MOUSE DRIVER STUDY

Project Report

Team Members

Submitted to: Dr. R. Karne

Towson University

Objective: To study and understand the functioning of mouse driver for Linux system (Fedora 11).

Linux today supports more hardware devices than any other operating system in the history of the world. It does this using a development model significantly different from the familiar Windows device driver model. The Linux development process continues to evolve to better support the needs of **Independent Hardware Vendors (IHVs)**, distributions, and other members of the community, and the advantages of the Linux model are increasing with time. When IHVs engage with the Linux community, they almost invariably find that the Linux driver model provides significant benefits that lower their costs while producing better drivers. Our objective for this project is to understand the functioning of a Linux mouse driver.

A mouse as a pointing device may be connected using a) serial port b) PS/2 port or c) USB port. For this study we consider only the mice connected using USB ports.

Introduction:

A fundamental purpose for operating systems (OSes) is to serve as an abstraction layer between applications and hardware to enable interoperability. An IHV wants their hardware to be able to make use of all the relevant features of an OS, and an OS wants to take full advantage of the hardware it's running on. Since both the OS and the hardware tend to add and to rearrange features over time, it is a dynamic interaction. What everybody wants is for the hardware to "Just Work" without hassles or support calls.

Today, Linux works with more devices than any other OS in the history of the world. However, this is no comfort when there's no support for the device you need to use. In the past, Windows has benefited from the fact that IHVs expend significantly more resources on their Windows drivers. A review of the strengths and weaknesses of the Windows and Linux driver models helps explain the relative situation today, and provides the context to understand how Linux device support is finally overtaking Windows in both quantity and quality of device support.

a) Windows device driver model:

Microsoft commits to a stable set of Application Binary Interface (ABI) calls that can be used by driver developers. These interfaces represent a set of services that the core OS makes available to device drivers. IHVs test and certify their binary drivers against the released binary OS. Microsoft promises to provide the same binary interfaces over time so that IHVs can rely on those interfaces.

The reasoning behind this approach is to enable driver development to be decoupled from the OS development. The idea is that Microsoft controls their closed source OS and the IHV develops their closed source driver, with the stable ABI being the only interface between the two. This also means that drivers are external, second class citizens compared to the core operating system. Driver developers do not have access or visibility into the core OS beyond this ABI, although a buggy driver can still crash the entire OS.

Binary drivers tend to be buggier, because bugs can generally only be identified and fixed by the IHV, not by the community of system vendors, developers, and users. Another key disadvantage of the closed nature of Windows drivers is that large amounts of important subsystem code is duplicated (in different forms) across different Windows drivers. This duplication is also due to the small number of people who have access to the code: because no one person can see all the code, it is very difficult to factor out and optimize common subsystem code. Another problem with the Windows model is those stable device

drivers ABIs do not actually remain stable. Microsoft has modified the Windows ABI in every Windows release, resulting in a relentless succession of hardware support issues. Any change in the ABI can cause hardware to stop working correctly, and can even crash the entire OS.

b) Linux device driver model:

The Linux model is that IHVs get the source code for their driver accepted into the mainline kernel. This entails a public peer review process to ensure that the driver code is of sufficient quality and does not have obvious bugs or security risks. Linux has neither a stable binary driver ABI nor a stable source-code driver Application Programming Interface (API). That is, there is no guarantee that an interface provided in one version of the kernel will be available in the next version, and portions of the ABI and API change in every kernel release.

By contrast, the Linux kernel does provide a stable userspace interface for Linux applications. These applications essentially have a contract with the Linux kernel that the userspace binary interfaces they rely on will continue to work consistently over time. That's why a pre-compiled Linux application can run correctly on multiple distributions and multiple versions.

The driver code is an integral part of the Linux operating system, not a second-class add-on. Once a driver is accepted into the mainline kernel, it will be maintained over time as internal kernel interfaces change. That is, when a subsystem maintainer accepts a patch to make an incompatible change to a kernel interface, that patch will simultaneously upgrade every driver that relies on the interface. And, new drivers and any upgrades to them automatically flow downstream from the mainline kernel to all Linux distributions.

In this Project we study the mouse driver for the Fedora11 operating system.

Implementation:

We followed the steps described below for the implementation of this project:

1) Setting up Fedora 11 system for the project involved

   a) Downloading the fedora 11 iso file from www.fedora.org

   b) Writing a disc using “iso recorder” application

   c) Formatting initial Windows XP system

   d) Fedora installation

2) Downloading Source code

   > cd /tmp

   > wget http://www.kernel/org/pub/linux/kernel/v2.6/linux-2.6.30.tar.bz2

   >tar –xjvf linux-2.6.30.tar.bz2 –C /usr/src

>yum install gcc

--gcc already installed with latest versions. Nothing to do.

> cd linux-2.6.30

3) Finding the USB mouse driver file and modifying it with printk statements

> cd /usr/src/linux-2.6.30/drivers/hid/usbhid

> gedit usbmouse.c

4) Running make menuconfig to generate a config file (other configurators are: oldconfig, gconfig, and xconfig). The attempt to execute the kernel compilation script failed in absence of the config file.

>cd /home/xyz/Desktop

>./k2.txt A > compilation_log

-- config file missing

> cd /usr/src/linux-2.6.30

> make menuconfig

5) Kernel and module compilation (using the provided script **Appendix A**)

Modified the kernel compilation script by changing the variable to _aVersion=2.6.30 which is our current source directory. Now we execute the kernel compilation script.

>./k2.txt A> compilation_log

```
##
Compile kernels ...
*** Warning: make dep is unnecessary now.
  CHK     include/linux/version.h
  CHK     include/linux/utsrelease.h
  SYMLINK include/asm -> include/asm-x86
```

Results:

usbmouse.c file analysis:

In this section we focus on analyzing the source file for the mouse driver.

The following header files were needed in order to compile and execute the source code:

- linux/kernel.h – Needed for KERN_INFO
- linux/slab.h
- linux/module.h – Needed by all modules
- linux/init.h – Needed for the macros
- linux/usb/input.h – Needed for device input
- linux/hid.h -
- ../hid-ids.h – USB HID quirk support for Linux it defines all USB_VENDOR_ID parameters

In this section we will dissect the source code with primary focus on giving brief description of each method within the code. Overall, there are six primary methods and three structures within the code.

Structures and Descriptions:

I. struct **usb_mouse** – Mouse structures, used to describe the mouse device. This structure has the following members: name, phys, *usbdev, *dev, * irq, *data, and data_dma.

- **name** – a char variable for the mouse device name, including the manufacturer, product category, product and other information.
- **phys** – a char variable for the device node name
- ***usbdev** – a structure of (struct usb_device). The USB device for this device. It requires an embedded structure to describe its USB attribute.
- ***dev** – a structure to describe the properties of the input device
- ***irq** – the interrupt handler. URB request packet structure for transmission of data
- ***data** – the interrupt buffer to receive data
- **data_dma** – DMA transmit an address

II. struct **usb_device_id** – a structure used to represent the driver supported by the device. The USB_INTERFACE_INFO is used to match a specific type of interface. This structure has the following members: USB_INTERFACE_CLASS_HID, USB_INTERFACE_SUBCLASS_BOOT, and USB_INTERFACE_PROTOCOL_MOUSE.

- USB_INTERFACE_CLASS_HID – human-computer interaction device categories
- USB_INTERFACE_SUBCLASS_BOOT – a sub-category boot stage of use of HID
- USB_INTERFACE_PROTOCOL_MOUSE – representation of the mouse device; follows the mouse protocol.

III. struct **usb_driver** – Mouse Driver structure.

Methods and Descriptions:

I. static void **usb_mouse_irq** – This function is shaped as a function of usb_fill_int_urb parameters, in order to build urb set callback function. It takes a pointer parameter of urb. *urb pointer parameter is used in the context of USB drivers to save some data.

II. static int **usb_mouse_open** – This method takes an input_dev pointer (*dev) as its parameter. Opens the mouse devices, and submits to the probe function to build urb, and enter the urb cycle.

III. static void **usb_mouse_close** – also takes an input_dev pointer (*dev) as its parameter. It is responsible for turning off the mouse device at the end of the urb life cycle.

IV. static int **usb_mouse_probe** – this method is for detection of driver functions. It includes several private/helper methods. These helper methods are responsible for different actions such as filling the mouse device structure of the node name; obtaining USB device Sysfs path, the format; inputting the numbers assigned to the input subsystem structure; and providing interface structure included in the device structure body, etc. usb_host_interface is used to describe the interface to set the structure, embedded in the interface structure usb_interface. usb_endpoint_descriptor is the endpoint descriptor structure, embedded in the endpoint structure usb_host_endpoint. Generally, in the probe function, we need the equipment information stored in a usb_interface structure, with a view later adopted by usb_get_intfdata access to use.

V. static void **usb_mouse_disconnect** – this is the mouse device handler when unplugged. It gets the mouse device structure and the interface structure of the mouse pointer to set the empty device. It ends the life cycle of urb (usb_kill_urb()) and frees storage space urb and storage of the mouse structure of storage space, and then release the mouse event data store storage space as well.

VI. static int __init **usb_mouse_init** – drivers, life cycle starting point of the USB core register.

VII. static void __exit **usb_mouse_exit** – driver, life-cycle end point to the USB core write off the mouse driver.

Problems faced :

Solved:

- Fedora Installation:

To avoid all installation disk writing related problems we recommend the use of ISO recorder.

- Kernel Compilation:

For the given script, download the source code from kernel.org. The script does not work with src.rpm files.

- Reboot after compilation does not load the kernel.

Press F8 during boot. This will take you to the kernel options. Select the desired kernel and press 'c'. This displays the grub command prompt. Execute the commands as follows.

grub> find /boot /stage1

grub> find /boot /grub /stage1

one of the above two command gives boot partition and the other one gives an error. Use the displayed partition in the next command.

Grub> root (hdx,y)

Grub>kernel /vmlinuz-2.6.30A

Grub>initrd /initrd- 2.6.30A.img

Grub>boot

Unsolved:

Printk statements are not displayed in the log files.

The possible reasons may be,

- The printk statements are going to some other file i.e. not var/log/messages, or dmesg or spooler or /etc/rsyslog
- They are being suppressed by the /proc/sys/kernel/printk file which currently has "3 4 1 7" value

  >cat /proc/sys/kernel/printk

  3 4 1 7

  Each value has a specific purpose. They are described as follows respectively
  - Console_loglevel: messages with higher priority than this will be printed to the console.
  - Default_message_level: messages without an explicit priority will be printed with this priority.

- Minimum console loglevel: minimum (highest) value to which console log level can be set.
- Default console log level: default value for the console log level.
- Something else?

Note:

The followingcommand resulted in something that grabs our attention.

>cat/dev/input/mice

When we execute this command it gives some weird characters on the screen when we move the mouse.

Code:

```
#include <linux/kernel.h>
#include <linux/slab.h>
#include <linux/module.h>
#include <linux/init.h>
#include <linux/usb/input.h>
#include <linux/hid.h>

/* for apple IDs */
#ifdef CONFIG_USB_HID_MODULE
#include "../hid-ids.h"
#endif

/*
 * Version Information
 */
#define DRIVER_VERSION "v1.6"
#define DRIVER_AUTHOR "Vojtech Pavlik <vojtech@ucw.cz>"
#define DRIVER_DESC "USB HID Boot Protocol mouse driver"
#define DRIVER_LICENSE "GPL"

MODULE_AUTHOR(DRIVER_AUTHOR);
MODULE_DESCRIPTION(DRIVER_DESC);
MODULE_LICENSE(DRIVER_LICENSE);

struct usb_mouse {
	char name[128];
	char phys[64];
	struct usb_device *usbdev;
	struct input_dev *dev;
	struct urb *irq;

	signed char *data;
	dma_addr_t data_dma;
		printk(KERN_INFO "Something important happened here  11...\n");
};

static void usb_mouse_irq(struct urb *urb)
{
	struct usb_mouse *mouse = urb->context;
	signed char *data = mouse->data;
	struct input_dev *dev = mouse->dev;
	int status;

	switch (urb->status) {
	case 0:			/* success */
		break;
	case -ECONNRESET:	/* unlink */
	case -ENOENT:
	case -ESHUTDOWN:
		return;
	/* -EPIPE:  should clear the halt */
	default:		/* error */
		goto resubmit;
```

```
                        printk(KERN_INFO "Something important happened here  12...\n");
        }

        input_report_key(dev, BTN_LEFT,   data[0] & 0x01);
        input_report_key(dev, BTN_RIGHT,  data[0] & 0x02);
        input_report_key(dev, BTN_MIDDLE, data[0] & 0x04);
        input_report_key(dev, BTN_SIDE,   data[0] & 0x08);
        input_report_key(dev, BTN_EXTRA,  data[0] & 0x10);

        input_report_rel(dev, REL_X,     data[1]);
        input_report_rel(dev, REL_Y,     data[2]);
        input_report_rel(dev, REL_WHEEL, data[3]);

        input_sync(dev);
resubmit:
        status = usb_submit_urb (urb, GFP_ATOMIC);
        if (status)
                err ("can't resubmit intr, %s-%s/input0, status %d",
                                mouse->usbdev->bus->bus_name,
                                mouse->usbdev->devpath, status);
                printk(KERN_INFO "Something important happened here  13...\n");
}

static int usb_mouse_open(struct input_dev *dev)
{
        struct usb_mouse *mouse = input_get_drvdata(dev);
        printk(KERN_INFO "Something important happened here  14...\n");
        mouse->irq->dev = mouse->usbdev;
        if (usb_submit_urb(mouse->irq, GFP_KERNEL))
                return -EIO;

        return 0;
}

static void usb_mouse_close(struct input_dev *dev)
{
        struct usb_mouse *mouse = input_get_drvdata(dev);

        usb_kill_urb(mouse->irq);
                printk(KERN_INFO "Something important happened here  16...\n");
}

static int usb_mouse_probe(struct usb_interface *intf, const struct usb_device_id *id)
{
        struct usb_device *dev = interface_to_usbdev(intf);
        struct usb_host_interface *interface;
        struct usb_endpoint_descriptor *endpoint;
        struct usb_mouse *mouse;
        struct input_dev *input_dev;
        int pipe, maxp;
        int error = -ENOMEM;

        interface = intf->cur_altsetting;

        if (interface->desc.bNumEndpoints != 1)
                return -ENODEV;
```

```
endpoint = &interface->endpoint[0].desc;
if (!usb_endpoint_is_int_in(endpoint))
        return -ENODEV;

pipe = usb_rcvintpipe(dev, endpoint->bEndpointAddress);
maxp = usb_maxpacket(dev, pipe, usb_pipeout(pipe));

mouse = kzalloc(sizeof(struct usb_mouse), GFP_KERNEL);
input_dev = input_allocate_device();
if (!mouse || !input_dev)
        goto fail1;

mouse->data = usb_buffer_alloc(dev, 8, GFP_ATOMIC, &mouse->data_dma);
if (!mouse->data)
        goto fail1;

mouse->irq = usb_alloc_urb(0, GFP_KERNEL);
if (!mouse->irq)
        goto fail2;

mouse->usbdev = dev;
mouse->dev = input_dev;
printk(KERN_INFO "Something important happened here  16...\n");
if (dev->manufacturer)
        strlcpy(mouse->name, dev->manufacturer, sizeof(mouse->name));

if (dev->product) {
        if (dev->manufacturer)
                strlcat(mouse->name, " ", sizeof(mouse->name));
        strlcat(mouse->name, dev->product, sizeof(mouse->name));
}

if (!strlen(mouse->name))
        snprintf(mouse->name, sizeof(mouse->name),
                 "USB HIDBP Mouse %04x:%04x",
                 le16_to_cpu(dev->descriptor.idVendor),
                 le16_to_cpu(dev->descriptor.idProduct));

usb_make_path(dev, mouse->phys, sizeof(mouse->phys));
strlcat(mouse->phys, "/input0", sizeof(mouse->phys));

input_dev->name = mouse->name;
input_dev->phys = mouse->phys;
usb_to_input_id(dev, &input_dev->id);
input_dev->dev.parent = &intf->dev;

input_dev->evbit[0] = BIT_MASK(EV_KEY) | BIT_MASK(EV_REL);
input_dev->keybit[BIT_WORD(BTN_MOUSE)] = BIT_MASK(BTN_LEFT) |
        BIT_MASK(BTN_RIGHT) | BIT_MASK(BTN_MIDDLE);
input_dev->relbit[0] = BIT_MASK(REL_X) | BIT_MASK(REL_Y);
input_dev->keybit[BIT_WORD(BTN_MOUSE)] |= BIT_MASK(BTN_SIDE) |
        BIT_MASK(BTN_EXTRA);
input_dev->relbit[0] |= BIT_MASK(REL_WHEEL);

input_set_drvdata(input_dev, mouse);
```

```
	input_dev->open = usb_mouse_open;
	input_dev->close = usb_mouse_close;

	usb_fill_int_urb(mouse->irq, dev, pipe, mouse->data,
			 (maxp > 8 ? 8 : maxp),
			 usb_mouse_irq, mouse, endpoint->bInterval);
	mouse->irq->transfer_dma = mouse->data_dma;
	mouse->irq->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;

	error = input_register_device(mouse->dev);
	if (error)
		goto fail3;

	usb_set_intfdata(intf, mouse);
	return 0;

fail3:
	usb_free_urb(mouse->irq);
fail2:
	usb_buffer_free(dev, 8, mouse->data, mouse->data_dma);
fail1:
	input_free_device(input_dev);
	kfree(mouse);
	return error;
}

static void usb_mouse_disconnect(struct usb_interface *intf)
{
	struct usb_mouse *mouse = usb_get_intfdata (intf);
	printk(KERN_INFO "Something important happened here  17...\n");
	usb_set_intfdata(intf, NULL);
	if (mouse) {
		usb_kill_urb(mouse->irq);
		input_unregister_device(mouse->dev);
		usb_free_urb(mouse->irq);
		usb_buffer_free(interface_to_usbdev(intf), 8, mouse->data, mouse->data_dma);
		kfree(mouse);
	}
}

static struct usb_device_id usb_mouse_id_table [] = {
	{ USB_INTERFACE_INFO(USB_INTERFACE_CLASS_HID, USB_INTERFACE_SUBCLASS_BOOT,
		USB_INTERFACE_PROTOCOL_MOUSE) },
	{ }	/* Terminating entry */
};

MODULE_DEVICE_TABLE (usb, usb_mouse_id_table);

static struct usb_driver usb_mouse_driver = {
	.name		= "usbmouse",
	.probe		= usb_mouse_probe,
	.disconnect	= usb_mouse_disconnect,
	.id_table	= usb_mouse_id_table,
};
```

```
static int __init usb_mouse_init(void)
{
	int retval = usb_register(&usb_mouse_driver);
	if (retval == 0)
		printk(KERN_INFO KBUILD_MODNAME ": " DRIVER_VERSION ":"
				DRIVER_DESC "\n");
		printk(KERN_INFO "Something important happened here  18...\n");
	return retval;
}

static void __exit usb_mouse_exit(void)
{
	usb_deregister(&usb_mouse_driver);
		printk(KERN_INFO "Something important happened here  19...\n");
}

module_init(usb_mouse_init);
module_exit(usb_mouse_exit);
```

Appendix A:

```
#!/bin/sh
#########################################################
# Script: k2.txt
# Description:  compile new kernels
#
# Usage:
# sh compile_kernels.sh <Kernel_Version>
# The default version is 2.6.30.5
#
# Songjie "Jeff" Liang 8/20/2009
# Modified by Team1 12/14/2009
#########################################################
#

if [ "$#" -ne 1 ]; then
  echo "$0 <subversion>"
  echo "For example,"
  echo "$0  C"
  exit 1
fi

if [ $( id |awk -F\( '{print $2}' |awk -F\) '{print $1}' ) != 'root' ]; then
  echo "Must run this script: $0 as the user: root"
  exit 1
fi



_aVERSION=2.6.30
_aSUBVERSION=$1

_BTime=$(date +%T)
cd /usr/src/linux-${_aVERSION}
compile_kernels()
```

```
{
  make dep
  make clean
  make bzImage
  sleep 10

  if [ ! -f arch/x86/boot/bzImage -o ! -f System.map]; then
    echo "'make bzIMage' did not perform well. Check and re-run it again."
    exit 1
  fi

  cp -fp arch/x86/boot/bzImage /boot/vmlinuz-${_aVERSION}${_aSUBVERSION}
  sleep 4
  chmod a+x /boot/vmlinuz-${_aVERSION}${_aSUBVERSION}
  cp -fp System.map /boot/System.map-${_aVERSION}${_aSUBVERSION}
  sleep 2

  if [ ! -f /boot/vmlinuz-${_aVERSION}${_aSUBVERSION} -o ! -f /boot/System.map-
${_aVERSION}${_aSUBVERSION} ]; then
    echo "'make bzIMage in vmlinuz-${_aVERSION}${_aSUBVERSION}' did not perform well. Check
and re-run it again."
    exit 1
  fi
}

compile_kernel_modules()
{
  make modules
  sleep 5
  make modules_install
  sleep 5
}

echo "##"
```

```
echo "Compile kernels ..."
compile_kernels

echo "##"
echo "Compile kernel modules: It takes a long time. Be patient..."
compile_kernel_modules

echo "Create a new initial ramdisk ..."

mkinitrd /boot/initrd-${_aVERSION}${_aSUBVERSION}.img ${_aVERSION}

if [ ! -f /boot/initrd-${_aVERSION}${_aSUBVERSION}.img ]; then
  echo "'mkinitrd /boot/initrd...' did not perform well. Please check and rerun it."
  exit 1
fi

echo "Update GRUB ..."
grubFL=/boot/grub/grub.conf
cp -pf ${grubFL} ${grubFL}.bkp

echo "title Fedora (${_aVERSION}${_aSUBVERSION})" >> ${grubFL}
echo "root (hd0,0)" >> ${grubFL}
echo "kernel /vmlinuz-${_aVERSION}${_aSUBVERSION} ro root=/dev/mapper/vg_barelinux2-lv_root
rhgb quiet" >>${grubFL}
echo "initrd /initrd-${_aVERSION}${_aSUBVERSION}.img" >>${grubFL}

echo "Completed!"
echo "Begin: ${_BTime}    End: $(date +%T)"
##shutdown -P now
```