Tutorial on Linux Device Driver

Programming Embedded Systems

1 Basics

The role of a driver is to provide mechanisms which allows normal user to access protected parts of its system, in particular ports, registers and memory addresses normally managed by the operating system. One of the good features of Linux is the ability to extend at runtime the set of the features offered by the kernel. Users can add or remove functionalities to the kernel while the system is running. These "programs" that can be added to the kernel at runtime are called "module" and built into individual files with `.ko` (Kernel object) extension.

The Linux kernel takes advantages of the possibility to write kernel drivers as modules which can be uploaded on request. This method has different advantages:

- The kernel can be highly modularized, in order to be compatible with the most possible hardware
- A kernel module can be modified without need of recompiling the full kernel

Linux driver modules can be found in: `/lib/modules/<version>/kernel/drivers/` where `<version>` would be the output of the command `"uname -r"` on the system.

In order to write, modify, compile and upload a device driver, the user needs temporarily superuser (`root`) permissions. Module management can be done with four basic shell commands:

Command	Function
`lsmod`	List the currently loaded modules
`insmod module`	Load the module specified by `module`
`modprobe module`	Load the module along with its dependencies
`rmmod module`	Remove/Unload the module

The Linux kernel offers support for different classes of device modules:

- "char" devices are devices that can be accessed as a stream of bytes (like a file).
- "block" devices are accessed by filesystem nodes (example: disks).
- "network" interfaces are able to exchange data with other hosts.

2 Structure of a Simple Driver

A device driver contains at least two functions:

- A function for the module initialization (executed when the module is loaded with the command `"insmod"`)
- A function to exit the module (executed when the module is removed with the command `"rmmod"`)

These two are like normal functions in the driver, except that these are specified as the init and exit functions, respectively, by the macros `module_init()` and `module_exit()`, which are defined in the kernel header `module.h`.

The following code (`module1.c`) shows a very simple kernel module:

```
#include <linux/module.h>
#include <linux/version.h>
#include <linux/kernel.h>

static int __init init_mod(void) /* Constructor */
{
   printk(KERN_INFO "Module1 started...\n");
   return 0;
}

static void __exit end_mod(void) /* Destructor */
{
   printk(KERN_INFO "Module1 ended...\n");
}

module_init(init_mod);
module_exit(end_mod);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Uppsala University");
MODULE_DESCRIPTION("Test Driver Module");
```

Given above is the complete code for our first driver which is a simple module called `module1`. Note that there is no `stdio.h` (a user-space header); instead, we use the analogous `kernel.h` (a kernel space header). `printk()` is the equivalent of `printf()`. Additionally, `version.h` is included for the module version to be compatible with the kernel into which it is going to be loaded. The `MODULE_*` macros populate module-related information, which acts like the module's signature.

Once we have the C code, we can compile it and create the module file `module1.ko`. We use the kernel build system to do this. To build a module/-driver, you need to have the kernel source (or, at least, the kernel headers) installed on your system. You can easily install the Linux kernel headers for currently running kernel version using the following commands at shell prompt. Header files and scripts for building modules for Linux kernel are included in linux-header-YOUR-Kernel-Version package. Open a terminal and type the commands as `root` user to install `linux-headers*` package for your running kernel (tested in a `Debian` distribution):

```
$ sudo apt-get update

$ sudo apt-get install linux-headers-$(uname -r)
```

The kernel source is assumed to be installed at `/usr/src/linux`. If it's at any other location on your system, specify the location in the `KERNEL_SOURCE` variable in the Makefile. The following Makefile invokes the kernel's build system from the kernel source, and the kernel's Makefile will, in turn, invoke our Makefile to build our first driver:

```
# Makefile: makefile of our first driver

# if KERNELRELEASE is defined, we've been invoked from the
# kernel build system and can use its language.

ifneq (${KERNELRELEASE},)
        obj-m := module1.o
# Otherwise we were called directly from the command line.
# Invoke the kernel build system.
else
    KERNEL_SOURCE := /usr/src/linux-headers-3.16.0-4-amd64
    PWD := $(shell pwd)
default:
    ${MAKE} -C ${KERNEL_SOURCE} SUBDIRS=${PWD} modules
clean:
    ${MAKE} -C ${KERNEL_SOURCE} SUBDIRS=${PWD} clean
endif
```

Now the module can be uploaded using the command "insmod" and removed using the command "rmmod".

```
$ insmod ./module1.ko
$ dmesg
$ rmmod module1
$ dmesg
```

The command "dmesg" allows to see the output of the modules, printed using the specific function "printk", corresponding to the function "printf" in user space. At the end of the output of "dmesg" the following lines are shown:

```
$ Module1 started...
$ Module1 ended...
```

3 Char Device Drivers

A physical device, for example a I/O port, can be accessed like a file, using the commands "open", "read", "write" and "close" provided by the prototypes described under `fcntl.h`. First, a device must be configured in the folder `/dev`, providing a class (for example "c" for a char device), a major and a minor number in order to identify it. Following shell command will create a device called `mydev` with major number 240 and minor number 0:

```
$ mknod -m 0666 /dev/mydev c 240 0
```

This command must be run as superuser. The device `/dev/mydev` is created with the as character device ("c"), with the major number 240 and the minor number 0.

The major number is used by the driver initialization for associating a module driver to a specific device. The minor number is used by the device driver programmer to access different functions in the same device. The most important part of the driver is represented by the "file operations" structure, which associates the different operations on the device (open, read, write, close etc) to the implemented ones in the driver. The module must register the programmed functions to the device functions using the "file operations" structure in the initializations procedure. This structure is defined in the `linux/fs.h` prototype.

```
struct file_operations mydev_fops =
{
  owner:      THIS_MODULE,
  open:       mydev_open,
  release:    mydev_release,
  read:       mydev_read,
  write:      mydev_write,
};
```

Next we will give brief description of the four file operation functions mentioned above.

3.1 Open the device

This function is automatically called when the user call for example

```
fd = open("/dev/mydev", O_RDWR);
```

in its user space program. Following is a simple example of this procedure tha only send a simple message to log file.

```
static int mydev_open(struct inode *inode, struct file *filep)
{
   printk("Device opened\n");
   return 0;
}
```

3.2 Close the device

This procedure will be associated to the command

```
close(fd);
```

Following is a simple example of this procedure that only sends a simple message to log file.

```
static int mydev_release(struct inode *inode, struct file *filep)
{
   printk("Device closed\n");
   return 0;
}
```

3.3 Write to the device

This function is associated to the function `write` in the user space program. Using a driver allows to access the hardware or the resource without needing to set particular permissions.

```
static ssize_t mydev_write(struct file *filep, const char *buf,
                           size_t count, loff_t *f_pos)
{
   printk("Writing to the device...\n");
   return 1;
}
```

3.4 Read from the device

This function is assigned to the user space operation `read`.

```
static ssize_t mydev_read(struct file *filep, char *buf,
                          size_t count, loff_t *f_pos)
{
   printk("Reading from the device...\n");
   return 1;
}
```

3.5 Initializing and Closing the driver

The main task of the initialization procedure is to register the driver and to associate it to a specific device. In particular, the function `register chardev` is used to associate the written module to the device with a specific major number, the name of the device. In our example the device will be registered with the major number 240.

```
static int __init init_mydevDriver(void)
{
   int result;
   if ((result = register_chrdev(240, "mydev", &mydev_fops)) < 0)
     goto fail;
   printk("mydev device driver loaded...\n");
   return 0;
   fail:
     return -1;
}
```

This procedure is responsible of unregistering the driver module.

```
static void __exit end_pp(void)
{
   unregister_chrdev(240, "mydev");
   printk("mydev driver unloaded...\n");
}
```

3.6 Sample program for calling the driver functions from user space

The following program can test the generated device and its driver. It writes and reads a byte to and from the mydev character device.

```
#include <fcntl.h>
#include <stdio.h>
int main()
{
  char buffer[1];
  int fd;
  fd=open("/dev/mydev",O_RDWR);
  buffer[0]=0x00;
  write(fd,buffer,1,NULL);
  read(fd,buffer,1,NULL);
  printf("Value : 0x%02x\n",buffer[0]);
  close(fd);
}
```