vinput.c vinput.h Makefile

vinput.c

```c
/*
 * vinput.c
 */

#include <linux/cdev.h>
#include <linux/input.h>
#include <linux/module.h>
#include <linux/slab.h>
#include <linux/spinlock.h>

#include <asm/uaccess.h>

#include "vinput.h"

#define DRIVER_NAME "vinput"

#define dev_to_vinput(dev) container_of(dev, struct vinput, dev)

static DECLARE_BITMAP(vinput_ids, VINPUT_MINORS);

static LIST_HEAD(vinput_devices);
static LIST_HEAD(vinput_vdevices);

static int vinput_dev;
static struct spinlock vinput_lock;
static struct class vinput_class;

/* Search the name of vinput device in the vinput_devices linked list,
 * which added at vinput_register().
 */
static struct vinput_device *vinput_get_device_by_type(const char *type)
{
    int found = 0;
    struct vinput_device *vinput;
    struct list_head *curr;

    spin_lock(&vinput_lock);
    list_for_each(curr, &vinput_devices)
    {
        vinput = list_entry(curr, struct vinput_device, list);
        if (vinput && strncmp(type, vinput->name, strlen(vinput->name)) == 0)
        {
            found = 1;
            break;
        }
    }
    spin_unlock(&vinput_lock);
```

```
    if (found)
        return vinput;
    return ERR_PTR(-ENODEV);
}

/* Search the id of virtual device in the vinput_vdevices linked list,
 * which added at vinput_alloc_vdevice().
 */
static struct vinput *vinput_get_vdevice_by_id(long id)
{
    struct vinput *vinput = NULL;
    struct list_head *curr;

    spin_lock(&vinput_lock);
    list_for_each(curr, &vinput_vdevices)
    {
        vinput = list_entry(curr, struct vinput, list);
        if (vinput && vinput->id == id)
            break;
    }
    spin_unlock(&vinput_lock);

    if (vinput && vinput->id == id)
        return vinput;
    return ERR_PTR(-ENODEV);
}

static int vinput_open(struct inode *inode, struct file *file)
{
    int err = 0;
    struct vinput *vinput = NULL;

    vinput = vinput_get_vdevice_by_id(iminor(inode));

    if (IS_ERR(vinput))
        err = PTR_ERR(vinput);
    else
        file->private_data = vinput;

    return err;
}

static int vinput_release(struct inode *inode, struct file *file)
{
    return 0;
}

static ssize_t vinput_read(struct file *file, char __user *buffer, size_t
count, loff_t *offset)
{
```

```
    int len;
    char buff[VINPUT_MAX_LEN + 1];
    struct vinput *vinput = file->private_data;

    len = vinput->type->ops->read(vinput, buff, count);

    if (*offset > len)
        count = 0;
    else if (count + *offset > VINPUT_MAX_LEN)
        count = len - *offset;

    if (raw_copy_to_user(buffer, buff + *offset, count))
        count = -EFAULT;

    *offset += count;
    return count;
}

static ssize_t vinput_write(struct file *file, const char __user *buffer,
size_t count, loff_t *offset)
{
    char buff[VINPUT_MAX_LEN + 1];
    struct vinput *vinput = file->private_data;

    memset(buff, 0, sizeof(char) * (VINPUT_MAX_LEN + 1));

    if (count > VINPUT_MAX_LEN)
    {
        dev_warn(&vinput->dev, "Too long. %d bytes allowed\n",
VINPUT_MAX_LEN);
        return -EINVAL;
    }

    if (raw_copy_from_user(buff, buffer, count))
        return -EFAULT;

    return vinput->type->ops->send(vinput, buff, count);
}

static const struct file_operations vinput_fops = {
    .owner = THIS_MODULE,
    .open = vinput_open,
    .release = vinput_release,
    .read = vinput_read,
    .write = vinput_write,
};

static void vinput_unregister_vdevice(struct vinput *vinput)
{
    input_unregister_device(vinput->input);
    if (vinput->type->ops->kill)
```

```
        vinput->type->ops->kill(vinput);
}

static void vinput_destroy_vdevice(struct vinput *vinput)
{
    /* Remove from the list first */
    spin_lock(&vinput_lock);
    list_del(&vinput->list);
    clear_bit(vinput->id, vinput_ids);
    spin_unlock(&vinput_lock);

    module_put(THIS_MODULE);

    kfree(vinput);
}

static void vinput_release_dev(struct device *dev)
{
    struct vinput *vinput = dev_to_vinput(dev);
    int id = vinput->id;

    vinput_destroy_vdevice(vinput);

    pr_debug("released vinput%d.\n", id);
}

static struct vinput *vinput_alloc_vdevice(void)
{
    int err;
    struct vinput *vinput = kzalloc(sizeof(struct vinput), GFP_KERNEL);

    try_module_get(THIS_MODULE);

    memset(vinput, 0, sizeof(struct vinput));

    spin_lock_init(&vinput->lock);

    spin_lock(&vinput_lock);
    vinput->id = find_first_zero_bit(vinput_ids, VINPUT_MINORS);
    if (vinput->id >= VINPUT_MINORS)
    {
        err = -ENOBUFS;
        goto fail_id;
    }
    set_bit(vinput->id, vinput_ids);
    list_add(&vinput->list, &vinput_vdevices);
    spin_unlock(&vinput_lock);

    /* allocate the input device */
    vinput->input = input_allocate_device();
    if (vinput->input == NULL)
```

```
    {
        pr_err("vinput: Cannot allocate vinput input device\n");
        err = -ENOMEM;
        goto fail_input_dev;
    }

    /* initialize device */
    vinput->dev.class = &vinput_class;
    vinput->dev.release = vinput_release_dev;
    vinput->dev.devt = MKDEV(vinput_dev, vinput->id);
    dev_set_name(&vinput->dev, DRIVER_NAME "%lu", vinput->id);

    return vinput;

fail_input_dev:
    spin_lock(&vinput_lock);
    list_del(&vinput->list);
fail_id:
    spin_unlock(&vinput_lock);
    module_put(THIS_MODULE);
    kfree(vinput);

    return ERR_PTR(err);
}

static int vinput_register_vdevice(struct vinput *vinput)
{
    int err = 0;

    /* register the input device */
    vinput->input->name = vinput->type->name;
    vinput->input->phys = "vinput";
    vinput->input->dev.parent = &vinput->dev;

    vinput->input->id.bustype = BUS_VIRTUAL;
    vinput->input->id.product = 0x0000;
    vinput->input->id.vendor = 0x0000;
    vinput->input->id.version = 0x0000;

    err = vinput->type->ops->init(vinput);

    if (err == 0)
        dev_info(&vinput->dev, "Registered virtual input %s %ld\n",
vinput->type->name, vinput->id);
    return err;
}

static ssize_t export_store(struct class *class, struct class_attribute *attr,
const char *buf, size_t len)
{
    int err;
```

```
    struct vinput *vinput;
    struct vinput_device *device;

    device = vinput_get_device_by_type(buf);
    if (IS_ERR(device))
    {
        pr_info("vinput: This virtual device isn't registered\n");
        err = PTR_ERR(device);
        goto fail;
    }

    vinput = vinput_alloc_vdevice();
    if (IS_ERR(vinput))
    {
        err = PTR_ERR(vinput);
        goto fail;
    }

    vinput->type = device;
    err = device_register(&vinput->dev);
    if (err < 0)
        goto fail_register;

    err = vinput_register_vdevice(vinput);
    if (err < 0)
        goto fail_register_vinput;

    return len;

fail_register_vinput:
    device_unregister(&vinput->dev);
fail_register:
    vinput_destroy_vdevice(vinput);
fail:
    return err;
}
/* This macro generates class_attr_export structure and export_store() */
static CLASS_ATTR_WO(export);

static ssize_t unexport_store(struct class *class, struct class_attribute
*attr, const char *buf, size_t len)
{
    int err;
    unsigned long id;
    struct vinput *vinput;

    err = kstrtol(buf, 10, &id);
    if (err)
    {
        err = -EINVAL;
        goto failed;
```

```
    }

    vinput = vinput_get_vdevice_by_id(id);
    if (IS_ERR(vinput))
    {
        pr_err("vinput: No such vinput device %ld\n", id);
        err = PTR_ERR(vinput);
        goto failed;
    }

    vinput_unregister_vdevice(vinput);
    device_unregister(&vinput->dev);

    return len;
failed:
    return err;
}
/* This macro generates class_attr_unexport structure and unexport_store() */
static CLASS_ATTR_WO(unexport);

static struct attribute *vinput_class_attrs[] = {
    &class_attr_export.attr,
    &class_attr_unexport.attr,
    NULL,
};

/* This macro generates vinput_class_groups structure */
ATTRIBUTE_GROUPS(vinput_class);

static struct class vinput_class = {
    .name = "vinput",
    .owner = THIS_MODULE,
    .class_groups = vinput_class_groups,
};

int vinput_register(struct vinput_device *dev)
{
    spin_lock(&vinput_lock);
    list_add(&dev->list, &vinput_devices);
    spin_unlock(&vinput_lock);

    pr_info("vinput: registered new virtual input device '%s'\n", dev->name);

    return 0;
}
EXPORT_SYMBOL(vinput_register);

void vinput_unregister(struct vinput_device *dev)
{
    struct list_head *curr, *next;
```

```
    /* Remove from the list first */
    spin_lock(&vinput_lock);
    list_del(&dev->list);
    spin_unlock(&vinput_lock);

    /* unregister all devices of this type */
    list_for_each_safe(curr, next, &vinput_vdevices)
    {
        struct vinput *vinput = list_entry(curr, struct vinput, list);
        if (vinput && vinput->type == dev)
        {
            vinput_unregister_vdevice(vinput);
            device_unregister(&vinput->dev);
        }
    }

    pr_info("vinput: unregistered virtual input device '%s'\n", dev->name);
}
EXPORT_SYMBOL(vinput_unregister);

static int __init vinput_init(void)
{
    int err = 0;

    pr_info("vinput: Loading virtual input driver\n");

    vinput_dev = register_chrdev(0, DRIVER_NAME, &vinput_fops);
    if (vinput_dev < 0)
    {
        pr_err("vinput: Unable to allocate char dev region\n");
        goto failed_alloc;
    }

    spin_lock_init(&vinput_lock);

    err = class_register(&vinput_class);
    if (err < 0)
    {
        pr_err("vinput: Unable to register vinput class\n");
        goto failed_class;
    }

    return 0;
failed_class:
    class_unregister(&vinput_class);
failed_alloc:
    return err;
}

static void __exit vinput_end(void)
{
```

```
    pr_info("vinput: Unloading virtual input driver\n");

    unregister_chrdev(vinput_dev, DRIVER_NAME);
    class_unregister(&vinput_class);
}

module_init(vinput_init);
module_exit(vinput_end);

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Emulate input events");
```

vinput.h

```
/*
 * vinput.h
 */

#ifndef VINPUT_H
#define VINPUT_H

#include <linux/input.h>
#include <linux/spinlock.h>

#define VINPUT_MAX_LEN 128
#define MAX_VINPUT 32
#define VINPUT_MINORS MAX_VINPUT

#define dev_to_vinput(dev) container_of(dev, struct vinput, dev)

struct vinput_device;

struct vinput
{
    long id;
    long devno;
    long last_entry;
    spinlock_t lock;

    void *priv_data;

    struct device dev;
    struct list_head list;
    struct input_dev *input;
    struct vinput_device *type;
};

struct vinput_ops
{
    int (*init)(struct vinput *);
    int (*kill)(struct vinput *);
    int (*send)(struct vinput *, char *, int);
```

```
    int (*read)(struct vinput *, char *, int);
};

struct vinput_device
{
    char name[16];
    struct list_head list;
    struct vinput_ops *ops;
};

int vinput_register(struct vinput_device *dev);
void vinput_unregister(struct vinput_device *dev);

#endif
```

Makefile

```
obj-m += vinput.o

PWD := $(CURDIR)

all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Standard output

```
hougreat@republicofhoul:~/AdvancedOperatingSystem/11-28-hw$ echo "+34" | sudo tee /dev/vin
put0
+34
hougreat@republicofhoul:~/AdvancedOperatingSystem/11-28-hw$ echo "-34" | sudo tee /dev/vin
put0
-34
```

Kernel log message

```
hougreat@republicofhoul:~/AdvancedOperatingSystem/11-28-hw$ sudo dmesg | tail -1
[1952777.460501] vinput: Loading virtual input driver
```