Linux Module Programming in Small Diffs

Jeremiah Mahler (jmmahler@gmail.com)

August 9, 2013

Contents

1 Introduction

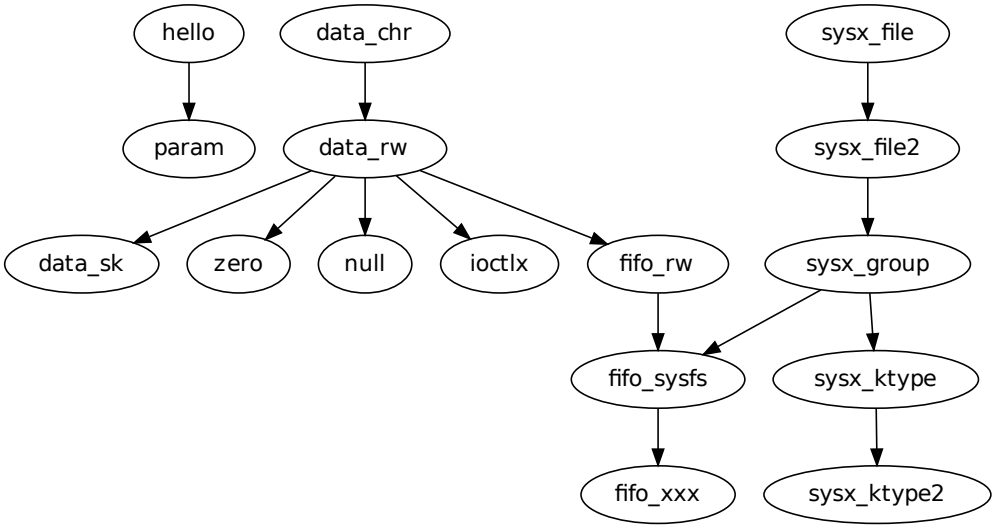

Figure 1: Hierarch of kernel module examples.

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 `__init` and `__exit` on lines 4 and 10 are optional hints for the compiler. For example in the case of `__init` it may discard that code after initiliazation has been completed.

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.

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

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

```
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

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 [1].

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

[1] 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.

The `DEVICE_NAME` on line 8 defines the string which will be used to define the module name in later functions.

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 and references to the open, close, read, write, and seek functions to this structure.

The `data_cleanup` function takes care of unregistering and removing the various that were created during `data_init`. It will be called if module initialization fails or during module removal. Different authors use different styles for cleanup. It is not always done with a separate function. Some use several goto labels each with levels of items to remove [2].

```
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 DEVICE_NAME "data"
9
10 static dev_t data_major;
11 static int cdev_add_done;
12 struct class *data_class;
13 struct device *data_device;
14
15 struct data_dev {
16     struct cdev cdev;
17 } *data_devp;
18
19 struct file_operations data_fops = {
20     .owner = THIS_MODULE,
21 };
22
23 static void data_cleanup(void)
24 {
25     if (data_major) {
26         unregister_chrdev_region(data_major, 1);
27     }
28
29     if (data_device) {
30         device_destroy(data_class, data_major);
31     }
32
33     if (cdev_add_done) {
34         cdev_del(&data_devp->cdev);
35     }
36
```

[2] Recalling the `__init` directive, it might be beneficial use the goto style since this code will be purged after init.

```
37      if (data_devp) {
38          kfree(data_devp);
39      }
40
41      if (data_class) {
42          class_destroy(data_class);
43      }
44  }
45
46  static int __init data_init(void)
47  {
48      int err = 0;
49
50      data_major = 0;
51      data_class = NULL;
52      data_device = NULL;
53      data_devp = NULL;
54      cdev_add_done = 0;
55
56      if (alloc_chrdev_region(&data_major, 0, 1, DEVICE_NAME) < 0) {
57          printk(KERN_WARNING "Unable to register device\n");
58          err = -1;
59          goto err_out;
60      }
61
62      /* populate sysfs entries */
63      /* /sys/class/data/data0/ */
64      data_class = class_create(THIS_MODULE, DEVICE_NAME);
65
66      data_devp = kmalloc(sizeof(struct data_dev), GFP_KERNEL);
67      if (!data_devp) {
68          printk(KERN_WARNING "Unable to kmalloc data_devp\n");
69          err = -ENOMEM;
70          goto err_out;
71      }
72
73      cdev_init(&data_devp->cdev, &data_fops);
74      data_devp->cdev.owner = THIS_MODULE;
75      err = cdev_add(&data_devp->cdev, data_major, 1);
76      if (err) {
77          printk(KERN_WARNING "cdev_add failed\n");
78          goto err_out;
79      } else {
80          cdev_add_done = 1;
81      }
82
83      /* send uevents to udev, so it'll create /dev nodes */
84      /* /dev/data0 */
85      data_device = device_create(data_class, NULL,
86                                  MKDEV(MAJOR(data_major), 0), NULL, "data%d",0);
87
88      return 0;  /* success */
89
90  err_out:
91      data_cleanup();
```

```
92      return err;
93  }
94
95  static void __exit data_exit(void)
96  {
97      data_cleanup();
98  }
99
100 MODULE_AUTHOR("Jeremiah Mahler <jmmahler@gmail.com>");
101 MODULE_LICENSE("GPL");
102
103 module_init(data_init);
104 module_exit(data_exit);
```

Listing 3: Data driver infrastructure.

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.