

ECE 468/668 Linux Device Driver Lab

Peng Xu

In this lab, you will learn how to write a Linux device driver. As opposed to previous lab on Linux Kernel, in which a full, detailed and step-by-step manual is given, this lab will need you to think more and read more to solve problems. By the end of the lab, you will be a Linux device driver developer and be able to write a device driver from scratch.

To finish this lab smoothly, you are expected to have a moderate understanding about the following topics. If not, at least, you need something for reference while you are working on this lab.

- C programming.
 - Make sure you understand what is pointer, function pointer.
 - References
- Microcomputer organization and interfacing.
 - Get familiar with the following concept again I/O Port, DMA, interrupt.

Kernel module and preparation

A device driver is no more than a piece of program that runs together with the kernel. It is fine to write a C file that compiles into kernel just like what you did in Lab 4. However, compile the whole kernel takes too much time and makes development very inefficient. Moreover, requiring user to recompile kernel whenever a new device is added is quite inconvenient.

To solve these problems, the Linux kernel has a mechanism to load in (insert) a piece of kernel program, which is called “kernel module” or simply “module”, during runtime. The relation of kernel image to kernel modules is analogous to program EXE file to DLLs in Windows and ELF binary file to “*.so” files in Linux. Actually, kernel modules usually bears a “*.ko” name in Linux system.

In Lab 4 in order to trim down the kernel image size, we do not include support for kernel modules in configuration. Now it is time make another kernel image with “Enable loadable module support” in configuration. Check all sub menu items as well, because those options can be handy for module development. Later we need to copy some sound into virtual machine. Hard drive is used to implement it.

DoNothing module

We will start with a kernel module that essentially does nothing just to smooth the steep learning curve. To qualify as a kernel module, an initialization function and a clean-up function need to be implemented. These two functions are called by the kernel when loading in or unloading the modules respectively. By default they are called `init_module` and `cleanup_module`, although it is possible to use two macros, `module_init` and `module_exit` to appoint functions with arbitrary name to serve as initialization and clean-up purpose.

Here I will show the example code for a module that essentially does nothing.

```
#include <linux/module.h> /* all modules need this*/
#include <linux/kernel.h> /* KERN_ALERT */
#include <linux/init.h> /* the macro definition */

static int donothing_init(void)
{
    printk(KERN_ALERT "This is a DoNothing module\n");
    return 0;
}

static void donothing_exit(void)
{
    printk(KERN_ALERT "DoNothing module is unloaded\n");
}

module_init(donothing_init); // donothing_init will be called when loading
                             // this kernel module
module_exit(donothing_exit); // donothing_exit will be called when unloading
                              // this kernel module
```

The example is simple and self-explanatory. At the beginning, necessary files are included. The `donothing_init` prints a message and return 0 signify it is successful. Notice here `KERN_ALERT` is used, which tell the kernel the message has a higher priority and will be displayed on the console directly. Moreover, in Linux kernel, it is a convention to return a 0 for successful and negative numbers for failures. The absolute value of the negative result is the error code. Most error codes are defined in the kernel header file "include/errno.h". The `donothing_exit` prints a message and returns nothing. At the end, the `module_init` and `module_exit` macros appoint the function for initialization and clean-up, respectively.

Make a directory under the kernel source tree (`~/kernel/linux-2.6.32.55`), for example "lab5". Please type in the code above into a file "nothing.c" in lab5 directory.

It is possible to use `gcc` command line to compile this C file into a module binary file (.ko file), however, it is a common practice to use `make` utility to automate the building process in software engineering in Linux. The benefit of using `make` file includes:

- Typing fewer commands for building target file (the file we wanted, the kernel module file in this case).

- Dependencies are automatically managed. Changes in source files will be correctly propagated to the target.
- Sharing configuration and options with other make files.

A makefile defines the rules for building the final file. The makefile for building the DoNothing kernel module is quite simple. It only has one line.

```
obj-m := nothing.o
```

Save the content above in a file named “Makefile” in the lab5 directory. Type the following command to compile the kernel module. Make sure the current directory is lab5.

```
make -C ~/kernel/linux-2.6.32.55 M=$PWD modules
```

The make command will look for “Makefile” and use the definition inside the file to compile the module. However, in this case, we are reusing the Makefile for building the kernel image since it has necessary definition. The option “-C” direct the make utility to look for “Makefile” in directory “~/kernel/linux-2.6.32.55”, which is the directory of kernel source tree. The “modules” directive inform the make utility to build modules and “M=\$PWD” will tell make where the module source code is. The \$PWD will be expended into the current directory and thus it is equivalent to write “M=~/kernel/linux-2.6.32.55/lab6” given the current working directory is “lab6”.

If the Makefile used by make is the one in “~/kernel/linux-2.6.32.55”, why we are writing another? The reason is that the Makefile in “~/kernel/linux-2.6.32.55” will actually insert the content of our Makefile into a general template and compile the kernel module.

If you saw something like this, the module is built successfully. You will see a nothing.ko file in lab5 directory.

```
make[1]: Entering directory `/home/user/kernel/linux-2.6.32.55'
CC [M] /home/user/kernel/linux-2.6.32.55/lab6/nothing.o
Building modules, stage 2.
MODPOST 1 modules
CC /home/user/kernel/linux-2.6.32.55/lab6/nothing.mod.o
LD [M] /home/user/kernel/linux-2.6.32.55/lab6/nothing.ko
make[1]: Leaving directory `/home/user/kernel/linux-2.6.32.55'
user@linuxlab668:~/kernel/linux-2.6.32.55/lab6$
```

It is probably a good choice to write the command line into a script because otherwise you are going to type it so many times in this lab. For more information about Makefile please read

<http://www.opussoftware.com/tutorial/TutMakefile.htm> and

<http://www.ravnborg.org/kbuild/makefiles.html>.

Now we have a module and it is time to load it into kernel. Type the following command:

```
sudo insmod ./nothing.ko
```

The insmod stands for INSert MODule. It will “insert” (load) the module into the kernel. Since it is a command that requires root privilege, “sudo” is used. During the loading process, the initialization

function is called and the message is printed. Kernel modules currently loaded into kernel can be viewed by “lsmod” command.

To unload the kernel module, another command, rmmod, is used. The following command will unload the module and the unload message is displayed because the clean-up function is called.

```
sudo rmmod nothing.ko
```

Check the currently load module again with “lsmod” to confirm it is unloaded.

```
user@linuxlab668:~/kernel/linux-2.6.32.55/lab6$ sudo insmod ./nothing.ko
[144274.496505] This is a DoNothing module
user@linuxlab668:~/kernel/linux-2.6.32.55/lab6$ lsmod
Module                Size  Used by
nothing                602  0
user@linuxlab668:~/kernel/linux-2.6.32.55/lab6$ sudo rmmod nothing
[144281.961420] DoNothing module is unloaded
user@linuxlab668:~/kernel/linux-2.6.32.55/lab6$ lsmod
Module                Size  Used by
user@linuxlab668:~/kernel/linux-2.6.32.55/lab6$ _
```

At this point, you should know how to write a simple kernel module and the commands to load and unload the module.

NOTE: This will be CHECKPOINT 1 for this lab. You need to save all files that are required for this demonstration.

Make it a device

In this section, you will extend the nothing.c to make it appear to be a device.

In Linux, almost everything is a file. This convention starts from UNIX and affects many modern operating systems. A common set of operations are defined on the broader sense files, which include files in the disk and devices such as keyboard, video card, serial cable, etc. Think of the scenario when you have to write a program to read a file. If nothing special is done to the files, the same program can be used to read data from a device, such as a serial cable, without any modification. This is quite convenient for both programmers and users.

All operations can be done to a file is defined in a structure named “file_operations”. The structure is defined in “include/linux/fs.h” in kernel source directory. Each field in “file_operations” is a function pointer pointing to the function that implements the operation. If you do not remember what function pointer is, Google it or read the textbook for C programming.

Here is the file_opeartions structures with only a few fields that will be used in this lab displayed.

```

struct file_operations {
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
    int (*open) (struct inode *, struct file *);
    int (*release) (struct inode *, struct file *);
    ...
};

```

If you still remember function pointers, you will quickly recognize that the `file_operations` is a structure of many function pointers. Each function pointer will point to the function that handles the operation. Each file is associated with a `file_operation` structure of its kind. The operating system will call to the corresponding functions in that associated `file_operation` when requested by the user program.

For example, when you call `fread` on a file in the disk, the operating system will actually call to the `read` function pointer in the `file_operation` associated with that file. This `read` pointer is pointing to a `read` function in a file system implementation, such as Ext2, Ext4 or FAT32, since it is a file on the disk. If the file is actually a device, the function pointers will point to some function in device driver implementation.

In the file operations listed:

- `read` by convention will implement read of a file/device.
 - 1st parameter: `struct file *`, is a pointer to file structure. The file structure is defined in...
 - 2nd parameter: `char *`, pointer to the data buffer
 - 3rd parameter: `size_t`(unsigned int), size of data to be read
 - 4th parameter: `loff_t *(unsigned long long *)` pointer to 64 bit current byte offset.
 - return value: `ssize_t(int)`: number of byte actually read.
- `write` by convention will implement write of a file/device.
 - Parameters and return value definition similar to `read`.
- `open` by convention will implement open of a file.
 - 1st parameter: `struct inode *`, a pointer to inode structure.
 - 2nd parameter: `struct file *`, pointer to a file structure
 - Return value: 0 if success or negative error number if failed.
- `release` by convention will implement close of a file.
 - Parameters and return value definition similar to `open`.

Notice that the parameter list of `read` is different to `fread`. This is due to `fread` is defined as a standard C library function which will be common in many different operating systems. In contrast, `read` is for Linux operating system only although it is heavily modeled after UNIX design. The `fread` will finally call to `read` function of the appropriate `file_operation`.

As we said before, devices in Linux are just files. Thus, it is possible to create a special file in the file system to denote our device. In this way, it is required to define a custom `file_operation` and register it to the kernel so that any access to that special file will be handled by functions in the custom `file_operations`. In Linux, all the special file are in the directory `/dev`.

When creating this special file, it is required to supply a “major number” and “minor number”. One instance of device is uniquely identified by a “major number” and “minor number” combination. The “major number” is used to differentiate type of devices and “minor number” is used to differentiate instance of device of the same type. For example, if you have two IDE hard disks in the computer of the same type, they will have the same “major number” but different “minor number”.

Also, when registering a device in the driver, it is required to supply a major number. The kernel matches the major number and associates the special file with corresponding driver.

There are two types of device in Linux, block device and character device. Block device is referring to devices that are written and read by chunks, such as hard disk or tape. Character device is referring to those written and read by bytes. Character device is simpler and lots of device driver is implemented as character device. In this lab, we are going to use character device only.

Now we can start writing a device driver. We will make a virtual device that has 1 byte storage in it. After this device module is loaded. You can write into the device, and the device will hold one byte.

Before writing the code, we can create the device special file in /dev use the following command.

```
sudo mknod -m666 /dev/onedev c 61 0
```

The mknod command will make the special file. In its parameter, “-m666” set the mode of file to read/write, “/dev/onedev” is the path of the special file, “c” means it is a character device and 61 and 0 are major and minor numbers respectively. The major number 61 is an arbitrary choice. (In Linux, major numbers in range of 60-63, 120-127, 240-254 are for experimental use.)

Now if you are trying to access the special file, since there is no corresponding driver loaded, the kernel will complain.

```
user@linuxlab668:~/kernel/linux-2.6.32.55/lab6$ cat /dev/one
cat: /dev/one: No such device or address
```

Here is the code for the one byte device. Code is commented and you are expected to understand the code by yourself.

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/slab.h>
#include <linux/errno.h>
#include <linux/types.h>
#include <linux/fs.h>
#include <linux/proc_fs.h>
#include <asm/uaccess.h>

#define MAJOR_NUMBER 61

/* forward declaration */
int onebyte_open(struct inode *inode, struct file *filep);
int onebyte_release(struct inode *inode, struct file *filep);
ssize_t onebyte_read(struct file *filep, char *buf,
                    size_t count, loff_t *f_pos);
ssize_t onebyte_write(struct file *filep, const char *buf,
```

```

        size_t count, loff_t *f_pos);

static void onebyte_exit(void);

/* definition of file operation structure */
struct file_operations onebyte_fops = {
    read:    onebyte_read,
    write:   onebyte_write,
    open:    onebyte_open,
    release: onebyte_release
};

char *onebyte_data = NULL;

int onebyte_open(struct inode *inode, struct file *filep)
{
    return 0; // always successful
}

int onebyte_release(struct inode *inode, struct file *filep)
{
    return 0; // always successful
}

ssize_t onebyte_read(struct file *filep, char *buf,
                    size_t count, loff_t *f_pos)
{
    if (*f_pos == 0) {
        // at position 0, need to use copy_to_user
        // because kernel space and user space
        // are different
        copy_to_user(buf, onebyte_data, 1);
        // advance the f_pos
        *f_pos += 1;
        // return 1, since the device only have 1 byte
        return 1;
    } else {
        return 0;
    }
}

ssize_t onebyte_write(struct file *filep, const char *buf,
                    size_t count, loff_t *f_pos)
{
    if (*f_pos == 0) {
        // Use copy_from_user due to pointer are in different
        // address space
        copy_from_user(onebyte_data, buf, 1);
        // advance f_pos
        *f_pos+=1;
        return 1;
    } else {
        // no more space to write
        return -ENOSPC;
    }
}

static int onebyte_init(void)
{
    int result;
    // register the device
    result = register_chrdev(MAJOR_NUMBER, "onebyte", &onebyte_fops);
    if (result < 0) {
        return result;
    }
    // allocate one byte of memory for storage
    // kcalloc is just like malloc, the second parameter is
    // the type of memory to be allocated.
    // To release the memory allocated by kcalloc, use kfree.
    onebyte_data = kcalloc(sizeof(char), GFP_KERNEL);
    if (!onebyte_data) {
        onebyte_exit();
    }
}

```

```

        // cannot allocate memory
        // return no memory error, negative signify a failure
        return -ENOMEM;
    }
    // initialize the value to be X
    *onebyte_data = 'X';
    printk(KERN_ALERT "This is a onebyte device module\n");
    return 0;
}

static void onebyte_exit(void)
{
    // if the pointer is pointing to something
    if (onebyte_data) {
        // free the memory and assign the pointer to NULL
        kfree(onebyte_data);
        onebyte_data = NULL;
    }
    // unregister the device
    unregister_chrdev(MAJOR_NUMBER, "onebyte");
    printk(KERN_ALERT "Onebyte device module is unloaded\n");
}

MODULE_LICENSE("GPL");
module_init(onebyte_init);
module_exit(onebyte_exit);

```

Read the code above, understand what it is doing. Here are some hints:

The function

```
int register_chrdev( unsigned int major_number,
                    const char* dev_name,
                    const struct file_operations* fops);
```

will register a character device and associate `major_number` with the `file_operations` structure. The `dev_name` will show up in system management software to help keep track of the device. Function returns 0 for success and negative numbers otherwise.

The function

```
void unregister_chrdev( unsigned int major_number,
                       const char* dev_name,
```

works the opposite way. It reverses everything done in `register_chrdev`.

Type the code into a file named “onebyte.c” and add one more item into Makefile. The Makefile will look like the following.

```
obj-m := nothing.o onebyte.o
```

Use the same command and to make the kernel module and you will see a “onebyte.ko” file in the directory lab6.

Insert this newly built module and we can do some testing.

```

user@linuxlab668:~/kernel/linux-2.6.32.55/lab6$ cat /dev/one
Xuser@linuxlab668:~/kernel/linux-2.6.32.55/lab6$ printf a>/dev/one
user@linuxlab668:~/kernel/linux-2.6.32.55/lab6$ cat /dev/one
auser@linuxlab668:~/kernel/linux-2.6.32.55/lab6$ printf pqr>/dev/one
-bash: printf: write error: No space left on device
user@linuxlab668:~/kernel/linux-2.6.32.55/lab6$ cat /dev/one
user@linuxlab668:~/kernel/linux-2.6.32.55/lab6$

```

The command `cat` following a file will display the content of that file. The “`printf`” command will output its parameter without change and the output is redirect (“`>`”) to `/dev/one`. Notice that since there is only one byte in this device. The output (high-lighted with red line) do not has a “`\n`” and mixed with the prompt. The example also shows that when more than one character is written, only the first character will be written and then a “no space” error will show. This is expected behavior as the code specified.

NOTE: This will be CHECKPOINT 2 for this lab. You need to save all files that are required for demonstrating read and write of this onebyte kernel device.

Interact with hardware

In this section, we will learn to interact with hardware. If you have ever take classes such as Computer Organization or Microcomputer Interfacing, you should be familiar with the following terms:

- Memory addressing
- I/O port
- Interrupt
- DMA (Direct memory access)

I/O port is similar to memory. You can read from and write to it and achieve information exchange between CPU and the peripheral hardware. However, I/O address space and memory spaces are completely separate in x86 architecture, which means the I/O address `0x10` has nothing to do with memory address `0x10`.

It is designed this way because in the early days of personal computer, memory is a scarce resource. The simple architecture at that time does not allow engineers to remap piece of memory to a different address. If there are 65536 I/O ports mapping to memory address space, 64KB of memory will be wasted, which is not acceptable at that time. Thus, I/O ports are given a different address space and the tradition follows until now.

Accessing I/O ports is as simple as accessing memory. In x86, there are two dedicated assembly instruction, `IN` and `OUT` for input from and output to an I/O port. In Linux, port I/O access is wrapped by functions `in?`, `out?`. The question mark can be `b` (for byte or 8-bit), `w`(for word or 16-bit), `l`(for long word or 32bit)... Here are the prototypes of `inb` and `outb`, I/O for other data type can be easily generalized.

```
unsigned char inb(unsigned int port);
```

```
void outb(unsigned char value, unsigned int port);
```

For example, the following code snippet reads 8-bit port `0x50`, adds the data by `0x10` and writes it back to port `0x51`.

```
unsigned char data;
```

```
data = inb(0x50);
```

```
data +=0x10;
```

```
outb(data, 0x51);
```

Please note that you need to include `asm/io.h` for using the `inb` and `outb`.

Now we know how to read and write ports. But which ports should be read from or write to if we are going to access the Sound Blaster 16 audio card?

Sound Blaster 16 is released before PCI and uses ISA bus originally. Lacking of plug and play support, ISA devices usually requires user to configure the address settings by changing the onboard jumpers. In the virtual machine we use, the base address for Sound Blaster is `0x220`, which means the I/O port address for Sound Blaster will start from `0x220`.

It is time to read the “Sound Blaster Series Hardware Programming Guide”(the Guide) to figure out the exact addresses for different ports on Sound Blaster 16 audio card. The Guide is provided in Lab6 Resource Pack. The file name is “SoundBlasterProgrammingGuide.pdf”.

Please read Chap. 1 of the Guide for an overview of the sound blaster sound card.

Please read first 4 Sections of Chap. 2 (DSP I/O address, Resetting DSP, Read from DSP and Write to DSP) of the Guide for an overview of the sound blaster sound card.

First, quiz yourself (open everything) about the following question. Please write the answer down, they may be useful for you later.

Assume the base address is `0x220`,

1. What is the port for resetting the DSP?
2. What is the port to read data from DSP?
3. Which bit of which port should be checked before reading data from DSP?
4. What is the port to write data to DSP?
5. Which bit of which port should be checked before writing data to DSP?

In the Guide, they have given out the source code for reading from and writing to DSP and code for resetting DSP. Unfortunately, the code is written in x86 assembly language, which I assume most of you do not remember how to interpret. Thus, I will show the C code for DSP read. Please compare the C code with the assembly code and try to write the C code fore write DSP and DSP reset. If you have further questions about interpreting the assembly code, here are some helpful tutorials.

<http://www.drpaulcarter.com/pcasm/>
<http://www.csn.ul.ie/~darkstar/assembler/tut3.html>

The code for DSP read is listed as below.

```
#include <asm/io.h>

#define SB16_BASE          0x220
#define SB16_DSP_READ      (SB16_BASE+0xA)
#define SB16_DSP_RSTATUS  (SB16_BASE+0xE)

#define SB16_RBUF_EMPTY()  (!(inb(SB16_DSP_RSTATUS) & 0x80))

unsigned char read_dsp(void)
{
    while(SB16_RBUF_EMPTY());
    return inb(SB16_DSP_READ);
}
```

Please write the code for DSP write (`write_dsp`) and DSP reset (`reset_dsp`). Defining ports and other numbers with `#define` helps increasing readability of your code and give the code a professional look. Thus, it is highly recommended.

Read the Guide page 6-29. It describes the command for retrieving the version number of the sound card DSP. Use the `read_dsp` and `write_dsp` function to retrieve the version number and print it out on screen (using `printf`). Do not forget you need to reset the DSP for before everything. Implement the code in a function and call that function in the module initialization. The version of DSP in the virtual machine is 4.05. In the other word, the major version number is 4 and minor version number is 5.

Make sure the sound card on the virtual machine is Sound Blaster 16 in the setting page. By default, it may be another type of card and the checkpoint 3 result cannot be reproduced

If you can successfully output the version number as below, you know how to talk to the hardware with your C code now. A big achievement!

```
user@linuxlab668:~/kernel/linux-2.6.32.55/lab6$ sudo insmod ./dspio.ko
[sudo] password for user:
[ 206.245204] Found SB16 card, DSP version: 4.05
[ 206.245241] This is a DoNothing module
user@linuxlab668:~/kernel/linux-2.6.32.55/lab6$ _
```

NOTE: This will be CHECKPOINT 3 for this lab. You need to save all files that are required for demonstrating reading the version number of SB16 onboard DSP.

When you would like to make some sound from the sound card, sound samples are sent to the sound card. The sample values are transformed into electric signals via D/A converter, which is then amplified and use to drive the speaker/earphone.

Our next step is DMA (Direct Memory Access) since DMA is used to send data to sound card. First, learn a fact: using CPU to copy data to I/O port is very inefficient. This is because compiled code for copying data is also in memory and CPU has to read the code while transferring data. Thus, doing it this way cannot fully use the available memory bandwidth. Moreover, many peripherals cannot keep up with the fast speed of CPU and making the CPU sitting there and idling most of time during the copy process.

DMA is invented to solve this very problem. The data transfer using DMA is controlled by a special hardware called DMA controller, which can be seen as a CPU that just knows how to copy data. DMA controller shares the memory bus with CPU by time. The sharing is coordinated with special signallines between CPU and DMA controller.

More detailed description of DMA can be found in textbook of Microcomputer Interfacing or the following places.

- http://en.wikipedia.org/wiki/Direct_memory_access
- <http://www.inversereality.org/files/dmaprogramming.pdf>
- <http://cires.colorado.edu/jimenez-group/QAMSResources/Docs/DMAFundamentals.pdf>

To initialize a DMA transfer, the DMA controller registers has to be setup. It is possible to use the port I/O method to setup DMA controller, although Linux provided wrapper functions to do the same job. For the details about how to use those functions, please read the following page from “DMA for ISA Devices” to the end.

- <http://www.xml.com/ldd/chapter/book/ch13.html>

Notice in the page above, a parameterized function (dad_dma_prepare) for setting up DMA transfer is already supplied. To use the function, the appropriate parameter should be passed.

- Channel: 1
- Mode: DMA_MODE_WRITE
- Buffer address and count will be specified later.

Do not forget to include header file asm/dma.h.

Along with setting up DMA controller, the sound card DSP have to be setup as well to interpret data in the right way. Read the Guide from page 3-2 to 3-5 and page 3-26 for the details. After finish reading, write a function for getting the DSP ready for a DMA 8-bit mono channel 11025Hz sound sample data transfer. Name the function init_dspdma.

Let’s take a look at the big picture of the process for playing sound using sound blaster 16. Suppose the sound samples are now at somewhere in the memory, the data has to be sent to the sound card. The data transfer is done using DMA. The DMA controller has to be initialized first. Then the DSP is setup for receiving 8bit 11025Hz mono channel sound sample data. DSP will automatically starts to play the sound from now until the data transferring finished. The code for playing sound is provided below.

```
void start_play(void)
{
    init_dmactl();           // init dma controller
    init_dspdma();         // init DSP dma setting

    printk("<1> started play!\n");
}
```

The function init_dmactl will initialize the DMA controller and the function init_dspdma will get the DSP in sound card ready for receiving data in the format fixed before. Both function is implemented by yourself.

You may notice there is one important link missing. Where is the sound sample data coming from? It is actually from a file. A program will read in the file and write the content to the device created in last section. The device driver stores the data into a dedicated DMA buffer. When the buffer is filled, it will initialize the DMA transfer.

One thing worth mentioning is that the DMA controller is kind of picky about the buffer. The buffer cannot go over a 64KB boundary and the address should be lower than 16M(0x100000). Thus, a special function will be used to allocate the DMA buffer.

Here is the code for allocating 64K of DMA buffer.

```
dma_buffer = (unsigned char*) get_free_pages(GFP_DMA, 4);
```

Here is the code for freeing the memory allocated.

```
free_pages( (long)dmabuffer, 4);
```

Put the allocation in module initialization and the free in module cleanup. The GFP_DMA flag in the code above will guarantee the returned buffer satisfy DMA requirement. The number 4 means $2^4=16$ pages. With each page 4KB long (x86 architecture default), it is equal to 64KB of buffer.

Now the buffer address and count of the DMA controller initialization is figured out.

- Buffer: dma_buffer
- Count: 64*1024

Interrupt is not mentioned in this section since it is desired to keep this tutorial as simple as possible. The side-effect is that the driver will not get notified when the 64KB data transfer finished and thus cannot play sound segment longer than a few seconds ($64*1024/11025=5.9$ sec). Adding interrupt handling will enable the driver to play continuous sound with arbitrary length. However, it will increase the length of the code by at least 200 lines. If you are interested, talk with the lab TA (me) about it.

Integration and Testing

Integrate what you learn from previous sections and come up with a working sound card driver. Skeleton of the code of driver is listed below for reference.

```
<include header files>
```

```
// forward declarations
int sb16drv_open(struct inode *inode, struct file *filp);
int sb16drv_release(struct inode *inode, struct file *filp);
ssize_t sb16drv_read(struct file *filp, char *buf, size_t count, loff_t *f_pos);
ssize_t sb16drv_write(struct file *filp, const char *buf, size_t count, loff_t *f_pos);
int sb16drv_init(void);
void sb16drv_exit(void);

// the file_operations structure
const struct file_operations sb16drv_fops = {
    read: sb16drv_read,
    write: sb16drv_write,
    open: sb16drv_open,
    release: sb16drv_release
};
```

```

// DMA buffer pointer
unsigned char *dma_buffer = NULL;

// See Manual page 2-4
void write_dsp(unsigned char value)
{
    // implement this function
}

// See Manual page 2-3
unsigned char read_dsp(void)
{
    // implement this function
}

// See Manual page 2-2
int reset_dsp(void)
{
    // implement this function
    // return value should indicate if the reset is successful
    // Hint: mdelay(x); will pause execution for x millisecond
}

int dad_dma_prepare( .... )
{
    // copy the function over
}

// DMA controller configuration
void init_dmactl(void)
{
    // config the dma controller by calling function
    // dad_dma_prepare with appropriate parameters
}

// Setup DMA transfer sampling rate, direction, format and length
// See manual 3-26
void init_dspdma(void)
{
    // implement the procedure described in the Guide
    // hint:
    // High byte of a word x .... ((x&0xff00)>>8)
    // Low byte of a word x .... (x&0xff)
}

// play sound
void start_play(void)
{
    init_dmactl();           // init dma controller
    init_dspdma();         // init DSP dma setting
    printk("<1> started play!\n");
}

// module init
int sb16drv_init() {
    // implement this function
    // This function need following components
    // 1. reset DSP.
    // 2. print out version number of DSP
    // 3. registering the device
    // 4. allocate DMA buffer
    // If any of these three steps fails, goto fail.
    // You many want to print some message to help you debugging.
    printk("<1> Inserted sb16drv module: successful\n");
    return 0;
}

fail:
    printk("<1> Inserted sb16drv module: FAILED \n");
}

```

```

    sb16drv_exit();
    return -1;
}

// module clean up.
void sb16drv_exit() {
    printk(KERN_INFO "sb16drv uninitialized");

    // implement this function
    // This function need following components
    // 1. free DMA buffer if dma_buffer is not NULL.
    // 2. unregistering the device
    // 3. reset DSP.
    printk("<1> Removed sb16drv module\n");
}

// open device, always return success
int sb16drv_open(struct inode *inode, struct file *filp){
    printk("<1> Device opened\n");
    return 0;
}

// close device, always return success
int sb16drv_release(struct inode *inode, struct file *flip){
    printk("<1> Device closed\n");
    return 0;
}

// read device, nothing to read from it.
ssize_t sb16drv_read(struct file *filp, char* buf, size_t count, loff_t *f_pos){
    return 0;
}

// write device use to write data to DMA buffer
ssize_t sb16drv_write(struct file *filp, const char *buf, size_t count, loff_t *f_pos) {
    // Implement this function, please note:
    // 1) *f_pos will indicate the current offset to write
    // 2) count will be the length of the data provided
    // 3) do not forget to use copy_from_user instead of memcpy
    // 4) If the count is more than the space left in the buffer, copy part of
    //    the data and return the number of byte actually copied.
    // 5) If the buffer is filled, e.g. current offset is at 64*1024, call start_play.
}

```

Necessary header files:

```

#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/slab.h>
#include <linux/fs.h>
#include <linux/errno.h>
#include <linux/fcntl.h>
#include <linux/ioport.h>
#include <linux/interrupt.h>
#include <linux/irq.h>
#include <linux/dma-mapping.h>
#include <linux/mm.h>
#include <asm/page.h>
#include <asm/signal.h>
#include <asm/system.h>
#include <asm/uaccess.h>
#include <asm/io.h>
#include <asm/dma.h>

```

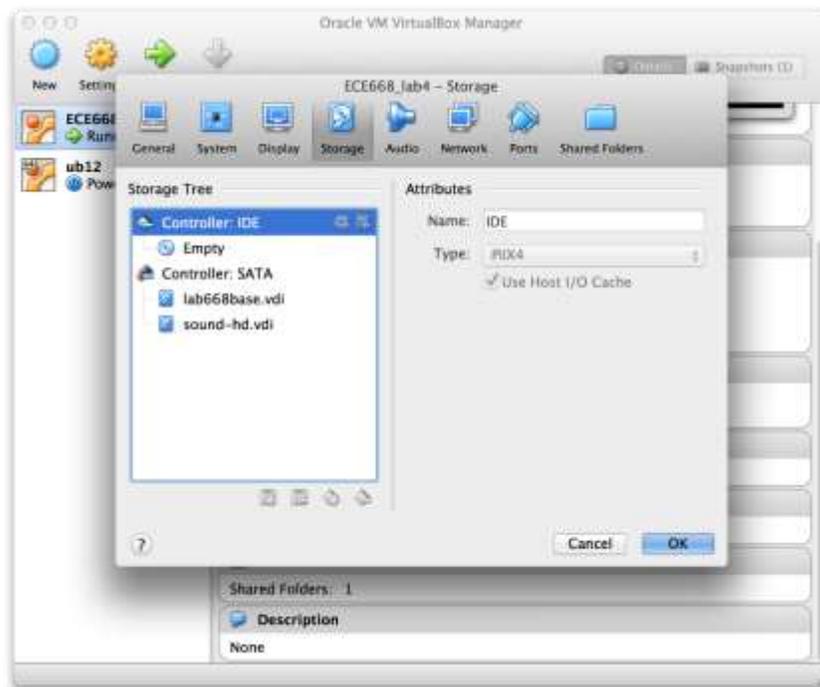
To test the device driver, a piece of program that writes data into the device driver is needed. The program is very simple: read a 64 KB chunk of data from a file and write it into the device file. The better

news is that you do not have to write the program by yourself. Linux comes with lots of utility program that achieve the same job. To send a file named “data” into the device file at “/dev/sb16driver”, just type the following command line

```
cat data>/dev/sb16driver
```

If you followed closely, you will soon found you do not have a playable sound sample files. We are going to use the hard drive to copy sound sample data into the virtual machine.

Install a hard drive in the virtual machine settings page, go to the **Settings** of the virtual machine. In the **Storage** page, write click in the space part of the Storage Tree and select Add a **hard driver**, then add an **Empty** device. Finally, OK to confirm the change.



After that, select the file “sound.img” in Lab 6 Resource Pack.

Then type in the following command to create a mount point. This only has to be done once.

```
sudo mkdir /mnt/hrdisk
```

```
sudo chmod 777 /mnt/hrdisk
```

To mount the hard drive at the mount point (this have to be done in every boot):

```
sudo mount /dev/sdb1 /mnt/hrdisk
```

To un-mount the hard drive, notice the current working directory cannot be inside the mount point directory else you get a file in use error:

```
sudo umount /mnt/hrdisk
```

The sound file linked at the course web site, named

<http://www.ces.clemson.edu/~ahoover/ece468/labs/sound-hd.vdi> , is the hard drive image that

contains the sound file. This file contains the sound sample data in 8 bit, 11025Hz, mono channel format. After you finish writing the driver C code, you need to

- Build the module
- Insert the module using **insmod**
- Create the special file in /dev using **mknod**
- Send the file into the driver file by

```
cat /mnt/hrdisk/sound1.raw > /dev/sb16driver
```

If everything goes well, which rarely happen, you should be able to hear some sound. Do not forget to bring an earphone because computers in the lab do not come with speakers.

NOTE: This will be CHECKPOINT 4 for this lab. You need to save all files that are required for demonstrating making sound with your driver

