vinput.c vinput.h Makefile

vinput.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)

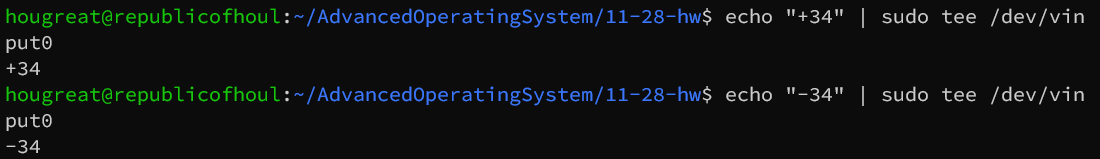
all:

    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



Kernel log message

