Keyboard Driver Study

Professor: Ramesh Karne

Objective: To study and understand the keyboard driver in a Linux System (Ubuntu 10).

A lot of processing is required for the kernel to know the correct character. When you press a key the keyboard generates scancodes that then assemble a keycode. Using kernel keymaps the keycodes are converted into TTY (generic terminal handling) input characters. Only after processing can the corresponding character be added to the TTY input buffers and normal 'stty' processing takes place, just as for any other terminal. Our objective for this study is to understand the keyboard driver functions.

For this assignment we will only look at keyboards connected through a USB 2.0 port.

Introduction

Universal Serial Bus (USB) has become today’s standard connection method for peripheral devices such as mice, keyboards, printers, media players, external hard drives and numerous other devices. It was originally designed for personal computers but has become common is other devices as well, such as smart phones, PDAs and video game consoles. USB device communication is based on a connection from the host controller to a logical entity found on a device. Most keyboards sold now are fitted with USB cables.

There are three sets of scan codes which most PC keyboards can produce. Scanmode 2 is the default, in which a key press produces a value s in the range 0x01-0x5f and the corresponding key release produces s+0x80. Scanmode 3 produces a scan code for the key releases of both Shift keys and the left Ctrl and Alt keys and all other keys note only the key press, producing the same as scancode mode 2. Scancode mode 1 uses most of the same key release values as in scanmode 2, but the key presses are entirely different.

A single key press can produce a sequence of up to six scancodes. The kernel has to parse the stream of scancodes and convert it into a series of key press and key release events. Pressing a key provides a unique keycode k in the range 1-127, while releasing if produces a keycode of k + 128. At present, the keycode is the same as the scancode for keys that produce a single scancode in the range of 0x01-0x58.

Since there are 127 values for all possible key presses (0 is an error condition) parsing could result in 1 + 127 + 126 = 254 distinct keycodes. At present, keycodes are restricted to the range 1-127. There are small tables that assign a keycode to a scancode pair 0xe0 s or to a scancode range 0x59-0x7f.

Keycodes are then converted to key symbols by looking them up on the appropriate keymap. There are eight possible modifiers such as Shift keys, Ctrl keys, Caps lock, and others. The keymap used is determined by combination of currently activated modifiers and locks, thus there are 256 possible keymaps.

Implementation:

Setting up Ubuntu 10.10 for use:

A) Download VMWare Server from the VMWare website: www.vmware.com

B) Install VMWare and set it up on a currently existing Windows7 partition.

C) Download the Ubuntu 10.10 x64 installation .ISO from conical (www.ubuntu.com)

D) Mount the ISO into VMWare and install it using the advanced configuration manager for an optimal setup for the host machine.

Set up the Ubuntu 10.10 installation for use:

A) Download and edit the driver that we used (It can be found at http://www.linuxjournal.com/files/linuxjournal.com/linuxjournal/articles/010/1080/1080l1.html)

   a. Because the input was sanitized, first we had to re-add elements, such as greater/less than symbols for the #includes.

   b. Afterwards, we then edited the file with print statements (Prints to stderr) to determine the path the program took during execution.

B) Compile, execute, and install the driver:

   a. gcc ./cmd_kbd

   b. mv ./a.out /usr/bin/cmd_kbd

   c. cmd_kbd enable set_scancode 2 f0 1; showkey –s;

      i. This first enables the driver, then sets the scancode, then starts grabbing keys. Showkey is a pre-installed program that shows what keys are grabbed by the driver.

Results:

Structures:

Command: This holds the information for the command.

--Unsigned char cmd: The hex number for the command

--Int argc: The number of arguments for the command

--Int resct: This appears to be deprecated.

--Char* name: A string representing the name of the command. Used when passing input.

Functions:

send_cmd: This function performs the command given if possible. It does this by sending the commands to /dev/port and that is the interface which is used here.

Int to_hex: This functions turns a command into a hex number.

Int hexd: This function is required for to_hex. It converts characters into hexadecimal numbers.

Problems Faced

Solved

- Scroll lock key lost functionality
    - o Solved by a reboot of the system.
- Understanding the code used
    - o Solved after researching code elements and reworking some unusual coding habits from found code
- After editing a hex value thought to change the keyboard's scancode set, the system crashed.
    - o Problem was solved by fixing the errant code after a reboot.
- Difficulty in understanding the hex values for keycodes (on both press and release)
    - o Internet search provided resources to determine that the values were in fact correct.

Unsolved

- Unresolved dependencies

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/file.h>
//Here is the commands to use
struct
{
	unsigned char cmd;
	int argct, resct;
	char *name;
}
commands[] =
{
	0xed, 1, 0,   "LED",     /* arg 0-7: 1 ScrollLock, 2 NumLock, 4 CapsLock */
	0xee, 0, 1,   "echo",               /* result: ee */
	0xf0, 1, 1, "get_scancodes",          /* arg: 0, result: 43, 41 or 3f */
	0xf0, 1, 0,   "set_scancodes",          /* arg: 1-3 */
	0xf2, 0, 2,   "identify_keyboard",      /* result: ab 41 */
```

```
        0xf3, 1, 0,    "set_repeat_rate",
        0xf4, 0, 0,    "enable",
        0xf5, 0, 0,    "reset_and_disable",
        0xf6, 0, 0,    "reset_and_enable",
        0xfe, 0, 1,    "resend",
        0xff, 0, 1,    "reset_and_selftest"      /* result: aa (OK) / fc (error) */
};
//This is a file pointer for the /dev/port
int fd;
int args_expected;

//Preforms the command given, if possible
send_cmd(unsigned char x)
{
        char z;
        int i;
        //Read through the /dev/port
        fprintf(stderr, "Reading through /dev/port\n");
        do
        {
                lseek( fd, 0x64, 0 );
                read( fd, &z, 1 );
        }
        while ((z & 2) == 2 );  /* wait */
        //Write the command to /dev/port, which sends it to the keyboard
        lseek( fd, 0x60, 0 );
        write( fd, &x, 1 );
        fprintf(stderr, "Writing to /dev/port\n"); //Added by group
        //Check for the args for the command given, if nessessary
        fprintf(stderr, "Dealing with the arguments\n");//Added by group//Added by group
        if (args_expected)
                args_expected--;
        else
        {
                for(i=0; i<sizeof(commands)/sizeof(commands[0]); i++)
                        if(x == commands[i].cmd)
                        {
                                args_expected = commands[i].argct;
                                break;
                        }
        }
        fprintf(stderr, "Command finished.\n");//Added by group
}
//Checks to see if what was given is a hexidecmal number
int hexd(char c)
{
        if ('0' <= c && c <= '9') return(c - '0');
        if ('a' <= c && c <= 'f') return(c - 'a' + 10);
        if ('A' <= c && c <= 'F') return(c - 'A' + 10);
        fprintf(stderr, "kbd_cmd: expected a hex digit, got _%c_ (0%o)\n", c);
        leave(1);
}
//Converts a char to a hex number.
//Deep magicks here.
unsigned char tohex(char *s)
{
        fprintf(stderr, "Converting to hex...\n");//Added by group
        if(!s[0])
```

```
                return(0);
        else if(!s[1])
                return(hexd(s[0]));
        else
                return((hexd(s[0])<<4) + hexd(s[1]));
}

void main(int argc, char **argv)
{
        int i, j;
        //Check for access to /dev/port to access the keyboard
        fprintf(stderr, "Opening /dev/port...\n");//Added by group
        if ( (fd = open("/dev/port", O_RDWR)) < 0)
        {
                perror("Cannot open /dev/port");
                exit(1);
        }
        //If no args are given, then just enable the keyboard.
        if (argc < 2)
        {
                fprintf(stderr, "Just enabling the keyboard.\n");//Added by group
                send_cmd(0xf4);     /* enable */
        }
        //Enable sane to check keymaps
        else if (argc == 2 && !strcmp(argv[1], "sane"))
        {
                fprintf(stderr, "Enabling sane for keymaps\n");//Added by group
                send_cmd(0);     /* Just in case the kbd was waiting */
                /* for the second byte of a command */
                send_cmd(0xf0);     /* Select scancode set */
                send_cmd(0x02);     /* set 2 */
                send_cmd(0xf4);     /* Enable keyboard */
        }
        else
                //Go through all the args
                for (i=1; i<argc; i++)
                {
                        //Wait, if the command was given
                        if (!strcmp(argv[i], "W") || !strcmp(argv[i], "w"))
                        {
                                sleep(1);
                        }
                        //Otherwise send the command, Break if invalid
                        else if (strlen(argv[i]) > 2)
                        {
                                for (j=0; j < sizeof(commands)/sizeof(commands[0]); j++)
                                        if(!strcmp(commands[j].name, argv[i]))
                                        {
                                                send_cmd(commands[j].cmd);
                                                goto fnd;
                                        }
                                        fprintf(stderr, "kbd_cmd: unrecognized command %s\n",
argv[i]);
                                        leave(1);
fnd:;
                        }
                        else
                                //convert to hex then send the command
                                send_cmd(tohex(argv[i]));
                }
                leave(0);
}

leave(int n)
{
        /* prevent frozen keyboards, waiting for command arguments */
        while(args_expected)
                send_cmd(0);
        exit(n);
}
```