

Driver for usb remote control (pid 13ec, vid 000a)

Source: <http://linux.derkeiler.com/Mailing-Lists/Kernel/2006-10/msg05236.html>

- *From:* "Marc-Antoine Avon Charreyron" <marcantoine@xxxxxxxxxx>
 - *Date:* Sun, 15 Oct 2006 12:51:24 -0400
-

Hello, the following is a driver for a usb remote control with product id 13ec and vendor id 000a (it came with a Asus tv card). I'm looking for comments and people to help test it.

Thank you

P.S. please CC me personally

```
/*
 * USB remote control driver
 *
 * 2006 Marc-Antoine Avon-Charreyron <mavon006@xxxxxxxxxx>
 *
 * This driver is based on drivers/usb/usb-skeleton.c
 */

/*
 * 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.
 */

#include <linux/config.h>
#include <linux/kernel.h>
#include <linux/errno.h>
#include <linux/init.h>
#include <linux/slab.h>
#include <linux/module.h>
#include <linux/kref.h>
#include <asm/uaccess.h>
#include <linux/usb.h>

/* Define these values to match your devices */
#define USB_REMCTL_VENDOR_ID 0x13ec
#define USB_REMCTL_PRODUCT_ID 0x000a
```

Driver for usb remote control (pid 13ec, vid 000a)

```
/* table of devices that work with this driver */
static struct usb_device_id rem_ctl_table [] = {
{ USB_DEVICE(USB_REMCTL_VENDOR_ID, USB_REMCTL_PRODUCT_ID) },
{ } /* Terminating entry */
};
MODULE_DEVICE_TABLE (usb, rem_ctl_table);

/* Get a minor range for your devices from the usb maintainer */
#define USB_REMCTL_MINOR_BASE 192

struct semaphore sem;

/* Structure to hold all of our device specific stuff */
struct usb_rem_ctl {
struct usb_device * udev; /* the usb device for this device */
struct usb_interface * interface; /* the interface for this device */
unsigned char * data_in_buffer1; /* the buffer to receive data from
endpoint 0x81*/
unsigned char * data_in_buffer2; /* the buffer to receive
data from endpoint 0x82*/
size_t data_in_size; /* the size of the receive buffer */
struct kref kref;
};
#define to_rem_ctl_dev(d) container_of(d, struct usb_rem_ctl, kref)

static struct usb_driver rem_ctl_driver;

static void rem_ctl_delete(struct kref *kref)
{
struct usb_rem_ctl *dev = to_rem_ctl_dev(kref);

usb_put_dev(dev->udev);
kfree (dev->data_in_buffer1);
kfree (dev->data_in_buffer2);
kfree (dev);
}

static void cb_read0(struct urb *urb, struct pt_regs *regs)
{
usb_submit_urb(urb, GFP_ATOMIC);
up(&sem);
}

static void cb_read1(struct urb *urb, struct pt_regs *regs)
{
usb_submit_urb(urb, GFP_ATOMIC);
}

static int rem_ctl_open(struct inode *inode, struct file *file)
{
struct usb_rem_ctl *dev;
```

Driver for usb remote control (pid 13ec, vid 000a)

```
struct usb_interface *interface;
int subminor;
int retval = 0;

subminor = iminor(inode);

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

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

/* increment our usage count for the device */
kref_get(&dev->kref);

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

struct urb *urb;
struct urb *urb2;
urb = usb_alloc_urb(0, GFP_KERNEL);
if (!urb) {
return -ENOMEM;
}
urb2 = usb_alloc_urb(0, GFP_KERNEL);
if (!urb2) {
return -ENOMEM;
}

usb_fill_int_urb(urb, dev->udev, usb_rcvintpipe(dev->udev, 0x81),
dev->data_in_buffer1,
dev->data_in_size, cb_read0, dev, 128);
usb_fill_int_urb(urb2, dev->udev, usb_rcvintpipe(dev->udev, 0x82),
dev->data_in_buffer2, dev->data_in_size, cb_read1, dev, 128);

retval = usb_submit_urb(urb, GFP_KERNEL);
retval = usb_submit_urb(urb2, GFP_KERNEL);
sema_init(&sem, 0);

exit:
return retval;
}
```

Driver for usb remote control (pid 13ec, vid 000a)

```
static int rem_ctl_release(struct inode *inode, struct file *file)
{
    struct usb_rem_ctl *dev;

    dev = (struct usb_rem_ctl *)file->private_data;
    if (dev == NULL)
        return -ENODEV;

    /* decrement the count on our device */
    kref_put(&dev->kref, rem_ctl_delete);
    return 0;
}

static ssize_t read(struct file *file, char *buffer, size_t count, loff_t *ppos)
{
    struct usb_rem_ctl *dev;
    int retval = 0;

    char buf[1]="0";

    dev = (struct usb_rem_ctl *)file->private_data;

    down_interruptible(&sem);

    if((dev->data_in_buffer1[1] == 0x07) &&
        (dev->data_in_buffer1[3] == 0x1f)) { //TV

        buf[0] = 0x01;
        retval = 1;

    } else if((dev->data_in_buffer1[1] == 0x07) &&
        (dev->data_in_buffer1[3] == 0x23)) { //FM

        buf[0] = 0x02;
        retval = 1;

    } else if((dev->data_in_buffer1[1] == 0x07) &&
        (dev->data_in_buffer1[3] == 0x1e)) { //DVD

        buf[0] = 0x03;
        retval = 1;

    } else if((dev->data_in_buffer1[1] == 0x04) &&
        //Close
        (dev->data_in_buffer1[3] == 0x3d)) {

        buf[0] = 0x04;
        retval = 1;

    }
}
```

Driver for usb remote control (pid 13ec, vid 000a)

```
} else if((dev->data_in_buffer1[1] == 0x03) && //<<
(dev->data_in_buffer1[3] == 0x05)) {
```

```
buf[0] = 0x05;
retval = 1;
```

```
} else if((dev->data_in_buffer1[1] == 0x01) &&
//Stop
(dev->data_in_buffer1[3] == 0x16)) {
```

```
buf[0] = 0x06;
retval = 1;
```

```
} else if((dev->data_in_buffer1[1] == 0x01) &&
//Play/Pause
(dev->data_in_buffer1[3] == 0x13)) {
```

```
buf[0] = 0x07;
retval = 1;
```

```
} else if((dev->data_in_buffer1[1] == 0x03) && //>>
(dev->data_in_buffer1[3] == 0x09)) {
```

```
buf[0] = 0x08;
retval = 1;
```

```
} else if((dev->data_in_buffer1[1] == 0x07) &&
//don't know
(dev->data_in_buffer1[3] == 0x18)) {
```

```
buf[0] = 0x09;
retval = 1;
```

```
} else if((dev->data_in_buffer1[1] == 0x01) &&
//record
(dev->data_in_buffer1[3] == 0x15)) {
```

```
buf[0] = 0x0a;
retval = 1;
```

```
} else if((dev->data_in_buffer1[1] == 0x00) && //pip
(dev->data_in_buffer1[3] == 0x1d)) {
```

```
buf[0] = 0x0b;
retval = 1;
```

```
} else if((dev->data_in_buffer1[1] == 0x00) && //1
(dev->data_in_buffer1[3] == 0x1e)) {
```

```
buf[0] = 0x0c;
retval = 1;
```

Driver for usb remote control (pid 13ec, vid 000a)

Driver for usb remote control (pid 13ec, vid 000a)

```
} else if((dev->data_in_buffer1[1] == 0x00) && //2
(dev->data_in_buffer1[3] == 0x1f)) {

buf[0] = 0x0d;
retval = 1;

} else if((dev->data_in_buffer1[1] == 0x00) && //3
(dev->data_in_buffer1[3] == 0x20)) {

buf[0] = 0x0e;
retval = 1;

} else if((dev->data_in_buffer1[1] == 0x01) &&
//ch up
(dev->data_in_buffer1[3] == 0x09)) {

buf[0] = 0x0f;
retval = 1;

} else if((dev->data_in_buffer1[1] == 0x00) && //4
(dev->data_in_buffer1[3] == 0x21)) {

buf[0] = 0x10;
retval = 1;

} else if((dev->data_in_buffer1[1] == 0x00) && //5
(dev->data_in_buffer1[3] == 0x22)) {

buf[0] = 0x11;
retval = 1;

} else if((dev->data_in_buffer1[1] == 0x00) && //6
(dev->data_in_buffer1[3] == 0x23)) {

buf[0] = 0x12;
retval = 1;

} else if((dev->data_in_buffer1[1] == 0x01) &&
//ch down
(dev->data_in_buffer1[3] == 0x05)) {

buf[0] = 0x13;
retval = 1;

} else if((dev->data_in_buffer1[1] == 0x00) && //7
(dev->data_in_buffer1[3] == 0x24)) {

buf[0] = 0x14;
retval = 1;
```

Driver for usb remote control (pid 13ec, vid 000a)

```
} else if((dev->data_in_buffer1[1] == 0x00) && //8  
(dev->data_in_buffer1[3] == 0x25)) {
```

```
buf[0] = 0x15;  
retval = 1;
```

```
} else if((dev->data_in_buffer1[1] == 0x00) && //9  
(dev->data_in_buffer1[3] == 0x26)) {
```

```
buf[0] = 0x16;  
retval = 1;
```

```
} else if((dev->data_in_buffer1[1] == 0x00) &&  
//Back  
(dev->data_in_buffer1[3] == 0x2a)) {
```

```
buf[0] = 0x17;  
retval = 1;
```

```
} else if((dev->data_in_buffer1[1] == 0x00) && //0  
(dev->data_in_buffer1[3] == 0x27)) {
```

```
buf[0] = 0x18;  
retval = 1;
```

```
} else if((dev->data_in_buffer1[1] == 0x00) && //up  
(dev->data_in_buffer1[3] == 0x52)) {
```

```
buf[0] = 0x1c;  
retval = 1;
```

```
} else if((dev->data_in_buffer1[1] == 0x00) &&  
//right  
(dev->data_in_buffer1[3] == 0x4f)) {
```

```
buf[0] = 0x1d;  
retval = 1;
```

```
} else if((dev->data_in_buffer1[1] == 0x00) &&  
//down  
(dev->data_in_buffer1[3] == 0x51)) {
```

```
buf[0] = 0x1e;  
retval = 1;
```

```
} else if((dev->data_in_buffer1[1] == 0x00) &&  
//left  
(dev->data_in_buffer1[3] == 0x50)) {
```

```
buf[0] = 0x1f;  
retval = 1;
```

Driver for usb remote control (pid 13ec, vid 000a)

Driver for usb remote control (pid 13ec, vid 000a)

```
} else if((dev->data_in_buffer1[1] == 0x00) &&
//enter
(dev->data_in_buffer1[3] == 0x28)) {

buf[0] = 0x20;
retval = 1;

} else if((dev->data_in_buffer1[1] == 0x00) &&
//Volumes
(dev->data_in_buffer1[3] == 0x00)) {

if(dev->data_in_buffer2[1] == 0x04) { //Vol Down
buf[0] = 0x19;
retval = 1;
} else if(dev->data_in_buffer2[1] == 0x02) { //Vol Up
buf[0] = 0x1a;
retval = 1;
} else if(dev->data_in_buffer2[1] == 0x01) { //Mute
buf[0] = 0x1b;
retval = 1;
}

} else {
return 0;
}

if (copy_to_user(buffer, buf, sizeof(buffer))) {
retval = -EFAULT;
}

return retval;
}

static struct file_operations rem_ctl_fops = {
.owner = THIS_MODULE,
.read = read,
.open = rem_ctl_open,
.release = rem_ctl_release,
};

/*
 * usb class driver info in order to get a minor number from the usb core,
 * and to have the device registered with devfs and the driver core
 */
static struct usb_class_driver rem_ctl_class = {
.name = "usb/remote%d",
.fops = &rem_ctl_fops,
.minor_base = USB_REMCTL_MINOR_BASE,
};
```

Driver for usb remote control (pid 13ec, vid 000a)

```
static int rem_ctl_probe(struct usb_interface *interface, const struct
usb_device_id *id)
{
    struct usb_rem_ctl *dev = NULL;
    struct usb_host_interface *iface_desc;
    struct usb_endpoint_descriptor *endpoint;
    size_t buffer_size;
    int retval = -ENOMEM;

    /* allocate memory for our device state and initialize it */
    dev = kmalloc(sizeof(*dev), GFP_KERNEL);
    if (dev == NULL) {
        err("Out of memory");
        goto error;
    }
    memset(dev, 0x00, sizeof(*dev));
    kref_init(&dev->kref);

    dev->udev = usb_get_dev(interface_to_usbdev(interface));
    dev->interface = interface;

    /* set up the endpoint information */
    iface_desc = interface->cur_altsetting;
    endpoint = &iface_desc->endpoint[0].desc;

    buffer_size = le16_to_cpu(endpoint->wMaxPacketSize);
    dev->data_in_size = buffer_size;
    dev->data_in_buffer1 = kmalloc(buffer_size, GFP_KERNEL);
    if (!dev->data_in_buffer1) {
        err("Could not allocate data_in_buffer1");
        goto error;
    }
    dev->data_in_buffer2 = kmalloc(buffer_size, GFP_KERNEL);
    if (!dev->data_in_buffer2) {
        err("Could not allocate data_in_buffer2");
        goto error;
    }

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

    /* we can register the device now, as it is ready */
    retval = usb_register_dev(interface, &rem_ctl_class);
    if (retval) {
        /* something prevented us from registering this driver */
        err("Not able to get a minor for this device.");
        usb_set_intfdata(interface, NULL);
        goto error;
    }
}
```

Driver for usb remote control (pid 13ec, vid 000a)

```
/* let the user know what node this device is now attached to */
info("USB remote control device now attached to USBrem-%d", interface->minor);
return 0;

error:
if (dev)
kref_put(&dev->kref, rem_ctl_delete);
return retval;
}

static void rem_ctl_disconnect(struct usb_interface *interface)
{
struct usb_rem_ctl *dev;
int minor = interface->minor;

/* prevent rem_open() from racing rem_disconnect() */
lock_kernel();

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

/* give back our minor */
usb_deregister_dev(interface, &rem_ctl_class);

unlock_kernel();

/* decrement our usage count */
kref_put(&dev->kref, rem_ctl_delete);

info("USB remote #d now disconnected", minor);
}

static struct usb_driver rem_ctl_driver = {
.name = "remote",
.probe = rem_ctl_probe,
.disconnect = rem_ctl_disconnect,
.id_table = rem_ctl_table,
};

static int __init usb_rem_ctl_init(void)
{
int result;

/* register this driver with the USB subsystem */
result = usb_register(&rem_ctl_driver);
if (result)
err("usb_register failed. Error number %d", result);

return result;
}
```

Driver for usb remote control (pid 13ec, vid 000a)

```
static void __exit usb_rem_ctl_exit(void)
{
/* deregister this driver with the USB subsystem */
usb_deregister(&rem_ctl_driver);
}
```

```
module_init (usb_rem_ctl_init);
module_exit (usb_rem_ctl_exit);
```

```
MODULE_LICENSE("GPL");
```

—

To unsubscribe from this list: send the line "unsubscribe linux-kernel" in the body of a message to majordomo@xxxxxxxxxxxxxxxxx
More majordomo info at <http://vger.kernel.org/majordomo-info.html>
Please read the FAQ at <http://www.tux.org/lkml/>