe` only consults the list of driver details and calls no Linux driver function.

As for other FreeBSD USB drivers, an array of the required driver functions and other required details are registered with the parent bus (driver) through the use of the `DRIVER_MODULE` macro. These declarations, shown below in Example 2, describe the skeleton of the driver.

Example 2. Declaration of the USB driver in `usb_compat_linux.c`

```

static device_method_t usb_linux_methods[] = {
    /* Device interface */
    DEVMETHOD(device_probe, usb_linux_probe),
    DEVMETHOD(device_attach, usb_linux_attach),
    DEVMETHOD(device_detach, usb_linux_detach),
    DEVMETHOD(device_suspend, usb_linux_suspend),
    DEVMETHOD(device_resume, usb_linux_resume),

    {0, 0}
};

static driver_t usb_linux_driver = {
    .name = "usb_linux",
    .methods = usb_linux_methods,
    .size = sizeof(struct usb_linux_softc),
};

static devclass_t usb_linux_devclass;

DRIVER_MODULE(usb_linux, uhub, usb_linux_driver, usb_linux_devclass, NULL, 0);

```

3.5. Wrapper functions for cdevs

An examination of the `ldev_stub.c` file belonging to the original `linux-kmod-compat` port reveals the use of a second set of driver (wrapper) functions. These functions are required for the operation of FreeBSD `cdevs`, they are basically wrappers around the Linux driver functions used for `cdevs` and referenced from the Linux `file_operations` structure.

The code of the `linux-kmod-compat` port contains the declaration of a `cdevsw` structure, as shown in Example 3, and the definitions of each of the functions referenced within this structure. Example 4 shows the `ldev_open` open function and how it wraps around the Linux driver function referenced by `sc->l_u_d.fops->open`.

Example 3. The declaration of a `cdevsw` structure in `ldev_stub.c`

```
static struct cdevsw ldev_cdevsw = {
    .d_version      = D_VERSION,
    .d_flags        = D_NEEDGIANT,
    .d_open         = ldev_open,
    .d_close        = ldev_close,
    .d_read         = ldev_read,
    .d_ioctl        = ldev_ioctl,
    .d_poll         = ldev_poll,
    .d_mmap         = ldev_mmap,
    .d_name         = __stringify(DRIVER_NAME),
};
```

Example 4. The `ldev_open` function from `ldev_stub.c`

```
static int
ldev_open(struct cdev *dev, int flag, int mode, usb_proc_ptr p)
{
    int err = 0;
    int unit = LDEVUNIT(dev);
    struct ldev_softc *sc = devclass_get_softc(ldev_devclass, unit);

    if (sc == NULL)
        return ENXIO;

    if (sc->error_status == EPIPE) /* detach in progress */
        return sc->error_status;

    /* if vopen is 0, set to 1 and return success */
    if (atomic_cmpset_int(&sc->vopen, 0, 1) == 0) /* already open */
        return EBUSY;
```

```

    err = -sc->l_u_d.fops->open((struct inode *)sc, &sc->l_u_d.file);
    if (err)
        goto bad;

    /* FILL HERE: fire the device */
    return 0;

bad:
    /* free memory */
    sc->vopen = 0;
    return err;
}

```

The `fops` field of the compatibility layer data (`sc->l_u_d` in Example 4), is set in a function that is called, from within Linux driver code, when the driver is associated with a `cdev`. The basic strategy taken is to use a single set of, `cdevsw`, functions that wrap around Linux driver functions associated with a `cdev` through the use of some other emulated Linux function.

3.6. Creating a driver using the compatibility layer

The compatibility layer, based upon the combination of the `linux-kmod-compat` port and compatibility code of Hans Petter Selasky's USB stack, was developed to enable the creation of a FreeBSD USB DVB driver from Linux source. Since the development of the layer, as a toolkit for compiling Linux driver source for FreeBSD, was intimately tied to the creation of a driver using the layer, the process of creating other drivers using the compatibility layer will involve similar use and development of the layer.

This process, which follows guidelines provided in the `linux-kmod-compat` port, mainly consists of incremental additions and fixes to the compatibility layer, and possibly driver sources, while attempting to compile a properly working driver module. The basic procedure used is set out below:

1. Assemble the sources, compatibility layer code and Linux driver sources that will be compiled.

For moderately complex drivers it is unlikely that the exact set of required Linux source files will be known before attempting to create a driver. Therefore, the Linux sources will probably consist of a larger collection of driver code from which files may be selected.

2. Create a makefile, based upon an existing (compatibility layer) makefile, for the new driver.

An initial set of Linux driver source files will need to be selected and specified in the makefile. At this point, the initial set of files should include only those that are obviously required.

3. Create a file containing the required driver declarations, driver function definitions, including the probe, attach and detach functions and also the wrapper functions for handling `cdev` operations.

It has been assumed that this code will be prepared for a particular Linux driver. Therefore, a new version of this code will be used for a new driver.

4. Attempt to make the driver module.

The compilation will probably fail with lots of error messages. These error messages will indicate which problems need to be addressed, there are several possibilities:

- A header file included by one of the driver sources cannot be found.

It is possible that paths, specified in the makefile for the module, may need to be adjusted in order for certain driver headers to be found. However, the more typical situation is that a compatibility version of a Linux header file, included by driver source, does not yet exist and a new, initially empty, header file will need to be added.

In some cases, any required content will already be available in other headers and no content need be added to the new header at all.

- Undefined functions and macros

Sometimes, this will be the result of one or more compatibility headers not being indirectly included in driver source as would be the case using the original Linux headers. Tracking down where, in the compatibility headers, an additional inclusion is required is not always easy.

Often, a Linux driver source file will contain uses of functions and/or macros not already implemented within the compatibility layer.

Adding such functions and/or macros starts with a search of Linux header files for the required declarations. Further Linux kernel code will then probably need to be examined and, in order to decide how to implement new functionality in the compatibility layer, it may also be necessary to consult FreeBSD manual pages and kernel sources.

Adding new functionality to the compatibility layer may simply involve adding macros or inline functions wrapping FreeBSD functions. Some Linux structure emulations may need to be defined. Possibly such structures can be treated as opaque types and pointers to such structures could actually refer to entirely different structures. Sometimes, this task requires getting familiar with more kernel code, Linux and FreeBSD than initially envisaged.

- Missing structure definitions

A similar situation as for missing functions. Basically, a new structure definition will need to be created. Linux kernel headers will probably need to be consulted, to see where the structure needs to be defined and roughly what its contents should be.

At this stage, the original Linux structure probably shouldn't be duplicated. All that is desired are the fields that are obviously used within driver code or whatever is necessary to implement any functionality associated with the structure.

- Missing fields from structure definitions

Occasionally a driver makes use of a field of a Linux structure not already present in the compatibility layer emulation of the structure.

This is not always easy to fix since the missing field may itself be of a type that is not yet present within the compatibility layer and/or the use of the field may indicate that some additional functionality is required.

- Syntactic errors in newly created code

Any newly created or modified code may contain syntactic errors that will need fixing.

5. Once that a kernel module has been built, it may be loaded for testing.

At this point, the module may successfully load but immediately fail with a panic. This could occur due to functions invoked at module loading time or due to a probe or attach driver function being automatically invoked for a device already attached. In either case, a complete module has been achieved and the process of creating a working driver moves into the debugging stage (step 6).

It is, however, quite possible that the module loading will fail due to undefined symbols.

Note that the `kldload` utility fails with the rather cryptic “No such file or directory” error message, the system log should contain the relevant message regarding which symbol was not defined.

This is generally the result of some function not being defined and will need to be resolved, in broadly the same manner as with compilation failure due to undefined functions (step 4), by adding the necessary definition and rebuilding the module.

In some cases, the definition will be found in a driver source file not yet included in the compilation and, therefore, at least one new source file will need to be specified in the module makefile.

6. Initial testing of the module and becoming familiar with kernel debugging.

It is probably very unlikely that a new driver will work without some kind of bad failure. It is likely that during attaching and operation, a new driver will fail in a way that causes a kernel panic or lock up of some sort.

In general, a kernel panic is probably preferable to some kind of lock-up since the kernel debugger can then be used to investigate. Bugs are most likely to be found in new code for emulating some Linux functionality or behaviour, but are certainly quite likely anywhere within the compatibility layer.

7. When a relatively stable driver has been created a second stage of debugging will be entered.

In this stage emphasis moves from preventing catastrophic failure to getting the driver to function correctly.

While there may be far fewer kernel panics, it is likely that there will be perplexing failures of driver functionality. Corrections may require examination of how the driver works and how, exactly, the compatibility layer needs to be adjusted in order to support the driver.

4. Resolving differences between Linux and FreeBSD

In this section some of the differences between Linux and FreeBSD are described and example compatibility layer code to handle these differences is presented.

4.1. Per-open instance data

On a Linux system an instance of an open `cdev` is associated with a file structure. Each time that `cdev` is opened, a new file structure is created. calls to any of the driver functions that are associated with a `cdev` (the functions of the `file_operations` structure) are accompanied with a reference to one of these file structures.

On FreeBSD things are a little different. Driver functions associated with a `cdev` (`cdevsw` structure) are not invoked with a particular instance of an open `cdev`, this context is not passed directly from the virtual filesystem layer to the driver functions. Also, the virtual filesystem does not invoke a `close` for each `open`, instead, a `close` is only invoked when the last instance of an open `cdev` is closed.

However, there is a mechanism by which FreeBSD can supply a driver with data specific to a particular opening of a cdev and by which the driver will be informed when each instance is closed.

Since FreeBSD 7.1 there has been devfs functionality to support the association of private driver data with an instance of an open cdev. The `devfs_set_cdevpriv`, `devfs_get_cdevpriv` and `devfs_clear_cdevpriv` functions are used, respectively, to set, retrieve and clear instance specific data. These functions may be used, in a driver, where an open file context exists (i.e. from cdev related functions).

When private data is set, a callback function may also be associated with the instance of the open cdev. This callback will be invoked when the instance is destroyed (closed) and is, therefore, the mechanism by which devfs informs a driver about the closing of a cdev. The use of the callback function makes the `cdevsw_d_close` function redundant.

In compatibility layer wrappers, the private data associated with a cdev includes a Linux file structure that may be passed to the, wrapped, Linux driver functions. The private data is allocated and set when a cdev is opened and retrieved in other `cdevsw` functions. This (wrapper) mechanism is illustrated in Example 5.

Example 5. The compatibility layer DVB driver `d_open` and `d_ioctl` functions

```
static int
linux_dvbdev_open(struct cdev *dev, int flags, int mode, struct thread *td)
{
    int err = 0;
    int unit = dvb_unit(dev);
    struct dvb_device *dvbdev;
    struct open_instance_data *instance_data;
    struct linux_dvbdev_softc *sc =
        devclass_get_softc(linux_dvbdev_devclass, unit);

    if (sc == NULL) {
        return ENXIO;
    }

    /* Get information for the linux (sub)device actually concerned */
    dvbdev = linux_dvbdev_get_dvbdev(sc, dev);

    if ( dvbdev == NULL ) {
        return ENXIO;
    }

    /* Create new open cdev instance data */
    instance_data = alloc_instance_data(dvbdev, dvbdev_release_callback);

    if ( instance_data == NULL ) {
        return ENOMEM;
    }

    set_up_file_structure(dvbdev, &(amp;instance_data->file), OFLAGS(flags));

    /* If there is an open function (reference), call the function. */
    if ( dvbdev->fops->open ) {
        err = dvbdev->fops->open((struct inode *)sc, &(amp;instance_data->file));
    }
}
```

```

        if ( err ) {
            /* If there won't be a valid open file, clean up the instance data. */
            clear_instance_data(instance_data, INSTANCE_DATA_NO_RELEASE);
        }

        return err;
    }

static int
linux_dvbdev_ioctl(struct cdev *dev, u_long cmd,
                  caddr_t addr, int fflag, struct thread *td)
{
    int err;
    int unit = dvb_unit(dev);
    struct open_instance_data *instance_data;
    struct file *file;
    struct linux_dvbdev_softc *sc =
        devclass_get_softc(linux_dvbdev_devclass, unit);

    /* Retrieve instance data */
    instance_data = retrieve_instance_data();

    if (instance_data == NULL) {
        return ENXIO;
    }

    file = &(instance_data->file);

    if ( file->f_op->ioctl ) {

        set_file_flags(file, OFLAGS(fflag));

        /* call the (sub)device-specific ioctl handler */
        err = -file->f_op->ioctl((struct inode *)sc, file,
                               cmd, (unsigned long)addr);
    } else {
        err = ENODEV;
    }

    return err;
}

```

4.2. Thread synchronization: Mutexes and locks

Both FreeBSD and Linux kernels include several methods for thread synchronization. There are various similar methods on both systems by which the execution and scheduling of threads may be affected.

One key method, used in both systems, is the use of locks that may be acquired (held, locked) if they have not already been exclusively acquired. Threads can be made to wait until they are able to acquire a lock and, therefore, synchronization achieved by requiring that a lock is held when performing certain operations.

Exclusively held *mutexes* and *spin locks* are used quite widely by Linux drivers. The FreeBSD kernel includes the equivalent *mutexes* and *spin mutexes*. At first sight, it would appear to be possible to wrap up the FreeBSD equivalents in order to emulate the Linux locks. However, there are some differences between the locks and their use that complicates matters.

- Linux *mutexes* and *spin locks* do not need to be destroyed when they are no longer required, they only need to be released (unlocked) before re-initialization or freeing. FreeBSD *mutexes* and locks should be destroyed (e.g. with (the `mtx_destroy` function).

It appears that there are no FreeBSD equivalents to Linux *mutexes* and *spin locks* that do not expect to be destroyed after use and that there is no good way to ensure that any FreeBSD equivalents do get destroyed. However, some careful examination of FreeBSD locks suggests that, while destruction should take place, it is currently possible to ignore this requirement.

- A FreeBSD thread may not sleep while holding a (normal) *mutex*, Linux *mutexes* may be held while sleeping.

Allowing a Linux driver to sleep while holding an emulated *mutex* means that a FreeBSD *mutex* cannot be used. Fortunately, there exists an alternative FreeBSD lock, the shared/exclusive lock, *sx*, that may be held while sleeping.

Using an *sx* lock to emulate a Linux *mutex* will allow Linux code to sleep while holding a *mutex*. However, since an *sx* lock may not be acquired while holding a FreeBSD *mutex*, any Linux code that may acquire a (Linux) *mutex* must not be called from any FreeBSD code where a (FreeBSD) *mutex* is held.

- The use of *spin locks* in Linux appears to be far more common than the use of *spin mutexes* within FreeBSD.

A Linux driver may hold a *spin lock* and call functions whose FreeBSD equivalents involve the acquisition of standard FreeBSD *mutexes*. Since it is not permissible to acquire a standard *mutex* while holding a *spin mutex*, it is not really possible to use FreeBSD *spin mutexes* in place of Linux *spin locks*.

The solution chosen is to assume that a *spin style lock*, in which a thread waiting to acquire the lock does not context switch, is not going to be necessary for the drivers being created on FreeBSD.

Instead, a standard FreeBSD *mutex* will be used.

The resulting emulation code basically consists of structure definitions containing the FreeBSD structure that will be used in place of the original Linux structure and a few simple wrapper functions. Example 6 shows some of the code used for emulating Linux *mutex* functionality, note the use of the `SX_NOWITNESS` flag used to avoid problems associated with not destroying a lock after use.

Example 6. Emulation of Linux *mutexes*

```
struct mutex {
    struct sx sx;
};

static inline void
mutex_init(struct mutex *_m)
{
    /* Clear the lock so that the lock never appears initialized. */
}
```

```

        bzero(_m, sizeof(struct mutex));
        sx_init_flags(&(_m)->sx, "ldevmtx", SX_NOWITNESS);
    }

#define mutex_lock(_m)      sx_xlock(&(_m)->sx)
#define mutex_unlock(_m)   sx_xunlock(&(_m)->sx)

```

4.3. Copying data to and from user-space

Moving data from memory used by a driver to memory used by a user space program requires copying data using specific functions for copying into or out of user space. FreeBSD and Linux use very similar functions. However, there are some differences in the manner in which drivers use these functions.

The `ioctl` system call, on Linux and FreeBSD may include a user space address that is to be used with the request. On FreeBSD, such a call for which a buffer has been specified will result in an automatic copy to or from the user space buffer (providing that the specified size is within limits). However, on Linux it is the task of the driver code to handle all the copying of data from and to user-space.

Fortunately, V4L drivers employ an additional function that handles the copying that FreeBSD performs implicitly. A replacement for this function, which takes into account the copying already performed, avoids the need to alter Linux code or for any emulated functions to be aware of the context in which they are called.

A second difference exists between the Linux and FreeBSD cdev read and write handling functions. FreeBSD's functions are provided with a pointer to a `uio` structure with which the FreeBSD `uiomove` function may be used. Linux driver functions just use buffer addresses and the Linux `copy_to_user` and `copy_from_user` functions.

Unfortunately, there exists driver code which expects and manipulates buffer addresses used for copying data to and from user space. If modifications to Linux driver code are to be avoided, this effectively rules out passing a FreeBSD `uio` structure down into emulated data copying functions (since there is no easy way to track the manipulations that the driver code may perform upon pointers).

To get around this problem, a compatibility layer IO wrapper function performs the function of `uiomove` using a Linux cdev read or write function in place of a FreeBSD `copying` or `copyout` function. The IO wrapper extracts the necessary address from a `uio` structure and passes it to the Linux cdev read or write and, ultimately to an emulated Linux `copy_from_user` or `copy_to_user` function, which will use the FreeBSD `copying` or `copyout`. Example 7 and Example 8 show, respectively, the wrapper function used for handling reading and the use of the wrapper.

Example 7. A compatibility IO wrapper used in place of `uiomove`

```

int
io_read_wrapper(int bytes_to_read, struct uio *uio, struct file *file,
                ssize_t (*read)(struct file *, char __user *, size_t, loff_t *))
{
    struct thread *td = curthread;
    struct iovec *iovc;
    u_int cnt;          /* Count of bytes */

```

```

int save = 0;
ssize_t res;

if (uio->uio_offset < 0 || uio->uio_resid < 0 ) {
    return (EINVAL);
}

/* This function will only deal with copying to user space */
if (uio->uio_segflg == UIO_SYSSPACE || uio->uio_rw != UIO_READ) {
    return (EFAULT);
}

if ( uio->uio_resid < bytes_to_read ) {
    bytes_to_read = uio->uio_resid;
}
if (bytes_to_read == 0) {
    return 0;
}

save = td->td_pflags & TDP_DEADLKTREAT;
td->td_pflags |= TDP_DEADLKTREAT;

/* Even if there are several io vectors. Only the first
   non-zero length buffer will be used. */
iovec = uio->uio_iov;
while ( (cnt = iovec->iiov_len) == 0 ) {
    uio->uio_iov++;
    iovec = uio->uio_iov;
    uio->uio_iovcnt--;
}

if (cnt > bytes_to_read) {
    cnt = bytes_to_read;
}

/* UIO_USERSPACE will result in a copy to userspace, UIO_NOCOPY will result
   in no reading of data at all (the uio structure will be adjusted). */
if (uio->uio_segflg == UIO_USERSPACE) {

    if (ticks - PCPU_GET(switchticks) >= hopticks) {
        uio_yield();
    }

    /* The linux read function should return a positive count of
       the bytes copied or a negative error value. */
    res = read(file, iovec->iiov_base, cnt, NULL);

    if ( res < 0 ) {
        if (save == 0) {
            td->td_pflags &= ~TDP_DEADLKTREAT;
        }
        return -res;
    }

    /* The number of bytes read may be fewer than requested. */
    cnt = res;
}

```

```

iov->iov_base = (char *)iov->iov_base + cnt;
iov->iov_len -= cnt;
uio->uio_resid -= cnt;
uio->uio_offset += cnt;

if (save == 0) {
    td->td_pflags &= ~TDP_DEADLK_TREAT;
}
return (0);
}

```

Example 8. A cdev `d_read` function using the `io_read_wrapper` function

```

static int
linux_dvbdev_read(struct cdev *dev, struct uio *uio, int ioflag)
{
    int err = 0;
    int count = uio->uio_resid;
    struct open_instance_data *instance_data;
    struct file *file;

    /* Retrieve instance data */
    instance_data = retrieve_instance_data();

    if (instance_data == NULL) {
        return ENXIO;
    }

    file = &(instance_data->file);

    if ( file->f_op->read ) {
        set_file_flags(file, 0);

        /* io_read_wrapper is used to handle the actual copy of data
           from kernel address space to user space. This function calls
           the specified Linux function (dvb->fops->read) which will,
           eventually, call the emulated copy_to_user function */
        err = io_read_wrapper(count, uio, file, file->f_op->read);
    } else {
        err = ENODEV;
    }

    return err;
}

```

5. The current status and future of the compatibility layer

The compatibility layer development and porting of a V4L DVB driver were performed predominantly on FreeBSD 7 systems. A working driver for dib0700 based USB2 DVB-T devices[5] was prepared and some basic testing performed with a Hauppague WinTV-NOVA-TD Stick.

The compatibility code for the DVB driver has now been updated for FreeBSD 8 (current) and a preliminary driver for a USB UVC video camera has been added (for FreeBSD 8 only). Testing of the drivers with the FreeBSD 8 code has been very limited.

There is still quite a lot that should really be done to the compatibility layer as it stands:

- General clean up of code and comments and a reorganization of source layout.

This could probably be said of a lot of projects. The makefiles do need to be fixed so that include paths are set up appropriately. It may also be worth revising function naming to avoid any potential name clashes.

- Revise the code to use a single shared set of driver and cdev function wrappers (as are present in the compatibility code of the new USB stack).
- Add new driver sources to the USB DVB and USB video driver modules in order to support more hardware.

There are also several directions in which further development could take place. For example:

- Support for buses other than USB could be added.
- Provision of more of the V4L framework so that drivers register with master DVB and/or video subsystems.

This would avoid the limitation that only one driver that uses particular cdev names can be used at any one time.

There is also another direction in which to take the whole strategy of using a compatibility layer to compile USB Linux driver code on FreeBSD. Through the use of libusb2 (libusb), it is possible for userland utilities to directly communicate with a USB device. This opens up the way to moving the bulk of a USB device driver into userland. Already, Hans Petter Selasky has created code for compiling Linux drivers as part of a userland application.

This certainly has advantages, perhaps the largest being that userland driver bugs won't cause kernel panics. The only current disadvantage is that applications must be linked with a userland driver library rather than using cdev (`/dev`) entries. Even this disadvantage could be removed by using some kind of bridge between userland and devfs (so that a userland process can create and receive requests to a `/dev` cdev). Therefore, this is a likely area where future development will focus.

References

- [1] *DVBUSB - Freecom DVB-T USB driver for FreeBSD* [<http://raaf.atspace.org/dvbusb/>]
- [2] *LinuxTV V4L-DVB Wiki* [http://www.linuxtv.org/wiki/index.php/Main_Page]

- [3] Laplanche, Ganaël. *FreeBSD Terratec Cinergy Piranha DVB driver* [<http://contribs.martymac.com/>]
- [4] Rizzo, Luigi. *Building Linux Device Drivers on FreeBSD*
[http://info.iet.unipi.it/~luigi/FreeBSD/linux_bsd_kld.html]
- [5] *DiB0700 USB2.0 DVB-T devices*
[http://www.linuxtv.org/wiki/index.php/DiBcom_USB_devices#DiB0700_USB2.0_DVB-T_devices]