# 在Ubuntu上为Android系统编写Linux内核驱动程序

 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](http://www.ifanr.com/39850" \t "_blank)，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](http://tech.ifeng.com/internet/detail_2011_06/13/6971800_0.shtml" \t "_blank)，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。

**《Android系统源代码情景分析》一书正在进击的程序员网（**[http://0xcc0xcd.com](http://0xcc0xcd.com/" \t "_blank)**）中连载，点击进入！**

        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](http://blog.csdn.net/luoshengyang/article/details/6557518" \t "_blank)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](http://blog.csdn.net/luoshengyang/article/details/6559955" \t "_blank)和[在Ubuntu上下载、编译和安装Android最新内核源代码（Linux Kernel）](http://blog.csdn.net/luoshengyang/article/details/6564592)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：

**[cpp]** [view plain](https://blog.csdn.net/luoshengyang/article/details/6568411) [copy](https://blog.csdn.net/luoshengyang/article/details/6568411)

1. #ifndef \_HELLO\_ANDROID\_H\_
2. #define \_HELLO\_ANDROID\_H\_
4. #include <linux/cdev.h>
5. #include <linux/semaphore.h>
7. #define HELLO\_DEVICE\_NODE\_NAME  "hello"
8. #define HELLO\_DEVICE\_FILE\_NAME  "hello"
9. #define HELLO\_DEVICE\_PROC\_NAME  "hello"
10. #define HELLO\_DEVICE\_CLASS\_NAME "hello"
12. **struct** hello\_android\_dev {
13. **int** val;
14. **struct** semaphore sem;
15. **struct** cdev dev;
16. };
18. #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：

**[cpp]** [view plain](https://blog.csdn.net/luoshengyang/article/details/6568411) [copy](https://blog.csdn.net/luoshengyang/article/details/6568411)

1. #include <linux/init.h>
2. #include <linux/module.h>
3. #include <linux/types.h>
4. #include <linux/fs.h>
5. #include <linux/proc\_fs.h>
6. #include <linux/device.h>
7. #include <asm/uaccess.h>
9. #include "hello.h"
11. /\*主设备和从设备号变量\*/
12. **static** **int** hello\_major = 0;
13. **static** **int** hello\_minor = 0;
15. /\*设备类别和设备变量\*/
16. **static** **struct** **class**\* hello\_class = NULL;
17. **static** **struct** hello\_android\_dev\* hello\_dev = NULL;
19. /\*传统的设备文件操作方法\*/
20. **static** **int** hello\_open(**struct** inode\* inode, **struct** file\* filp);
21. **static** **int** hello\_release(**struct** inode\* inode, **struct** file\* filp);
22. **static** ssize\_t hello\_read(**struct** file\* filp, **char** \_\_user \*buf, **size\_t** count, loff\_t\* f\_pos);
23. **static** ssize\_t hello\_write(**struct** file\* filp, **const** **char** \_\_user \*buf, **size\_t** count, loff\_t\* f\_pos);
25. /\*设备文件操作方法表\*/
26. **static** **struct** file\_operations hello\_fops = {
27. .owner = THIS\_MODULE,
28. .open = hello\_open,
29. .release = hello\_release,
30. .read = hello\_read,
31. .write = hello\_write,
32. };
34. /\*访问设置属性方法\*/
35. **static** ssize\_t hello\_val\_show(**struct** device\* dev, **struct** device\_attribute\* attr,  **char**\* buf);
36. **static** ssize\_t hello\_val\_store(**struct** device\* dev, **struct** device\_attribute\* attr, **const** **char**\* buf, **size\_t** count);
38. /\*定义设备属性\*/
39. **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：

**[cpp]** [view plain](https://blog.csdn.net/luoshengyang/article/details/6568411) [copy](https://blog.csdn.net/luoshengyang/article/details/6568411)

1. /\*打开设备方法\*/
2. **static** **int** hello\_open(**struct** inode\* inode, **struct** file\* filp) {
3. **struct** hello\_android\_dev\* dev;
5. /\*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\*/
6. dev = container\_of(inode->i\_cdev, **struct** hello\_android\_dev, dev);
7. filp->private\_data = dev;
9. **return** 0;
10. }
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. }
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;
22. /\*同步访问\*/
23. **if**(down\_interruptible(&(dev->sem))) {
24. **return** -ERESTARTSYS;
25. }
27. **if**(count < **sizeof**(dev->val)) {
28. **goto** out;
29. }
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. }
37. err = **sizeof**(dev->val);
39. out:
40. up(&(dev->sem));
41. **return** err;
42. }
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;
49. /\*同步访问\*/
50. **if**(down\_interruptible(&(dev->sem))) {
51. **return** -ERESTARTSYS;
52. }
54. **if**(count != **sizeof**(dev->val)) {
55. **goto** out;
56. }
58. /\*将用户提供的缓冲区的值写到设备寄存器去\*/
59. **if**(copy\_from\_user(&(dev->val), buf, count)) {
60. err = -EFAULT;
61. **goto** out;
62. }
64. err = **sizeof**(dev->val);
66. out:
67. up(&(dev->sem));
68. **return** err;
69. }

        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：

**[cpp]** [view plain](https://blog.csdn.net/luoshengyang/article/details/6568411) [copy](https://blog.csdn.net/luoshengyang/article/details/6568411)

1. /\*读取寄存器val的值到缓冲区buf中，内部使用\*/
2. **static** ssize\_t \_\_hello\_get\_val(**struct** hello\_android\_dev\* dev, **char**\* buf) {
3. **int** val = 0;
5. /\*同步访问\*/
6. **if**(down\_interruptible(&(dev->sem))) {
7. **return** -ERESTARTSYS;
8. }
10. val = dev->val;
11. up(&(dev->sem));
13. **return** snprintf(buf, PAGE\_SIZE, "%d\n", val);
14. }
16. /\*把缓冲区buf的值写到设备寄存器val中去，内部使用\*/
17. **static** ssize\_t \_\_hello\_set\_val(**struct** hello\_android\_dev\* dev, **const** **char**\* buf, **size\_t** count) {
18. **int** val = 0;
20. /\*将字符串转换成数字\*/
21. val = simple\_strtol(buf, NULL, 10);
23. /\*同步访问\*/
24. **if**(down\_interruptible(&(dev->sem))) {
25. **return** -ERESTARTSYS;
26. }
28. dev->val = val;
29. up(&(dev->sem));
31. **return** count;
32. }
34. /\*读取设备属性val\*/
35. **static** ssize\_t hello\_val\_show(**struct** device\* dev, **struct** device\_attribute\* attr, **char**\* buf) {
36. **struct** hello\_android\_dev\* hdev = (**struct** hello\_android\_dev\*)dev\_get\_drvdata(dev);
38. **return** \_\_hello\_get\_val(hdev, buf);
39. }
41. /\*写设备属性val\*/
42. **static** ssize\_t hello\_val\_store(**struct** device\* dev, **struct** device\_attribute\* attr, **const** **char**\* buf, **size\_t** count) {
43. **struct** hello\_android\_dev\* hdev = (**struct** hello\_android\_dev\*)dev\_get\_drvdata(dev);
45. **return** \_\_hello\_set\_val(hdev, buf, count);
46. }

        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：

**[cpp]** [view plain](https://blog.csdn.net/luoshengyang/article/details/6568411) [copy](https://blog.csdn.net/luoshengyang/article/details/6568411)

1. /\*读取设备寄存器val的值，保存在page缓冲区中\*/
2. **static** ssize\_t hello\_proc\_read(**char**\* page, **char**\*\* start, off\_t off, **int** count, **int**\* eof, **void**\* data) {
3. **if**(off > 0) {
4. \*eof = 1;
5. **return** 0;
6. }
8. **return** \_\_hello\_get\_val(hello\_dev, page);
9. }
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;
16. **if**(len > PAGE\_SIZE) {
17. printk(KERN\_ALERT"The buff is too large: %lu.\n", len);
18. **return** -EFAULT;
19. }
21. page = (**char**\*)\_\_get\_free\_page(GFP\_KERNEL);
22. **if**(!page) {
23. printk(KERN\_ALERT"Failed to alloc page.\n");
24. **return** -ENOMEM;
25. }
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. }
34. err = \_\_hello\_set\_val(hello\_dev, page, len);
36. out:
37. free\_page((unsigned **long**)page);
38. **return** err;
39. }
41. /\*创建/proc/hello文件\*/
42. **static** **void** hello\_create\_proc(**void**) {
43. **struct** proc\_dir\_entry\* entry;
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. }
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：

**[cpp]** [view plain](https://blog.csdn.net/luoshengyang/article/details/6568411) [copy](https://blog.csdn.net/luoshengyang/article/details/6568411)

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);
6. memset(dev, 0, **sizeof**(**struct** hello\_android\_dev));
8. cdev\_init(&(dev->dev), &hello\_fops);
9. dev->dev.owner = THIS\_MODULE;
10. dev->dev.ops = &hello\_fops;
12. /\*注册字符设备\*/
13. err = cdev\_add(&(dev->dev),devno, 1);
14. **if**(err) {
15. **return** err;
16. }
18. /\*初始化信号量和寄存器val的值\*/
19. init\_MUTEX(&(dev->sem));
20. dev->val = 0;
22. **return** 0;
23. }
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;
31. printk(KERN\_ALERT"Initializing hello device.\n");
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. }
40. hello\_major = MAJOR(dev);
41. hello\_minor = MINOR(dev);
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. }
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. }
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. }
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\_NAME);
68. **if**(IS\_ERR(temp)) {
69. err = PTR\_ERR(temp);
70. printk(KERN\_ALERT"Failed to create hello device.");
71. **goto** destroy\_class;
72. }
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. }
81. dev\_set\_drvdata(temp, hello\_dev);
83. /\*创建/proc/hello文件\*/
84. hello\_create\_proc();
86. printk(KERN\_ALERT"Succedded to initialize hello device.\n");
87. **return** 0;
89. destroy\_device:
90. device\_destroy(hello\_class, dev);
92. destroy\_class:
93. class\_destroy(hello\_class);
95. destroy\_cdev:
96. cdev\_del(&(hello\_dev->dev));
98. cleanup:
99. kfree(hello\_dev);
101. unregister:
102. unregister\_chrdev\_region(MKDEV(hello\_major, hello\_minor), 1);
104. fail:
105. **return** err;
106. }
108. /\*模块卸载方法\*/
109. **static** **void** \_\_exit hello\_exit(**void**) {
110. dev\_t devno = MKDEV(hello\_major, hello\_minor);
112. printk(KERN\_ALERT"Destroy hello device.\n");
114. /\*删除/proc/hello文件\*/
115. hello\_remove\_proc();
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. }
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. }
129. /\*释放设备号\*/
130. unregister\_chrdev\_region(devno, 1);
131. }
133. MODULE\_LICENSE("GPL");
134. MODULE\_DESCRIPTION("First Android Driver");
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）](http://blog.csdn.net/luoshengyang/article/details/6564592" \t "_blank)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。