دانشکده مهندسی
کامپیوتر و فناوری اطلاعات

In The Name of God

دانشگاه صنعتی امیرکبیر
(پلی تکنیک تهران)

OS Lab Final project

“Adding a misc char device driver to The Linux Kernel using kernel modules”

by:

Yasaman Mirmohammad

9431022

Winter 2018

1. Overview

Linux has a monolithic kernel. For this reason, writing a device driver for Linux requires performing a combined compilation with the kernel. Another way around is to implement your driver as a kernel module, in which case you won't need to recompile the kernel to add another driver. We'll be concerned with this second option: kernel modules.

At its base, a module is a specifically designed object file. When working with modules, Linux links them to its kernel by loading them to its address space. The Linux kernel was developed using the C programming language and Assembler. C implements the main part of the kernel, and Assembler implements parts that depend on the architecture. Unfortunately, these are the only two languages we can use for device driver programming in Linux. We cannot use C++, which is used for the Microsoft Windows operating system kernel, because some parts of the Linux kernel source code – header files, to be specific – may include keywords from C++ (for example, `delete` or `new`), while in Assembler we may encounter lexemes such as `' : : '`.

We run the module code in the kernel context. This requires a developer to be very attentive, as it entails extra responsibilities: if a developer makes a mistake when implementing a user-level application, this will not cause problems outside the user application in most cases; but if a developer makes a mistake when implementing a kernel module, the consequences will be problems at the system level. Luckily for us, the Linux kernel has a nice feature of being resistant to errors in module code. When the kernel encounters non-critical errors (for example, null pointer dereferencing), you'll see the `oops` message (insignificant malfunctions during Linux operation are called `oops`), after which the malfunctioning module will be unloaded, allowing the kernel and other modules to work as usual. In addition, you'll be able to find a record in the kernel log that precisely describes this error. But be aware that continuing work after an oops message is not recommended, as doing so may lead to instability and kernel panic.

The kernel and its modules essentially represent a single program module – so keep in mind that a single program module uses a single global namespace. In order to minimize it, you must watch what is being exported by the module: exported global characters must be named uniquely (a commonly used workaround is to simply use the name of the module that's exporting the characters as a prefix) and must be cut to the bare minimum.

What exactly is a kernel module?

Modules are pieces of code that can be loaded and unloaded into the kernel
upon demand. They extend the functionality of the kernel without the need to reboot the system. For example,
one type of module is the device driver, which allows the kernel to access hardware connected to the system.
Without modules, we would have to build monolithic kernels and add new functionality directly into the
kernel image. Besides having larger kernels, this has the disadvantage of requiring us to rebuild and reboot the
kernel every time we want new functionality.

A *character device* typically transfers data to and from a user application — they behave like pipes or serial ports, instantly reading or writing the byte data in a character-by-character stream.

They provide the framework for many typical drivers, such as those that are required for interfacing to serial communications, video capture, and audio devices. The main alternative to a character device is a *block device*. Block devices behave in a similar fashion to regular files, allowing a buffered array of cached data to be viewed or manipulated with operations such as reads, writes, and seeks. Both device types can be accessed through device files that are attached to the file system tree.

For example, the program code that is presented in this article builds to become a device **/dev/oslab**, which appears on your Linux system as follows:

```
molloyd@beaglebone:~/exploringBB/extras/kernel/oslab$ lsmod
Module                  Size  Used by
oslab              2754  0
molloyd@beaglebone:~/exploringBB/extras/kernel/oslab$ ls -l /dev/os*
crw-rw-rwT 1 root root 240, 0 Apr 11 15:34 /dev/oslab
```

Here we describe a straightforward character driver that can be used to pass information between a Linux user-space program and a loadable kernel module (LKM), which is running in Linux kernel space. In this example, a C user-space application sends a string to the LKM.

The LKM then responds with the message that was sent along with the number of letters that the sent message contains

Major and Minor Numbers

Device drivers have an associated *major* and *minor number*.

For example, **/dev/ram0** and **/dev/null** are associated with a driver with major number 1, and **/dev/tty0** and **/dev/ttyS0** are associated with a driver with major number 4. The major number is used by the kernel to identify the correct device driver when the device is accessed. The role of the minor number is device dependent, and is handled internally within the driver. You can see the major/minor number pair for each device if you perform a listing in the **/dev** directory.

For example:

```
molloyd@beaglebone:/dev$ ls -l
crw-rw---T 1 root i2c      89,   0 Jan  1  2000 i2c-0
brw-rw---T 1 root disk      1,   0 Mar  1 20:46 ram0
brw-rw---T 1 root floppy  179,   0 Mar  1 20:46 mmcblk0
crw-rw-rw- 1 root root      1,   3 Mar  1 20:46 null
crw------- 1 root root      4,   0 Mar  1 20:46 tty0
crw-rw---T 1 root dialout   4,  64 Mar  1 20:46 ttyS0
…
```

Character devices are identified by a '**c**' in the first column of a listing, and block devices are identified by a '**b**'. The access permissions, owner, and group of the device is provided for each device. Regular user accounts on the BeagleBone are members of some of these groups and therefore have permissions to use the **i2c-0** and **ttyS0** devices etc.

See:

```
molloyd@beaglebone:/dev$ groups
molloyd dialout cdrom floppy audio video plugdev users i2c spi
```

The device that is developed in this article appears as a device (**/dev/ebbchar**) in the **/dev** directory.

It is possible to manually create a block or character device file entry and later associate it with your device (e.g., **sudo mknod /dev/test c 92 1**), but this approach is prone to problems. One such problem is that you have to ensure that the number you choose (e.g., 92 in this case) is not already in use. On the BeagleBone, you could examine the file **/usr/src/linux-headers-3.8.13-bone70/include/uapi/linux/major.h** for a list of all system device major numbers. However, a device that idenfies a "unique" major number using this approach would not be very portable, as the major number of the device could clash with that of another device on another Linux SBC or Linux distribution. The code that is provided in this article automatically identifies an appropriate major number to use.

The File Operations Data Structure

The file_operations data structure that is defined in /linux/fs.h holds pointers to functions (function pointers) within a driver that allows you to define the behavior of certain file operations. For example, Listing 1 is a segment of the data structure from **/linux/fs.h**. The driver in this article provides an implementation for the read, write, open, and releasesystem call file operations. If you do not provide an implementation for one of the entries in this data structure then it will simply point to NULL, making it inaccessible. Listing 1 is somewhat intimidating, given the number of operations available. However, to build the ebbchar LKM we only need to provide an implementation for four of the entries. Therefore, Listing 1 is provided mainly as a reference that you can use if you need to provide additional functionality within the driver framework.

// Note: __user refers to a user-space address.

The Device Driver Implementation

there is an init() function and an exit() function. However, there are additional file_operations functions that are required for the character device:

- dev_open(): Called each time the device is opened from user space.
- dev_read(): Called when data is sent from the device to user space.
- dev_write(): Called when data is sent from user space to the device.
- dev_release(): Called when the device is closed in user space.

Drivers have a class name and a device name. In Listing 2, **ebb** (Exploring BeagleBone) is used as the class name, and **ebbchar** as the device name. This results in the creation of a device that appears on the file system at **/sys/class/ebb/ebbchar**.

Building and Testing the Linux Kernel Module

A Makefile is required to build the LKM, as provided in Listing 3. This Makefile is very similar to the Makefile in the first article in the series, with the exception that it also builds a user-space C program that interacts with the LKM.

obj-m+=ebbchar.o

```
all:
	make -C /lib/modules/$(shell uname -r)/build/ M=$(PWD) modules
	$(CC) testebbchar.c -o test
clean:
	make -C /lib/modules/$(shell uname -r)/build/ M=$(PWD) clean
	rm test
```

Listing 4 is a short program that requests a string from the user, and writes it to the **/dev/ebbchar** device. After a subsequent key press (ENTER) it then reads the response from the device and displays it in the terminal window.

All going well, the process to build the kernel module should be straightforward, provided that you have installed the Linux headers as described in the first article. The steps are as follows:

```
molloyd@beaglebone:~/exploringBB/extras/kernel/ebbchar$ make
 molloyd@beaglebone:~/exploringBB/extras/kernel/ebbchar$ ls -l *.ko
 -rw-r--r-- 1 molloyd molloyd 7075 Apr  8 19:04 ebbchar.ko
 molloyd@beaglebone:~/exploringBB/extras/kernel/ebbchar$ ls -l test
 -rwxr-xr-x 1 molloyd molloyd 6342 Apr  8 19:23 test
 molloyd@beaglebone:~/exploringBB/extras/kernel/ebbchar$ sudo insmod ebbchar.ko
 molloyd@beaglebone:~/exploringBB/extras/kernel/ebbchar$ lsmod
 Module                  Size  Used by
 ebbchar                 2521  0
```

The device is now present in the **/dev** directory, with the following attributes:

```
molloyd@beaglebone:~/exploringBB/extras/kernel/ebbchar$ cd /dev
 molloyd@beaglebone:/dev$ ls -l ebb*
 crw------- 1 root root 240, 0 Apr  8 19:28 ebbchar
```

You can see that the major number is 240 — this is automatically assigned by the code in Listing 2.

2. Loading and Unloading Modules

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

static int my_init(void)
{
                        return 0;
}

static void my_exit(void)
{
                        return;
}

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

3. Registering a character device

```
int register_chrdev (unsigned int  major,
                     const char *   name,
                     const struct  fops);
                     file_operations  *


struct file_operations {
       struct module  *owner;
       loff_t  (*llseek)  (struct file *, loff_t, int);
       ssize_t  (*read)  (struct file *, char *, size_t, loff_t *);
       ssize_t  (*write)  (struct file *, const char *, size_t, loff_t *);
       int (*readdir)  (struct file *, void *, filldir_t);
       unsigned int (*poll)  (struct file *, struct poll_table_struct *);
       int (*ioctl)  (struct inode *, struct file *, unsigned int, unsigned long
);
       int (*mmap)  (struct file *, struct vm_area_struct *);
       int (*open)  (struct inode *, struct file *);
       int (*flush)  (struct file *);
       int (*release)  (struct inode *, struct file *);
       int (*fsync)  (struct file *, struct dentry *, int datasync);
       int (*fasync)  (int, struct file *, int);
       int (*lock)  (struct file *, int, struct file_lock *);
```

```
    ssize_t (*readv) (struct file *, const struct iovec *, unsigned long,
          loff_t *);
    ssize_t (*writev) (struct file *, const struct iovec *, unsigned long,
          loff_t *);


   };

static struct file_operations simple_driver_fops =
{
    .owner   = THIS_MODULE,
    .read    = device_file_read,
};



    static int device_file_major_number = 0;
static const char device_name[] = "Simple-driver";
static int register_device(void)
{
        int result = 0;
        printk( KERN_NOTICE "Simple-driver: register_device() is called." );
        result = register_chrdev( 0, device_name, &simple_driver_fops );
        if( result < 0 )
        {
            printk( KERN_WARNING "Simple-driver:  can\'t register character d
evice with errorcode = %i", result );
            return result;
        }
        device_file_major_number = result;
        printk( KERN_NOTICE "Simple-driver: registered character device with
major number = %i and minor numbers 0...255"
             , device_file_major_number );
        return 0;
}




void unregister_device(void)
{
    printk( KERN_NOTICE "Simple-driver: unregister_device() is called" );
    if(device_file_major_number != 0)
    {
        unregister_chrdev(device_file_major_number, device_name);
    }
}
```

4. Using Memory Allocated in User Mode

```
ssize_t (*read) (struct file *, char *, size_t, loff_t *);
long copy_to_user( void __user *to, const void * from, unsigned long n );
```

implementing the *read* function:

```
static const char  g_s_Hello_World_string[] = "Hello world from kernel mode!\n\
0";
static const ssize_t g_s_Hello_World_size = sizeof(g_s_Hello_World_string);
static ssize_t device_file_read(
                        struct file *file_ptr
                       , char __user *user_buffer
                       , size_t count
                       , loff_t *position)
{
    printk( KERN_NOTICE "Simple-driver: Device file is read at offset = %i, r
ead bytes count = %u"
                , (int)*position
                , (unsigned int)count );
    /* If position is behind the end of a file we have nothing to read */
    if( *position >= g_s_Hello_World_size )
        return 0;
    /* If a user tries to read more than we have, read only as many bytes as
we have */
    if( *position + count > g_s_Hello_World_size )
        count = g_s_Hello_World_size - *position;
    if( copy_to_user(user_buffer, g_s_Hello_World_string + *position, count)
!= 0 )
        return -EFAULT;
    /* Move reading position */
    *position += count;
    return count;
}
```

5. Build System of a Kernel Module

```
obj-m := source_file_name.o
```

```
obj-m := module_name.o
module_name-objs := source_1.o source_2.o … source_n.o
```

The `make` command initializes the kernel build system:

To build the module:

```
make -C KERNEL_MODULE_BUILD_SYSTEM_FOLDER M=`pwd` modules
```

To clean up the build folder:

```
make -C KERNEL_MODULES_BUILD_SYSTEM_FOLDER M=`pwd` clean
```

The module build system is commonly located in /lib/modules/`uname -r`/build. Now it's time to prepare the module build system. To build the first module, execute the following command from the folder where the build system is located:

```
#> make modules_prepare
```

Finally, we combine everything we've learned into one makefile:

```
TARGET_MODULE:=simple-module
# If we are running by kernel building system
ifneq ($(KERNELRELEASE),)
    $(TARGET_MODULE)-objs := main.o device_file.o
    obj-m := $(TARGET_MODULE).o
# If we running without kernel build system
else
    BUILDSYSTEM_DIR:=/lib/modules/$(shell uname -r)/build
    PWD:=$(shell pwd)
all :
# run kernel build system to make module
    $(MAKE) -C $(BUILDSYSTEM_DIR) M=$(PWD) modules
clean:
# run kernel build system to cleanup in current directory
    $(MAKE) -C $(BUILDSYSTEM_DIR) M=$(PWD) clean
load:
    insmod ./$(TARGET_MODULE).ko
unload:
    rmmod ./$(TARGET_MODULE).ko
endif
```

The `load` target loads the build module and the `unload` target deletes it from the kernel.

In our tutorial, we've used code from main.c and device_file.c to compile the driver. The resulting driver is named simple-module.ko.

6. Loading and Using Module

The following command executed from the source file folder allows us to load the built module:

```
#> make load
```

After executing this command, the name of the driver is added to the /proc/modules file, while the device that the module registers is added to the /proc/devices file. The added records look like this:

```
Character devices: 1 mem 4 tty 4 ttyS … 250 Simple-driver …
```

The first three records contain the name of the added device and the major device number with which it's associated. The minor number range (0–255) allows the device files to be created in the /dev virtual file system.

```
#> mknod /dev/simple-driver c  250 0
```

After we've created the device file, we need to perform the final verification to make sure that what we've done works as expected. To verify, we can use the `cat` command to display the contents:

```
$> cat /dev/simple-driver
Hello world from kernel mode!
```

7. References

- 1. Linux Device Drivers, 3rd Edition by Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman: http://lwn.net/Kernel/LDD3/
- 2. The Linux Kernel Module Programming Guide by Peter Jay Salzman and Ori Pomeranz: http://tldp.org/LDP/lkmpg/2.6/html/lkmpg.html
- Linux Cross Reference http://lxr.free-electrons.com/ident

“The End”