USB Keyboard Driver
Project Report

Group #7: Stefan Bruno, Kevin Kuo,
Mary Snyder, Akhila Todupunuri

COSC 519: Operating Systems
Dr. Ramesh K. Karne
Fall 2015
Towson University

Table of Contents

- Objective
- Introduction
- Implementation and Approach
  - Environment and Setup
    - Operation System Determination
    - Virtual Machine (VM) Setup
  - USB Keyboard Driver
    - Software Configuration and Source Code
    - Natively Loaded Modules and USB Devices
    - Hardware: USB Keyboard
    - Building Driver
    - Inserting/Removing Driver
    - Modified Key Mapping
- Results and Conclusions
  - Final Analysis
  - Group Roles
  - Problems Faced/Lessons Learned

Objective

The purpose of this study is to understand the USB keyboard driver in a Linux Operating System (CentOS/Ubuntu). To demonstrate our knowledge we will attempt to reprogram the functionality performed for a single keystroke.

Introduction

A device driver is software that acts as an interface between the operating system (OS) and the hardware. These drivers control and manage the devices as well as help convert requests from the kernel to that of the hardware.

Universal Serial Bus (USB) acts as a connection media between the host and all its peripheral devices. It was originally developed to replace all the slow and multiple buses required for parallel/serial and keyboard connections with a standardized bus to which all the devices could be connected in a “plug and play” fashion. The keyboard is one of the most popular USB devices that belong to the USBHID (Human Interface Device) class. USB keyboards are connected to the host system using a USB cable. Keystrokes are converted to electrical signals that are sent over the USB link. These signals are then received by the host machine and translated into scan codes.

In this project, the group explored and modified the existing “usbkbd.c” from the official Linux kernel repository. This module was used as a basis on which to learn and demonstrate our knowledge of the Linux kernel and USB keyboard device interaction.

Implementation and Approach

Environment and Setup

Operation System Determination

When we began the study into the USB keyboard device driver, our group decided to use Virtual Machines (VMs) instead of a standalone laptop or desktop for our investigation, implementation, and testing purposes. This not only allowed each group member the flexibility to work on his or her own time, but also allowed each group member to work on different aspects of the project simultaneously. The virtualization approach ensured we could minimize variation between different host machine environments as well as maintain a consistent virtual machine configuration.

We selected CentOS (version 7 x64 with kernel version 3.10.0-229.20.1.el7.x86_64) as our first candidate operating system, since CentOS was the operating system being used for the class work. As we began to work with the drivers in the operating system, we discovered many of the details were not as easily accessible as other Linux operating systems such as Ubuntu. After struggling to find the details we were interested in, we decided to switch gears and determine if other Linux distributions were more easily configurable. We determined Ubuntu (version 14.04.03 LTS x64 with kernel version 3.19.0-37-generic) to be a much better development environment and operating system for our needs and switched our investigations from CentOS to Ubuntu. CentOS was designed with an enterprise environment in mind and the loading/unloading of modules is more involved.

Virtual Machine (VM) Setup

To install the Ubuntu VM, follow the steps below (CentOS would be similar only with an .iso from the CentOS website):

1. Download and install Oracle VirtualBox (www.virtualbox.org/wiki/Downloads)
2. Download the 64 bit desktop .iso from the Ubuntu website (www.ubuntu.com/download/desktop)
3. Open VirtualBox and create a Ubuntu virtual machine
    a. Click New (or Machine->New) to create a new machine
    b. *Name and operating system:* Select Type “Linux” and Version “Ubuntu (64-bit)” from the pull-down menus, enter a name, and press “Next”
    c. *Memory size:* Take the default/recommended memory size and press “Next”
    d. *Hard disk:* Select “Create a virtual hard disk now”; press “Create”
    e. *Hard disk file type:* Select “VDI”; press “Next”
    f. *Storage on physical hard disk:* Select “Dynamically allocated”; press “Next”
    g. *File location and size:* Use the default/recommended location/size; press “Create”
    h. Open the created machine and it will ask to select the start-up disk; select the location of the .iso downloaded in step 2; press “Start”
    i. When prompted:
        i. *Welcome:* press “Install Ubuntu”
        ii. *Preparing to install Ubuntu:* press “Continue”
        iii. *Installation type:* press “Install Now”
        iv. *Write the changes to disk?:* press “Continue”
        v. *Where are you?:* press “Continue”
        vi. *Keyboard layout:* press “Continue”
        vii. *Who are you?:* create a username and password; press “Continue”
        viii. *Installation Complete:* press “Restart Now”

USB Keyboard Driver

Software Configuration and Source Code

Version control was implemented using Git. This allowed changes to source code to be tracked as well as easier collaboration among all group members.

Eclipse Mars 4.5.1 served as the integrated development environment, which includes Git source control capability. If replicating environment, ensure Eclipse has the appropriate add-ons to develop in C/C++ and interface with a remote Git repository.

The source code used for our project is from the open source Linux kernel trunk available online at https://github.com/torvalds/linux/tree/master/drivers/hid/usbhid. We also chose to use GitHub to host our source code repository, which can be found at https://github.com/Polo08816/USBKeyboardDriverLinux. As a convenience, a copy of the source code has been attached to the end of this report.

Natively Loaded Modules and USB Devices

To start our investigation, the first step was to determine which kernel modules are loaded in the base operating system before loading any additional modules. To do so we used “lsmod” in a terminal, which shows all modules that are currently loaded. The list of modules loaded looked very similar for both the Ubuntu and CentOS operating system VMs. The module we were most interested in for this study was the “usbhid” module. It manages the USB protocol for all USB devices, including USB keyboards and mice.

We also needed to determine which USB devices were currently loaded and/or recognized by the operating system before inserting or attaching any other USB devices to interact with the VM. To do so we used “lsusb” in a terminal, which shows all USB devices currently loaded. Again, the USB device list for both the Ubuntu and CentOS VM operating systems were very similar.

Hardware: USB Keyboard

In this project, we discovered it is necessary to pay special attention to the type of keyboard used. In our initial testing, we were unable to disable the use of the built-in/form factor laptop keyboard. We determined that a built-in laptop keyboard is not using the USB protocol and therefore does not interact with the same kernel module as a physical USB keyboard would. We then acquired a standard Hewlett Packard USB 2.0 keyboard as well as a Dell Smart Card Reader USB 2.0 Keyboard. It is important to ensure that the keyboard used is physically plugged into a USB port on the host machine, not through a docking station or other means, so the VM is able to recognize it.

To guarantee the VM recognizes the USB keyboard, for both Ubuntu and CentOS click Devices → USB → CHICONY HP Basic USB Keyboard. As an added note, if multiple VMs are being used, only one virtual machine at a time can capture the input from the physical keyboard. The USB keyboard must be disconnected from the VM Devices menu of the current VM before it can be attached to another VM.

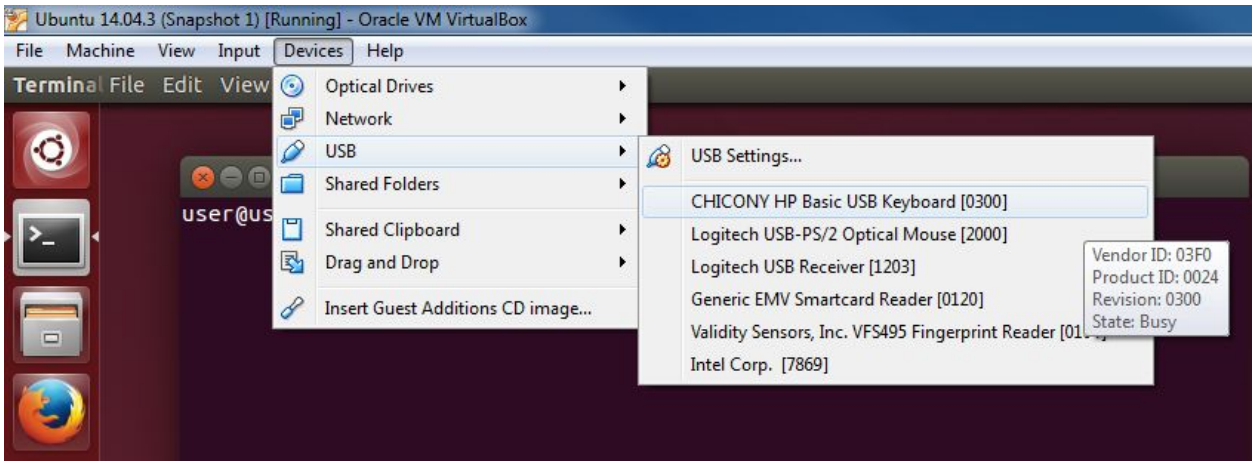
Standard keyboard (HP) example for Ubuntu:

user@user-VirtualBox:~$ lsusb

Bus 001 Device 003: ID 03f0:0024 Hewlett-Packard KU-0316 Keyboard

Bus 001 Device 002: ID 80ee:0021 VirtualBox USB Tablet

Bus 001 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub

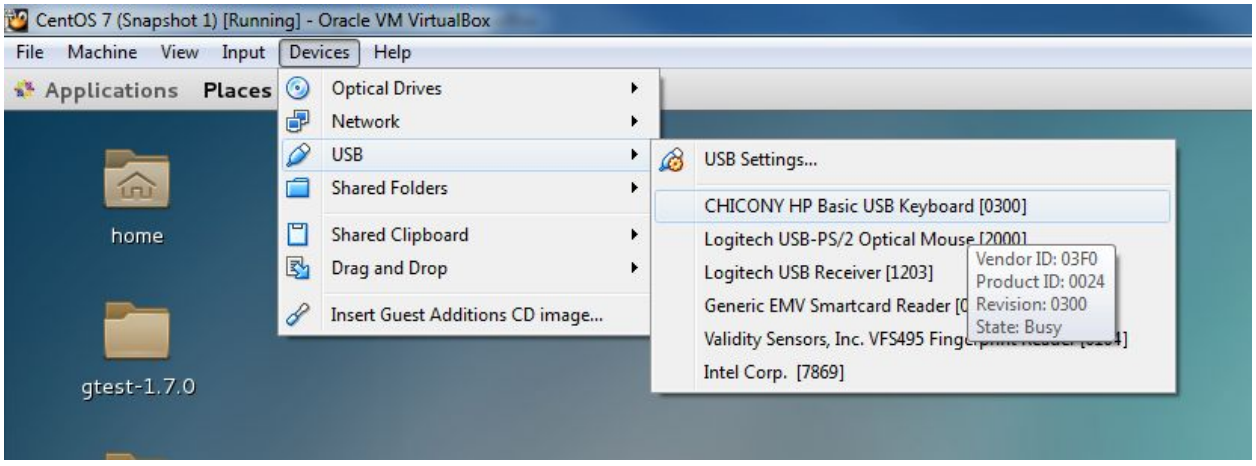


Standard keyboard (HP) example for CentOS:

[user@localhost Desktop]$ lsusb

Bus 001 Device 002: ID 03f0:0024 Hewlett-Packard KU-0316 Keyboard

Bus 001 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub



After the keyboard is physically attached and the VirtualBox Devices menu updates have been made, the keyboard will behave normally inside the VM. This is because both Ubuntu and CentOS load the “usbhid” kernel module by default at boot time. To remove this module, execute the following command:

user@user-VirtualBox:~$ ***sudo rmmod usbhid***

Immediately after entering the root password in Ubuntu, the VM will no longer register keystrokes from the USB keyboard; however, the operating system still recognizes the USB keyboard are being attached.

user@user-VirtualBox:~$ lsusb
Bus 001 Device 004: ID 03f0:0024 Hewlett-Packard KU-0316 Keyboard
Bus 001 Device 002: ID 80ee:0021 VirtualBox USB Tablet
Bus 001 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub

Unfortunately, limitations arose in CentOS 7 by not having privileges to remove the “usbhid” module even as a root user.

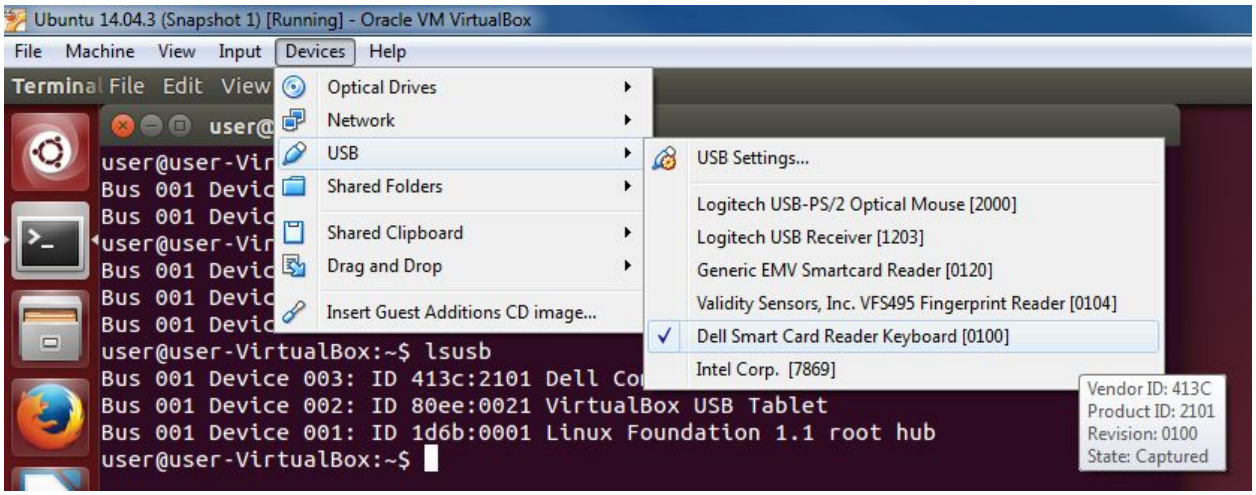
[root@localhost boot]# rmmod usbhid
rmmod: ERROR: ***Module usbhid is builtin****.*

For this reason, we chose to switch our development to Ubuntu 14.04.3 LTS and continue the rest of our implementation and testing in Ubuntu 14.04.3 LTS.

Continuing with Ubuntu, we also tested the keyboard with a smart card reader and saw similar results to the standard keyboard.

Smart Card Reader Keyboard example for Ubuntu:

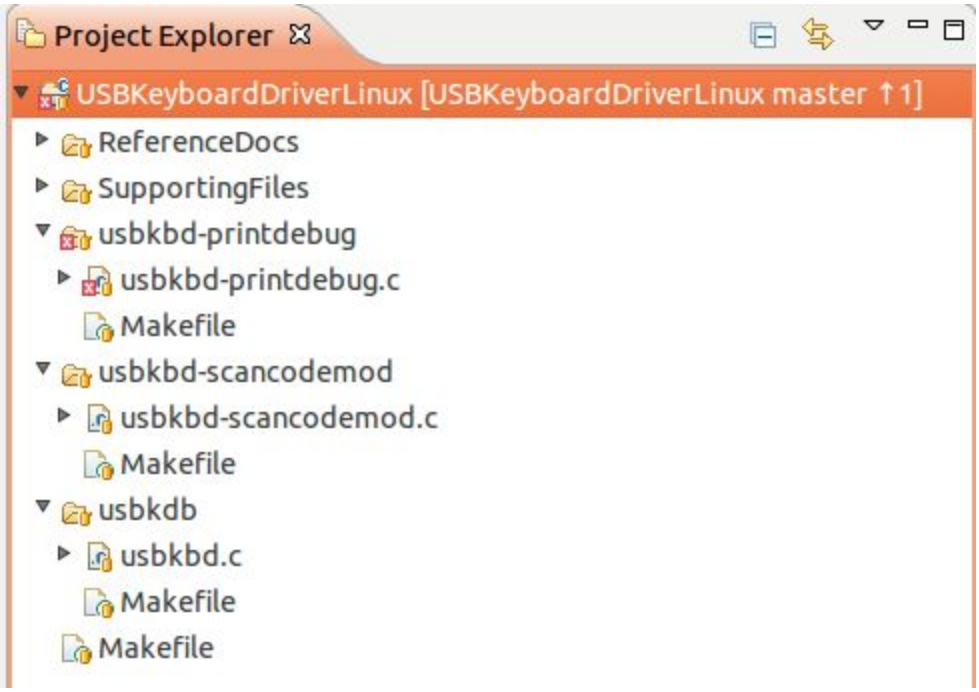
user@user-VirtualBox:~$ lsusb
Bus 001 Device 004: ID 058f:9540 Alcor Micro Corp.
Bus 001 Device 003: ID 413c:2101 Dell Computer Corp. SmartCard Reader Keyboard
Bus 001 Device 002: ID 80ee:0021 VirtualBox USB Tablet
Bus 001 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub



After removing the “usbhid” kernel module, the Dell Smart Card Reader Keyboard also ceased to register keystrokes despite still being listed as a USB device attached to the VM.

Building Driver

The structure of the project source code contains a main Makefile in the project root directory (Eclipse: USBKeyboardDriverLinux). This Makefile will execute three other Makefiles for each sub-project: usbkbd, usbkbd-scancodemod, and usbkbd-printdebug.



To build the modules simply execute the “make” command in the project’s main directory and the output will contain a loadable kernel module (*.ko) within each subproject folder (3 in total).

Inserting/Removing Driver

Before including our module, it is necessary to remove the usbhid module so the keyboard will have no choice but to use our module.
This is accomplished using the following command:

user@user-VirtualBox:~$ sudo rmmod usbhid

When the remove command is executed, the following information is printed in the /var/log/syslog file:

Dec 6 21:24:56 user-VirtualBox kernel: [3360.662002] usbcore: deregistering interface driver usbhid

Dec 6 21:24:56 user-VirtualBox acpid: input device has been disconnected, fd 11

This shows the module was successfully removed.

With the usbhid module removed, one of our new modules can be inserted, such as the ‘usbkbd-printdebug’ module that includes our debugging statements.
This is accomplished using the following command:

user@user-VirtualBox:~/git/USBKeyboardDriverLinux/usbkbd-printdebug$ sudo insmod usbkbd-printdebug.ko

To verify the module has been inserted, an “lsmod” can be performed:

user@user-VirtualBox:~$ lsmod
Module Size Used by
usbkbd_printdebug 16384 0

When the insert command is executed, the following information is printed in the /var/log/syslog file:

Dec 6 21:29:11 user-VirtualBox kernel: [3615.911146] usbkbd_printdebug: module verification Dec 6 21:29:11 user-VirtualBox kernel: [3615.911596] ***Group7*** *usbkbd probe*
Dec 6 21:29:11 user-VirtualBox kernel: [3615.911640] input: CHICONY HP Basic USB Keyboard as /devices/pci0000:00/0000:00:06.0/usb1/1-2/1-2:1.0/input/input9
Dec 6 21:29:12 user-VirtualBox kernel: [3615.911643] ***Group7*** *usbkbd open*
Dec 6 21:29:12 user-VirtualBox kernel: [3615.930098] usbcore: registered new interface driver usbkbd

This shows our usbkbd_printdebug module was able to be loaded into the kernel and the HP keyboard will be using our newly inserted module.

To restore the original setup after completing demonstration of the updated kernel module, remove the new module that was inserted (such as usbkbd_printdebug) using the “rmmod” command mentioned above.

user@user-VirtualBox:~$ sudo rmmod usbkbd_printdebug

When the remove command is executed, the following information is printed in the /var/log/syslog file:

Dec 6 21:40:44 user-VirtualBox kernel: [4308.638084] usbcore: deregistering interface driver usbkbd
Dec 6 21:40:44 user-VirtualBox kernel: [4308.649049] ***Group7*** *usbkbd disconnect*

Then insert the original usbhid module again using the “insmod” command mentioned above.

user@user-VirtualBox:~$ sudo insmod usbhid

Modified Key Mapping

The USB keyboard module code includes a mapping from the signal or keycode produced when a key is pressed to the associated scan code. This mapping can contain up to 256 different scan codes, one for each of the possible key-map combinations. A sample of the original key mapping is in the table below:

Key	Scan Code	Key	Scan Code	Key	Scan Code	Key	Scan Code	Key	Scan Code
ESC	1	eE	18	hH	35	.>	52	NUM	69
1!	2	rR	19	jJ	36	/?	53	SCROLL	70
2@	3	tT	20	kK	37	R SHIFT	54	HOME	71
3#	4	yY	21	lL	38	PRISC	55	UP	72
4$	5	uU	22	;:	39	ALT	56	PGUP	73
0.05	6	iI	23	"	40	SPACE	57	GRAY-	74
6^	7	oO	24	`~	41	CAPS	58	LEFT	75
7&	8	pP	25	L SHIFT	42	F1	59	CENTER	76
8*	9	[{	26	|	43	F2	60	RIGHT	77
9(	10	]}	27	zZ	44	F3	61	GRAY+	78
0)	11	ENTER	28	xX	45	F4	62	END	79
-_	12	CTRL	29	cC	46	F5	63	DOWN	80
=+	13	aA	30	vV	47	F6	64	PGDN	81
BCSP	14	sS	31	bB	48	F7	65	INS	82
TAB	15	dD	32	nN	49	F8	66	DEL	83
qQ	16	fF	33	mM	50	F9	67	F11	87
wW	17	gG	34	,<	51	F10	68	F12	88

To verify the loaded module was being used by the USB keyboard, the key mapping was updated in the module to swap two of the keycodes, the scan code 50 (“mM”) with the scan code 31 (“sS”).

Once the updates were complete and built, the new module, ‘usbkbd-scancodemod’, was able to be inserted in the kernel.

user@user-VirtualBox:~/git/USBKeyboardDriverLinux$ sudo insmod usbkbd-scancodemod.ko

To verify the module has been inserted, an “lsmod” can be performed:

user@user-VirtualBox:~$ lsmod | grep usb
usbkbd_scancodemod 16384 0

When the USB keyboard has been plugged into the system, it will appear in a listing of the “/dev/input/by-path” directory.

user@user-VirtualBox:/dev/input/by-path$ pwd
/dev/input/by-path
user@user-VirtualBox:/dev/input/by-path$ ls
pci-0000:00:04.0-event-mouse platform-i8042-serio-0-event-kbd
pci-0000:00:04.0-mouse platform-i8042-serio-1-event-mouse
pci-0000:00:06.0-usb-0:2:1.0-event-kbd platform-i8042-serio-1-mouse

In this output, the ‘platform-i8042-serio-0-event-kbd’ is the build-in laptop keyboard; therefore, the keystrokes of ‘pci-0000:00:06.0-usb-0:2:1.0-event-kbd’, which is the attached USB keyboard, needed to instead analyzed instead.

user@user-VirtualBox:/dev/input/by-path$ ***sudo hd***
pci-0000\:00\:06.0-usb-0\:2\:1.0-event-kbd
00000000 d9 0f 66 56 00 00 00 00 1d cb 0b 00 00 00 00 00 |..fV............|
00000010 01 00 32 00 01 00 00 00 d9 0f 66 56 00 00 00 00 |..2.......fV....|
00000020 1d cb 0b 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
m*00000030 d9 0f 66 56 00 00 00 00 77 03 0d 00 00 00 00 00 |..fV....w.......|*
00000040 01 00 32 00 00 00 00 00 d9 0f 66 56 00 00 00 00 |..2.......fV....|
00000050 77 03 0d 00 00 00 00 00 00 00 00 00 00 00 00 00 |w...............|
00000060 db 0f 66 56 00 00 00 00 5f f9 0d 00 00 00 00 00 |..fV...._.......|
00000070 01 00 1f 00 01 00 00 00 db 0f 66 56 00 00 00 00 |..........fV....|
00000080 5f f9 0d 00 00 00 00 00 00 00 00 00 00 00 00 00 |_...............|
s*00000090 db 0f 66 56 00 00 00 00 8c 12 0f 00 00 00 00 00 |..fV............|*
000000a0 01 00 1f 00 00 00 00 00 db 0f 66 56 00 00 00 00 |..........fV....|
000000b0 8c 12 0f 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|

We were successfully able to demonstrate pressing the ’s’ or ‘S’ key produced a ‘m’ or ‘M’ (respectively) on the screen and vice versa (‘m’ or ‘M’ keys produced a ‘s’ or ‘S’ on the screen).

Results and Conclusions

Final Analysis

In our project, we first tried to find the distribution of the keys or the key mapping. For this purpose, we used an already existing USB keyboard module and modified the code to fit our requirements. Through our research, we learned the modules loaded by default differ between Linux operating systems and each Linux operating system may embed certain default modules protecting them from being removed or altered. After determining the best OS for our needs, we unloaded the current USB module and loaded our updated module into the kernel for use by the USB keyboard.

We were able to study how the module interprets and maps the key strokes from the USB keyboard to scan codes representing each of the different keys through debug in our updated module. It is important to recognize the default key code mapping for our standard keyboard does not use all the available indexes in the scan code mapping data structure. Those indexes with a value of ‘0’ have no key mapped to it. This allows for the handling of other keyboards that may contain a larger number of keys than our standard keyboard. Through the manipulation of the key/scan code mapping to switch the scan codes of two keys, we were able to demonstrate the knowledge we have gained in this exercise.

Group Roles

All members of the team participated in research, software development, testing, as well as documentation. To ensure each team member was able to take part in each aspect of the project, the following specialized roles were assigned on a rotational basis:

- System Engineer
    - Research USB protocols
    - Research scan code/keycode standards
    - Organize and guide group presentation/demonstration
- Software Developer
    - Implement system design via source code changes
- Software Tester
    - Develop test cases
    - Evaluate performance
- Configuration Management
    - Quality control source code changes
    - Manage software repository
    - Determine best operating systems and virtual machine settings
- Documenter
    - Compile information and results from the group research project into the final research paper

Problems Faced/Lessons Learned

Some of our biggest challenges were due to the Linux operating system we had chosen for development. While CentOS was very useful for homework assignments in class, it took a lot of research and trial-error to try to gain access to the USB module(s). While many of the problems we were able to find an answer for, in the end we were not able to determine how to unload the main USB module and opted to choose a different Linux operating system (Ubuntu) more suited to our needs.

Another smaller problem we faced was trying to remove the module used for the built-in keyboard. We originally assumed it was necessary to remove all USB and/or keyboard related modules in order for us to test the USB keyboard was using our kernel module. To accomplish this we tested whether or not we could disable the use of the built-in keyboard; however, we soon discovered this was not feasible and correctly changed paths to determine which modules should be removed to disable use of an attached USB keyboard.

Text Compare
Produced: 12/6/2015 9:04:18 PM

Mode: All
Left file: C:\Users\J14688\git\USBKeyboardDriverLinux\usbkdb\usbkbd.c
Right file: C:\Users\J14688\git\USBKeyboardDriverLinux\usbkbd-printdebug\usbkbd-printdebug.c

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

#define pr_fmt(fmt) KBUILD_MODNAME ": " fmt

#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>

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

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

static const unsigned char usb_kbd_keycode[256] = {
      0,  0,  0,  0, 30, 48, 46, 32, 18, 33, 34, 35, 23, 36, 37, 38,
     50, 49, 24, 25, 16, 19, 31, 20, 22, 47, 17, 45, 21, 44,  2,  3,
      4,  5,  6,  7,  8,  9, 10, 11, 28,  1, 14, 15, 57, 12, 13, 26,
     27, 43, 43, 39, 40, 41, 51, 52, 53, 58, 59, 60, 61, 62, 63, 64,
     65, 66, 67, 68, 87, 88, 99, 70,119,110,102,104,111,107,109,106,
    105,108,103, 69, 98, 55, 74, 78, 96, 79, 80, 81, 75, 76, 77, 71,
     72, 73, 82, 83, 86,127,116,117,183,184,185,186,187,188,189,190,
    191,192,193,194,134,138,130,132,128,129,131,137,133,135,136,113,
    115,114,  0,  0,  0,121,  0, 89, 93,124, 92, 94, 95,  0,  0,  0,
    122,123, 90, 91, 85,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
      0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
      0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
      0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
      0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
     29, 42, 56,125, 97, 54,100,126,164,166,165,163,161,115,114,113,
    150,158,159,128,136,177,178,176,142,152,173,140
};


/**
 * struct usb_kbd - state of each attached keyboard
 * @dev:    input device associated with this keyboard
 * @usbdev: usb device associated with this keyboard
 * @old:    data received in the past from the @irq URB representing which
 *      keys were pressed. By comparing with the current list of keys
 *      that are pressed, we are able to see key releases.
 * @irq:    URB for receiving a list of keys that are pressed when a
 *      new key is pressed or a key that was pressed is released.
 * @led:    URB for sending LEDs (e.g. numlock, ...)
 * @newleds:    data that will be sent with the @led URB representing which LEDs
 *      should be on
 * @name:   Name of the keyboard. @dev's name field points to this buffer
 * @phys:   Physical path of the keyboard. @dev's phys field points to this
 *      buffer
```

```
 * @new:    Buffer for the @irq URB
 * @cr:     Control request for @led URB
 * @leds:   Buffer for the @led URB
 * @new_dma:    DMA address for @irq URB
 * @leds_dma:   DMA address for @led URB
 * @leds_lock:  spinlock that protects @leds, @newleds, and @led_urb_submitted
 * @led_urb_submitted: indicates whether @led is in progress, i.e. it has been
 *      submitted and its completion handler has not returned yet
 *      without resubmitting @led
 */
struct usb_kbd {
    struct input_dev *dev;
    struct usb_device *usbdev;
    unsigned char old[8];
    struct urb *irq, *led;
    unsigned char newleds;
    char name[128];
    char phys[64];

    unsigned char *new;
    struct usb_ctrlrequest *cr;
    unsigned char *leds;
    dma_addr_t new_dma;
    dma_addr_t leds_dma;

    spinlock_t leds_lock;
    bool led_urb_submitted;

};

static void usb_kbd_irq(struct urb *urb)
{
    struct usb_kbd *kbd = urb->context;
    int i;

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

    for (i = 0; i < 8; i++)
        input_report_key(kbd->dev, usb_kbd_keycode[i + 224], (kbd->new[0] >> i) & 1);

    for (i = 2; i < 8; i++) {

        if (kbd->old[i] > 3 && memscan(kbd->new + 2, kbd->old[i], 6) == kbd->new + 8) {
            if (usb_kbd_keycode[kbd->old[i]])
                input_report_key(kbd->dev, usb_kbd_keycode[kbd->old[i]], 0);
            else
                hid_info(urb->dev,
                     "Unknown key (scancode %#x) released.\n",
                     kbd->old[i]);
        }

        if (kbd->new[i] > 3 && memscan(kbd->old + 2, kbd->new[i], 6) == kbd->old + 8) {
            if (usb_kbd_keycode[kbd->new[i]])
                input_report_key(kbd->dev, usb_kbd_keycode[kbd->new[i]], 1);
            else
                hid_info(urb->dev,
                     "Unknown key (scancode %#x) pressed.\n",
                     kbd->new[i]);
        }
    }

    input_sync(kbd->dev);

    memcpy(kbd->old, kbd->new, 8);

resubmit:
    i = usb_submit_urb (urb, GFP_ATOMIC);
    if (i)
        hid_err(urb->dev, "can't resubmit intr, %s-%s/input0, status %d",
            kbd->usbdev->bus->bus_name,
            kbd->usbdev->devpath, i);
}

static int usb_kbd_event(struct input_dev *dev, unsigned int type,
             unsigned int code, int value)
{
    printk(KERN_INFO "Group7 usbkbd event");
    unsigned long flags;
```

```
170 171     struct usb_kbd *kbd = input_get_drvdata(dev);
171 172
172 173     if (type != EV_LED)
173 174         return -1;
174 175
175 176     spin_lock_irqsave(&kbd->leds_lock, flags);
176 177     kbd->newleds = (!!test_bit(LED_KANA,    dev->led) << 3) | (!!test_bit(LED_COMPOSE, dev->led) << 3) |
177 178                (!!test_bit(LED_SCROLLL, dev->led) << 2) | (!!test_bit(LED_CAPSL,   dev->led) << 1) |
178 179                (!!test_bit(LED_NUML,    dev->led));
179 180
180 181     if (kbd->led_urb_submitted){
181 182         spin_unlock_irqrestore(&kbd->leds_lock, flags);
182 183         return 0;
183 184     }
184 185
185 186     if (*(kbd->leds) == kbd->newleds){
186 187         spin_unlock_irqrestore(&kbd->leds_lock, flags);
187 188         return 0;
188 189     }
189 190
190 191     *(kbd->leds) = kbd->newleds;
191 192
192 193     kbd->led->dev = kbd->usbdev;
193 194     if (usb_submit_urb(kbd->led, GFP_ATOMIC))
194 195         pr_err("usb_submit_urb(leds) failed\n");
195 196     else
196 197         kbd->led_urb_submitted = true;
197 198
198 199     spin_unlock_irqrestore(&kbd->leds_lock, flags);
199 200
200 201     return 0;
201 202 }
202 203
203 204 static void usb_kbd_led(struct urb *urb)
204 205 {
205 206     unsigned long flags;
206 207     struct usb_kbd *kbd = urb->context;
207 208
208 209     if (urb->status)
209 210         hid_warn(urb->dev, "led urb status %d received\n",
210 211              urb->status);
211 212
212 213     spin_lock_irqsave(&kbd->leds_lock, flags);
213 214
214 215     if (*(kbd->leds) == kbd->newleds){
215 216         kbd->led_urb_submitted = false;
216 217         spin_unlock_irqrestore(&kbd->leds_lock, flags);
217 218         return;
218 219     }
219 220
220 221     *(kbd->leds) = kbd->newleds;
221 222
222 223     kbd->led->dev = kbd->usbdev;
223 224     if (usb_submit_urb(kbd->led, GFP_ATOMIC)){
224 225         hid_err(urb->dev, "usb_submit_urb(leds) failed\n");
225 226         kbd->led_urb_submitted = false;
226 227     }
227 228     spin_unlock_irqrestore(&kbd->leds_lock, flags);
228 229
229 230 }
230 231
231 232 static int usb_kbd_open(struct input_dev *dev)
232 233 {
    234     printk(KERN_INFO "Group7 usbkbd open");
233 235     struct usb_kbd *kbd = input_get_drvdata(dev);
234 236
235 237     kbd->irq->dev = kbd->usbdev;
236 238     if (usb_submit_urb(kbd->irq, GFP_KERNEL))
237 239         return -EIO;
238 240
239 241     return 0;
240 242 }
241 243
242 244 static void usb_kbd_close(struct input_dev *dev)
243 245 {
    246     printk(KERN_INFO "Group7 usbkbd close");
244 247     struct usb_kbd *kbd = input_get_drvdata(dev);
245 248
246 249     usb_kill_urb(kbd->irq);
247 250 }
248 251
249 252 static int usb_kbd_alloc_mem(struct usb_device *dev, struct usb_kbd *kbd)
250 253 {
251 254     if (!(kbd->irq = usb_alloc_urb(0, GFP_KERNEL)))
252 255         return -1;
253 256     if (!(kbd->led = usb_alloc_urb(0, GFP_KERNEL)))
254 257         return -1;
255 258     if (!(kbd->new = usb_alloc_coherent(dev, 8, GFP_ATOMIC, &kbd->new_dma)))
```

```
256 259         return -1;
257 260     if (!(kbd->cr = kmalloc(sizeof(struct usb_ctrlrequest), GFP_KERNEL)))
258 261         return -1;
259 262     if (!(kbd->leds = usb_alloc_coherent(dev, 1, GFP_ATOMIC, &kbd->leds_dma)))
260 263         return -1;
261 264
262 265     return 0;
263 266 }
264 267
265 268 static void usb_kbd_free_mem(struct usb_device *dev, struct usb_kbd *kbd)
266 269 {
267 270     usb_free_urb(kbd->irq);
268 271     usb_free_urb(kbd->led);
269 272     usb_free_coherent(dev, 8, kbd->new, kbd->new_dma);
270 273     kfree(kbd->cr);
271 274     usb_free_coherent(dev, 1, kbd->leds, kbd->leds_dma);
272 275 }
273 276
274 277 static int usb_kbd_probe(struct usb_interface *iface,
275 278             const struct usb_device_id *id)
276 279 {
    280     printk(KERN_INFO "Group7 usbkbd probe");
277 281     struct usb_device *dev = interface_to_usbdev(iface);
278 282     struct usb_host_interface *interface;
279 283     struct usb_endpoint_descriptor *endpoint;
280 284     struct usb_kbd *kbd;
281 285     struct input_dev *input_dev;
282 286     int i, pipe, maxp;
283 287     int error = -ENOMEM;
284 288
285 289     interface = iface->cur_altsetting;
286 290
287 291     if (interface->desc.bNumEndpoints != 1)
288 292         return -ENODEV;
289 293
290 294     endpoint = &interface->endpoint[0].desc;
291 295     if (!usb_endpoint_is_int_in(endpoint))
292 296         return -ENODEV;
293 297
294 298     pipe = usb_rcvintpipe(dev, endpoint->bEndpointAddress);
295 299     maxp = usb_maxpacket(dev, pipe, usb_pipeout(pipe));
296 300
297 301     kbd = kzalloc(sizeof(struct usb_kbd), GFP_KERNEL);
298 302     input_dev = input_allocate_device();
299 303     if (!kbd || !input_dev)
300 304         goto fail1;
301 305
302 306     if (usb_kbd_alloc_mem(dev, kbd))
303 307         goto fail2;
304 308
305 309     kbd->usbdev = dev;
306 310     kbd->dev = input_dev;
307 311     spin_lock_init(&kbd->leds_lock);
308 312
309 313     if (dev->manufacturer)
310 314         strlcpy(kbd->name, dev->manufacturer, sizeof(kbd->name));
311 315
312 316     if (dev->product) {
313 317         if (dev->manufacturer)
314 318             strlcat(kbd->name, " ", sizeof(kbd->name));
315 319         strlcat(kbd->name, dev->product, sizeof(kbd->name));
316 320     }
317 321
318 322     if (!strlen(kbd->name))
319 323         snprintf(kbd->name, sizeof(kbd->name),
320 324              "USB HIDBP Keyboard %04x:%04x",
321 325              le16_to_cpu(dev->descriptor.idVendor),
322 326              le16_to_cpu(dev->descriptor.idProduct));
323 327
324 328     usb_make_path(dev, kbd->phys, sizeof(kbd->phys));
325 329     strlcat(kbd->phys, "/input0", sizeof(kbd->phys));
326 330
327 331     input_dev->name = kbd->name;
328 332     input_dev->phys = kbd->phys;
329 333     usb_to_input_id(dev, &input_dev->id);
330 334     input_dev->dev.parent = &iface->dev;
331 335
332 336     input_set_drvdata(input_dev, kbd);
333 337
334 338     input_dev->evbit[0] = BIT_MASK(EV_KEY) | BIT_MASK(EV_LED) |
335 339         BIT_MASK(EV_REP);
336 340     input_dev->ledbit[0] = BIT_MASK(LED_NUML) | BIT_MASK(LED_CAPSL) |
337 341         BIT_MASK(LED_SCROLLL) | BIT_MASK(LED_COMPOSE) |
338 342         BIT_MASK(LED_KANA);
339 343
340 344     for (i = 0; i < 255; i++)
341 345         set_bit(usb_kbd_keycode[i], input_dev->keybit);
342 346     clear_bit(0, input_dev->keybit);
```

```
343 347
344 348     input_dev->event = usb_kbd_event;
345 349     input_dev->open = usb_kbd_open;
346 350     input_dev->close = usb_kbd_close;
347 351
348 352     usb_fill_int_urb(kbd->irq, dev, pipe,
349 353              kbd->new, (maxp > 8 ? 8 : maxp),
350 354              usb_kbd_irq, kbd, endpoint->bInterval);
351 355     kbd->irq->transfer_dma = kbd->new_dma;
352 356     kbd->irq->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;
353 357
354 358     kbd->cr->bRequestType = USB_TYPE_CLASS | USB_RECIP_INTERFACE;
355 359     kbd->cr->bRequest = 0x09;
356 360     kbd->cr->wValue = cpu_to_le16(0x200);
357 361     kbd->cr->wIndex = cpu_to_le16(interface->desc.bInterfaceNumber);
358 362     kbd->cr->wLength = cpu_to_le16(1);
359 363
360 364     usb_fill_control_urb(kbd->led, dev, usb_sndctrlpipe(dev, 0),
361 365                  (void *) kbd->cr, kbd->leds, 1,
362 366                  usb_kbd_led, kbd);
363 367     kbd->led->transfer_dma = kbd->leds_dma;
364 368     kbd->led->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;
365 369
366 370     error = input_register_device(kbd->dev);
367 371     if (error)
368 372         goto fail2;
369 373
370 374     usb_set_intfdata(iface, kbd);
371 375     device_set_wakeup_enable(&dev->dev, 1);
372 376     return 0;
373 377
374 378 fail2:
375 379     usb_kbd_free_mem(dev, kbd);
376 380 fail1:
377 381     input_free_device(input_dev);
378 382     kfree(kbd);
379 383     return error;
380 384 }
381 385
382 386 static void usb_kbd_disconnect(struct usb_interface *intf)
383 387 {
    388     printk(KERN_INFO "Group7 usbkbd disconnect");
384 389     struct usb_kbd *kbd = usb_get_intfdata (intf);
385 390
386 391     usb_set_intfdata(intf, NULL);
387 392     if (kbd) {
388 393         usb_kill_urb(kbd->irq);
389 394         input_unregister_device(kbd->dev);
390 395         usb_kill_urb(kbd->led);
391 396         usb_kbd_free_mem(interface_to_usbdev(intf), kbd);
392 397         kfree(kbd);
393 398     }
394 399 }
395 400
396 401 static struct usb_device_id usb_kbd_id_table [] = {
397 402     { USB_INTERFACE_INFO(USB_INTERFACE_CLASS_HID, USB_INTERFACE_SUBCLASS_BOOT,
398 403         USB_INTERFACE_PROTOCOL_KEYBOARD) },
399 404     { }                     /* Terminating entry */
400 405 };
401 406
402 407 MODULE_DEVICE_TABLE (usb, usb_kbd_id_table);
403 408
404 409 static struct usb_driver usb_kbd_driver = {
405 410     .name =     "usbkbd",
406 411     .probe =    usb_kbd_probe,
407 412     .disconnect =   usb_kbd_disconnect,
408 413     .id_table = usb_kbd_id_table,
409 414 };
410 415
411 416 module_usb_driver(usb_kbd_driver);
```

Text Compare
Produced: 12/7/2015 6:07:15 PM

Mode: All
Left file: C:\Users\J14688\git\USBKeyboardDriverLinux\usbkdb\usbkbd.c
Right file: C:\Users\J14688\git\USBKeyboardDriverLinux\usbkbd-scancodemod\usbkbd-scancodemod.c

```
 1   1 /*
 2   2  *  Copyright (c) 1999-2001 Vojtech Pavlik
 3   3  *
 4   4  *  USB HIDBP Keyboard support
 5   5  */
 6   6
 7   7 /*
 8   8  * This program is free software; you can redistribute it and/or modify
 9   9  * it under the terms of the GNU General Public License as published by
10  10  * the Free Software Foundation; either version 2 of the License, or
11  11  * (at your option) any later version.
12  12  *
13  13  * This program is distributed in the hope that it will be useful,
14  14  * but WITHOUT ANY WARRANTY; without even the implied warranty of
15  15  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
16  16  * GNU General Public License for more details.
17  17  *
18  18  * You should have received a copy of the GNU General Public License
19  19  * along with this program; if not, write to the Free Software
20  20  * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
21  21  *
22  22  * Should you need to contact me, the author, you can do so either by
23  23  * e-mail - mail your message to <vojtech@ucw.cz>, or by paper mail:
24  24  * Vojtech Pavlik, Simunkova 1594, Prague 8, 182 00 Czech Republic
25  25  */
26  26
27  27 #define pr_fmt(fmt) KBUILD_MODNAME ": " fmt
28  28
29  29 #include <linux/kernel.h>
30  30 #include <linux/slab.h>
31  31 #include <linux/module.h>
32  32 #include <linux/init.h>
33  33 #include <linux/usb/input.h>
34  34 #include <linux/hid.h>
35  35
36  36 /*
37  37  * Version Information
38  38  */
39  39 #define DRIVER_VERSION ""
40  40 #define DRIVER_AUTHOR "Vojtech Pavlik <vojtech@ucw.cz>"
41  41 #define DRIVER_DESC "USB HID Boot Protocol keyboard driver"
42  42 #define DRIVER_LICENSE "GPL"
43  43
44  44 MODULE_AUTHOR(DRIVER_AUTHOR);
45  45 MODULE_DESCRIPTION(DRIVER_DESC);
46  46 MODULE_LICENSE(DRIVER_LICENSE);
47  47
    48 /*
    49  * Group7
    50  *
    51  * Swapped the position of elements in the 2nd row.  '50' and '31' have been swapped.
    52  *
    53  * Expectation: USB Keyboard physical 'm' key will become 's' and vice versa.
    54  *
    55  */
    56
48  57 static const unsigned char usb_kbd_keycode[256] = {
    58 //    0,  0,  0,  0, 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
49  59       0,  0,  0,  0, 30, 48, 46, 32, 18, 33, 34, 35, 23, 36, 37, 38,
50      50, 49, 24, 25, 16, 19, 31, 20, 22, 47, 17, 45, 21, 44,  2,  3,
    60 //   50, 49, 24, 25, 16, 19, 31, 20, 22, 47, 17, 45, 21, 44,  2,  3,
    61      31, 49, 24, 25, 16, 19, 50, 20, 22, 47, 17, 45, 21, 44,  2,  3,
51  62       4,  5,  6,  7,  8,  9, 10, 11, 28,  1, 14, 15, 57, 12, 13, 26,
52  63      27, 43, 43, 39, 40, 41, 51, 52, 53, 58, 59, 60, 61, 62, 63, 64,
53  64      65, 66, 67, 68, 87, 88, 99, 70,119,110,102,104,111,107,109,106,
54  65     105,108,103, 69, 98, 55, 74, 78, 96, 79, 80, 81, 75, 76, 77, 71,
55  66      72, 73, 82, 83, 86,127,116,117,183,184,185,186,187,188,189,190,
56  67     191,192,193,194,134,138,130,132,128,129,131,137,133,135,136,113,
57  68     115,114,  0,  0,  0,121,  0, 89, 93,124, 92, 94, 95,  0,  0,  0,
58  69     122,123, 90, 91, 85,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
59  70       0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
60  71       0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
61  72       0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
62  73       0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
63  74      29, 42, 56,125, 97, 54,100,126,164,166,165,163,161,115,114,113,
64  75     150,158,159,128,136,177,178,176,142,152,173,140
65  76 };
66  77
67  78
68  79 /**
69  80  * struct usb_kbd - state of each attached keyboard
```

```
 * @dev:    input device associated with this keyboard
 * @usbdev: usb device associated with this keyboard
 * @old:    data received in the past from the @irq URB representing which
 *      keys were pressed. By comparing with the current list of keys
 *      that are pressed, we are able to see key releases.
 * @irq:    URB for receiving a list of keys that are pressed when a
 *      new key is pressed or a key that was pressed is released.
 * @led:    URB for sending LEDs (e.g. numlock, ...)
 * @newleds:    data that will be sent with the @led URB representing which LEDs
 *      should be on
 * @name:   Name of the keyboard. @dev's name field points to this buffer
 * @phys:   Physical path of the keyboard. @dev's phys field points to this
 *      buffer
 * @new:    Buffer for the @irq URB
 * @cr:     Control request for @led URB
 * @leds:   Buffer for the @led URB
 * @new_dma:    DMA address for @irq URB
 * @leds_dma:   DMA address for @led URB
 * @leds_lock:  spinlock that protects @leds, @newleds, and @led_urb_submitted
 * @led_urb_submitted: indicates whether @led is in progress, i.e. it has been
 *      submitted and its completion handler has not returned yet
 *      without resubmitting @led
 */
struct usb_kbd {
    struct input_dev *dev;
    struct usb_device *usbdev;
    unsigned char old[8];
    struct urb *irq, *led;
    unsigned char newleds;
    char name[128];
    char phys[64];

    unsigned char *new;
    struct usb_ctrlrequest *cr;
    unsigned char *leds;
    dma_addr_t new_dma;
    dma_addr_t leds_dma;

    spinlock_t leds_lock;
    bool led_urb_submitted;

};

static void usb_kbd_irq(struct urb *urb)
{
    struct usb_kbd *kbd = urb->context;
    int i;

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

    for (i = 0; i < 8; i++)
        input_report_key(kbd->dev, usb_kbd_keycode[i + 224], (kbd->new[0] >> i) & 1);

    for (i = 2; i < 8; i++) {

        if (kbd->old[i] > 3 && memscan(kbd->new + 2, kbd->old[i], 6) == kbd->new + 8) {
            if (usb_kbd_keycode[kbd->old[i]])
                input_report_key(kbd->dev, usb_kbd_keycode[kbd->old[i]], 0);
            else
                hid_info(urb->dev,
                     "Unknown key (scancode %#x) released.\n",
                     kbd->old[i]);
        }

        if (kbd->new[i] > 3 && memscan(kbd->old + 2, kbd->new[i], 6) == kbd->old + 8) {
            if (usb_kbd_keycode[kbd->new[i]])
                input_report_key(kbd->dev, usb_kbd_keycode[kbd->new[i]], 1);
            else
                hid_info(urb->dev,
                     "Unknown key (scancode %#x) pressed.\n",
                     kbd->new[i]);
        }
    }

    input_sync(kbd->dev);

    memcpy(kbd->old, kbd->new, 8);

resubmit:
```

```
    i = usb_submit_urb (urb, GFP_ATOMIC);
    if (i)
        hid_err(urb->dev, "can't resubmit intr, %s-%s/input0, status %d",
            kbd->usbdev->bus->bus_name,
            kbd->usbdev->devpath, i);
}

static int usb_kbd_event(struct input_dev *dev, unsigned int type,
             unsigned int code, int value)
{
    unsigned long flags;
    struct usb_kbd *kbd = input_get_drvdata(dev);

    if (type != EV_LED)
        return -1;

    spin_lock_irqsave(&kbd->leds_lock, flags);
    kbd->newleds = (!!test_bit(LED_KANA,    dev->led) << 3) | (!!test_bit(LED_COMPOSE, dev->led) << 3) |
               (!!test_bit(LED_SCROLLL, dev->led) << 2) | (!!test_bit(LED_CAPSL,   dev->led) << 1) |
               (!!test_bit(LED_NUML,    dev->led));

    if (kbd->led_urb_submitted){
        spin_unlock_irqrestore(&kbd->leds_lock, flags);
        return 0;
    }

    if (*(kbd->leds) == kbd->newleds){
        spin_unlock_irqrestore(&kbd->leds_lock, flags);
        return 0;
    }

    *(kbd->leds) = kbd->newleds;

    kbd->led->dev = kbd->usbdev;
    if (usb_submit_urb(kbd->led, GFP_ATOMIC))
        pr_err("usb_submit_urb(leds) failed\n");
    else
        kbd->led_urb_submitted = true;

    spin_unlock_irqrestore(&kbd->leds_lock, flags);

    return 0;
}

static void usb_kbd_led(struct urb *urb)
{
    unsigned long flags;
    struct usb_kbd *kbd = urb->context;

    if (urb->status)
        hid_warn(urb->dev, "led urb status %d received\n",
             urb->status);

    spin_lock_irqsave(&kbd->leds_lock, flags);

    if (*(kbd->leds) == kbd->newleds){
        kbd->led_urb_submitted = false;
        spin_unlock_irqrestore(&kbd->leds_lock, flags);
        return;
    }

    *(kbd->leds) = kbd->newleds;

    kbd->led->dev = kbd->usbdev;
    if (usb_submit_urb(kbd->led, GFP_ATOMIC)){
        hid_err(urb->dev, "usb_submit_urb(leds) failed\n");
        kbd->led_urb_submitted = false;
    }
    spin_unlock_irqrestore(&kbd->leds_lock, flags);

}

static int usb_kbd_open(struct input_dev *dev)
{
    struct usb_kbd *kbd = input_get_drvdata(dev);

    kbd->irq->dev = kbd->usbdev;
    if (usb_submit_urb(kbd->irq, GFP_KERNEL))
        return -EIO;

    return 0;
}

static void usb_kbd_close(struct input_dev *dev)
{
    struct usb_kbd *kbd = input_get_drvdata(dev);

    usb_kill_urb(kbd->irq);
}
```

```
248 259
249 260 static int usb_kbd_alloc_mem(struct usb_device *dev, struct usb_kbd *kbd)
250 261 {
251 262     if (!(kbd->irq = usb_alloc_urb(0, GFP_KERNEL)))
252 263         return -1;
253 264     if (!(kbd->led = usb_alloc_urb(0, GFP_KERNEL)))
254 265         return -1;
255 266     if (!(kbd->new = usb_alloc_coherent(dev, 8, GFP_ATOMIC, &kbd->new_dma)))
256 267         return -1;
257 268     if (!(kbd->cr = kmalloc(sizeof(struct usb_ctrlrequest), GFP_KERNEL)))
258 269         return -1;
259 270     if (!(kbd->leds = usb_alloc_coherent(dev, 1, GFP_ATOMIC, &kbd->leds_dma)))
260 271         return -1;
261 272
262 273     return 0;
263 274 }
264 275
265 276 static void usb_kbd_free_mem(struct usb_device *dev, struct usb_kbd *kbd)
266 277 {
267 278     usb_free_urb(kbd->irq);
268 279     usb_free_urb(kbd->led);
269 280     usb_free_coherent(dev, 8, kbd->new, kbd->new_dma);
270 281     kfree(kbd->cr);
271 282     usb_free_coherent(dev, 1, kbd->leds, kbd->leds_dma);
272 283 }
273 284
274 285 static int usb_kbd_probe(struct usb_interface *iface,
275 286                          const struct usb_device_id *id)
276 287 {
277 288     struct usb_device *dev = interface_to_usbdev(iface);
278 289     struct usb_host_interface *interface;
279 290     struct usb_endpoint_descriptor *endpoint;
280 291     struct usb_kbd *kbd;
281 292     struct input_dev *input_dev;
282 293     int i, pipe, maxp;
283 294     int error = -ENOMEM;
284 295
285 296     interface = iface->cur_altsetting;
286 297
287 298     if (interface->desc.bNumEndpoints != 1)
288 299         return -ENODEV;
289 300
290 301     endpoint = &interface->endpoint[0].desc;
291 302     if (!usb_endpoint_is_int_in(endpoint))
292 303         return -ENODEV;
293 304
294 305     pipe = usb_rcvintpipe(dev, endpoint->bEndpointAddress);
295 306     maxp = usb_maxpacket(dev, pipe, usb_pipeout(pipe));
296 307
297 308     kbd = kzalloc(sizeof(struct usb_kbd), GFP_KERNEL);
298 309     input_dev = input_allocate_device();
299 310     if (!kbd || !input_dev)
300 311         goto fail1;
301 312
302 313     if (usb_kbd_alloc_mem(dev, kbd))
303 314         goto fail2;
304 315
305 316     kbd->usbdev = dev;
306 317     kbd->dev = input_dev;
307 318     spin_lock_init(&kbd->leds_lock);
308 319
309 320     if (dev->manufacturer)
310 321         strlcpy(kbd->name, dev->manufacturer, sizeof(kbd->name));
311 322
312 323     if (dev->product) {
313 324         if (dev->manufacturer)
314 325             strlcat(kbd->name, " ", sizeof(kbd->name));
315 326         strlcat(kbd->name, dev->product, sizeof(kbd->name));
316 327     }
317 328
318 329     if (!strlen(kbd->name))
319 330         snprintf(kbd->name, sizeof(kbd->name),
320 331              "USB HIDBP Keyboard %04x:%04x",
321 332              le16_to_cpu(dev->descriptor.idVendor),
322 333              le16_to_cpu(dev->descriptor.idProduct));
323 334
324 335     usb_make_path(dev, kbd->phys, sizeof(kbd->phys));
325 336     strlcat(kbd->phys, "/input0", sizeof(kbd->phys));
326 337
327 338     input_dev->name = kbd->name;
328 339     input_dev->phys = kbd->phys;
329 340     usb_to_input_id(dev, &input_dev->id);
330 341     input_dev->dev.parent = &iface->dev;
331 342
332 343     input_set_drvdata(input_dev, kbd);
333 344
334 345     input_dev->evbit[0] = BIT_MASK(EV_KEY) | BIT_MASK(EV_LED) |
335 346         BIT_MASK(EV_REP);
```

```
    input_dev->ledbit[0] = BIT_MASK(LED_NUML) | BIT_MASK(LED_CAPSL) |
        BIT_MASK(LED_SCROLLL) | BIT_MASK(LED_COMPOSE) |
        BIT_MASK(LED_KANA);

    for (i = 0; i < 255; i++)
        set_bit(usb_kbd_keycode[i], input_dev->keybit);
    clear_bit(0, input_dev->keybit);

    input_dev->event = usb_kbd_event;
    input_dev->open = usb_kbd_open;
    input_dev->close = usb_kbd_close;

    usb_fill_int_urb(kbd->irq, dev, pipe,
             kbd->new, (maxp > 8 ? 8 : maxp),
             usb_kbd_irq, kbd, endpoint->bInterval);
    kbd->irq->transfer_dma = kbd->new_dma;
    kbd->irq->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;

    kbd->cr->bRequestType = USB_TYPE_CLASS | USB_RECIP_INTERFACE;
    kbd->cr->bRequest = 0x09;
    kbd->cr->wValue = cpu_to_le16(0x200);
    kbd->cr->wIndex = cpu_to_le16(interface->desc.bInterfaceNumber);
    kbd->cr->wLength = cpu_to_le16(1);

    usb_fill_control_urb(kbd->led, dev, usb_sndctrlpipe(dev, 0),
                (void *) kbd->cr, kbd->leds, 1,
                usb_kbd_led, kbd);
    kbd->led->transfer_dma = kbd->leds_dma;
    kbd->led->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;

    error = input_register_device(kbd->dev);
    if (error)
        goto fail2;

    usb_set_intfdata(iface, kbd);
    device_set_wakeup_enable(&dev->dev, 1);
    return 0;

fail2:
    usb_kbd_free_mem(dev, kbd);
fail1:
    input_free_device(input_dev);
    kfree(kbd);
    return error;
}

static void usb_kbd_disconnect(struct usb_interface *intf)
{
    struct usb_kbd *kbd = usb_get_intfdata (intf);

    usb_set_intfdata(intf, NULL);
    if (kbd) {
        usb_kill_urb(kbd->irq);
        input_unregister_device(kbd->dev);
        usb_kill_urb(kbd->led);
        usb_kbd_free_mem(interface_to_usbdev(intf), kbd);
        kfree(kbd);
    }
}

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

MODULE_DEVICE_TABLE (usb, usb_kbd_id_table);

static struct usb_driver usb_kbd_driver = {
    .name =     "usbkbd",
    .probe =    usb_kbd_probe,
    .disconnect =   usb_kbd_disconnect,
    .id_table = usb_kbd_id_table,
};

module_usb_driver(usb_kbd_driver);
```