在智能手机时代，每个品牌的手机都有自己的个性特点。正是依靠这种与众不同的个性来吸引用户，营造品牌凝聚力和用户忠城度，典型的代表非 iphone 莫属了。据统计，截止 2011 年 5 月，AppStore 的应用软件数量达 381062 个，位居第一，而 Android Market 的应用软件数量达 294738，紧随 AppStore 后面，并有望在 8 月份越过 AppStore。随着 Android 系统逐步扩大市场占有率，终端设备的多样性亟需更多的移动开发人员的参与。据业内统计，Android 研发人才缺口至少 30 万。目前，对 Android 人才需求一类是偏向硬件驱动的 Android 人才需求，一类是偏向软件应用的 Android 人才需求。总的来说，对有志于从事 Android 硬件驱动的开发工程师来说，现在是一个大展拳脚的机会。那么，就让我们一起来看看如何为 Android 系统编写内核驱动程序吧。

这里，我们不会为真实的硬件设备编写内核驱动程序。为了方便描述为 Android 系统编写内核驱动程序的过程，我们使用一个虚拟的硬件设备，这个设备只有一个 4 字节的寄存器，它可读可写。想起我们第一次学习程序语言时，都喜欢用“Hello, World”作为例子，这里，我们就把这个虚拟的设备命名为“hello”，而这个内核驱动程序也命名为 hello 驱动程序。其实，Android 内核驱动程序和一般 Linux 内核驱动程序的编写方法是一样的，都是以 Linux 模块的形式实现的，具体可参考前面 Android 学习启动篇一文中提到的 Linux Device Drivers 一书。不过，这里我们还是从 Android 系统的角度来描述 Android 内核驱动程序的编写和编译过程。

一. 参照前面两篇文章在 Ubuntu 上下载、编译和安装 Android 最新源代码和在 Ubuntu 上下载、编译和安装 Android 最新内核源代码（Linux Kernel）准备好 Android 内核驱动程序开发环境。

二. 进入到 kernel/common/drivers 目录，新建 hello 目录：

USER-NAME@MACHINE-NAME:~/Android$ cd kernel/common/drivers

USER-NAME@MACHINE-NAME:~/Android/kernel/common/drivers$ mkdir hello

三. 在 hello 目录中增加 hello.h 文件：

view plain

```
1.  #ifndef _HELLO_ANDROID_H_
2.  #define _HELLO_ANDROID_H_
3.
4.  #include <linux/cdev.h>
```

```
#include <linux/semaphore.h>

#define HELLO_DEVICE_NODE_NAME  "hello"
#define HELLO_DEVICE_FILE_NAME  "hello"
#define HELLO_DEVICE_PROC_NAME  "hello"
#define HELLO_DEVICE_CLASS_NAME "hello"

struct hello_android_dev {
    int val;
    struct semaphore sem;
    struct cdev dev;
};

#endif
```

这个头文件定义了一些字符串常量宏，在后面我们要用到。此外，还定义了一个字符设备结构体 hello_android_dev，这个就是我们虚拟的硬件设备了，val 成员变量就代表设备里面的寄存器，它的类型为 int，sem 成员变量是一个信号量，是用同步访问寄存器 val 的，dev 成员变量是一个内嵌的字符设备，这个 Linux 驱动程序自定义字符设备结构体的标准方法。

四.在 hello 目录中增加 hello.c 文件，这是驱动程序的实现部分。驱动程序的功能主要是向上层提供访问设备的寄存器的值，包括读和写。这里，提供了三种访问设备寄存器的方法，一是通过 proc 文件系统来访问，二是通过传统的设备文件的方法来访问，三是通过 devfs 文件系统来访问。下面分段描述该驱动程序的实现。

首先是包含必要的头文件和定义三种访问设备的方法：

view plain

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/types.h>
#include <linux/fs.h>
#include <linux/proc_fs.h>
#include <linux/device.h>
#include <asm/uaccess.h>

#include "hello.h"

/*主设备和从设备号变量*/
static int hello_major = 0;
static int hello_minor = 0;

```

```
/*设备类别和设备变量*/
static struct class* hello_class = NULL;
static struct hello_android_dev* hello_dev = NULL;

/*传统的设备文件操作方法*/
static int hello_open(struct inode* inode, struct file* filp);
static int hello_release(struct inode* inode, struct file* filp);
static ssize_t hello_read(struct file* filp, char __user *buf, size_t count,
    loff_t* f_pos);
static ssize_t hello_write(struct file* filp, const char __user *buf, size_t
    count, loff_t* f_pos);

/*设备文件操作方法表*/
static struct file_operations hello_fops = {
    .owner = THIS_MODULE,
    .open = hello_open,
    .release = hello_release,
    .read = hello_read,
    .write = hello_write,
};

/*访问设置属性方法*/
static ssize_t hello_val_show(struct device* dev, struct device_attribute* a
ttr,  char* buf);
static ssize_t hello_val_store(struct device* dev, struct device_attribute*
attr, const char* buf, size_t count);

/*定义设备属性*/
static DEVICE_ATTR(val, S_IRUGO | S_IWUSR, hello_val_show, hello_val_store);
```

定义传统的设备文件访问方法，主要是定义 hello_open、hello_release、hello_read 和 hello_write 这四个打开、释放、读和写设备文件的方法：

view plain

```
/*打开设备方法*/
static int hello_open(struct inode* inode, struct file* filp) {
    struct hello_android_dev* dev;

    /*将自定义设备结构体保存在文件指针的私有数据域中，以便访问设备时拿来用*/
    dev = container_of(inode->i_cdev, struct hello_android_dev, dev);
    filp->private_data = dev;

```

```
9.      return 0;
10. }
11.
12. /*设备文件释放时调用，空实现*/
13. static int hello_release(struct inode* inode, struct file* filp) {
14.     return 0;
15. }
16.
17. /*读取设备的寄存器 val 的值*/
18. static ssize_t hello_read(struct file* filp, char __user *buf, size_t count,
     loff_t* f_pos) {
19.     ssize_t err = 0;
20.     struct hello_android_dev* dev = filp->private_data;
21.
22.     /*同步访问*/
23.     if(down_interruptible(&(dev->sem))) {
24.         return -ERESTARTSYS;
25.     }
26.
27.     if(count < sizeof(dev->val)) {
28.         goto out;
29.     }
30.
31.     /*将寄存器 val 的值拷贝到用户提供的缓冲区*/
32.     if(copy_to_user(buf, &(dev->val), sizeof(dev->val))) {
33.         err = -EFAULT;
34.         goto out;
35.     }
36.
37.     err = sizeof(dev->val);
38.
39. out:
40.     up(&(dev->sem));
41.     return err;
42. }
43.
44. /*写设备的寄存器值 val*/
45. static ssize_t hello_write(struct file* filp, const char __user *buf, size_t
     count, loff_t* f_pos) {
46.     struct hello_android_dev* dev = filp->private_data;
47.     ssize_t err = 0;
48.
49.     /*同步访问*/
50.     if(down_interruptible(&(dev->sem))) {
```

```
        return -ERESTARTSYS;
    }

    if(count != sizeof(dev->val)) {
        goto out;
    }

    /*将用户提供的缓冲区的值写到设备寄存器去*/
    if(copy_from_user(&(dev->val), buf, count)) {
        err = -EFAULT;
        goto out;
    }

    err = sizeof(dev->val);

out:
    up(&(dev->sem));
    return err;
}
```

定义通过 devfs 文件系统访问方法，这里把设备的寄存器 val 看成是设备的一个属性，通过读写这个属性来对设备进行访问，主要是实现 hello_val_show 和 hello_val_store 两个方法，同时定义了两个内部使用的访问 val 值的方法__hello_get_val 和 __hello_set_val：

view plain

```
/*读取寄存器 val 的值到缓冲区 buf 中，内部使用*/
static ssize_t __hello_get_val(struct hello_android_dev* dev, char* buf) {
    int val = 0;

    /*同步访问*/
    if(down_interruptible(&(dev->sem))) {
        return -ERESTARTSYS;
    }

    val = dev->val;
    up(&(dev->sem));

    return snprintf(buf, PAGE_SIZE, "%d/n", val);
}

/*把缓冲区 buf 的值写到设备寄存器 val 中去，内部使用*/
static ssize_t __hello_set_val(struct hello_android_dev* dev, const char* bu
f, size_t count) {
```

```
    int val = 0;

    /*将字符串转换成数字*/
    val = simple_strtol(buf, NULL, 10);

    /*同步访问*/
    if(down_interruptible(&(dev->sem))) {
        return -ERESTARTSYS;
    }

    dev->val = val;
    up(&(dev->sem));

    return count;
}

/*读取设备属性 val*/
static ssize_t hello_val_show(struct device* dev, struct device_attribute* attr, char* buf) {
    struct hello_android_dev* hdev = (struct hello_android_dev*)dev_get_drvdata(dev);

    return __hello_get_val(hdev, buf);
}

/*写设备属性 val*/
static ssize_t hello_val_store(struct device* dev, struct device_attribute* attr, const char* buf, size_t count) {
    struct hello_android_dev* hdev = (struct hello_android_dev*)dev_get_drvdata(dev);

    return __hello_set_val(hdev, buf, count);
}
```

定义通过 proc 文件系统访问方法，主要实现了 hello_proc_read 和 hello_proc_write 两个方法，同时定义了在 proc 文件系统创建和删除文件的方法 hello_create_proc 和 hello_remove_proc：

view plain

```
/*读取设备寄存器 val 的值，保存在 page 缓冲区中*/
static ssize_t hello_proc_read(char* page, char** start, off_t off, int count, int* eof, void* data) {
    if(off > 0) {
        *eof = 1;
```

```
5.          return 0;
6.      }
7.
8.      return __hello_get_val(hello_dev, page);
9.  }
10.
11. /*把缓冲区的值 buff 保存到设备寄存器 val 中去*/
12. static ssize_t hello_proc_write(struct file* filp, const char __user *buff,
    unsigned long len, void* data) {
13.     int err = 0;
14.     char* page = NULL;
15.
16.     if(len > PAGE_SIZE) {
17.         printk(KERN_ALERT"The buff is too large: %lu./n", len);
18.         return -EFAULT;
19.     }
20.
21.     page = (char*)__get_free_page(GFP_KERNEL);
22.     if(!page) {
23.         printk(KERN_ALERT"Failed to alloc page./n");
24.         return -ENOMEM;
25.     }
26.
27.     /*先把用户提供的缓冲区值拷贝到内核缓冲区中去*/
28.     if(copy_from_user(page, buff, len)) {
29.         printk(KERN_ALERT"Failed to copy buff from user./n");

30.         err = -EFAULT;
31.         goto out;
32.     }
33.
34.     err = __hello_set_val(hello_dev, page, len);
35.
36. out:
37.     free_page((unsigned long)page);
38.     return err;
39. }
40.
41. /*创建/proc/hello 文件*/
42. static void hello_create_proc(void) {
43.     struct proc_dir_entry* entry;
44.
45.     entry = create_proc_entry(HELLO_DEVICE_PROC_NAME, 0, NULL);
46.     if(entry) {
```

```
47.         entry->owner = THIS_MODULE;
48.         entry->read_proc = hello_proc_read;
49.         entry->write_proc = hello_proc_write;
50.     }
51. }
52.
53. /*删除/proc/hello 文件*/
54. static void hello_remove_proc(void) {
55.     remove_proc_entry(HELLO_DEVICE_PROC_NAME, NULL);
56. }
```

最后，定义模块加载和卸载方法，这里只要是执行设备注册和初始化操作：

view plain

```
1.  /*初始化设备*/
2.  static int  __hello_setup_dev(struct hello_android_dev* dev) {
3.      int err;
4.      dev_t devno = MKDEV(hello_major, hello_minor);
5.
6.      memset(dev, 0, sizeof(struct hello_android_dev));
7.
8.      cdev_init(&(dev->dev), &hello_fops);
9.      dev->dev.owner = THIS_MODULE;
10.     dev->dev.ops = &hello_fops;
11.
12.     /*注册字符设备*/
13.     err = cdev_add(&(dev->dev),devno, 1);
14.     if(err) {
15.         return err;
16.     }
17.
18.     /*初始化信号量和寄存器 val 的值*/
19.     init_MUTEX(&(dev->sem));
20.     dev->val = 0;
21.
22.     return 0;
23. }
24.
25. /*模块加载方法*/
26. static int __init hello_init(void){
27.     int err = -1;
28.     dev_t dev = 0;
29.     struct device* temp = NULL;
```

```
30.
31.     printk(KERN_ALERT"Initializing hello device./n");
32.
33.     /*动态分配主设备和从设备号*/
34.     err = alloc_chrdev_region(&dev, 0, 1, HELLO_DEVICE_NODE_NAME);
35.     if(err < 0) {
36.         printk(KERN_ALERT"Failed to alloc char dev region./n");
37.         goto fail;
38.     }
39.
40.     hello_major = MAJOR(dev);
41.     hello_minor = MINOR(dev);
42.
43.     /*分配 helo 设备结构体变量*/
44.     hello_dev = kmalloc(sizeof(struct hello_android_dev), GFP_KERNEL);
45.     if(!hello_dev) {
46.         err = -ENOMEM;
47.         printk(KERN_ALERT"Failed to alloc hello_dev./n");
48.         goto unregister;
49.     }
50.
51.     /*初始化设备*/
52.     err = __hello_setup_dev(hello_dev);
53.     if(err) {
54.         printk(KERN_ALERT"Failed to setup dev: %d./n", err);
55.         goto cleanup;
56.     }
57.
58.     /*在/sys/class/目录下创建设备类别目录 hello*/
59.     hello_class = class_create(THIS_MODULE, HELLO_DEVICE_CLASS_NAME);
60.     if(IS_ERR(hello_class)) {
61.         err = PTR_ERR(hello_class);
62.         printk(KERN_ALERT"Failed to create hello class./n");
63.         goto destroy_cdev;
64.     }
65.
66.     /*在/dev/目录和/sys/class/hello 目录下分别创建设备文件 hello*/
67.     temp = device_create(hello_class, NULL, dev, "%s", HELLO_DEVICE_FILE_NAM
    E);
68.     if(IS_ERR(temp)) {
69.         err = PTR_ERR(temp);
70.         printk(KERN_ALERT"Failed to create hello device.");
71.         goto destroy_class;
72.     }
```

```
73.
74.     /*在/sys/class/hello/hello 目录下创建属性文件 val*/
75.     err = device_create_file(temp, &dev_attr_val);
76.     if(err < 0) {
77.         printk(KERN_ALERT"Failed to create attribute val.");
78.         goto destroy_device;
79.     }
80.
81.     dev_set_drvdata(temp, hello_dev);
82.
83.     /*创建/proc/hello 文件*/
84.     hello_create_proc();
85.
86.     printk(KERN_ALERT"Succedded to initialize hello device./n");
87.     return 0;
88.
89. destroy_device:
90.     device_destroy(hello_class, dev);
91.
92. destroy_class:
93.     class_destroy(hello_class);
94.
95. destroy_cdev:
96.     cdev_del(&(hello_dev->dev));
97.
98. cleanup:
99.     kfree(hello_dev);
100.
101. unregister:
102.     unregister_chrdev_region(MKDEV(hello_major, hello_minor), 1);
103.
104. fail:
105.     return err;
106. }
107.
108. /*模块卸载方法*/
109. static void __exit hello_exit(void) {
110.     dev_t devno = MKDEV(hello_major, hello_minor);
111.
112.     printk(KERN_ALERT"Destroy hello device./n");
113.
114.     /*删除/proc/hello 文件*/
115.     hello_remove_proc();
```

```
116.
117.     /*销毁设备类别和设备*/
118.     if(hello_class) {
119.         device_destroy(hello_class, MKDEV(hello_major, hello_minor));
120.         class_destroy(hello_class);
121.     }
122.
123.     /*删除字符设备和释放设备内存*/
124.     if(hello_dev) {
125.         cdev_del(&(hello_dev->dev));
126.         kfree(hello_dev);
127.     }
128.
129.     /*释放设备号*/
130.     unregister_chrdev_region(devno, 1);
131. }
132.
133. MODULE_LICENSE("GPL");
134. MODULE_DESCRIPTION("First Android Driver");
135.
136. module_init(hello_init);
137. module_exit(hello_exit);
```

五.在 hello 目录中新增 Kconfig 和 Makefile 两个文件，其中 Kconfig 是在编译前执行配置命令 make menuconfig 时用到的，而 Makefile 是执行编译命令 make 是用到的：

Kconfig 文件的内容

```
config HELLO
    tristate "First Android Driver"
    default n
    help
    This is the first android driver.
```

Makefile 文件的内容

```
obj-$(CONFIG_HELLO) += hello.o
```

在 Kconfig 文件中，tristate 表示编译选项 HELLO 支持在编译内核时，hello 模块支持以模块、内建和不编译三种编译方法，默认是不编译，因此，在编译内核前，我们还需要执行 make menuconfig 命令来配置编译选项，使得 hello 可以以模块或者内建的方法进行编译。

在 Makefile 文件中，根据选项 HELLO 的值，执行不同的编译方法。

六．修改 arch/arm/Kconfig 和 drivers/kconfig 两个文件，在 menu "Device Drivers"和 endmenu 之间添加一行：

source "drivers/hello/Kconfig"

这样，执行 make menuconfig 时，就可以配置 hello 模块的编译选项了。.

七. 修改 drivers/Makefile 文件，添加一行：

obj-$(CONFIG_HELLO) += hello/

八．配置编译选项：

USER-NAME@MACHINE-NAME:~/Android/kernel/common$ make menuconfig

找到"Device Drivers" => "First Android Drivers"选项，设置为 y。

注意，如果内核不支持动态加载模块，这里不能选择 m，虽然我们在 Kconfig 文件中配置了 HELLO 选项为 tristate。要支持动态加载模块选项，必须要在配置菜单中选择 Enable loadable module support 选项；在支持动态卸载模块选项，必须要在 Enable loadable module support 菜单项中，选择 Module unloading 选项。

九．编译：

USER-NAME@MACHINE-NAME:~/Android/kernel/common$ make

编译成功后，就可以在 hello 目录下看到 hello.o 文件了，这时候编译出来的 zImage 已经包含了 hello 驱动。

十．参照在 Ubuntu 上下载、编译和安装 Android 最新内核源代码（Linux Kernel）一文所示，运行新编译的内核文件，验证 hello 驱动程序是否已经正常安装：

USER-NAME@MACHINE-NAME:~/Android$ emulator -kernel ./kernel/common/arch/arm/boot/zImage &

USER-NAME@MACHINE-NAME:~/Android$ adb shell

进入到 dev 目录，可以看到 hello 设备文件：

root@android:/ # cd dev

root@android:/dev # ls

进入到 proc 目录，可以看到 hello 文件：

root@android:/ # cd proc

root@android:/proc # ls

访问 hello 文件的值：

root@android:/proc # cat hello

0

root@android:/proc # echo '5' > hello

root@android:/proc # cat hello

5

进入到 sys/class 目录，可以看到 hello 目录：

```
root@android:/ # cd sys/class
root@android:/sys/class # ls
```

进入到 hello 目录，可以看到 hello 目录：

```
root@android:/sys/class # cd hello
root@android:/sys/class/hello # ls
```

进入到下一层 hello 目录，可以看到 val 文件：

```
root@android:/sys/class/hello # cd hello
root@android:/sys/class/hello/hello # ls
```

访问属性文件 val 的值：

```
root@android:/sys/class/hello/hello # cat val
5
root@android:/sys/class/hello/hello # echo '0'   > val
root@android:/sys/class/hello/hello # cat val
0
```

至此，我们的 hello 内核驱动程序就完成了，并且验证一切正常。这里我们采用的是系统提供的方法和驱动程序进行交互，也就是通过 proc 文件系统和 devfs 文件系统的方法，下一篇文章中，我们将通过自己编译的 C 语言程序来访问/dev/hello 文件来和 hello 驱动程序交互，敬请期待。