CS 411 Project 5: USB Missile Turret

Doug Dziggel David Merrick Michael Phan

June 7, 2013

Abstract

This paper describes the design process and implementation of creating a driver for the Dream Cheeky USB Missile Launcher.

1 Background

2 Design

To design our USB driver we first needed to download and run the Dream Cheeky software in Windows. Once we had successfully installed the software and Windows driver we then found a USB sniffer called USBlyzer. I found that the following Commands controlled the USB Missile Launcher.

Up: 01 00 00 00 00 00 00 00
Down: 02 01 00 00 00 00 00 00
Left: 02 04 00 00 00 00 00 00
Right: 02 08 00 00 00 00 00 00
Fire: 02 10 00 00 00 00 00 00
LED on: 30 01 00 00 00 00 00 00
LED off: 30 00 00 00 00 00 00 00
Filler: 01 00 00 00 00 00 00 00
Stop: 02 20 00 00 00 00 00 00

The next step was to create a driver that would use the above commands. For our design we found code that was written specifically for the Dream Cheeky USB Missile Launcher on Linux. The driver was set up to initalize, remove and run input commands. The file operations was set up to tie the userspace with the driver. We had two userspace programs, one to bind the missile launcher to our driver and the other to control the missile launcher. For the missile launch controller we decided to control it by holding the keys down rather than specifying a direction and a time to move in that direction.

3 Implementation

The struct usb_ml holds all of the different variables associated with the USB device. It holds the name, interface, IDs, number of times opened, locks, and pointers.

The struct launcher_table creates an entry of the form usb_device_id and inserts it into MODULE_DEVICE_TABLE, which tells the kernel that this driver code is able to support the specified hardware.

The struct launcher_driver sets the name of the driver that supports the hardware.

The function launcher_ctrl_callback will grab the usb device from the request and then set the usb to have correction_required = 0. The correction_required field is a variable for setting up the packets that are sent to USB.

The function launcher_abort_transfers performs checks to see if the usb device that is passed in the arguments exists, if its udev exists, and if it is actually connected. If so, it will set the USB to be seen as no longer running by setting int_in_running = 0. It will then kill any current request by looking at int_in_urb.

The function launcher_int_in_callback will take a request and print out debug messages that alert the function is called and state what the data and number of characters is. It then prints out the status of the request and if the status is something recoverable, it will try to resubmit. It then checks the request, locks it for use, and checks the position of the launcher in comparison with the max ranges. If the launcher is at its max, it will disable the command and mark that correction_required occured. If correction_required is set, it will insert the command to stop the direction of movement, unlock the data, and then submit the new request. The resubmit will send what is current in the request.

The function launcher_delete will call launcher_abort_transfers to cancel all transfers, then will free all data structures.

The function launcher_open will create pointers to structs usb_ml and usb_interface. It creates an int variable called subminor and sets it to the MINOR of the inode inodep. It creates a mutex and attempts to find the interface of the minor using usb_find_interface, then tries to find dev using usb_get_intfdata. Afterward, it will lock the device using down_interruptible, then increment open_count. It then instantiates the interrupt URB and sets itself to running through int_in_running. It submits the request in int_in_urb, then saves the object in the file pointer's private_data. It then unlocks the device, unlocks the mutex, and then return the result of usb_submit_urb.

The function launcher_close will create a file over to the device and grab it from the filer pointer's private_data. It checks if it exists, then locks the device. The open_count is checked to see if the device is opened before. If the device isn't mounted, then it will unlock the device, delete it, then exit. It then calls launcher_abort_transfers and unlocks the device.

The function launcher_read will return an error -EFAULT.

The function launcher_write will set retval to -EFAULT and change it to other errors should they occur. It grabs the device from the file pointer's private_data, and then checks to see if dev exists, is open, has data, and sets count to 1. It copies over the command and stores it into the buffer user_buf. Then it will begins clearing the buffer, and sets the device command to cmd. When it verifies that the byte has been submitted, it will unlock and exit.

The struct fops will set the name of open, release, read, and write functions to be using the launcher functions.

The function launcher_probe will set up the class, which contains the name of the launcher node and the list of functions that replace the file operations. It then registers the interface and class to the kernel. If the device isn't opened in the kernel, it will print an error. Otherwise it continues and allocates dev to usb_ml. It sets up the initial values with command set to stop, semaphores initialized, and then sets the device variables. The interrupt endpoint information is created through each byte in a for loop that sets the int_in_endpoint in dev to the endpoints. It allocates spec for the different variables in the dev struct and grabs the serial number from the device. The data pointer is saved using usb_set_intfdata and the interface minor is saved to the device, then exits.

The function launcher_disconnect will lock the mutex, grab the inferface, and completely unlocks the device. The minor node is deregistered using usb_deregister_dev and deletes the device, then unlocks the mutex.

The functions launcher_init will set the driver to recognize the launcher_probe and launcher_disconnect, then connects the driver to the usb, and then returns the result.

The function launcher_exit will deregister the driver.

4 Driver Code

```
/*
 * Dream Cheeky USB Thunder Launcher driver
 *
 * Copyright (C) 2012 Nick Glynn <Nick.Glynn@feabhas.com>
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License as
 * published by the Free Software Foundation, version 2.
 *
 * derived from the USB Missile Launcher driver
 * Copyright (C) 2007 Matthias Vallentin <vallentin@icsi.berkeley.edu>
 *
 * Note: The device node "/dev/launcher0" is created with root:root so you
 *       will need to chgrp, chmod or sudo your way to access
 */

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/usb.h>
#include <linux/slab.h>
#include <linux/usb.h>
#include <linux/mutex.h>
#include <linux/ioctl.h>
#include <asm/uaccess.h>

/* Rocket launcher specifics */
#define LAUNCHER_VENDOR_ID              0x2123
#define LAUNCHER_PRODUCT_ID             0x1010

#define LAUNCHER_NODE                   "launcher"
#define LAUNCHER_CTRL_BUFFER_SIZE       8
#define LAUNCHER_CTRL_REQUEST_TYPE      0x21
#define LAUNCHER_CTRL_REQUEST           0x09
#define LAUNCHER_CTRL_VALUE             0x0
#define LAUNCHER_CTRL_INDEX             0x0
#define LAUNCHER_CTRL_COMMAND_PREFIX    0x02

#define LAUNCHER_STOP                   0x20
#define LAUNCHER_UP                     0x02
#define LAUNCHER_DOWN                   0x01
#define LAUNCHER_LEFT                   0x04
#define LAUNCHER_RIGHT                  0x08
#define LAUNCHER_UP_LEFT                (LAUNCHER_UP | LAUNCHER_LEFT)
#define LAUNCHER_DOWN_LEFT              (LAUNCHER_DOWN | LAUNCHER_LEFT)
#define LAUNCHER_UP_RIGHT               (LAUNCHER_UP | LAUNCHER_RIGHT)
#define LAUNCHER_DOWN_RIGHT             (LAUNCHER_DOWN | LAUNCHER_RIGHT)
#define LAUNCHER_FIRE                   0x10

#define LAUNCHER_MAX_UP                 0x80            /* 80 00 00 00 00 00 00 00 */
```

```
#define LAUNCHER_MAX_DOWN               0x40            /* 40 00 00 00 00 00 00 00 */
#define LAUNCHER_MAX_LEFT               0x04            /* 00 04 00 00 00 00 00 00 */
#define LAUNCHER_MAX_RIGHT              0x08            /* 00 08 00 00 00 00 00 00 */

static struct usb_class_driver class;
static DEFINE_MUTEX(disconnect_mutex);

struct usb_ml {
        struct usb_device               *udev;
        struct usb_interface            *interface;
        unsigned char                   minor;
        char                            serial_number[8];

        int                             open_count;             /* Open count for this port */
        struct                          semaphore sem;          /* Locks this structure */
        spinlock_t                      cmd_spinlock;           /* locks dev->command */

        char                            *int_in_buffer;
        struct usb_endpoint_descriptor  *int_in_endpoint;
        struct urb                      *int_in_urb;
        int                             int_in_running;

        char                            *ctrl_buffer;           /* 8 byte buffer for the control msg
        struct urb                      *ctrl_urb;
        struct usb_ctrlrequest          *ctrl_dr;               /* Setup packet information */
        int                             correction_required;
        unsigned char                   command;                /* Last issued command */
};

/* Table of devices that work with this driver */
static struct usb_device_id launcher_table[] =
{
        { USB_DEVICE(LAUNCHER_VENDOR_ID, LAUNCHER_PRODUCT_ID) },
        {} /* Terminating entry */
};
MODULE_DEVICE_TABLE (usb, launcher_table);

static struct usb_driver launcher_driver =
{
        .name = "launcher_driver",
        .id_table = launcher_table,
};

static void launcher_ctrl_callback(struct urb *urb)
{
        struct usb_ml *dev = urb->context;
        pr_debug("launcher_ctrl_callback\n");
        dev->correction_required = 0;        /* TODO: do we need race protection? */
}

static void launcher_abort_transfers(struct usb_ml *dev)
```

```
{
        if (! dev) {
                pr_err("dev is NULL");
                return;
        }

        if (! dev->udev) {
                pr_err("udev is NULL");
                return;
        }

        if (dev->udev->state == USB_STATE_NOTATTACHED) {
                pr_err("udev not attached");
                return;
        }

        /* Shutdown transfer */
        if (dev->int_in_running) {
                dev->int_in_running = 0;
                mb();
                if (dev->int_in_urb) {
                        usb_kill_urb(dev->int_in_urb);
                }
        }

        if (dev->ctrl_urb) {
                usb_kill_urb(dev->ctrl_urb);
        }
}

static void launcher_int_in_callback(struct urb *urb)
{
        struct usb_ml *dev = urb->context;
        int retval;
        int i;

        pr_debug("launcher_int_in_callback\n");

        pr_debug("actual_length: 0x%X - data was: ", urb->actual_length);
        for (i = 0; i < urb->actual_length; ++i) {
                pr_debug("0x%X ", (const unsigned char)(urb->transfer_buffer) + i);
        }

        if (urb->status) {
                if (urb->status == -ENOENT) {
                        pr_debug("received -NOENT\n");
                        return;
                } else if (urb->status == -ECONNRESET) {
                        pr_debug("received  -ECONNRESET\n");
                        return;
                } else if (urb->status == -ESHUTDOWN) {
```

```
                pr_debug("received -ESHUTDOWN\n");
                return;
        } else {
                pr_err("non-zero urb status (%d)", urb->status);
                goto resubmit; /* Maybe we can recover. */
        }
    }

    if (urb->actual_length > 0) {
        spin_lock(&dev->cmd_spinlock);

        if (dev->int_in_buffer[0] & LAUNCHER_MAX_UP && dev->command & LAUNCHER_UP) {
                dev->command &= ~LAUNCHER_UP;
                dev->correction_required = 1;
        } else if (dev->int_in_buffer[0] & LAUNCHER_MAX_DOWN &&
                   dev->command & LAUNCHER_DOWN) {
                dev->command &= ~LAUNCHER_DOWN;
                dev->correction_required = 1;
        }

        if (dev->int_in_buffer[1] & LAUNCHER_MAX_LEFT && dev->command & LAUNCHER_LEFT) {
                dev->command &= ~LAUNCHER_LEFT;
                dev->correction_required = 1;
        } else if (dev->int_in_buffer[1] & LAUNCHER_MAX_RIGHT &&
                   dev->command & LAUNCHER_RIGHT) {
                dev->command &= ~LAUNCHER_RIGHT;
                dev->correction_required = 1;
        }


        if (dev->correction_required) {
                dev->ctrl_buffer[0] = dev->command;
                spin_unlock(&dev->cmd_spinlock);
                retval = usb_submit_urb(dev->ctrl_urb, GFP_ATOMIC);
                if (retval) {
                        pr_err("submitting correction control URB failed (%d)", retval);
                }
        } else {
                spin_unlock(&dev->cmd_spinlock);
        }
    }

resubmit:
    /* Resubmit if we're still running. */
    if (dev->int_in_running && dev->udev) {
        retval = usb_submit_urb(dev->int_in_urb, GFP_ATOMIC);
        if (retval) {
                pr_err("resubmitting urb failed (%d)", retval);
                dev->int_in_running = 0;
        }
    }
```

```
}

static inline void launcher_delete(struct usb_ml *dev)
{
        launcher_abort_transfers(dev);

        /* Free data structures. */
        if (dev->int_in_urb) {
                usb_free_urb(dev->int_in_urb);
        }
        if (dev->ctrl_urb) {
                usb_free_urb(dev->ctrl_urb);
        }

        kfree(dev->int_in_buffer);
        kfree(dev->ctrl_buffer);
        kfree(dev->ctrl_dr);
        kfree(dev);
}

static int launcher_open(struct inode *inodep, struct file *filp)
{
        struct usb_ml *dev = NULL;
        struct usb_interface *interface;
        int subminor;
        int retval = 0;

        pr_debug("launcher_open\n");
        subminor = iminor(inodep);

        mutex_lock(&disconnect_mutex);

        interface = usb_find_interface(&launcher_driver, subminor);
        if (!interface) {
                pr_err("can't find device for minor %d", subminor);
                retval = -ENODEV;
                goto exit;
        }

        dev = usb_get_intfdata(interface);
        if (!dev) {
                retval = -ENODEV;
                goto exit;
        }

        /* lock this device */
        if (down_interruptible(&dev->sem)) {
                pr_err("sem down failed");
                retval = -ERESTARTSYS;
                goto exit;
        }
```

```
        /* Increment our usage count for the device. */
        ++dev->open_count;
        if (dev->open_count > 1) {
                pr_info("open_count = %d", dev->open_count);
        }

        /* Initialize interrupt URB. */
        usb_fill_int_urb(dev->int_in_urb, dev->udev,
                         usb_rcvintpipe(dev->udev, dev->int_in_endpoint->bEndpointAddress),
                         dev->int_in_buffer,
                         le16_to_cpu(dev->int_in_endpoint->wMaxPacketSize),
                         launcher_int_in_callback,
                         dev,
                         dev->int_in_endpoint->bInterval);

        dev->int_in_running = 1;
        mb();

        retval = usb_submit_urb(dev->int_in_urb, GFP_KERNEL);
        if (retval) {
                pr_err("submitting int urb failed (%d)", retval);
                dev->int_in_running = 0;
                --dev->open_count;
                goto unlock_exit;
        }

        /* Save our object in the file's private structure. */
        filp->private_data = dev;

unlock_exit:
        up(&dev->sem);

exit:
        mutex_unlock(&disconnect_mutex);
        return retval;
}

static int launcher_close(struct inode *inodep, struct file *filp)
{       struct usb_ml *dev = NULL;
        int retval = 0;

        pr_debug("launcher_close\n");
        dev = filp->private_data;

        if (! dev) {
                pr_err("dev is NULL");
                retval =  -ENODEV;
                goto exit;
        }
```

```
        /* Lock our device */
        if (down_interruptible(&dev->sem)) {
                retval = -ERESTARTSYS;
                goto exit;
        }

        if (dev->open_count <= 0) {
                pr_err("device not opened");
                retval = -ENODEV;
                goto unlock_exit;
        }

        if (! dev->udev) {
                pr_warn("device unplugged before the file was released");
                up (&dev->sem);         /* Unlock here as LAUNCHER_delete frees dev. */
                launcher_delete(dev);
                goto exit;
        }

        if (dev->open_count > 1) {
                pr_info("open_count = %d", dev->open_count);
        }

        launcher_abort_transfers(dev);
        --dev->open_count;

unlock_exit:
        up(&dev->sem);

exit:
        return retval;
}

static ssize_t launcher_read(struct file *f, char __user *buf, size_t cnt, loff_t *off)
{
        int retval = -EFAULT;
        pr_debug("launcher_read\n");
        return retval;
}

static ssize_t launcher_write(struct file *filp, const char __user *user_buf,
                              size_t count, loff_t *off)
{
        int retval = -EFAULT;
        struct usb_ml *dev;
        unsigned char buf[8];
        unsigned char cmd = LAUNCHER_STOP;

        pr_debug("launcher_write\n");
        dev = filp->private_data;
```

```
/* Lock this object. */
if (down_interruptible(&dev->sem)) {
        retval = -ERESTARTSYS;
        goto exit;
}

/* Verify that the device wasn't unplugged. */
if (! dev->udev) {
        retval = -ENODEV;
        pr_err("No device or device unplugged (%d)", retval);
        goto unlock_exit;
}

/* Verify that we actually have some data to write. */
if (count == 0) {
        goto unlock_exit;
}

/* We only accept one-byte writes. */
if (count != 1) {
        count = 1;
}

if (copy_from_user(&cmd, user_buf, count)) {
        retval = -EFAULT;
        goto unlock_exit;
}

pr_info("Received command 0x%x\n", cmd);

/* TODO: Check the range of the commands allowed - otherwise we're
 *        trusting the user not to be silly
 */

memset(&buf, 0, sizeof(buf));
buf[0] = LAUNCHER_CTRL_COMMAND_PREFIX;
buf[1] = cmd;

/* The interrupt-in-endpoint handler also modifies dev->command. */
spin_lock(&dev->cmd_spinlock);
dev->command = cmd;
spin_unlock(&dev->cmd_spinlock);

pr_debug("Sending usb_control_message()\n");
retval = usb_control_msg(dev->udev,
                usb_sndctrlpipe(dev->udev, 0),
                LAUNCHER_CTRL_REQUEST,
                LAUNCHER_CTRL_REQUEST_TYPE,
                LAUNCHER_CTRL_VALUE,
                LAUNCHER_CTRL_INDEX,
                &buf,
```

```
                        sizeof(buf),
                        HZ*5);


        if (retval < 0) {
                pr_err("usb_control_msg failed (%d)", retval);
                goto unlock_exit;
        }

        /* We've only written one byte hopefully! */
        retval = count;

unlock_exit:
        up(&dev->sem);

exit:
        return retval;
}

static struct file_operations fops =
{
        .open = launcher_open,
        .release = launcher_close,
        .read = launcher_read,
        .write = launcher_write,
};

static int launcher_probe(struct usb_interface *interface, const struct usb_device_id *id)
{
        struct usb_device *udev = interface_to_usbdev(interface);
        struct usb_ml *dev = NULL;
        struct usb_host_interface *iface_desc;
        struct usb_endpoint_descriptor *endpoint;
        int i, int_end_size;
        int retval = -ENODEV;

        pr_debug("launcher_probe\n");

        /* Set up our class */
        class.name = LAUNCHER_NODE"%d";
        class.fops = &fops;

        if ((retval = usb_register_dev(interface, &class)) < 0) {
                /* Something stopped us from registering this driver */
                pr_err("Not able to get a minor for this device.");
                goto exit;
        } else {
                pr_info("Minor receieved - %d\n", interface->minor);
        }

        if (!udev) {
```

```
        /* Something has gone bad */
        pr_err("udev is NULL");
        goto exit;
}

dev = kzalloc(sizeof(struct usb_ml), GFP_KERNEL);
if (!dev) {
        pr_err("cannot allocate memory for struct usb_ml");
        retval = -ENOMEM;
        goto exit;
}

dev->command = LAUNCHER_STOP;

sema_init(&dev->sem, 1);
spin_lock_init(&dev->cmd_spinlock);

dev->udev = udev;
dev->interface = interface;
iface_desc = interface->cur_altsetting;

/* Set up interrupt endpoint information. */
for (i = 0; i < iface_desc->desc.bNumEndpoints; ++i) {
        endpoint = &iface_desc->endpoint[i].desc;

        if (((endpoint->bEndpointAddress & USB_ENDPOINT_DIR_MASK) == USB_DIR_IN)
                        && ((endpoint->bmAttributes & USB_ENDPOINT_XFERTYPE_MASK) ==
                                USB_ENDPOINT_XFER_INT))
                dev->int_in_endpoint = endpoint;

}

if (!dev->int_in_endpoint) {
        pr_err("could not find interrupt in endpoint");
        goto error;
}

int_end_size = le16_to_cpu(dev->int_in_endpoint->wMaxPacketSize);

dev->int_in_buffer = kmalloc(int_end_size, GFP_KERNEL);
if (!dev->int_in_buffer) {
        pr_err("could not allocate int_in_buffer");
        retval = -ENOMEM;
        goto error;
}

dev->int_in_urb = usb_alloc_urb(0, GFP_KERNEL);
if (!dev->int_in_urb) {
        pr_err("could not allocate int_in_urb");
        retval = -ENOMEM;
        goto error;
```

```
        }

        /* Set up the control URB. */
        dev->ctrl_urb = usb_alloc_urb(0, GFP_KERNEL);
        if (!dev->ctrl_urb) {
                pr_err("could not allocate ctrl_urb");
                retval = -ENOMEM;
                goto error;
        }

        dev->ctrl_buffer = kzalloc(LAUNCHER_CTRL_BUFFER_SIZE, GFP_KERNEL);
        if (!dev->ctrl_buffer) {
                pr_err("could not allocate ctrl_buffer");
                retval = -ENOMEM;
                goto error;
        }

        dev->ctrl_dr = kmalloc(sizeof(struct usb_ctrlrequest), GFP_KERNEL);
        if (!dev->ctrl_dr) {
                pr_err("could not allocate usb_ctrlrequest");
                retval = -ENOMEM;
                goto error;
        }
        dev->ctrl_dr->bRequestType = LAUNCHER_CTRL_REQUEST_TYPE;
        dev->ctrl_dr->bRequest = LAUNCHER_CTRL_REQUEST;
        dev->ctrl_dr->wValue = cpu_to_le16(LAUNCHER_CTRL_VALUE);
        dev->ctrl_dr->wIndex = cpu_to_le16(LAUNCHER_CTRL_INDEX);
        dev->ctrl_dr->wLength = cpu_to_le16(LAUNCHER_CTRL_BUFFER_SIZE);

        usb_fill_control_urb(dev->ctrl_urb, dev->udev,
                        usb_sndctrlpipe(dev->udev, 0),
                        (unsigned char *)dev->ctrl_dr,
                        dev->ctrl_buffer,
                        LAUNCHER_CTRL_BUFFER_SIZE,
                        launcher_ctrl_callback,
                        dev);

        /* Retrieve a serial. */
        if (!usb_string(udev, udev->descriptor.iSerialNumber, dev->serial_number,
                                sizeof(dev->serial_number))) {
                pr_err("could not retrieve serial number");
                goto error;
        }

        /* Save our data pointer in this interface device. */
        usb_set_intfdata(interface, dev);

        dev->minor = interface->minor;

exit:
        return retval;
```

```
error:
        launcher_delete(dev);
        return retval;
}

static void launcher_disconnect(struct usb_interface *interface)
{
        struct usb_ml *dev;
        int minor;

        pr_debug("launcher_disconnect\n");
        mutex_lock(&disconnect_mutex);        /* Not interruptible */

        dev = usb_get_intfdata(interface);
        usb_set_intfdata(interface, NULL);

        down(&dev->sem); /* Not interruptible */

        minor = dev->minor;

        /* Give back our minor. */
        usb_deregister_dev(interface, &class);

        /* If the device is not opened, then we clean up right now. */
        if (! dev->open_count) {
                up(&dev->sem);
                launcher_delete(dev);
        } else {
                dev->udev = NULL;
                up(&dev->sem);
        }

        mutex_unlock(&disconnect_mutex);

}

static int __init launcher_init(void)
{
        int result;

        pr_debug("launcher_init\n");

        /* Wire up our probe/disconnect */
        launcher_driver.probe = launcher_probe;
        launcher_driver.disconnect = launcher_disconnect;

        /* Register this driver with the USB subsystem */
        if ((result = usb_register(&launcher_driver))) {
                pr_err("usb_register() failed. Error number %d", result);
        }
```

```
        return result;
}

static void __exit launcher_exit(void)
{
        pr_debug("launcher_exit\n");
        /* Deregister this driver with the USB subsystem */
        usb_deregister(&launcher_driver);
}

module_init(launcher_init);
module_exit(launcher_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Nick Glynn <Nick.Glynn@feabhas.com>");
MODULE_DESCRIPTION("USB Launcher Driver");
```

5 Controller Code

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define LAUNCHER_NODE           "/dev/launcher0"
#define LAUNCHER_FIRE           0x10
#define LAUNCHER_STOP           0x20
#define LAUNCHER_UP             0x02
#define LAUNCHER_DOWN           0x01
#define LAUNCHER_LEFT           0x04
#define LAUNCHER_RIGHT          0x08
#define LAUNCHER_UP_LEFT        (LAUNCHER_UP | LAUNCHER_LEFT)
#define LAUNCHER_DOWN_LEFT      (LAUNCHER_DOWN | LAUNCHER_LEFT)
#define LAUNCHER_UP_RIGHT       (LAUNCHER_UP | LAUNCHER_RIGHT)
#define LAUNCHER_DOWN_RIGHT     (LAUNCHER_DOWN | LAUNCHER_RIGHT)

#define SEC 1000

static void launcher_cmd(int fd, int cmd)
{
        int retval = 0;

        retval = write(fd, &cmd, 1);
        if (retval < 0) {
                fprintf(stderr, "Could not send command to " LAUNCHER_NODE
                        " (error %d)\n", retval);
        } else if (LAUNCHER_FIRE == cmd) {
                usleep(5000000);
        }
}

int main(int argc, char **argv)
{
        char c;
        int fd;
        int cmd = LAUNCHER_STOP;
        char *dev = LAUNCHER_NODE;
        unsigned int duration = 50;

        fd = open(dev, O_RDWR);
        if (fd == -1) {
                perror("Couldn't open file: %m");
                exit(1);
        }

        system("/bin/stty raw");
        while ((c = getchar()) != 'q') {
                switch (c) {
```

```
                case 'a':
                        cmd = LAUNCHER_LEFT;
                        break;
                case 'd':
                        cmd = LAUNCHER_RIGHT;
                        break;
                case 'w':
                        cmd = LAUNCHER_UP;
                        break;
                case 's':
                        cmd = LAUNCHER_DOWN;
                        break;
                case ' ':
                        cmd = LAUNCHER_FIRE;
                        break;
                case 'e':
                        cmd = LAUNCHER_STOP;
                        break;
                }
                launcher_cmd(fd, cmd);
                usleep(duration *SEC);
                launcher_cmd(fd, LAUNCHER_STOP);
        }

        close(fd);
        system("/bin/stty cooked");
        return EXIT_SUCCESS;
}
```