> 作 者：道哥，10+年嵌入式开发老兵，专注于：C/C++、嵌入式、Linux。
>
> 关注下方公众号，回复【书籍】，获取 Linux、嵌入式领域经典书籍；回复【PDF】，获取所有原创文章(PDF 格式)。

目录

别人的经验，我们的阶梯！

大家好，我是道哥，今天我为大伙儿解说的技术知识点是：【字符设备的驱动程序】。

在上一篇文章中，讨论的是Linux系统中，驱动模块的两种编译方式。

我们就继续以此为基础，用保姆级的粒度一步一步操作，来讨论一下字符设备驱动程序的编写方法。

> 1. 这篇文章的实际操作部分，使用的是的 API 函数;
> 2. 下一篇文章，再来演示新的 API 函数;

混乱的 API 函数

我在刚开始接触Linux驱动的时候，非常的困扰：注册一个字符设备，怎么有这么多的 API 函数啊?

参考的每一篇文章中，使用的函数都不一样，但是执行结果都是符合预期的！

比如下面这几个：

> 1. register_chrdev(...);

2. register_chrdev_regin(...);
3. cdev_add(...);

它们的功能都是向系统注册字符设备，但是只从函数名上看，初学者谁能分得清它们的区别？！

这也难怪，Linux系统经过这么多年的发展，代码更新是很正常的事情。

但是，我们参考的文章就没法做到：很详细的把文章中所描述内容的背景介绍清楚，往往都是文章作者在自己的实际工作环境中，测试某种方法解决了自己的问题，于是就记录成文。

不同的文章、不同的工作上下文、不同的API函数调用，这往往就苦了我们初学者，特别是我这种有选择障碍症的人！

其实，上面这个几个函数都是正确的，它们的功能都是类似的，它们是Linux 系统中不同阶段的产物。

旧的 API 函数

在Linux内核代码2.4版本和早期的2.6版本中，注册、卸载字符设备驱动程序的经典方式是：

注册设备：

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

> 参数1 major： 如果为0 - 由操作系统动态分配一个主设备号给这个设备；如果非0 - 驱动程序向系统申请，使用这个主设备号；
>
> 参数2 name： 设备名称；
>
> 参数3 fops： file_operations 类型的指针变量，用于操作设备；

如果是动态分配，那么这个函数的返回值就是：操作系统动态分配给这个设备的主设备号。

这个动态分配的设备号，我们要把它记住，因为在其他的API函数中需要使用它。

卸载设备：

```
int unregister_chrdev(unsigned int major,const char *name)
```

> 参数1 major： 设备的主设备号，也就是 register_chrdev() 函数的返回值(动态)，或者驱动程序指定的设备号(静态方式)；
>
> 参数2 name： 设备名称；

新的 API 函数

注册设备：

```
int register_chrdev_region(dev_t from, unsigned count, const char *name);
int alloc_chrdev_region(dev_t *dev, unsigned baseminor, unsigned count,const char
*name);
```

上面这2个注册设备的函数，其实对应着旧的 API 函数 `register_chrdev`：把参数 1 表示的动态分配、静态分配，拆分成2个函数而已。

也就是说：

> `register_chrdev_region()`: 静态注册设备;
>
> `alloc_chrdev_region()`: 动态注册设备;

这两个函数的参数含义是：

`register_chrdev_region` 参数：

> 参数1 from: 注册指定的设备号，这是静态指定的，例如：MKDEV(200, 0) 表示起始主设备号 200, 起始次设备号为 0;
>
> 参数2 count: 驱动程序指定连续注册的次设备号的个数，例如：起始次设备号是 0，count 为 10，表示驱动程序将会使用 0 ~ 9 这 10 个次设备号;
>
> 参数3 name：设备名称;

`alloc_chrdev_region` 参数：

> 参数1 dev: 动态注册就是系统来分配设备号，那么驱动程序就要提供一个指针变量来接收系统分配的结果(设备号);
>
> 参数2 baseminor: 驱动程序指定此设备号的起始值;
>
> 参数3 count: 驱动程序指定连续注册的次设备号的个数，例如：起始次设备号是 0，count 为 10，表示驱动程序将会使用 0 ~ 9 这 10 个次设备号;
>
> 参数4 name：设备名称;

补充一下关于设备号的内容：

这里的结构体 `dev_t`，用来保存设备号，包括主设备号和次设备号。

它本质上是一个 32 位的数，其中的 12 位用来表示主设备号，而其余 20 位用来表示次设备号。

系统中定义了3宏，来实现`dev_t`变量、主设备号、次设备号之间的转换：

> MAJOR(dev_t dev): 从 dev_t 类型中获取主设备号;
>
> MINOR(dev_t dev): 从 dev_t 类型中获取次设备号;
>
> MKDEV(int major,int minor): 把主设备号和次设备号转换为 dev_t 类型;

卸载设备：

```
void unregister_chrdev_region(dev_t from, unsigned count);
```

> 参数1 from: 注销的设备号;
>
> 参数2 count: 注销的连续次设备号的个数;

代码实操

下面，我们就用旧的API函数，一步一步的描述字符设备驱动程序的：编写、加载和卸载过程。

> 如何使用新的 API 函数来编写字符设备驱动程序，下一篇文章再详细讨论。

以下所有操作的工作目录，都是与上一篇文章相同的，即：`~/tmp/linux-4.15/drivers/`。

创建驱动目录和驱动程序

```
$ cd linux-4.15/drivers/
$ mkdir my_driver1
$ cd my_driver1
$ touch driver1.c
```

`driver1.c` 文件的内容如下(不需要手敲，文末有代码下载链接):

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/init.h>
#include <linux/delay.h>
#include <linux/uaccess.h>
#include <linux/ctype.h>
#include <linux/irq.h>
#include <linux/io.h>
#include <linux/device.h>

static unsigned int major;

int driver1_open(struct inode *inode, struct file *file)
{
  printk("driver1_open is called. \n");
  return 0;
}

ssize_t driver1_read(struct file *file, char __user *buf, size_t size, loff_t *ppos)
{
  printk("driver1_read is called. \n");
  return 0;
}

ssize_t driver1_write (struct file *file, const char __user *buf, size_t size, loff_t
*ppos)
{
```

```
  printk("driver1_write is called. \n");
  return 0;
}

static const struct file_operations driver1_ops={
  .owner = THIS_MODULE,
  .open  = driver1_open,
  .read  = driver1_read,
  .write = driver1_write,
};

static int __init driver1_init(void)
{
  printk("driver1_init is called. \n");

  major = register_chrdev(0, "driver1", &driver1_ops);
  printk("register_chrdev. major = %d\n",major);
  return 0;
}

static void __exit driver1_exit(void)
{
  printk("driver1_exit is called. \n");
  unregister_chrdev(major,"driver1");
}

MODULE_LICENSE("GPL");
module_init(driver1_init);
module_exit(driver1_exit);
```

创建 Makefile 文件

```
$ touch Makefile
```

内容如下:

```
ifneq ($(KERNELRELEASE),)
  obj-m := driver1.o
else
  KERNELDIR ?= /lib/modules/$(shell uname -r)/build
  PWD := $(shell pwd)
default:
  $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
clean:
  $(MAKE) -C $(KERNEL_PATH) M=$(PWD) clean
endif
```

编译驱动模块

```
$ make
```

得到驱动程序： driver1.ko 。

加载驱动模块

在加载驱动模块之前，先来看一下系统中，几个与驱动设备相关的地方。

先看一下 /dev 目录下，目前还没有我们的设备节点(/dev/driver1)。

再来查看一下 /proc/devices 目录下，也没有 driver1 设备的设备号。

```
cat /proc/devices  | grep driver1
```

> /proc/devices 文件: 列出字符和块设备的主设备号，以及分配到这些设备号的设备名称。

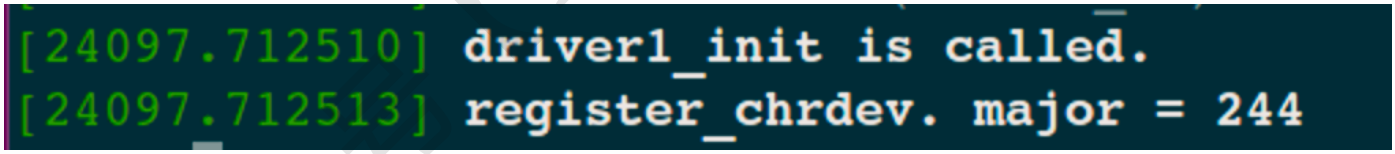
执行如下指令，加载驱动各模块:

```
$ sudo insmod driver1.ko
```

通过上一篇文章我们知道，当驱动程序被加载的时候，通过 `module_init(driver1_init);` 注册的函数 `driver1_init()` 将会被执行，那么其中的打印信息就会输出。

还是通过 dmesg 指令来查看驱动模块的打印信息：

```
$ dmesg
```



> 如果输入信息太多，可以使用dmesg | tail指令;

此时，驱动模块已经被加载了!

来查看一下 /proc/devices 目录下显示的设备号：

可以看到 driver1 已经挂载好了，并且它的主设备号是244。

此时，虽然已经向系统注册了这个设备，并且主设备号已经分配了，但是，在/dev目录下，还不存在这个设备的节点，需要我们手动创建：

```
sudo mknod -m 660 /dev/driver1 c 244 0
```

检查一下设备节点是否创建成功：

```
$ ls -l /dev
```

```
crw-rw----  1 root root    244,   0 Nov 15 14:50 driver1
```

> 关于设备节点，Linux 的应用层有一个 udev 服务，可以自动创建设备节点;
>
> 也就是：当驱动模块被加载的时候，自动在 /dev 目录下创建设备节点。当然了，我们需要在驱动程序中，提前告诉 udev 如何去创建;
>
> 下面会介绍：如何自动创建设备节点。

现在，设备的驱动程序已经加载了，设备节点也被创建好了，应用程序就可以来操作(读、写)这个设备了。

应用程序

我们把所有的应用程序，放在 ~/tmp/App/ 目录下。

```
$ cd ~/tmp
$ mkdir -p App/app_driver1
$ touch app_driver1.c
```

app_driver1.c 文件的内容如下：

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>


int main(void)
{
  int ret;
  int read_data[4] = { 0 };
  int write_data[4] = {1, 2, 3, 4};
  int fd = open("/dev/driver1", O_RDWR);
  if (-1 != fd)
  {
    ret = read(fd, read_data, 4);
    printf("read ret = %d \n", ret);

    ret = write(fd, write_data, 4);
    printf("write ret = %d \n", ret);
  }
  else
  {
    printf("open /dev/driver1 failed! \n");
  }

  return 0;
}
```

这里演示的仅仅是通过打印信息来体现函数的调用，并没有实际的读取数据和写入数据。

因为，读写数据又涉及到复杂的用户空间和内核空间的数据拷贝问题。

应用程序准备妥当，接下来就是编译和测试了：

```
$ gcc app_driver1.c -o app_driver1
$ sudo ./app_driver1
```

应用程序的输出信息如下：

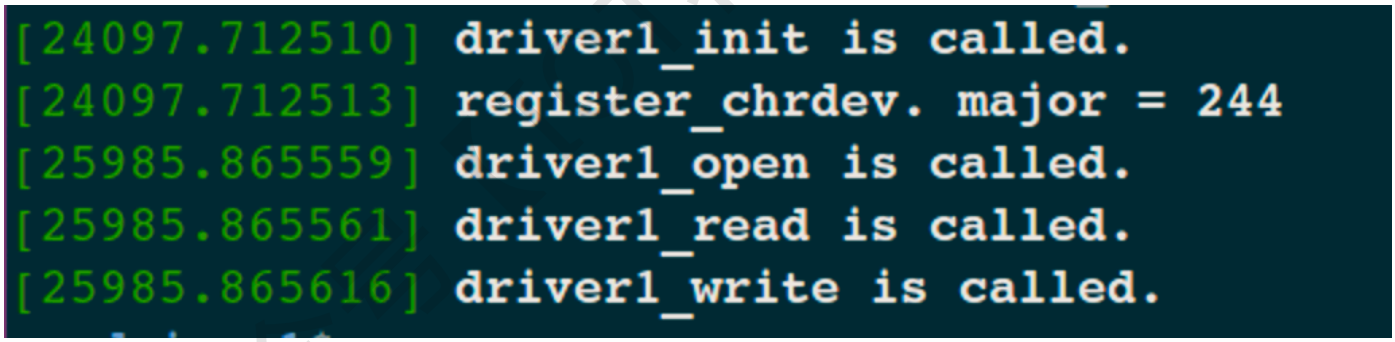
```
app_driver1$ sudo ./app_driver1
[sudo] password for xxxx: <输入用户密码>
read ret = 0
write ret = 0
```

从返回值来看，成功打开了设备，并且调用读函数、写函数都成功了！

根据Linux系统的驱动框架，应用层的 open、read、write 函数被调用的时候，驱动程序中对应的函数就会被执行：

```
static const struct file_operations driver1_ops={
  .owner = THIS_MODULE,
  .open  = driver1_open,
  .read  = driver1_read,
  .write = driver1_write,
};
```

我们已经在驱动程序的这三个函数中打印了信息，继续用dmesg命令查看一下：

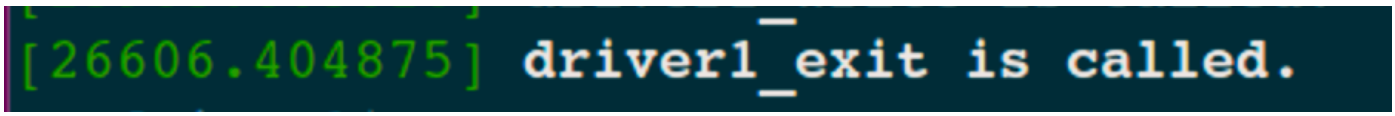


卸载驱动模块

卸载指令：

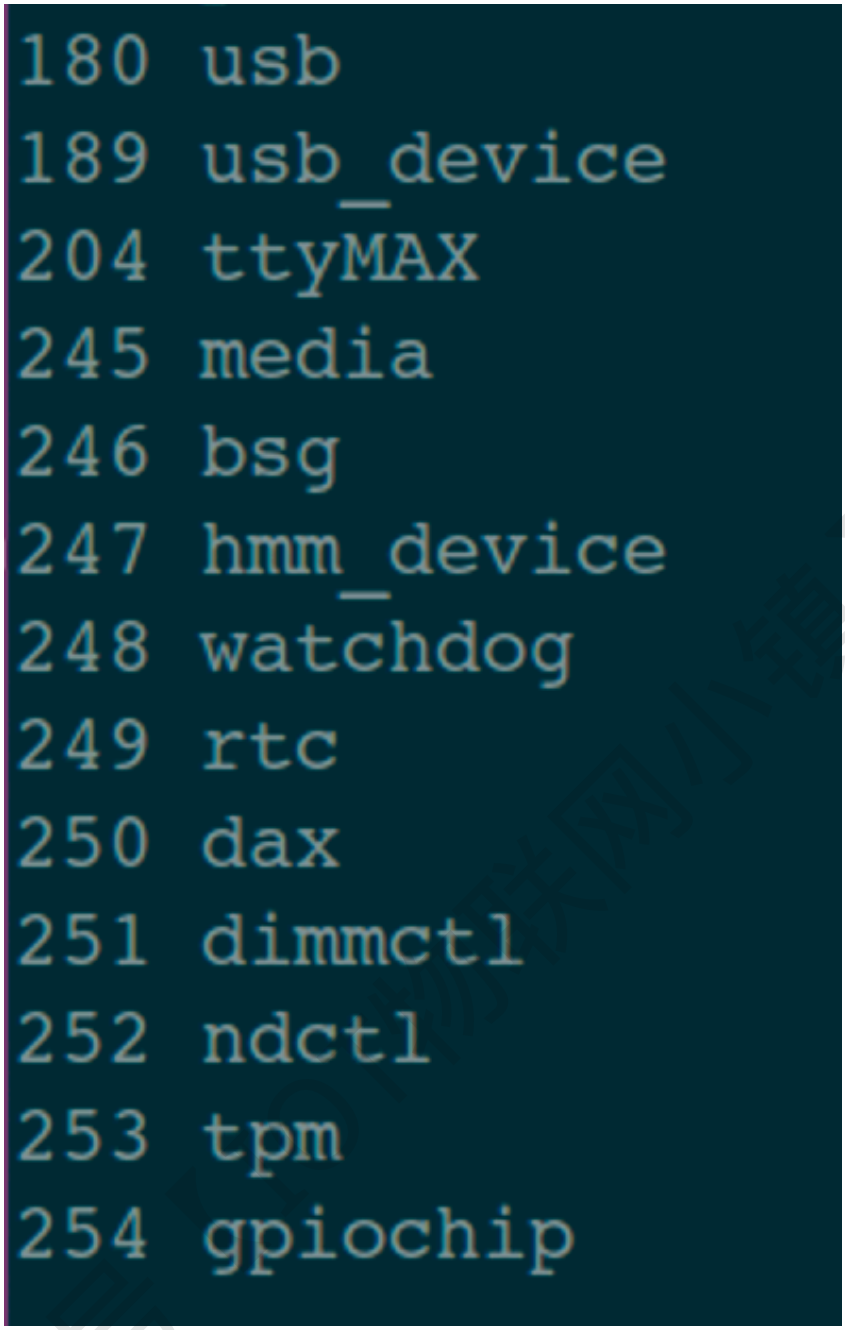
```
$ sudo rmmod driver1
```

继续用dmesg指令来查看驱动程序中的打印信息：



说明驱动程序中的 driver1_exit() 函数被调用了。

此时，我们来看一下 /proc/devices 目录下变化：



可以看到：刚才设备号为244的 driver1 已经被系统卸载了！因为驱动程序中的 `unregister_chrdev(major,"driver1");` 函数被执行了。

但是，由于 /dev 目录下的设备节点 driver1 ，是刚才手动创建的，因此需要我们手动删除。

```
$ sudo rm /dev/driver1
```

小结

以上，就是字符设备的最简单驱动程序！

从编写过程可以看出：Linux系统已经设计好了一套驱动程序的框架。

我们只需要按照它要求，按部就班地把每一个函数或者是结构体，注册到系统中就可以了。

自动在 /dev 目录下创建设备节点

在上面的操作过程中，设备节点 /dev/driver1 是手动创建的。

Linux系统的应用层提供了 udev 这个服务，可以帮助我们自动创建设备节点。我们现在就来把这个功能补上。

修改驱动程序

为了方便比较，添加的代码全部用宏定义 UDEV_ENABLE 控制起来。

driver1.c代码中，有 3 处变化：

1. 定义 2 个全局变量

```
#ifdef UDEV_ENABLE
static struct class  *driver1_class;
static struct device *driver1_dev;
#endif
```

2. driver1_init() 函数

```
static int __init driver1_init(void)
{
  printk("driver1_init is called. \n");

  major = register_chrdev(0, "driver1", &driver1_ops);
  printk("register_chrdev. major = %d\n",major);

#ifdef UDEV_ENABLE
  driver1_class = class_create(THIS_MODULE, "driver1");
  driver1_dev = device_create(driver1_class, NULL, MKDEV(major, 0), NULL, "driver1");
#endif

  return 0;
}
```

3. driver1_exit() 函数

```
static void __exit driver1_exit(void)
{
  printk("driver1_exit is called. \n");
#ifdef UDEV_ENABLE
  class_destroy(driver1_class);
#endif
  unregister_chrdev(major,"driver1");
}
```

代码修改之后(也可以直接下载我放在网盘里的源代码)，重新编译驱动模块：

```
$ make
```

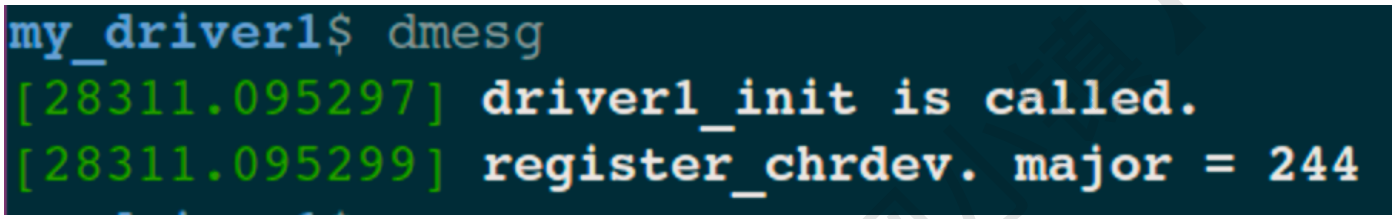
生成driver1.ko驱动模块，然后加载它：

> 先确定一下：/proc/devices，/dev 目录下，已经没有刚才测试的设备了；
>
> 为了便于查看驱动程序中的打印信息，最好把 dmesg 输出的打印信息清理一下(指令：sudo dmesg -c);

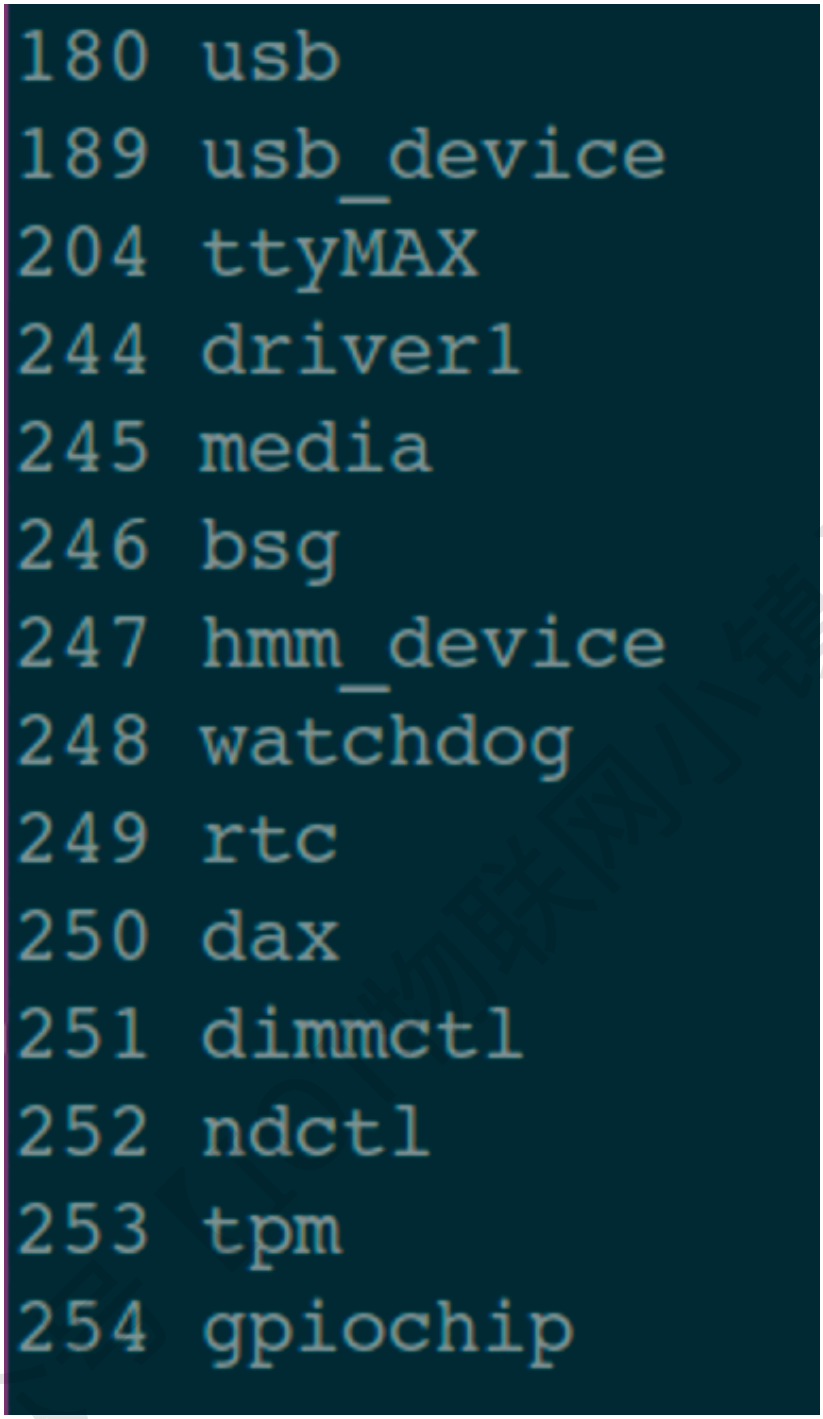
```
$ sudo insmod driver1.ko
```

按照刚才的操作流程，我们需要来验证3个信息：

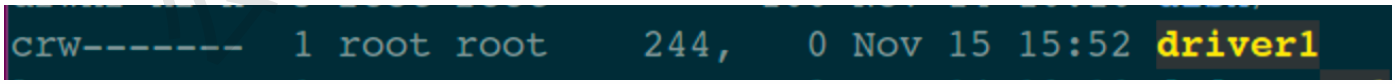
(1) 看一下驱动程序的打印信息(指令：dmesg)：



(2) 看一下 /proc/devices 下的设备注册情况：



(3) 看一下 /dev 下，是否自动创建了设备节点：



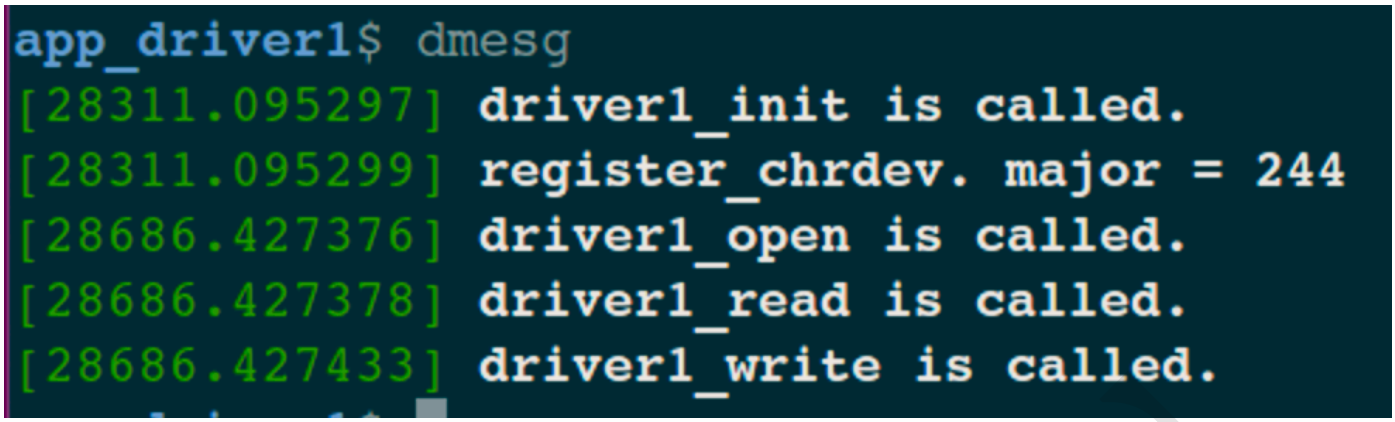
通过以上3张图片，可以得到结论：驱动程序正确加载了，设备节点被自动创建了！

下面，就应该是应用程序登场测试了，代码不用修改，直接执行即可：

```
$ sudo ./app_driver1
[sudo] password for xxx: <输入用户密码>
read ret = 0
write ret = 0
```

应用层的函数返回值正确！

再看一下 dmesg 的输出信息：



完美！

代码下载

文中的所有代码，已经放在网盘中了。

在公众号【IOT物联网小镇】后台回复关键字：1115，获取下列文件的网盘地址。

------ End ------

推荐阅读

【1】《Linux 从头学》系列文章

【2】C语言指针-从底层原理到花式技巧，用图文和代码帮你讲解透彻

【3】原来gdb的底层调试原理这么简单

【4】内联汇编很可怕吗？看完这篇文章，终结它！

其他系列专辑：精选文章、应用程序设计、物联网、 C语言。