Develop a fan driver for Raspberry Pi:
a very simple tutorial for beginners

Bin Zeng [1]

CS @ UMass Lowell
bin_zeng@student.uml.edu

August 21, 2022

[1]Supervised by Joel Savitz ⟨jsavitz@redhat.com⟩.

Acknowledgement

This is a project for my summer break. It has been an exciting project for me. As a beginner, I learned a lot about Linux and Linux device driver in the process. I want to thank Thomas Guma and Joel Savitz for their help. It wouldn't be possible for me to develop the driver without their help.

Thomas started the project and handed over to me. He introduced the basics about the project and showed me how to run Linux on Raspberry Pi. Everything was so interesting and amazing. Thank you Thomas for being so generous on this project.

Most of my thanks goes to Joel. He is an expert on Linux. We have been meeting every other week during the summer break. He taught me a lot about Linux device driver and how to debug problems in Linux. For many details and techniques he showed me, I am pretty sure that I wouldn't learn them anywhere else. Joel was also very patient and answered many of my silly questions, I really appreciate that. It was also a great pleasure to watch him coding live. Although I cannot follow it most of the time, I know this is what I want to be as a software engineer.

There have been other people developing the emc2301 fan driver. Their driver is more professional, please refer to this link.

Chapter 1

Driver is a module

When we develop a Linux device driver, we are adding code and functions to the Linux kernel. A basic rule is that it must be added as a module. A module is a piece of code that can be loaded and unloaded to the Linux kernel, it is the way how the kernel manages the external code dynamically.

Let's make an analogy to understand this. Imagine the kernel is a rail transport system. If we want to ship our goods, we must use a train. Here, the train is the module. It is the container of our goods, which is our code. It is the channel, through which we can integrate our code into the kernel.

1.1 Preparation: the kernel header files

Before we can build a module, we need the kernel header files. They are the source files of the kernel, and when we build the module, we will need to call some functions in these header files.

The Linux we use is *RPI OS 64 bit Lite*. To install the kernel header files, we need to run the command in the terminal:

sudo apt install raspberrypi-kernel-headers

If you are running other versions of Linux, you can also try this command, and see if it works. Otherwise, you have to figure out how to install the kernel header files.

1.2 The module

Let's build a simplest module called *emc2301*. It does not do anything, but we can load and unload it to the kernel.

Step 1: Create a directory called */emc2301*, we will put all the source code here.

Step 2: Create a new file called *emc2301.c*. Type in the following code and save.

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

static int __init emc2301_init(void)
{
   return 0;
}

static void __exit emc2301_exit(void)
{
}

module_init(emc2301_init);
module_exit(emc2301_exit);
MODULE_LICENSE("GPL");
```

A module must have at least the three parts: an initialization function, excecuted when the module is loaded; an exit or cleanup function, executed when the module is unloaded; a statement of the license of the code, where the open-source license, like *GPL*, is preferred.

In our code, we defined two functions *emc2301_init()* and *emc2301_exit()*. Apparently, they are for the module initialization and cleanup. But we need to tell the kernel about them. In order to do this, we use the macros *module_init()* and *module_exit()*. These macros are defined in the header file *linux/module.h*. As for the license statement, we use the macro *MODULE_LICENSE()*, it is also defined in the header file *linux/module.h*.

Step 3: Create the *Makefile* and type in the following code. Note that the file name must be literally *Makefile*, with no extension.

```
obj-m += emc2301.o
PWD := $(CURDIR)

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

Note: for the two lines starting with *make*, the indentation must be a *tab*. If you type multiple spaces instead, it won't work.

The *Makefile* is used to compile the source code *emc2301.c* and create a few new files in other format. Among them is the file with extension *.ko*. It is called the kernel object file and this is the module which we can insert into the kernel.

Step 4: Compile the source code with the *Makefile*.
Make sure we are in the directory */emc2301*. In the terminal, type the command *make*. This will compile the source file according to the instructions in the *Makefile* and create the kernel object file.
Type the command *ls*, you will see a file called *emc2301.ko*. This is the kernel object file we need.

Step5: Insert the module.
Run the command *sudo insmod emc2301.ko*. We need root level permission to insert the module, so we need *sudo*.

Step 6: Check the module is inserted.
Run the command *lsmod*. This will show all the modules loaded in the kernel. On top of the list, you will find the module *emc2301*, which we just loaded.

Step 7: Remove the module.
If you want to unload the module, run the command *sudo rmmod emc2301*. Note that you don't need the extension *.ko* here.

So we have made a simple module. It cannot do anything yet. But it is the container of our code and the channel to the kernel. Starting from here, we can add our code and make it do something.

Chapter 2

A simple driver

2.1 Driver basics

There are different kinds of Linux drivers, such as character driver, block driver and network driver. We will focus on the character driver, which is the most common driver.

2.1.1 Device number

In Linux, each device has a unique ID number. It is stored in a data type called *dev_t*. The size of this data type is 4 bytes, or 32 bits. The 12 most significant bits are called the *major number*, which refers to a specific driver. The 20 least significant bits are called the *minor number*, which refers to a device which uses the driver.

When we create the driver, we must allocate major and minor numbers for the driver and its devices. This can be done with the function *alloc_chrdrv_region()*. The kernel will automatically find the major and minor numbers which are not occupied. This must be done in the module initialization step.

2.1.2 Device file and *cdev* struct

In Linux, the device is treated as a file. Communications with device is treated as read and write system calls. In the /dev folder, there are device files for each device.

We can find the device file for each device in the */dev* folder. Communications with the device is done by reading and writing to the device file. So we must define the functions for the file operations. In the kernel, there is a struct called *file_operations*, and it is used to define the functions of file operations on the device file. These operations include *read*, *write*, *open*, *close*, etc.

When we create a device file, the *file_operations* struct is needed. Their relationship is stored in another struct in the kernel called *cdev*. After we have the device number and defined the *file_operations*, we need to use the functions *cdev_init()* and *cdev_add()* to create the *cdev* struct and store the information. This is also done in the module initialization.

2.1.3 Create device file programmatically

Device file can be created manually in the terminal with the command *mknod*. But here we will introduce how to do it programmatically, so the device file is created when the module is loaded. This needs to be done in two steps:

1. Create a device class with the macro *class_create()*. This is required when we create the device file.
2. Create the device file with the function *device_create()*.

After the module is loaded, we can we the device file created in the */dev* directory with the name which we pass into the *device_create()* function.

Note that it will also create a folder with the name of the device class in the directory of */sys/class*. In this directory, you will find another directory with the name of the device file.

2.2 The code

Putting all these together, here is the new code for the file *emc2301.c*:

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/fs.h>
#include <linux/device.h>
#include <linux/cdev.h>

dev_t dev;
static struct cdev emc2301_cdev;
static struct class *emc2301_class;
struct device *emc2301_device;

// 1. Declare file_operations struct
ssize_t emc2301_read(struct file *file, char __user *user, size_t s,
    loff_t *l)
{
   return 0;
}
ssize_t emc2301_write(struct file *file, const char __user *user, size_t
    s, loff_t *l)
{
```

```
   return 0;
}
static const struct file_operations emc2301_fops = {
   .owner = THIS_MODULE,
   .read = emc2301_read,
   .write = emc2301_write,
};

// 2. Define a module
static int __init emc2301_init(void)
{
   int ret;
   // Allocate device number
   ret = alloc_chrdev_region(&dev, 0, 1, "emc2301");
   // Create cdev struct
   cdev_init(&emc2301_cdev, &emc2301_fops);
   ret = cdev_add(&emc2301_cdev, dev, 1);

   // Create device file
   emc2301_class = class_create(THIS_MODULE, "emc2301_class");
   emc2301_device = device_create(emc2301_class, NULL, dev, NULL,
       "emc2301_device");

   return 0;
}
static void __exit emc2301_exit(void)
{
   device_destroy(emc2301_class, dev);
   class_destroy(emc2301_class);
   cdev_del(&emc2301_cdev);
   unregister_chrdev_region(dev,1);
}

module_init(emc2301_init);
module_exit(emc2301_exit);
MODULE_LICENSE("GPL");
```

Same as before, we need to run the *make* command in the terminal to compile the source code and run *sudo insmod emc2301.ko* to load the module. Note that we don't need to modify the *Makefile*.

We can check that the device file is created. Type *ls /dev* in the terminal to list all the device files, we can find the file *emc2301_device*. This is the device file we created.

We also created a directory */emc2301_class* in */sys/class*. This is created by *class_create()*. In this directory, there is another directory called *emc2301_device*, this is created by *device_create()*. We can check them with the command *ls*.

Chapter 3

Interact with device file

After we create the device file in the folder */dev*, we can interact with it using the system calls *read()* and *write()*. We can also use the command *cat* and *echo* to read and write to it in the terminal. These operations are just like normal file operations, and they will invoke the corresponding functions defined in the *file_operations* struct.

3.1 Print message

A simple way to interact with the device file is to print message in the terminal, so we know that the driver is working. But there are two catches:

1. Linux module runs in the kernel space, while the terminal runs in user space. As these two spaces are isolated, the terminal cannot display the message generated by the module.

2. The module can only use the functions defined in the kernel header files, it cannot use the standard C library. It means the *printf* function does not work in the module. The module has its own print function *pr_info()*, but it prints message in the system log file, not in the terminal.

The way to solve this problem is to use the *copy_to_user()* function. It can copy the data generated by the module, which is located in the kernel space, to the user space, which will display in the terminal. We will show how to do this.

3.2 Use *cat* command in the terminal

Just like we can use *cat* command to show the content of a text file, we can also use *cat* on the device file. It will invoke the *read()* function defined in the *file_operations* struct. We can add code to the *read()* function, so the kernel will communicate with the device and receive message. Then we use the *copy_to_user()* function to display the message in the terminal.

We will show an example without communicating with the device. It just displays message in the terminal.

Step 1: Overwrite *emc2301.c* with the following code:

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/fs.h>
#include <linux/device.h>
#include <linux/cdev.h>

dev_t dev;
static struct cdev emc2301_cdev;
static struct class *emc2301_class;
struct device *emc2301_device;

// 1. Declare file_operations struct
ssize_t emc2301_read(struct file *file, char __user *user, size_t
    s, loff_t *l)
{
   char msg[] = "Hello world, emc2301_read is called!\n";
   copy_to_user(user, msg, sizeof(msg));
   return sizeof(msg);
}
ssize_t emc2301_write(struct file *file, const char __user *user,
    size_t s, loff_t *l)
{
   return 0;
}
static const struct file_operations emc2301_fops = {
   .owner = THIS_MODULE,
   .read = emc2301_read,
   .write = emc2301_write,
};

// 2. Define a module
static int __init emc2301_init(void)
{
   int ret;
   // Allocate device number
   ret = alloc_chrdev_region(&dev, 0, 1, "emc2301");
   // Create cdev struct
   cdev_init(&emc2301_cdev, &emc2301_fops);
   ret = cdev_add(&emc2301_cdev, dev, 1);

   // Create device file
   emc2301_class = class_create(THIS_MODULE, "emc2301_class");
   emc2301_device = device_create(emc2301_class, NULL, dev, NULL,
       "emc2301_device");
```

```
    return 0;
}
static void __exit emc2301_exit(void)
{
    device_destroy(emc2301_class, dev);
    class_destroy(emc2301_class);
    cdev_del(&emc2301_cdev);
    unregister_chrdev_region(dev,1);
}

module_init(emc2301_init);
module_exit(emc2301_exit);
MODULE_LICENSE("GPL");
```

Step 2: Run the command *make* in the terminal to compile the source code, then load the module, same as before.

Step 3: Go to the directory */dev* and find the device file *emc2301_device*.

Step 4: Run the command *cat emc2301_device* in the terminal. We will see the message "*Hello world, emc2301_read is called!*" is printed endlessly.

To stop the endless printing, press *ctrl + c*.

Step 5: Remove the module, run the command *sudo rmmod emc2301*.

3.3 Avoid the endless printing

If we use *cat* command on a text file, the content is only printed once. So apparently there is something wrong with our driver. We need to understand what happens behind the *cat* command.

When we use *cat* command on the device file, the kernel will look up the *file_operations* struct assocated with the device and use the functions defined in it:

- First, the kernel will call the *open()* function.
- Then the kernel will call the *read()* function. If the function returns a positive value, the kernel will print the message copied from the kernel space in the terminal, then call the *read()* function again. If the function returns 0, the kernel will not print anything.
- Once the *read()* function returns 0, the kernel will call the *close()* function. This is the end of the *cat* command.

In the *read()* funciton of our code, we returned the size of the message, which is a positive value. The kernel will print the message in the terminal, then call

the *read()* function again. Not surprising, the return value is positive, so this process will go on and on, and we will see the message is printed endlessly.

To solve the problem, we need the *read()* function to return 0 after the message is printed. So the *read()* function will be called twice. For the first time, it returns a positive value, so the message is printed in the terminal. For the second time, it will return 0, no more message will be printed and it will terminate the *cat* command.

There might be different ways to do it. Here, we will use the parameter *loff_t *l* in the *read()* function. *l* is a pointer and points to an address outside the *read()* function. We can use it to record how many times *read()* is called.

The first time *read()* is called, the value of **l* is 0. We can increment it by 1, and return the value. The kernel will print the message and call *read()* again. Then we will check the value of **l*. If it is non-zero, it indicates *read()* has been called once and message was displayed. Then we will return 0 to prevent more calls to the *read()* function and terminates *cat* command.

Modify the code in section 3.2, just replace the *emc2301_read()* function.

```
ssize_t emc2301_read(struct file *file, char __user *user, size_t s,
    loff_t *l)
{
   char msg[] = "Hello world, emc2301_read is called!\n";
   if (*l == 0) {
      copy_to_user(user, msg, sizeof(msg));
      *l += 1;
      return *l;
   }
   else
      return 0;
}
```

Then repeat the steps in section 3.2. We can see that the message is only printed once. Now the device file is acting as a normal text file with the command *cat*.

Chapter 4

emc2301 on Raspberry Pi

Starting from this chapter, we will build a real driver that can control the fan on a Raspberry Pi. In this chapter, we will give a brief introduction to the hardware and system setup.

4.1 emc2301 fan controller

We are using the Raspberry Pi compute module 4 with the IO board . On the chassis, there is a fan to cool down the CPU. And by default, it runs at max speed. It is actually controlled by an IC chip on the IO board, called *emc2301*.

This chip provides two ways of controlling the fan speed: closed-loop speed control and direct power control. We will use the direct power control in our driver, because it is easier. We will implement two functions in our driver: set the power level (for the write funciton) and read the fan speed (for the read function).

Also, you can see why our driver is called *emc2301* in previous chapters.

4.2 I2C communication

The CPU communicates with the chip *emc2301* by the I2C protocol, which stands for "Inter-Integreted Circuit". Basically, they are connected by two wires, one called the serial data line(SDA), the other called the serial clock line(SCL).

High voltage and low voltage on these lines represent 1 and 0, respectively. The data is sent in a specific format of 0s and 1s, defined by the protocol. Note that, up to 128 devices can be attached to the same SDA and SCL wires. Each of the devices has a unique address, which is included in every data communication.

For the driver development, we care about two things: the I2C bus number and the device address. A pair of SDA and SCL wires is called an I2C bus. There could be many pairs of SDA and SCL wires on the board. To distinguish between them, each pair has a unique bus number. The chip *emc2301* is connected to bus 10. The bus is shown as *i2c-10* in Linux, and it has an alias *i2c_vc*.

The address of chip *emc2301* is 0x2f. These information can be found in the Raspberry Pi datasheet.

4.3 System setup

On Raspberry Pi, the I2C communication is disabled by default when we power it on. To enable the I2C communication, we need to change the configuration file. It is located at */boot*, and the file name is *config.txt*.

Open the file */boot/config.txt* with any text editor, and add one line:

dtparam=i2c_vc=on

Reboot the Pi and the bus *i2c-10* is turned on. To check this, go to the directory */dev* and type the command *ls*. In the device list, you will find *i2c-10*, and that is what we need.

4.4 Direct communication with emc2301

We can directly communicate with *emc2301* in the terminal. This is an alternative way to the device driver.

We need to install the *i2c-tools* package in Linux. Run the command in the terminal:

sudo apt-get install i2c-tools

Then we can use *i2cget* and *i2cset* to communicate with the device. Here is an example to set the fan to max speed:

i2cset 10 0x2f 0x30 0xff

To understand the command,

- *i2cset* means we are sending command to the device.
- *10* means the i2c bus number is 10.
- *0x2f* means the device address on the bus is 0x2f.
- *0x30* is the address of one register on *emc2301*. This controls the how much power is applied to the fan and hence controls the fan speed.

- *0xff* means max power for the fan, which leads to max speed. If we use *0x00* instead, it will set the fan power to zero.

There are many registers on *emc2301*. Each of them represents a status or a parameter of the fan, and has a unique address inside *emc2301*. When we want to read the fan status or set fan parameter, we basically read or write the value of corresponding registers.

For more information, refer to the datasheet of *emc2301*.

Chapter 5

Driver for emc2301

In Chapter 3, we developed a driver to print message in the terminal. Now we will add the communication part to it, so we can receive message from the device and print the message in the terminal. This will make it a real driver.

5.1 Communicate with device in the driver

5.1.1 Create the i2c_client

In Linux kernel, there are dedicated functions for the I2C communications. These functions are wrapped in the header file *linux/i2c.h*. We need to include this header file in our driver.

The core functions for the I2C communications are *i2c_smbus_read_byte_data()* and the *i2c_smbus_write_byte_data()*. They are used to receive message from the device and send command to the device.

These functions require the information of the bus number and the address of the device. These information are wrapped in a class called *i2c_client*. We need to create an instance of this class and pass it to the I2C functions. This is done in two steps:

```
// 1. Create i2c bus
struct i2c_adapter *my_i2c_adap = i2c_get_adapter(10); // 10 means i2c
    bus numer 10

// 2. Create i2c client
struct i2c_client *my_i2c_client = i2c_new_dummy_device(my_adap, 0x2f);
    // 0x2f is the address of emc2301 on the bus
```

The standard way to get the *i2c_client* is from the device tree. In that case, we don't need to hardcode the bus number and address in our driver. But it

requires more steps and is more complicated. We will not cover it here.

5.1.2 Use i2c_client for communication

Like we discussed in section 4.4, communications with the device are basically reading data from and writing data to a specific address in the device, a.k.a. register. We will do the same thing with *i2c_client* and I2C functions.

Here are the examples:

```
u8 ret;
ret = i2c_smbus_read_byte_data(my_i2c_client, 0x3e); // read fan speed,
    register address 0x3e
ret = i2c_smbus_write_byte_data(my_i2c_client, 0x30, 0xff); // set fan
    power to max, register address 0x30
```

5.1.3 Unregister i2c_client

After we finish the communication, we need to unregister the *i2c_client*. Otherwise, the system will mark the device as busy, and it cannot be accessed anymore. This is the code to use:

```
i2c_unregister_device(my_i2c_client);
```

Note: if we use the standard way of the device tree, we don't have to this. The kernel will manage the *i2c_client* automatically.

5.2 Interact with device file: use system calls

Here we will introduce a new way to interact with the device file: use system calls. The system calls like *read()* and *write()* will invoke the functions defined in the *file_operations* struct associated with the driver.

We can put the communications part in the functions of the *file_operations*. Then write a script to interact with the device file, just like the normal text file. We can also pass parameters to the device file and get the return value from the device file.

5.3 Code of emc2301.c

Here is the code for *emc2301.c*:

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

```c
#include <linux/device.h>
#include <linux/cdev.h>
#include <linux/i2c.h>

dev_t dev;
static struct cdev emc2301_cdev;
static struct class *emc2301_class;
struct device *emc2301_device;

// 1. Declare file_operations struct
ssize_t emc2301_read(struct file *file, char __user *user, size_t s,
    loff_t *l)
{
   struct i2c_adapter *my_i2c_adap = i2c_get_adapter(10);
   struct i2c_client *my_i2c_client = i2c_new_dummy_device(my_i2c_adap,
       0x2f);

   int ret = i2c_smbus_read_byte_data(my_i2c_client, 0x3e); // read fan
       speed register
   ret = 122880 / ret; // convert register value to fan speed in RPM

   i2c_unregister_device(my_i2c_client);

   return ret; // return fan speed in RPM
}
ssize_t emc2301_write(struct file *file, const char __user *user, size_t
    s, loff_t *l)
{
   struct i2c_adapter *my_i2c_adap = i2c_get_adapter(10);
   struct i2c_client *my_i2c_client = i2c_new_dummy_device(my_i2c_adap,
       0x2f);

   int ret = i2c_smbus_write_byte_data(my_i2c_client, 0x30, s); // set
       fan power, we use parameter s to represent the power level

   i2c_unregister_device(my_i2c_client);

   return ret;
}
static const struct file_operations emc2301_fops = {
   .owner = THIS_MODULE,
   .read = emc2301_read,
   .write = emc2301_write,
};

// 2. Define a module
static int __init emc2301_init(void)
{
   int ret;
   // Allocate device number
```

```
   ret = alloc_chrdev_region(&dev, 0, 1, "emc2301");
   // Create cdev struct
   cdev_init(&emc2301_cdev, &emc2301_fops);
   ret = cdev_add(&emc2301_cdev, dev, 1);

   // Create device file
   emc2301_class = class_create(THIS_MODULE, "emc2301_class");
   emc2301_device = device_create(emc2301_class, NULL, dev, NULL,
       "emc2301_device");

   return 0;
}
static void __exit emc2301_exit(void)
{
   device_destroy(emc2301_class, dev);
   class_destroy(emc2301_class);
   cdev_del(&emc2301_cdev);
   unregister_chrdev_region(dev,1);
}

module_init(emc2301_init);
module_exit(emc2301_exit);
MODULE_LICENSE("GPL");
```

5.4 Code for test

In this version of driver, we cannot use *cat* command to interact with the device file, because it does not copy message to the user space. Instead, we can write a program and use system calls *read()* and *write()* to interact with the device file.

Here are the code and steps to do this:

Step 1: Create a new file called *fanspeed.c*

Step 2: Copy the following code to the file and save.

```
#include <stdio.h>
#include <sys/ioctl.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
   int ret = 0;
   int my_dev = open("/dev/emc2301_device", O_RDWR);

   switch(*argv[1]) {
```

```
        // set fan power level
        case 's':
            ret = write(my_dev, NULL, atoi(argv[2]));
            break;
        // read fan speed
        case 'g':
            ret = read(my_dev, NULL, 0);
            printf("fan speed is %d RPM.\n", ret);
            break;
    }

    ret = close(my_dev);
    return ret;
}
```

Step 3: Compile the source file and make an executable. Running the command:

gcc fanspeed.c -o fanspeed

5.5 Test the driver

Step 1: Compile the driver source code and insert the module. Follow the steps in previous chapters.

Step 2: Change the permissions of the device file, so we can open it as a regular user. Run the command in the terminal:

sudo chmod 777 /dev/emc2301_device

Step 3: Compile the test code to make the executable, as described in previous section 5.4.

Step 4: Set fan speed to minimum. (It may not stop.) Run the command in the terminal:

./fanspeed s 0

Step 5: Set fan speed to maximum. Run the command in the terminal:

./fanspeed s 255

Step 6: Read the fan speed. Run the command in the terminal:

./fanspeed g

When we set the fan power level, the number should be between 0 and 255. For different power level, we can notice the fan is running at different speed. If we read the fan speed, it also shows different RPMs. So our driver is working.

Now we have developed a simple driver to control the fan on Raspberry Pi.