

# Learning Linux Device Drivers

Jeremiah Mahler (jmmahler@gmail.com)

August 10, 2013

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Hello, World</b>	<b>2</b>
2.1	hello . . . . .	2
2.2	param . . . . .	4
<b>3</b>	<b>Read/Write Data</b>	<b>5</b>
3.1	data_chr . . . . .	5
3.2	data_rw . . . . .	7
3.3	data_sk . . . . .	7
3.4	ioctlx . . . . .	7
3.5	null . . . . .	7
3.6	zero . . . . .	7
<b>4</b>	<b>Sysfs</b>	<b>7</b>
4.1	sysx_file . . . . .	7
4.2	sysx_file2 . . . . .	7
4.3	sysx_group . . . . .	7
4.4	sysx_ktype . . . . .	7
4.5	sysx_ktype2 . . . . .	7
<b>5</b>	<b>Concurrency</b>	<b>7</b>
5.1	fifo_rw . . . . .	7
5.2	fifo_sysfs . . . . .	7
5.3	fifo_xxx . . . . .	7
5.4	fifo_fix . . . . .	7

# 1 Introduction

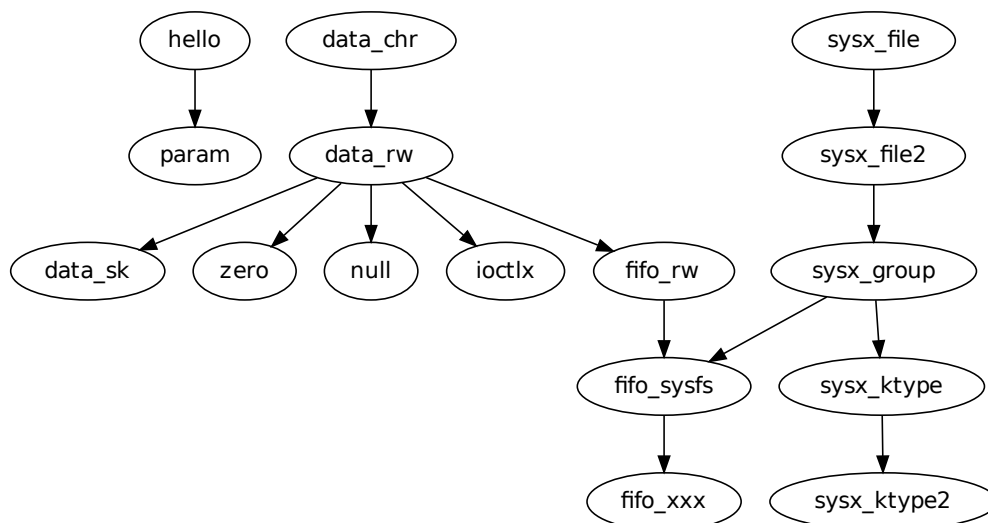


Figure 1: Hierarchy of kernel module examples.

## 2 Hello, World

### 2.1 hello

The `hello` module (Listing 1) simply prints message when it is loaded and unload.

```
hello$ make
(should compile without error, resulting in hello.ko)
hello$ sudo insmod hello.ko
Hello, World
hello$ sudo rmmod hello
Goodbye, cruel world
```

The `module_init` (line 18) and `module_exit` (line 19) tell the kernel which functions to call when this module is loaded (`insmod`) and unloaded (`rmmod`).

The `__init` (line 4) and `__exit` (line 10) are optional hints for the compiler. For example in the case of `__init`, this tells the kernel that it may discard the code after initialization has been completed.

Both the init function (line 4) and the exit function (line 10) are declared `static`. Since these functions are not meant to be used outside the scope of this file, declaring them `static` enforces this constraint.<sup>1</sup>

The `printk` statements are the `printf` of the kernel domain. There are various levels, in this case `KERN_ALERT` is used which will cause the messages to appear on the console. Notice that there is no comma between the level and the message.

<sup>1</sup>J. Corbet, A. Rubini, and G. Kroah-Hartman. *Linux Device Drivers*. O'Reilly Media, 2009. ISBN: 9780596555382, Pg. 52.

```

1 #include <linux/init.h>
2 #include <linux/module.h>
3
4 static int __init hello_init(void)
5 {
6     printk(KERN_ALERT "Hello , World\n");
7     return 0;
8 }
9
10 static void __exit hello_exit(void)
11 {
12     printk(KERN_ALERT "Goodbye, cruel world\n");
13 }
14
15 MODULE_AUTHOR("Jeremiah Mahler <jmmahler@gmail.com>");
16 MODULE_LICENSE("GPL");
17
18 module_init( hello_init );
19 module_exit( hello_exit );

```

Listing 1: Hello, World module in hello/hello.c

The `MODULE_AUTHOR` and `MODULE_LICENSE` on lines 15 and 16 are optional but recommended.<sup>2</sup> There are various other `MODULE_*` options as well (`linux/module.h`).

---

<sup>2</sup>Corbet, Rubini, and Kroah-Hartman, see n. 1, Pg. 51.

## 2.2 param

The `param` module expands upon the `hello` module to take a parameter specifying how many times to print the message.

```
param$ sudo insmod hello.ko howmany=2
Hello, World
Hello, World
param$ sudo rmmod hello
Goodbye, cruel world
Goodbye, cruel world
```

Listing 2 shows the differences between this parameterized hello world module and the previous `hello` module.

```
1  --- ../hello/hello.c      2013-08-09 12:23:58.222416131 -0700
2  +++ hello.c 2013-08-09 12:53:38.082434726 -0700
3  @@ -1,15 +1,28 @@
4  #include <linux/init.h>
5  #include <linux/module.h>
6  +#include <linux/moduleparam.h>
7  +
8  +static int howmany = 1;
9  +module_param(howmany, int, S_IRUGO);
10
11  static int __init hello_init(void)
12  {
13  -   printk(KERN_ALERT "Hello , World\n");
14  +   int i;
15  +
16  +   for (i = 0; i < howmany; i++) {
17  +       printk(KERN_ALERT "Hello , World\n");
18  +   }
19  +
20  +   return 0;
21  }
22
23  static void __exit hello_exit(void)
24  {
25  -   printk(KERN_ALERT "Goodbye, cruel world\n");
26  +   int i;
27  +
28  +   for (i = 0; i < howmany; i++) {
29  +       printk(KERN_ALERT "Goodbye, cruel world\n");
30  +   }
31  }
32
33  MODULE_AUTHOR("Jeremiah Mahler <jmmahler@gmail.com>");
```

Listing 2: `param$ diff -u hello.c ../hello/hello.c`

To use a parameter a global variable has been created named `howmany` on line 8. And on line 9 the `module_param` function is used to tell the kernel about this parameter <sup>3</sup>.

On lines 13-19 and 25-30 it can be seen that the same message is printed `howmany` times.

---

<sup>3</sup>The `module_param` function create a `sysfs` entry in `/sys/module/parameters/howmany`. `sysfs` will be discussed in detail in later modules.

## 3 Read/Write Data

The `data` module allocates a memory from ram which can be read from and written to. This is accomplished as a character device and supports all the usual file operations.

### 3.1 `data_chr`

The first step is to construct the basic infrastructure for a character driver as shown in Listing 3. It doesn't do anything useful but it will simplify the description of upcoming drivers.

The `DEVICE_NAME` (line 8) is just a shortcut for the name which is used in several places.

Lines 10-17 are the global variables that will be used. The `struct data_dev` is the per device structure. Notice that a character device is placed inside.

The `file_operations` (line 19-21) in this case only defines the `.owner`. Upcoming modules will add references to the open, close, read, write, and seek functions to this structure.

`alloc_chrdev_region` (line 26) allocates a major and minor number for the character device.<sup>4</sup> In this case only one major and minor pair is needed.

Functions such as `alloc_chrdev_region` may fail and when they do anything that has been created up to that point must be undone to ensure the kernel is left in a consistent state. A common way this is done is using `goto` statements which branch to different steps in the exit sequence.<sup>5</sup> It can be seen that if `alloc_chrdev_region` fails its `goto` (line 29) will branch to line 66. Since nothing was created up to that point nothing has to be undone.

`class_create` (line 32) establishes a "class" for this module which is also represented in sysfs under `/sys/class/data`. This object will be used later as an argument to `device_create`.

Since the per device structure is just a pointer it must be allocated before it is used (line 32).

To establish the character device it must be initialized and added (lines 41-43). And finally the device is created (line 49). This device will now appear under `/dev/fifo0`.

```
1  #include <linux/cdev.h>
2  #include <linux/device.h>
3  #include <linux/fs.h>
4  #include <linux/module.h>
5  #include <linux/slab.h>
6  #include <linux/uaccess.h>
7
8  #define DEVICENAME "data"
9
10 static dev_t data_major;
11 struct class *data_class;
12 struct device *data_device;
13
14 struct data_dev {
15     struct cdev cdev;
16 } *data_devp;
17
18 struct file_operations data_fops = {
19     .owner = THIS_MODULE,
20 };
21
```

---

<sup>4</sup>Corbet, Rubini, and Kroah-Hartman, see n. 1, Pg. 66.

<sup>5</sup>Ibid., Pg. 53.

```

22 static int __init data_init(void)
23 {
24     int err = 0;
25
26     err = alloc_chrdev_region(&data_major, 0, 1, DEVICENAME);
27     if (err < 0) {
28         printk(KERN_WARNING "Unable to register device\n");
29         goto err_chrdev_region;
30     }
31
32     data_class = class_create(THIS_MODULE, DEVICENAME);
33
34     data_devp = kmalloc(sizeof(struct data_dev), GFP_KERNEL);
35     if (!data_devp) {
36         printk(KERN_WARNING "Unable to kmalloc data_devp\n");
37         err = -ENOMEM;
38         goto err_malloc_data_devp;
39     }
40
41     cdev_init(&data_devp->cdev, &data_fops);
42     data_devp->cdev.owner = THIS_MODULE;
43     err = cdev_add(&data_devp->cdev, data_major, 1);
44     if (err) {
45         printk(KERN_WARNING "cdev_add failed\n");
46         goto err_cdev_add;
47     }
48
49     data_device = device_create(data_class, NULL,
50                               MKDEV(MAJOR(data_major), 0), NULL, "data%d", 0);
51     if (IS_ERR(data_device)) {
52         printk(KERN_WARNING "device_create failed\n");
53         err = PTR_ERR(data_device);
54         goto err_device_create;
55     }
56
57     return 0; /* success */
58
59 err_device_create:
60     cdev_del(&data_devp->cdev);
61 err_cdev_add:
62     kfree(data_devp);
63 err_malloc_data_devp:
64     class_destroy(data_class);
65     unregister_chrdev_region(data_major, 1);
66 err_chrdev_region:
67
68     return err;
69 }
70
71 static void __exit data_exit(void)
72 {
73     device_destroy(data_class, data_major);
74
75     cdev_del(&data_devp->cdev);
76

```

```

77     kfree (data_devp);
78
79     class_destroy (data_class);
80
81     unregister_chrdev_region (data_major , 1);
82 }
83
84 MODULE_AUTHOR("Jeremiah Mahler <jmmahler@gmail.com>");
85 MODULE_LICENSE("GPL");
86
87 module_init (data_init);
88 module_exit (data_exit);

```

Listing 3: data\_chr driver.

### 3.2 data\_rw

### 3.3 data\_sk

### 3.4 ioctlx

### 3.5 null

### 3.6 zero

## 4 Sysfs

### 4.1 sysx\_file

### 4.2 sysx\_file2

### 4.3 sysx\_group

### 4.4 sysx\_ktype

### 4.5 sysx\_ktype2

## 5 Concurrency

### 5.1 fifo\_rw

### 5.2 fifo\_sysfs

### 5.3 fifo\_xxx

### 5.4 fifo\_fix

## References

- Corbet, J., A. Rubini, and G. Kroah-Hartman. *Linux Device Drivers*. O'Reilly Media, 2009. ISBN: 9780596555382.
- Love, R. *Linux Kernel Development*. Developer's Library. Pearson Education, 2010. ISBN: 9780768696790.
- Love, Robert. *Linux System Programming: Talking Directly to the Kernel and C Library*. O'Reilly Media, 2013. ISBN: 9781449341541.
- Venkateswaran, S. *Essential Linux Device Drivers*. Pearson Education, 2008. ISBN: 9780132715812.