Lab 4 Linux Driver

Wen-Lin Sun

2018-10-19

In last lab, we used PXA270's LCD driver module to control the I/O on it. This time, similarly, we will use a driver to control the I/O, but the driver and the device are created by yourself on ZC702. There are several kinds of devices in Linux, such as block devices, character devices, pipe devices and socket devices. In this lab, we will create our own character devices step by step.

1 Environment Setup

Before we start to build our own devices, make sure you have done the environment setup in Lab2. The way we build up driver modules is just the same as the way we build up the boot image. So now, go to the directory where you build up the boot image (maybe `linux-xlnx`), create a directory to place your driver sources, and check the environment variables (PATH and CROSS_COMPILE) before you start this lab.

2 A Basic Driver

A simple driver can be implemented in two lines.

```
1 #include <linux/module.h>
2 MODULE_LICENSE("GPL");
```

And you can compile the source with the Makefile as following.

```
1 CC=arm-linux-gnueabihf-gcc
2
3 obj-m := mydev.o
4
5 all:
6         make -C ../ M=$(PWD) modules
7 clean:
8         make -C ../ M=$(PWD) clean
```

Now, you can just build the driver with the make tool.

```
SHELL> make ARCH=arm
```

In this example, the source file is mydev.c, and the output file is mydev.ko.

3 Initial/Remove Module

We have created a driver module without any functionality so far. In this section, we are going to add two functions to define the actions that the driver will do on its initialization

and remove.

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/kernel.h>
MODULE_LICENSE("GPL");

static int my_init(void) {
   printk("call␣init\n");

   return 0;
}

static void my_exit(void) {
   printk("call␣exit\n");
}

module_init(my_init);
module_exit(my_exit);
```

my_init is the function which will be called when we use insmod to load the driver. Relatively, my_exit is called when we use rmmod to unload the driver. To declare the relationship between the function and the action, the functions module_init and module_exit should be used. So now, after compiling the source, you can test the driver on your board. Try to load and unload it, and check if the behavior is just the same as you defined. By the way, the function printk is the way we use to print out the message in kernel space, instead of printf.

4 Define the Operations

To define the reactions of the actions which may be done on our device, such as, read, write, open, we should use the library linux/fs.h, which defines the basic operations use on file. (Everything is seen as file in Linux, including the devices.) The definition of the file operations in the linux/fs.h are showed as following.

```
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 *);
        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 *);
```

```
16          int (*fsync) (struct file *, loff_t, loff_t, int datasync);
17          int (*aio_fsync) (struct kiocb *, int datasync);
18          int (*fasync) (int, struct file *, int);
19          int (*lock) (struct file *, int, struct file_lock *);
20          ssize_t (*sendpage) (struct file *, struct page *, int, size_t,
                loff_t *, int);
21          unsigned long (*get_unmapped_area)(struct file *, unsigned long,
                unsigned long, unsigned long, unsigned long);
22          int (*check_flags)(int);
23          int (*flock) (struct file *, int, struct file_lock *);
24          ssize_t (*splice_write)(struct pipe_inode_info *, struct file *,
                loff_t *, size_t, unsigned int);
25          ssize_t (*splice_read)(struct file *, loff_t *, struct
                pipe_inode_info *, size_t, unsigned int);
26          int (*setlease)(struct file *, long, struct file_lock **, void **)
                ;
27          long (*fallocate)(struct file *file, int mode, loff_t offset,
28                            loff_t len);
29          void (*show_fdinfo)(struct seq_file *m, struct file *f);
30  #ifndef CONFIG_MMU
31          unsigned (*mmap_capabilities)(struct file *);
32  #endif
33          ssize_t (*copy_file_range)(struct file *, loff_t, struct file *,
34                          loff_t, size_t, unsigned int);
35          int (*clone_file_range)(struct file *, loff_t, struct file *,
                loff_t,
36                          u64);
37          ssize_t (*dedupe_file_range)(struct file *, u64, u64, struct file
                *,
38                          u64);
39  };
```

We don't have to define all the operations. In this example, we define read, write and open.

```
1   #include <linux/init.h>
2   #include <linux/module.h>
3   #include <linux/kernel.h>
4   #include <linux/fs.h>
5   MODULE_LICENSE("GPL");
6
7   // File Operations
8   static ssize_t my_read(struct file *fp, char *buf, size_t count, loff_t *
        fpos) {
9
10      printk("call␣read\n");
11
12      return count;
13  }
14
15  static ssize_t my_write(struct file *fp,const char *buf, size_t count,
        loff_t *fpos) {
16
```

```
17    printk("call␣write\n");
18
19    return count;
20 }
21
22 static int my_open(struct inode *inode, struct file *fp) {
23
24    printk("call␣open\n");
25
26    return 0;
27 }
28
29 struct file_operations my_fops = {
30    read: my_read,
31    write:   my_write,
32    open: my_open
33 };
34
35 #define MAJOR_NUM 244
36 #define DEVICE_NAME "my_dev"
37
38 static int my_init(void) {
39    printk("call␣init\n");
40    if(register_chrdev(MAJOR_NUM, DEVICE_NAME, &my_fops) < 0) {
41          printk("Can␣not␣get␣major␣%d\n", MAJOR_NUM);
42          return (-EBUSY);
43    }
44
45    printk("My␣device␣is␣started␣and␣the␣major␣is␣%d\n", MAJOR_NUM);
46    return 0;
47 }
48
49 static void my_exit(void) {
50    unregister_chrdev(MAJOR_NUM, DEVICE_NAME);
51    printk("call␣exit\n");
52 }
53
54
55 module_init(my_init);
56 module_exit(my_exit);
```

To declare the relationship between, for example, my_read and the read operation, we initialize a structure file_operations called my_fops. So now, we can register a character device with my_fops to the system when the driver is initialized. And don't forget to unregister the character device when the driver is removed.

To see your device under /dev, use should load your module first and get the major number. And then you can create the device node with the major number (in this example is 244).

```
SHELL> mknod /dev/mydev c 244 0
```

The c in the command means character device.

So now, you can design and test your own device and its driver!