Towson University
Department of Computer and Information Sciences

A Student Group Research Paper

For

Dr. Ramesh Karne

COSC 519: Operating System Principles

USB Mouse

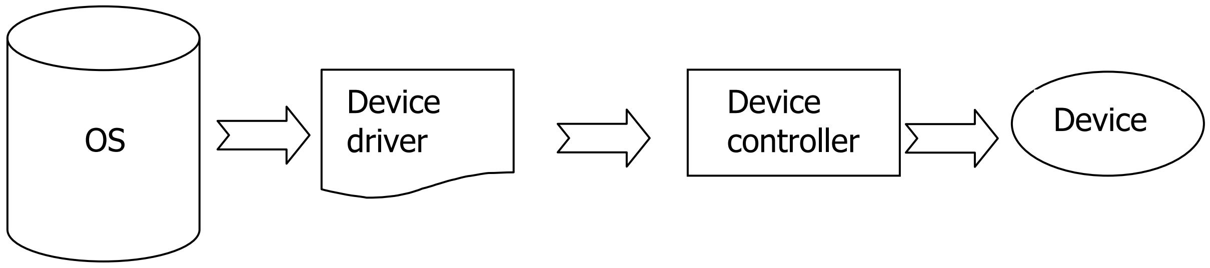
Driver Analysis

Table of Contents

Introduction

This group research project's objective was to research, analyze and reproduce a working USB device driver. More specifically, our project focused on a USB mouse driver on the Linux operating system. Our research included discovering the components, functions, and subsystems required by Linux for the development of a USB device driver for a mouse.

By general definition, a device driver is a program that controls a particular type of I/O device attached to a computer system. While most device drivers are built into the operating system, whenever adding a new type of device that the operating system, there may be a requirement to install the new device driver. A device driver basically converts the input/output instructions of the operating system to messages that the device type can understand. For any hardware device that is present in computer system, there should be a device controller that gets the message from device driver, which in turn has received the input from operating system as a command like "retrieve block #123". Therefore, a device driver acts as a translator between the OS and the device controller. This abstract schema below shows the interaction of OS with hardware with the help of device driver and hardware controller.



USB Subsystem

The USB core in Linux is built up from endpoints, interfaces, and configurations. We will briefly go over endpoints and interfaces, but going into detail on interface settings and configurations is outside of the scope of this report.

Endpoints

Endpoints are the most basic for of USB communication and data is carried only in one direction for an endpoint. Endpoints can be of four different types:

CONTROL Endpoint:

Control endpoints are used to allow access to different parts of the USB device. They are commonly used for configuring the device, retrieving information about the device, sending commands to the device, or retrieving status reports about the device.

INTERRUPT Endpoint:

Interrupt endpoints transfer small amounts of data at a fixed rate every time the USB host asks the device for data. These endpoints are the primary transport method for USB keyboards and mice.

BULK Endpoint:

Bulk endpoints transfer large amounts of data. These endpoints are usually much larger (they can hold more characters at once) than interrupt endpoints.

ISOCHRONOUS Endpoint:

Isochronous endpoints also transfer large amounts of data, but the data is not always guaranteed to make it through. These endpoints are used in devices that can handle loss of data, and rely more on keeping a constant stream of data flowing. Real-time data collections, such as audio and video devices, almost always use these endpoints.

Interfaces

Endpoints are bundled into interfaces that handle only one type of USB logical connection. Interfaces are used by the kernel as a structure (struct usb_interface) that is passed to the driver. There are four fields in the structure and they are “struct usb_host_interface *altsetting”, “unsigned num_altsetting”, “struct usb_host_interface *cur_altsetting”, and “int minor”.

USB URBS

USB request blocks (urbs) are used by the driver module to send or receive data to or from a USB endpoint in an asynchronous manner. The urb gives the driver module a means of communicating with the device through an endpoint.

Structure urb:

dev – This points to the USB device that this urb is sent to.

pipe – This is the endpoint information for the specific USB device that this urb is being sent to.

transfer_buffer – This is a pointer to the buffer to be used when sending or receiving data.

transfer_buffer_length – This is the length of the buffer pointed to by the transfer_buffer.

complete – A pointer to the completion handler function that is called by the USB core when the urb is completely transferred or when an error occurs to the urb.

context – A pointer to data that can be set by the USB driver.

status – This is the current status of the urb. The only time the USB driver can safely access this variable is inside the urb completion handler.

start_frame – This is the initial fram number for isochronous transfers to use.

interval – This is the interval that the urb should be polled for data. This is only used for interrupt or isochronous urbs. The units of measure depend on the type of device being used. Low-speed or full-speed devices use frames and they are equivalent to 1 millisecond, other devices use microframes and they are equivalent to 1/8 millisecond.

Classes of Devices and Modules

Before coding the driver module the question needs to be asked “What class of device am I writing this module for?” The decision will have ramifications on what is required in the driver module. The Linux kernel separates hardware devices into three main classes: character devices, block devices, and network interfaces.

A Character device can be accessed as a stream of characters or bytes, similar to a file. These devices include the text console, the keyboard and the serial ports. This device module will typically implement open, close, read, and write system calls.

A block device is a device that can only be accessed in blocks of data (usually a kilobyte or another power of 2) and can host a file system, like a disk drive. Linux allows applications to read and write to block devices like a character device, so they are managed in a special way by the kernel. These implement similar system calls to character devices but use an entirely different interface.

A network device allows the exchange of data with other hosts. These devices are usually hardware, but can also be purely software. The kernel has a different way to communicate with network devices than with character or block devices. Instead of reading and writing to the device, it handles packet transmission.

Device Identification

In Linux the usb_device_id structure is used to define a specific device. A usb_device_id table that is declared in the device driver is used to match to the usb_device_id structure associated with device to determine if the driver is intended for that device.

Structure usb_device_id:

> match_flags – Used to determine which of the fields in the structure the device should be matched against.
> id_Vendor – This is the USB vendor ID for the device.
> id_Product –This is the USB product ID for the device.
> bcdDevice_lo – Defines the low end of the range for vendor assigned product version numbers.
> bcdDevice_hi – Defines the high end of the range for vendor assigned product version numbers.
> bDeviceClass – Defines the class of the device.
> bDeviceSubClass – Defines the subclass of the device.
> bDeviceProtocol – Defines the protocol of the device.
> bInterfaceClass – Defines the class of the interface.
> bInterfaceSubClass – Defines the subclass of the interface.
> bInterfaceProtocol – Defines the protocol of the interface.
> driver_info – Provides information that the driver can use to differentiate the different devices from one another in the probe callback functions. This member is not used for matching.

Linux System Calls and Error Codes

While writing a USB device driver module there are several important system calls and error codes that the developer should be aware of and will most likely need for the driver.

System Calls

usb_buffer_alloc:

This function call is used to allocate memory for a USB buffer. Return value is either null (indicating no buffer could be allocated), or the cpu-space pointer to a buffer that may be used to perform DMA to the specified device. When the buffer is no longer used, free it with usb_buffer_free.

usb_submit_urb:

This function submits a transfer request, and transfers control of the urb describing that request to the USB subsystem. Request completion will be indicated later, asynchronously, by calling the completion handler. Successful submissions return 0; otherwise this routine returns a negative error number. If the submission is successful, the complete callback from the urb will be called exactly once, when the USB core and Host Controller Driver (HCD) are finished with the urb. When the completion function is called, control of the URB is returned to the device driver which issued the request. The completion handler may then immediately free or reuse that URB.

input_sync:

Informs the event handler that the device has transmitted an internally consistent set of data using the input subsystem.

usb_endpoint_is_int_in:

Returns true if the endpoint has interrupt transfer type and IN direction, otherwise it returns false.

usb_rcvintpipe:

Used to set up a receive pipe value for the URB.

usb_maxpacket:

Returns the max size of the USB packet for this device.

usb_pipeout:

Returns true if the pipe is an out pipe.

input_allocate_device:

Allocates memory for new input device and returns a prepared struct input_dev.

usb_alloc_urb:

Creates a new urb for a USB driver to use and returns a pointer to it. If no memory is available, NULL is returned. If the driver wants to use this urb for interrupt, control, or bulk endpoints, then pass '0' as the number of iso packets. The driver should call usb_free_urb when it is finished with the urb.

usb_make_path:

This function returns a stable device path in the USB tree.

usb_to_input_id:

This function is used to uniformly produce a struct input_id for USB input devices.

input_set_drvdata:

This function sets the driver data in the input core.

input_get_drvdata:

This function gets the driver data from the input core.

usb_fill_int_urb:

This function initializes an interrupt urb with the proper information needed to submit it to a device.

input_register_device:

This function registers the device with the input core. The device must be allocated with input_allocate_device and all of its capabilities must be set up before registering.

usb_free_urb:

This must be called when a user of a urb is finished with it. When the last user of the urb calls this function, the memory of the urb is freed.

usb_buffer_free:

This reclaims an I/O buffer, letting it be reused. The memory must have been allocated using usb_buffer_alloc, and the parameters must match those provided in that allocation request.

kfree:

This function frees up the memory allocated by kmalloc.

usb_kill_urb:

This routine cancels an in-progress urb. It is guaranteed that upon return all completion handlers will have finished and the urb will be totally idle and available for reuse or for calling usb_free_urb.

input_unregister_device:

This function unregisters an input device. Once device is unregistered the caller should not try to access it as it may get freed at any moment.

USB_INTERFACE_INFO:

This macro is used to create a struct usb_device_id that matches a specific class of interfaces (bInterfaceClass, bInterfaceSubClass, bInterfaceProtocol).

input_report_key:

This call is used to send “key” events from the device driver to the input core.

input_report_rel:

This call is used to send “rel” events from the device driver to the input core.

Error Values

ECONNRESET:

The urb was unlinked by a call to usb_unlink_urb.

ENOENT:

The urb was stopped by a call to usb_kill_urb.

ESHUTDOWN:

There was a severe error with the USB host controller driver or the device was disconnected from the system before the urb was submitted.

EPIPE:

The endpoint is now stalled.

EIO:

I/O error is a catch all error code for anything that doesn't fall under a predefined error code for an input device

ENODEV:

The USB device is now gone from the system.

ENOMEM:

No memory for allocation of internal structures.

Requirements for Driver Module

When creating a Linux device driver there are several things that are required to be defined in the source code such as header files, system macros, structures, and functions.

Header Files

- linux/module.h
- linux/init.h
- linux/kernel.h
- linux/slab.h
- linux/usb/input.h
- linux/hid.h

Descriptive Declarations and Macros

MODULE_AUTHOR(driver_author):

This declaration is used to state who wrote the module.

MODULE_DESCRIPTION(driver_desc):

This declaration provides a human readable statement of what the module does.

MODULE_LICENSE(driver_license):

This declaration specifies the license for the module. Linux recognizes "GPL", "GPL v2", "GPL and additional rights", "Dual BSD/GPL", and "Proprietary." If this is omitted the kernel will assume the module is "Proprietary" and will now be considered "tainted" by a non-open license.

MODULE_DEVICE_TABLE(usb, id_table):

This declaration is used to tell the user space about which devices the module supports.

module_init(device_init):

This macro is used to specify what function should be used for initialization. "device_init" should be the name of whatever function is being used for initializing the module.

module_exit(device_exit):

This macro is used to specify what function should be used for exit. "device_exit" should be the name of whatever function is being used for exiting the module.

Structures and Variables

For the following structures not all members are listed, only those members that are set in the driver or are important to a USB mouse device driver are listed.

Structure usb_driver:

owner – This is a pointer to the module owner of this driver and should be set to "THIS_MODULE".

name – This is a pointer to the name of the driver. This must be unique among all of the other USB drivers and is normally set to the same name as the module.

id_table – This is a pointer to the struct usb_device_id table that contains a list of devices this driver can be used for.

probe – This is a pointer to the function in the driver to use when a matching device is found connected to the system. The interface for the device is passed to this function; the function then will claim and initialize the device and return a zero or will generate an error and return a negative error value.

disconnect – This is a pointer to the function in the driver to use when the device is disconnected from the system.

Structure usb_device_id Array:

Each device or interface that this driver is intended for should be listed in this struct array. At a minimum there must be one element that is set to a usb_device_id and the last element should be empty to terminate the array.

Functions

static int __init device_init(void):

This is the initialization function that is pointed to by module_init. This function is used to register the struct for the driver to the USB core using usb_register(&device_driver).

static void __exit device_exit(void):

This is the exit function that is pointed to by module_exit. This function is used to deregister the struct for the driver from the USB core using usb_deregister(&device_driver).

static int device_probe(struct usb_interface, const struct usb_device_id):

The probe function is called when a device that this driver can be used for is connected to the system. The function is used to initialize any local structures, endpoints, interfaces, urbs, etc… that will be needed by the driver. Typically usb_set_intfdata(data) is used to make the local objects available later by the driver.

static void device_disconnect(struct usb_interface):

The disconnect function is called when the device is disconnected from the system. The function is used to clean up the memory being used by the driver and deregister the USB device. The clean-up of memory is typically accomplished by calling usb_get_intfdata(interface) and then setting.

static int usb_device_open(struct input_dev *dev):

This function is called when the input_register_device call is made to register the device. This function will set the urb device member to the value of the usb_device and then submit the urb to the usb core.

static int usb_device_close(struct input_dev *dev):

This function is called when the input_unregister_device call is made to unregister the device. This function will call the usb_killurb function to cancel the urb from any current processing and make it sit idle.

static void usb_device_irq(struct urb *urb):

This function is set as the completion handler for the urb when the urb is initialized. Whenever the urb completes or has an error this function will be called. This function is responsible for checking for errors in the urb and taking appropriate action or if there are no errors then it is responsible for processing any data that the urb has received from the device.

Linking Driver Modules

Linux allows for the linking and unlinking of device drivers modules into the kernel. This allows for the flexibility for users of Linux to use the provided drivers, create their own, or modify existing ones.

Linking to the Kernel

The device driver module is linked into the kernel using the “insmod” command. For example if our module is “helloworld.c” we would first compile our code to an object file and then use a makefile to create a kernel module (helloworld.ko). It is then linked to the kernel via “$ insmod helloworld.ko”. Upon linking the module to the kernel, the kernel will look for “module_init(device_init)” in the code. When the kernel finds the reference it will go to the function listed as device_init which is used to register the driver.

Delinking from the Kernel

The device driver module is delinked from the kernel using the “rmmod” command. For example if our linked module is “helloworld.ko” we would delink it from the kernel via “$ rmmod helloworld.ko”. Upon delinking the module from the kernel, the kernel will look for “module_exit(device_exit)” in the code. When the kernel finds the reference it will go to the function listed as device_exit which is used to deregister the driver.

Implementation Steps

1. Install Linux Kubuntu 11.04
2. Find the driver source:

   Linux/drivers/hid/usbhid/usbmouse.c and modifying it with printk statements
3. Set path to:

   ```
   /usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/src/linux-headers-2.6.38-8-generic/arc/ia64/include:/usr/src/linux-headers-2.6.38-8-generic/include/linux:/usr/src/linux-headers-2.6.38-8-generic/include:/usr/src/linux-headers-2.6.38-8/arc/ia64/include:/usr/src/linux-headers-2.6.38-8/include:/usr/src/linux-headers-2.6.38-8/include/linux/drivers
   ```
4. Create Makefile:

   ```
   # obj-m is a list of what kernel modules to build. The .o and other
   # objects will be automatically built from the corresponding .c file -
   # no need to list the source files explicitly.
   ```

```
PATH := $(PATH)
obj-m := main.o
# KDIR is the location of the kernel source. The current standard is
# to link to the associated source tree from the directory containing
# the compiled modules.
KDIR := /lib/modules/$(shell uname -r)/build
# PWD is the current working directory and the location of our module
# source files.
PWD := $(shell pwd)
# default is the default make target. The rule here says to run make
# with a working directory of the directory containing the kernel
# source and compile only the modules in the PWD (local) directory.
default:
$(MAKE) -C $(KDIR) M=$(PWD) modules
```

5. Create module by compiling the makefile:
```
$ make
```
6. Install module:
```
$ sudo insmod ./main.ko
```
7. Testing the module
8. Uninstall module:
```
$ sudo rmmod main
```

Operating System Events Timeline

- Load the module (insmod) – driver registration with USB core
```
static struct usb_driver usb_mouse_driver = {
      .name          = "usbmouse",
      .probe         = usb_mouse_probe,
      .disconnect    = usb_mouse_disconnect,
      .id_table      = usb_mouse_id_table,
 };
```
- Declare supported devices (interfaces), and bind them to probe() and disconnect() functions – this happens when device is plugged in
```
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);
```

- Probe() function called – it records the interface information and registers resources
- Now, any functions from file_operations structure can be called and user program can actually talk to the device. URBs are initialized for data transfer. When transfer is done, completion handler function is called.
- Device removed – disconnect() function called and the driver is unloaded.

Source Code

```c
/*
 *  Copyright (c) 1999-2001 Vojtech Pavlik
 *
 *  USB HIDBP Mouse support
 */

/*
 * 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; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
 *
 * Should you need to contact me, the author, you can do so either by
 * e-mail - mail your message to <vojtech@ucw.cz>, or by paper mail:
 * Vojtech Pavlik, Simunkova 1594, Prague 8, 182 00 Czech Republic
 */

#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;
};

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;
	}

	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);
}

static int usb_mouse_open(struct input_dev *dev)
{
        struct usb_mouse *mouse = input_get_drvdata(dev);

        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);
}

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;

	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);

	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");
	return retval;
}

static void __exit usb_mouse_exit(void)
{
	usb_deregister(&usb_mouse_driver);
}

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

Potential Issues

Portability

A device driver is written very specifically for an operating system. So much so that minor updates to the operating system can render a device driver inoperable. We experience this fact repetitively as we attempted to implement the USB mouse driver on a Linux installation. The difficulties we experienced in our implementation are reflective of the issue of portability that device driver creators and operating system programmers face continually. The issue is exacerbated when a device is intended to be used by entirely different operating systems, as our device was designed to do. Engineers creating device drivers and disseminating them to end users have a continual maintenance requirement to ensure that their drivers continue to function properly. This task not only encompasses the most recent versions of operating systems but continual support for historical versions as well since an end-user's specific 'version' of the OS can't be anticipated.

Documentation

In many cases documentation is either minimal or not publicly available. In these cases, there is a need to effectively reverse engineer the device driver. One way to accomplish this is by using a program such as SnoopyPro (usbsnoop.sourceforge.net) that provides a capture of all USB data sent from the computer to the device. By analyzing this captured data, you can start to determine what calls and functions are used to accomplish what tasks. Another option to reverse engineer a driver is to use a virtual environment via VMWare. Since the virtual OS has the ability to talk to all devices connected to the host OS, the virtual OS can similarly capture USB data traffic that can then be analyzed. Regardless of the method, capturing the USB data is necessary to decipher how the device expects data to be sent to the device and the corresponding functions the data will invoke.

Security

Security is not usually addressed heavily within a device driver. Security is typically left to the kernel code to enforce. In fact, in most cases attempting to writing security policies into a device driver can create serious problems. As a rule, the kernel should govern security as the security strength of the kernel is the effective security strength of the entire computer including all devices connected to it.

From the perspective of a malicious attack, device driver can actually be used to penetrate and exploit the security strength of a computer as they have system level access to the kernel. A device driver can use interrupts to affect global resources, load firmware to damage hardware, or change the default block size on an external storage device. All of these actions could severely damage and/or compromise a computer system. Furthermore, these types of problems can be not introduced purposely but also accidentally by a device driver creator through simple programming mistakes. Buffer overruns are a common mistake by coders and will cause an entire system to have issues.

While operating systems need to be suspicious of device driver code, device drivers need to be equally suspicious about data received from user processes or uninitialized memory from the kernel. Drivers need to be cognizant their devices do not become the unintentional vehicles for data leakage. Device drivers can take certain precautions such as zeroing or initializing all memory received from the kernel before utilizing it, to minimize the risk that the device will accidentally process or transmit data that could compromise the system.

Conclusion

Through the development and completion of this project, the group's understanding of how an operating system and device drivers interact grew significantly. It was critical to discover and then understand the role a device driver has and the expectations the operating system has of that role. The project educated us on the entire set of components contained within a device driver. For instance, we had to figure out that Linux requires header files, system macros, structures, and functions to be defined within the driver source code. Gaining this level of understanding was crucial to being able to reproduce a driver of our own. Finally, the actual reproduction of the driver itself challenged our group technically and forced us to expand our abilities in order to actually create a functioning device driver. This project has provided the group with a solid, practical understanding of the function and workings of a USB mouse driver for the Linux operating system.

References

Corbet, J., Kroah-Hartman, G., and Rubini, A. (2005). Linux Device Drivers, 3rd Edition. Retrieved April 12, 2011 from http://www.makelinux.net/ldd3/.

Kroah-Hartman, G. (2004). Writing a Simple USB Driver. Retrieved on April 4, 2011 from http://www.linuxjournal.com/article/7353.

Fliegl, Detlef. (2000). Programming Guide for Linux USB Device Drivers. Retrieved May 9, 2011 from http://www.lrr.in.tum.de/Par/arch/usb/download/usbdoc/usbdoc-1.32.pdf.

Linus Torvalds. (n.d.). Linux Cross-Reference. Retrieved May 2, 2011 from http://lxr.free-electrons.com/source/drivers/usb/.