How to Implement a Simple USB Driver for FreeBSD

BY MARIUSZ ZABORSKI

Have you ever bought new equipment for your FreeBSD box and it turned out that some functionality didn't work? Instead of returning it, it may be possible to write your own driver without much effort. We will explain how to write a simple USB driver for FreeBSD.

Case Study

In this article, we will look into a driver for Razer Ornata V2. The device is a Mecha-membrane keyboard which works perfectly on FreeBSD with one small issue: you can't change the backlight color. In some cases, you may find a keyboard that has a built-in color change. This means that the color will change independently on the software run on your machine under some key combination. In the case of this keyboard, the driver in operating systems controls the backlight. Thanks to that, you can have fancy patterns on your keyboard like fire flames. The disadvantage is that you have to have a driver for it. The device is shown in Figure 1.

Figure 1. Razer Ornata V2

Gathering the Information

First of all, we have to understand the protocol used in the driver. In the case of drivers for Razer, we have two ways of doing it:

- Look into an openrazer (unofficial collection of Linux drivers for Razer devices)
- Sniff the USB protocol from the Windows driver

In this article, we will combine these two methods. When we initially looked into the problem, there wasn't support for Razer Ornata V2 in the openrazer, so we had to deduct some of the parts from a USB protocol dump. The support for this keyboard was recently added to the openrazer, but when you try to write your driver, parts of it may not be available anywhere else than in the official Windows drivers. For educational purposes, we will assume that the openrazer doesn't support this keyboard.

OpenRazer

To get a needed context about the driver, we will try to find the package structure used to communicate with the keyboard, as this allows us to understand the dump from the USB sniff. The source code for openrazer is available under https://github.com/openrazer/openrazer. In a driver/razercommon.h file, we will find a `razer_report` structure, which is the main structure of the driver. It is used across all of the devices from this product. The structure is shown in Listing 1.

Listing 1. The razer_report structure defined by openrazer

```
struct razer_report {
    unsigned char status;
    union transaction_id_union transaction_id; /* */
    unsigned short remaining_packets; /* Big Endian */
    unsigned char protocol_type; /*0x0*/
    unsigned char data_size;
    unsigned char command_class;
    union command_id_union command_id;
    unsigned char arguments[80];
    unsigned char crc;/*xor'ed bytes of report*/
    unsigned char reserved; /*0x0*/
};
```

Sniffing a Windows Driver

To sniff a Windows USB driver, we can use a usbpcap (https://desowin.org/usbpcap/) tool. It is a command-line tool that is very simple to use (in Listing 2, we have an example). When we run the command tool, it will show us available devices; next, it will ask us which device we want to sniff and where to save a pcap file. The generated pcap file is easily viewable using Wireshark.

We will be targeting a Razer Windows driver. On Windows, the Razer Synapse tool allows you to customize the backlight colors of the keyboard. Let's try to set up different colors of the keyboard while the usbpcap is running. Thanks to this tool, we will record all requests sent to the keyboard (the Razer Synapse is shown in Figure 2). At this point, we will apply the red scheme on the whole keyboard.

Listing 2. Usage of usbpcap to capture the USB protocol

```
Following filter control devices are available:
1 \\.\USBPcap1
  \??\USB#ROOT_HUB20#4&19d0fd2a&0#{f18a0e88-c30c-11d0-8815-00a0c906bed8}
    [Port 1] Generic USB Hub
      [Port 4] ThinkPad Bluetooth 4.0
      [Port 6] Integrated Camera

2 \\.\USBPcap2
\??\USB#ROOT_HUB20#4&182122df&0#{f18a0e88-c30c-11d0-8815-00a0c906bed8}
    [Port 1] Generic USB Hub

3 \\.\USBPcap3
\??\USB#ROOT_HUB30#4&23ace5cb&0&0#{f18a0e88-c30c-11d0-8815-00a0c906bed8}
    [Port 1] Generic USB Hub
      Razer Ornata V2
        Razer Ornata V2
           Razer Ornata V2
        Razer Ornata V2
          Razer Control Device
Select filter to monitor (q to quit): 3
Output file name (.pcap): t1.pcap
```

Combining Methods

Now that we have a pcap from the dump, we can start analyzing the recorded protocol. Don't get into too much detail on how USB drivers work; instead, glean the general idea about the protocol. Most of the values we will just copy, as we might need to change them. We only want to generate similar requests as the original driver.

In Figure 3, we can see a dump created using usbpcap under Wireshark; in this case, the driver uses a setup packet. The setup packets are used for detection and configuration of the USB devices. In Table 1, we can see a package defined by the USB specification as well as the values that were sent by the driver.

Figure 2. Razer Synapse tool. The tool is used to configure the backlight color.

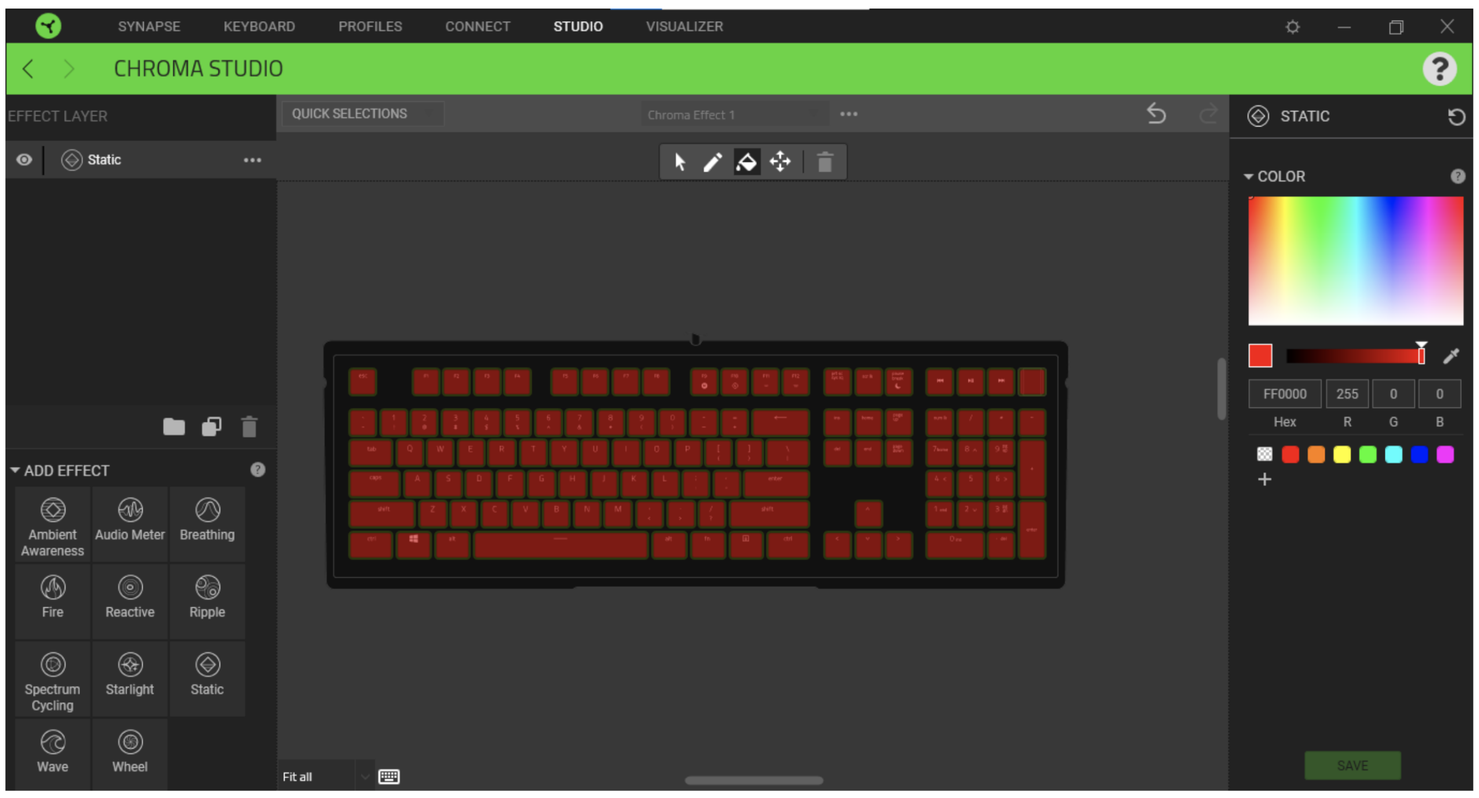


Table 1. Format of Setup Data from USB documentation, with the values from the dump.

Offset	Field	Size	Value	Description	Values from pcap
0	*bmRequestType*	1	Bitmap	Characteristics of request: • D7: Data transfer direction 0 = Host-to-device 1 = Device-to-host • D6...5:Type 0 = Standard 1 = Class 2 = Vendor 3 = Reserved • D4...0:Recipient 0 = Device 1 = Interface 2 = Endpoint 3 = Other • 4...31 = Reserved	0x21 • Data transfer direction: *Host-to-device* • Type: *Class* • Recipient: *Interface*
1	*bRequest*	1	Value	Specific request (for more details please refer to the USB Specification)	0x09 SET_REPORT
2	*wValue*	2	Value	Word-sized field that varies according to request	0x300
4	*wIndex*	2	Value or Offset	Word-sized field that varies according to request; typically used to pass an index or offset	2
6	*wLength*	2	Count	Number of bytes to transfer if there is a data stage	90

After the setup data, we have specific data for a Razer Driver. In Figure 4, we combined the pcap data with the `razer_report` structure from the openrazer project. Next, we can easily see some more things about the arguments. First, we have 2 bytes set to 0, which we can assume are reserved. Next, we have a one byte set to 1. When we look into the pcap, we can see many similar packages that, in this place, have this value in range from 0 to 5. We can verify this later, but actually it seems that this is the row number on the keyboard. Then, we have a value 0x15 (21), which is actually the number of keys in a row. Finally, there is a 21-times repeated value 0xff0000, which seems to refer to the red color that we set in RGB (R: 255, G:0, B:0).

Figure 3. The pcap generated using usbpcap under Wireshark. The Setup packet is highlighted.

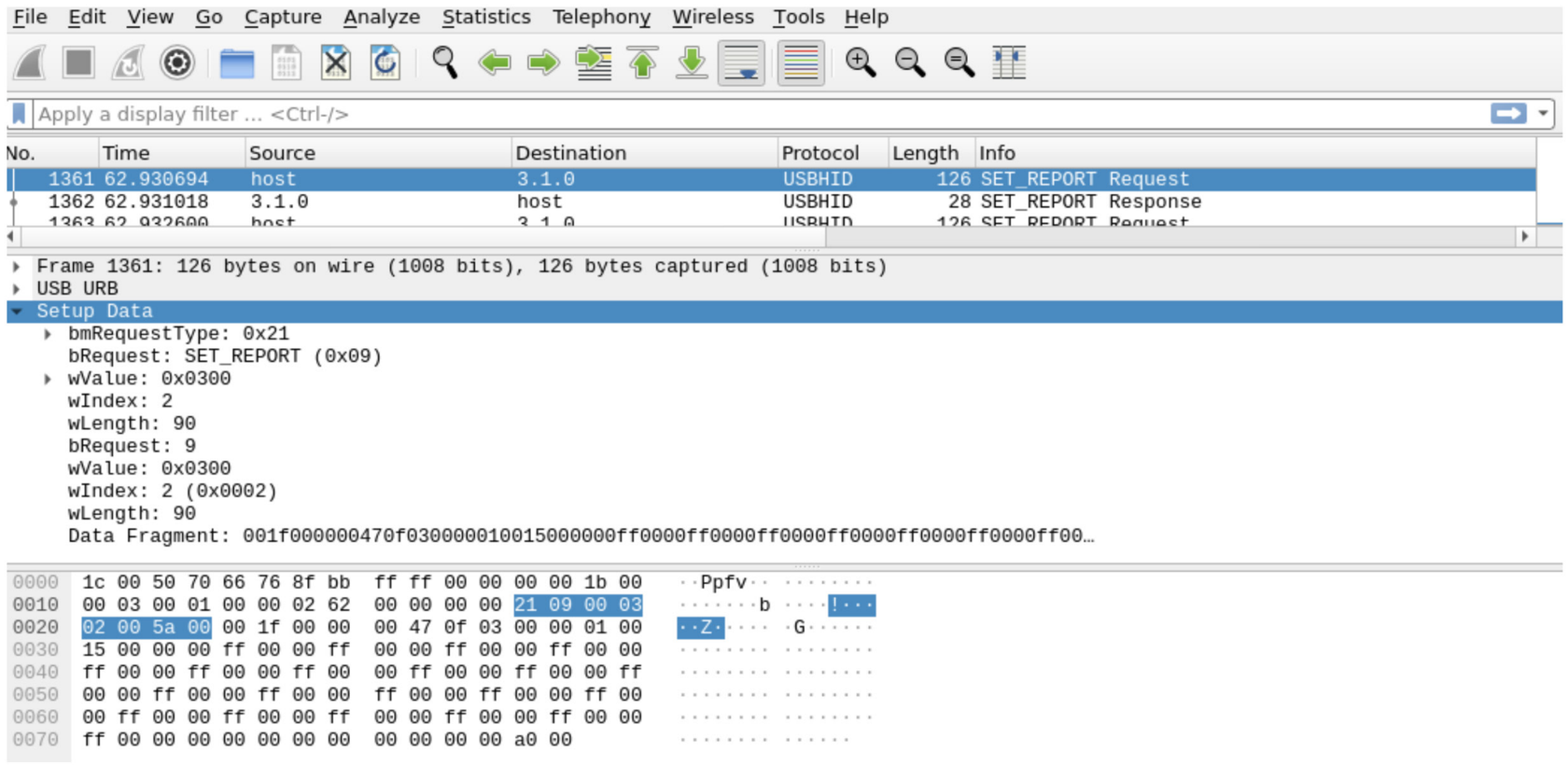


Figure 4. The setup data with the structure obtained from openrazer.

```
0000 00 1f 00 00 00 47 0f 03 00 00 01 00 15 00 00 00
0010 ff 00 00 ff 00 00 ff 00 00 ff 00 00 ff 00 00 ff
0020 00 00 ff 00 00 ff 00 00 ff 00 00 ff 00 00 ff 00
0030 00 ff 00 00 ff 00 00 ff 00 00 ff 00 00 ff 00 00
0040 ff 00 00 ff 00 00 ff 00 00 ff 00 00 ff 00 00 00
0050 00 00 00 00 00 00 00 00 a0 00
```

Fields	Values
Status	0x00
Transaction ID	0x1f
Remaining packets	0x0000
Protocol Type	0x00
Data Size	0x47
Command class	0x0f
Command ID	0x03
Arguments	0000010015
CRC	0xa0
Reserved	0x00

The packages used by the Razer Synapse seem to allow us to set each key to a different color. Each package refers to a single row on the keyboard. Without going into too much detail of USB and Razer protocol, this should be enough to implement any type of backlight effect we might like.

Lastly, we need to find a vendor and product identifier which will allow us to find the correct USB device. To do that, we can use the usbconfig(8) tool on the FreeBSD box. The utility has a special option, dump_device_desc, which allows us to print the details of all USB devices connected to our box. The example of usage is shown in Listing 3.

Listing 3. Identifying vendor and product ID using usbconfig(8) tool.

```
# usbconfig dump_device_desc
ugen0.4: <Razer Razer Ornata V2> at usbus0, cfg=0 md=HOST spd=FULL (12Mbps)
pwr=ON (500mA)

  bLength = 0x0012
  bDescriptorType = 0x0001
  bcdUSB = 0x0200
  bDeviceClass = 0x0000  <Probed by interface class>
  bDeviceSubClass = 0x0000
  bDeviceProtocol = 0x0000
  bMaxPacketSize0 = 0x0040
  idVendor = 0x1532
  idProduct = 0x025d
  bcdDevice = 0x0200
  iManufacturer = 0x0001  <Razer>
  iProduct = 0x0002  <Razer Ornata V2>
  iSerialNumber = 0x0000  <no string>
  bNumConfigurations = 0x0001
```

libusb&PyUSB

The first way of implementing a simple driver is to use libusb and PyUSB. This method allows us to write a USB driver in a userland without any additional kernel modules. Writing drivers in a userland is the most secure, because if there is a bug, it will expose only the kernel part for attack.

The libusb is a library for USB devices. It is a cross platform library, so we can see a port of it in FreeBSD, Linux, OpenBSD or even Windows. To simplify the task even more, instead of writing a driver in C, we can implement it using Python, which is possible thanks to the PyUSB module. PyUSB provides easy access to a host USB system. We can simply install pyusb using the pkg(8) tool (e.g. `pkg install py38-pyusb`).

First, we have to find a valid device. To do that, we use a usb.core.find function. To identify the right device, we can provide a product and vendor ID obtained from usbconfig(8), which is shown in Listing 4.

Listing 4. Finding a device using PyUSB.

```
# python
Python 3.8.10 (default, Jul  6 2021, 01:34:57)
>>> import usb.core
>>> dev = usb.core.find(idVendor=0x1532, idProduct=0x025d)
>>> dev.product
'Razer Ornata V2'
```

To send a Setup packet, we use the `ctrl_transfer` function. The interface of this function corresponds to the parameters described in the Setup packet. The simplest thing to do here is to copy all sniffed parameters. The last step is to rebuild the package. In our driver, we will assume that the color is hardcoded. Besides the color, row and CRC field, we will copy all of them from the sniffed part (the whole process is shown in Listing 5). At the end, we also have to recalculate the CRC field.

Listing 5. Sending a request to change a color using PyUSB.

```
import usb.core

# Color
r = 0xff
g = 0x00
b = 0x00

def change_color(dev, line, r, g, b):
    # Recreate package
    package = bytes([
        0x00,           # Status
        0x1f,           # Transaction ID
        0x00, 0x00,     # Remaining packets
        0x00,           # Protocol Type
        0x47,           # Data Size
        0x0f,           # Command Class
        0x03,           # Command ID
```

```
                0x00,       # - unknown
                0x00,       # - unknown
                line,       # - line
                0x00,       # - unknown
                0x15,       # - number of keys
                0x00, 0x00, # - unknown
                0x00        # - unknown
    ])
    for _ in range(0x15):
        package += bytes([r, g, b])

    # Fill up to 0x47 bytes size
    for _ in range(0x3):
        package += bytes([0, 0, 0])

    # Recalculate crc
    crc = 0x00
    for x in package:
        crc ^= x
    package += bytes([crc, 0x00]) # crc and reserved
    dev.ctrl_transfer(
        bmRequestType = 0x21,
        bRequest = 0x09,
        wValue = 0x300,
        wIndex = 0x02,
        data_or_wLength = package
    )

dev = usb.core.find(idVendor=0x1532, idProduct=0x025d)
for line in range(6):
    change_color(dev, line, r, g, b)
```

Kernel Module

In the case of a native driver, we have to write a FreeBSD kernel module. We also have to implement some kind of communication between the kernel and the userland to tell the module what color we want. To accomplish this, we can expose some additional sysctl, implement a iocotl(9) or read the input from the USB dev node. In this article, we will look at the ioctl(9) method.

Building a Kernel Module

First, we have to know how to compile the kernel module. The simplest way of doing this is using a Makefile and including the bsd.kmod.mk file. Thanks to that, it will auto generate all additional required files and headers. We also have to remember to include files like opt_usb.h, buf_if.h and device_if.h., which is common for all kernel modules. In the KMOD detective, we provide the name of the compiled driver. The example of Makefile is shown in Listing 6.

Listing 6. Makefile for building kernel module in FreeBSD.

```
SRCS=ornata.c
SRCS+=opt_usb.h bus_if.h device_if.h

KMOD=ornata

.include <bsd.kmod.mk>
```

The three standard methods that almost all drivers have to implement is probe, attach and detach. There are also additional methods, like suspend and resume, but we won't look into them.

The probe is executed first to examine the device and decide if the driver is supported or not. Here, we can use a VendorID and ProductID to decide if this is the device we are looking for. We can accomplish that using a `usbd_lookup_id_by_uaa` function, which will iterate over the given array of vendors and products to find a matching pair. We also have to check if the device is in host mode (`USB_MODE_HOST`), which is needed to initiate data transfers. Next, we want to be sure the device is actually a keyboard. The probe function is shown in Listing 7.

Listing 7. The USB probe function

```
static const STRUCT_USB_HOST_ID ornata_devs[] = {
	{USB_VPI(0x1532, 0x025d, 0)},
};

static int
ornata_probe(device_t self)
{
	struct usb_attach_arg *uaa = device_get_ivars(self);

	if (uaa->usb_mode != USB_MODE_HOST)
		return (ENXIO);

	if (uaa->info.bInterfaceProtocol == UIPROTO_BOOT_KEYBOARD)
		return (ENXIO);

	if (uaa->info.bInterfaceClass != UICLASS_HID)
		return (ENXIO);

	return (usbd_lookup_id_by_uaa(ornata_devs, sizeof(ornata_devs), uaa));
}
```

Two other methods that are useful are attach and detach. The attach function is called when the probe phase is finished and the probe function returns success. It is an entry point that allows the driver to initialize all required resources. At the opposite side, we have a detach function that allows us to clean up after the device disappears.

In case of this, the driver in the attached function will initialize mutex needed for synchro-

nizing and allocate the USB driver's entry points under /dev. The last part is done by the `usb_fifo_attach` function. While creating a new node, we have to also define what operations it supports (the `ornata_fifo_methods` variable), but we will look into that in the next phases. While creating a node, we can define which user and group should be an owner (in our case root(0) and wheel(0) group) and in what mode the node should be initialized (in our case everyone can read and write (666)). At this moment, we also introduce a helping structure which stores all device specific variables. At the opposite side, in the detach routine, we call the `usb_fifo_detach` function, which destroys its associated USB device node. These functions are shown in Listing 8.

Listing 8. Attach and detach function for the driver.

```
struct ornata_softc {
        struct usb_fifo_sc sc_fifo;
        struct mtx sc_mtx;

        struct usb_device  *sc_udev;
};

static int
ornata_attach(device_t self)
{
        struct usb_attach_arg *uaa = device_get_ivars(self);
        struct ornata_softc *sc = device_get_softc(self);
        int unit = device_get_unit(self);
        int error;

        device_set_usb_desc(self);
        mtx_init(&sc->sc_mtx, "ornata lock", NULL, MTX_DEF);

        error = usb_fifo_attach(uaa->device, sc, &sc->sc_mtx,
            &ornata_fifo_methods, &sc->sc_fifo,
            unit, -1, uaa->info.bIfaceIndex,
            0, 0, 0666);
        if (error)
                goto detach;

        sc->sc_udev = uaa->device;

        return (0);
detach:
        mtx_destroy(&sc->sc_mtx);
        return (error);
}

static int
ornata_detach(device_t self)
{
        struct ornata_softc *sc = device_get_softc(self);
```

```
        usb_fifo_detach(&sc->sc_fifo);
        mtx_destroy(&sc->sc_mtx);

        return (0);
}
```

Finally, we can define the driver module, which is shown in Listing 9. We are creating a kernel driver using a DRIVER_MODULE macro. In this part, we are setting the probe, attach and detach function to the strcuture driver. The MODULE_DEPEND macro is used to set the dependency on another kernel module. This is only used to help the operating system to load all required modules before loading this one; however, this does not dictate the order of the load.

Listing 9. Definition of kernel module.

```
static device_method_t ornata_methods[] = {
        DEVMETHOD(device_probe, ornata_probe),
        DEVMETHOD(device_attach, ornata_attach),
        DEVMETHOD(device_detach, ornata_detach),

         DEVMETHOD_END
};

static driver_t ornata_driver = {
        .name   = "ornata",
        .methods = ornata_methods,
        .size = sizeof(struct ornata_softc)
};

static devclass_t ornata_devclass;

DRIVER_MODULE(ornata, uhub, ornata_driver, ornata_devclass, NULL, 0);
MODULE_DEPEND(ornata, ukbd, 1, 1, 1);
MODULE_VERSION(ornata, 1);
USB_PNP_HOST_INFO(ornata_devs);
```

At this point, we can implement a function that will send setup data to the device. This can be done using the `usbd_do_request_flags` function and the `usb_device_request` structure representing the request. For the data part, we can use the structure from openrazer, as it is already implemented in the C language. For example, in the case of the python driver, the function will expect the color and the line to set, and most of the variables are just copied from our sniffed requests. We also have to remember to recalculate the CRC field. The USET macros allow us to set data independent of CPU endianness. The function for setting the backlight color is shown in Listing 10.

Listing 10. Attach and detach function for the driver.

```
static void
ornata_set_color(struct ornata_softc *sc, uint8_t r, uint8_t g, uint8_t b, uint8_t
line)
{
        struct razer_report rr;
        struct usb_device_request req;
```

```
char crc, *ptr;
int i;

memset(&rr, 0, sizeof(rr));

req.bmRequestType = 0x21;
req.bRequest = 0x09;
USETW(req.wValue, 0x300);
USETW(req.wIndex, 2);
USETW(req.wLength, sizeof(rr));

rr.status = 0x00;
rr.transaction_id = 0x1f;
rr.remaining_packets = 0x00;
rr.protocol_type = 0x00;
rr.data_size = 0x47;
rr.command_class = 0x0f;
rr.command_id = 0x03;

rr.arguments[2] = line;
rr.arguments[4] = 0x15;

for (i = 8; i < 8 + 0x15 * 3; i += 3) {
        rr.arguments[i] = r;
        rr.arguments[i + 1] = g;
        rr.arguments[i + 2] = b;
}

crc = 0;
for (ptr = (char *)&rr; ptr != (char *)&rr + sizeof(rr); ptr++) {
        crc ^= *ptr;
}

rr.crc = crc;

usbd_do_request_flags(sc->sc_udev, &sc->sc_mtx, &req,
    &rr, 0, NULL, 2000);
}
```

Implementing ioctl

The only missing part in the driver is the methods used to communicate with the USB device node. We will implement the ioctl method, as it is the simplest (but requires an additional program to send an ioctl).

First, we have to define the ioctl. To accomplish this, we can use _IOW macro, which defines a macro for a write operation — which means that the memory will be copied from userland to the kernel. For other purposes, we can use _IOR to define a read ioctl, or _IOWR for read/write operation, or _IO, which transfers no data. We will also use an additional structure, `ornata_color`, just to transfer the data in an organized way.

The definition of ioctl is shared between the userland and the kernel, so a good idea is to define a C file header that contains these definitions. The header is shown in Listing 11.

Listing 11. Attach and detach function for the driver.

```
#ifndef _ORNATA_H_
#define _ORNATA_H_

#include <sys/ioccom.h>

struct ornata_color {
        uint8_t r;
        uint8_t g;
        uint8_t b;
};

#defineORNATA_SET_COLOR _IOW('U', 205, struct ornata_color)

#endif
```

Now, getting back to the `usb_fifo_attach`, we use a structure `ornata_fifo_methods` that hasn't yet been defined. This structure defines supported operations on the device; for example, open or close. In our case, we want to support ioctl operations. The basename field describes the name of the node that should be created under /dev. When using the ioctl, the memory is already safely copied from the userland to the kernel, so we can just use color structure. The implementation of ioctl is shown in Listing 12.

Listing 12. Implementation of ioctl method.

```
static int
ornata_ioctl(struct usb_fifo *fifo, u_long cmd, void *addr, int fflags)
{
        struct ornata_softc *sc;
        struct ornata_color color;
        int error;
        uint8_t line;

        sc = usb_fifo_softc(fifo);
        error = 0;

        mtx_lock(&sc->sc_mtx);

        switch(cmd) {
        case ORNATA_SET_COLOR:
                color = *(struct ornata_color *)addr;
                for (line = 0; line < 6; line ++) {
                        ornata_set_color(sc,
                          color.r,
                          color.g,
                          color.b,
                          line);
                }
```

```
                break;
        default:
                error = ENOTTY;
                break;
        }

        mtx_unlock(&sc->sc_mtx);
        return (error);
}

static struct usb_fifo_methods ornata_fifo_methods = {
        .f_ioctl = &ornata_ioctl,
        .basename[0] = "ornata"
};
```

The disadvantage of this approach is that we have to implement an additional userland program, because there is no way of generating the ioctl(2) from a command line. This program is shown in Listing 13.

Listing 13. Example of usage of ioctl in userland.

```
int
main(void)
{
        int fd = open("/dev/ornata0", 0);
        struct ornata_color color;

        color.r = 0xFF;
        color.g = 0x00;
        color.b = 0x00;

        ioctl(fd, ORNATA_SET_COLOR, &color);

        return (0);
}
```

Summary

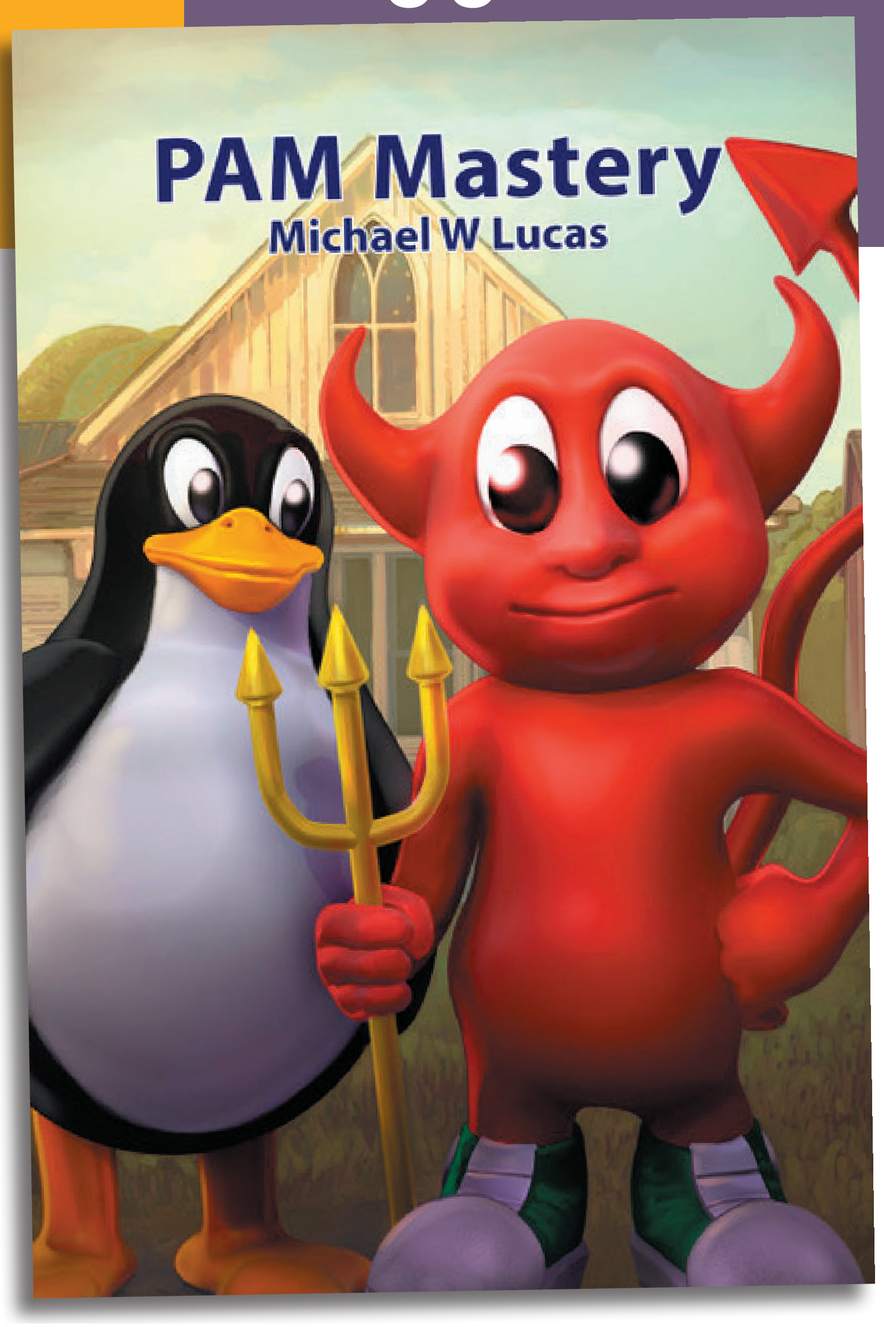
Implementing a userland driver isn't that complicated, thanks to libusb and pyusb. The most complicated part is actually understanding the protocol used by the device. If the protocol is simple, we can just sniff a lot of data from existing drivers on different platforms. If the protocol is more complicated, maybe there is an open-source project and we can port some part of it to FreeBSD. In the case of writing a native driver, we have to be patient, as the routines are more challenging. Implementing the kernel driver, we have to be very careful, as we can introduce bugs. Also, if we mess up something, the kernel may just panic, and we will need to restart the machine.

Bibliography

- USB 2.0 Specification — https://www.usb.org/document-library/usb-20-specification
- *FreeBSD Device Drivers A Guide for the Intrepid* by Joseph Kong
- Openrazer source code — https://github.com/openrazer/openrazer
- Roland's homepage — Setting the Razer ornata chroma color from userspace (https://rsmith.home.xs4all.nl/hardware/setting-the-razer-ornata-chroma-color-from-userspace.html)

MARIUSZ ZABORSKI currently works as a security expert at 4Prime. Since 2015, he has been the proud owner of the FreeBSD commit bit. His main areas of interest are OS security and low-level programming. In the past, he worked at Fudo Security, where he led a team developing the most advanced PAM solution in IT infrastructure. In 2018, Mariusz organized the Polish BSD User Group. In his free time, he enjoys blogging at https://oshogbo.vexillium.org.

Pluggable Authentication Modules: Threat or Menace?



PAM is one of the most misunderstood parts of systems administration. Many sysadmins live with authentication problems rather than risk making them worse. PAM's very nature makes it unlike any other Unix access control system.

If you have PAM misery or PAM mysteries, you need PAM Mastery!

"Once again Michael W Lucas nailed it." — nixCraft

PAM Mastery **by Michael W Lucas**

https://mwl.io