

Linux驱动实践：如何编写【GPIO】设备的驱动程序？



IOT物联网小镇

2 人赞同了该文章

别人的经验，我们的阶梯！

大家好，我是道哥。

在前几篇文章中，我们一块讨论了：在 Linux 系统中，编写字符设备驱动程序的基本框架，主要是从代码流程和 API 函数这两方面触发。

这篇文章，我们就以此为基础，写一个有实际应用功能的驱动程序：

在驱动程序中，初始化 GPIO 设备，自动创建设备节点；

在应用程序中，打开 GPIO 设备，并发送控制指令设置 GPIO 口的状态；

示例程序目标

编写一个驱动程序模块：mygpio.ko。

当这个驱动模块被加载的时候，在系统中创建一个 mygpio 类设备，并且在 /dev 目录下，创建 4 个设备节点：

```
/dev/mygpio0  
/dev/mygpio1  
/dev/mygpio2  
/dev/mygpio3
```

因为我们现在是在 x86 平台上来模拟 GPIO 的控制操作，并没有实际的 GPIO 硬件设备。

因此，在驱动代码中，与硬件相关部分的代码，使用宏 MYGPIO_HW_ENABLE 控制起来，并且在其中使用 printk 输出打印信息来体现硬件的操作。

在应用程序中，可以分别打开以上这 4 个 GPIO 设备，并且通过发送控制指令，来设置 GPIO 的状态。

编写驱动程序

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 mygpio_driver  
$ cd mygpio_driver  
$ touch mygpio.c
```

mygpio.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/ctype.h>  
#include <linux/device.h>  
#include <linux/cdev.h>  
  
// GPIO 硬件相关宏定义  
#define MYGPIO_HW_ENABLE  
  
// 设备名称  
#define MYGPIO_NAME          "mygpio"  
  
// 一共有4个 GPIO 口  
#define MYGPIO_NUMBER        4  
  
// 设备类  
static struct class *gpio_class;  
  
// 用来保存设备  
struct cdev gpio_cdev[MYGPIO_NUMBER];  
  
// 用来保存设备号  
int gpio_major = 0;  
int gpio_minor = 0;  
  
#ifdef MYGPIO_HW_ENABLE  
// 硬件初始化函数，在驱动程序被加载的时候(gpio_driver_init)被调用  
static void gpio_hw_init(int gpio)  
{  
    printk("gpio_hw_init is called: %d. \n", gpio);  
}
```

```
// 硬件释放
static void gpio_hw_release(int gpio)
{
    printk("gpio_hw_release is called: %d. \n", gpio);
}

// 设置硬件GPIO的状态，在控制GPIO的时候(gpio_ioctl)被调研
static void gpio_hw_set(unsigned long gpio_no, unsigned int val)
{
    printk("gpio_hw_set is called. gpio_no = %ld, val = %d. \n", gpio_no, val);
}
#endif

// 当应用程序打开设备的时候被调用
static int gpio_open(struct inode *inode, struct file *file)
{
    printk("gpio_open is called. \n");
    return 0;
}

// 当应用程序控制GPIO的时候被调用
static long gpio_ioctl(struct file* file, unsigned int val, unsigned long gpio_no)
{
    printk("gpio_ioctl is called. \n");

    // 检查设置的状态值是否合法
    if (0 != val && 1 != val)
    {
        printk("val is NOT valid! \n");
        return 0;
    }

    // 检查设备范围是否合法
    if (gpio_no >= MYGPIO_NUMBER)
    {
        printk("dev_no is invalid! \n");
        return 0;
    }

    printk("set GPIO: %ld to %d. \n", gpio_no, val);

#ifdef MYGPIO_HW_ENABLE
    // 操作 GPIO 硬件
    gpio_hw_set(gpio_no, val);
#endif
}
```

```
#endif

    return 0;
}

static const struct file_operations gpio_ops={
    .owner = THIS_MODULE,
    .open  = gpio_open,
    .unlocked_ioctl = gpio_ioctl
};

static int __init gpio_driver_init(void)
{
    int i, devno;
    dev_t num_dev;

    printk("gpio_driver_init is called. \n");

    // 动态申请设备号(严谨点的话, 应该检查函数返回值)
    alloc_chrdev_region(&num_dev, gpio_minor, MYGPIO_NUMBER, MYGPIO_NAME);

    // 获取主设备号
    gpio_major = MAJOR(num_dev);
    printk("gpio_major = %d. \n", gpio_major);

    // 创建设备类
    gpio_class = class_create(THIS_MODULE, MYGPIO_NAME);

    // 创建设备节点
    for (i = 0; i < MYGPIO_NUMBER; ++i)
    {
        // 设备号
        devno = MKDEV(gpio_major, gpio_minor + i);

        // 初始化 cdev 结构
        cdev_init(&gpio_cdev[i], &gpio_ops);

        // 注册字符设备
        cdev_add(&gpio_cdev[i], devno, 1);

        // 创建设备节点
        device_create(gpio_class, NULL, devno, NULL, MYGPIO_NAME"%d", i);
    }

#ifdef MYGPIO_HW_ENABLE
```

```
// 初始化 GPIO 硬件
for (i = 0; i < MYGPIO_NUMBER; ++i)
{
    gpio_hw_init(i);
}
#endif

return 0;
}

static void __exit gpio_driver_exit(void)
{
    int i;
    printk("gpio_driver_exit is called. \n");

    // 删除设备和设备节点
    for (i = 0; i < MYGPIO_NUMBER; ++i)
    {
        cdev_del(&gpio_cdev[i]);
        device_destroy(gpio_class, MKDEV(gpio_major, gpio_minor + i));
    }

    // 释放设备类
    class_destroy(gpio_class);

#ifdef MYGPIO_HW_ENABLE
    // 释放 GPIO 硬件
    for (i = 0; i < MYGPIO_NUMBER; ++i)
    {
        gpio_hw_release(i);
    }
#endif

    // 注销设备号
    unregister_chrdev_region(MKDEV(gpio_major, gpio_minor), MYGPIO_NUMBER);
}

MODULE_LICENSE("GPL");
module_init(gpio_driver_init);
module_exit(gpio_driver_exit);
```

相对于前几篇文章来说，上面的代码稍微有一点点复杂，主要是多了宏定义 MYGPIO_HW_ENABLE 控制部分的代码。

比如：在这个宏定义控制下的三个与硬件相关的函数：

```
gpio_hw_init()
gpio_hw_release()
gpio_hw_set()
```

就是与 GPIO 硬件的初始化、释放、状态设置相关的操作。

代码中的注释已经比较完善了，结合前几篇文章中的函数说明，还是比较容易理解的。

从代码中可以看出：驱动程序使用 `alloc_chrdev_region` 函数，来动态注册设备号，并且利用了 Linux 应用层中的 `udev` 服务，自动在 `/dev` 目录下创建了设备节点。

另外还有一点：在上面示例代码中，对设备的操作函数只实现了 `open` 和 `ioctl` 这两个函数，这是根据实际的使用场景来决定的。

这个示例中，只演示了如何控制 GPIO 的状态。

你也可以稍微补充一下，增加一个 `read` 函数，来读取某个 GPIO 口的状态。

控制 GPIO 设备，使用 `write` 或者 `ioctl` 函数都可以达到目的，只是 `ioctl` 更灵活一些。

创建 Makefile 文件

```
$ touch Makefile
```

内容如下：

```
ifneq ($(KERNELRELEASE),)
    obj-m := mygpio.o
else
    KERNELDIR ?= /lib/modules/$(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：mygpio.ko。

加载驱动模块

在加载驱动模块之前，先来检查一下系统中，几个与驱动设备相关的地方。

先看一下 /dev 目录下，目前还没有设备节点(/dev/mygpio[0-3])。

```
$ ls -l /dev/mygpio*  
ls: cannot access '/dev/mygpio*': No such file or directory
```

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

```
$ cat /proc/devices
```



```
180  usb  
189  usb_device  
204  ttyMAX  
245  media  
246  bsg  
247  hmm_device  
248  watchdog  
249  rtc
```

知乎 @道哥

为了方便查看打印信息，把 dmesg 输出信息清理一下：

```
$ sudo dmesg -c
```

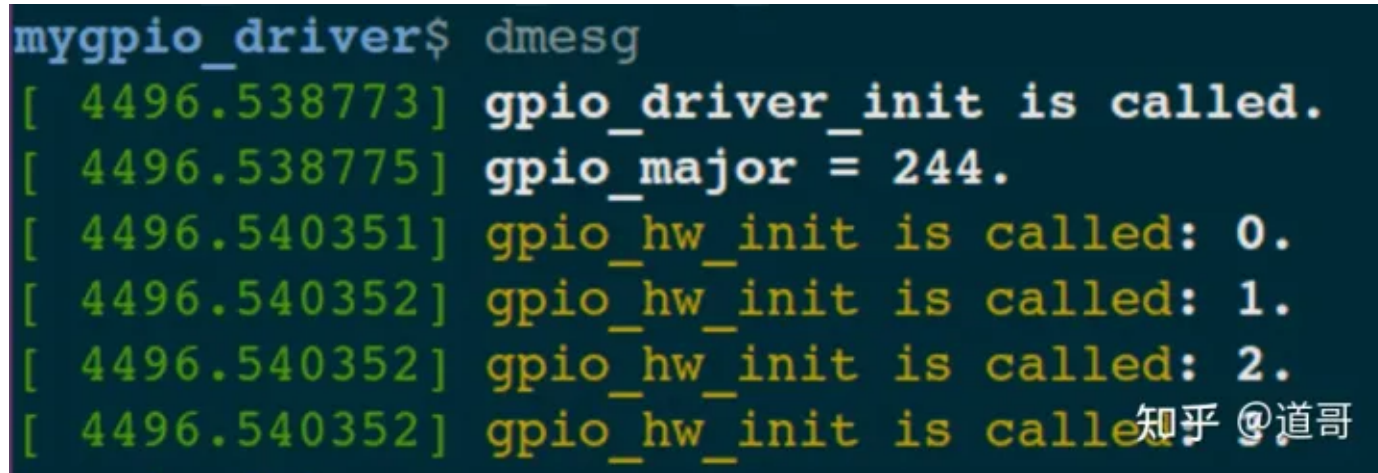
现在来加载驱动模块，执行如下指令：

```
$ sudo insmod mygpio.ko
```

当驱动程序被加载的时候，通过 `module_init()` 注册的函数 `gpio_driver_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
```



```
mygpio_driver$ dmesg
[ 4496.538773] gpio_driver_init is called.
[ 4496.538775] gpio_major = 244.
[ 4496.540351] gpio_hw_init is called: 0.
[ 4496.540352] gpio_hw_init is called: 1.
[ 4496.540352] gpio_hw_init is called: 2.
[ 4496.540352] gpio_hw_init is called: 3.
```

知乎 @道哥

可以看到：操作系统为这个设备分配的主设备号是 244，并且也打印了 GPIO 硬件的初始化函数的调用信息。

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：

```
$ cat /proc/devices
```



```
180 usb
189 usb_device
204 ttyMAX
244 mygpio
245 media
246 bsg
247 hmm_device
248 watchdog
249 rtc
250 dax
```

知乎 @道哥

设备已经注册了，主设备号是: 244。

设备节点

由于在驱动程序的初始化函数中，使用 `cdev_add` 和 `device_create` 这两个函数，自动创建设备节点。

所以，此时我们在 `/dev` 目录下，就可以看到下面这 4 个设备节点：

```
mygpio_driver$ ll /dev/mygpio*
crw----- 1 root root 244, 0 Nov 28 11:07 /dev/mygpio0
crw----- 1 root root 244, 1 Nov 28 11:07 /dev/mygpio1
crw----- 1 root root 244, 2 Nov 28 11:07 /dev/mygpio2
crw----- 1 root root 244, 3 Nov 28 11:07 /dev/mygpio3
```

知乎 @道哥

现在，设备的驱动程序已经加载了，设备节点也被创建好了，应用程序就可以来控制 GPIO 硬件设备了。

应用程序

应用程序仍然放在 `~/tmp/App/` 目录下。

```
$ mkdir ~/tmp/App/app_mygpio
$ cd ~/tmp/App/app_mygpio
$ touch app_mygpio.c
```

文件内容如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <assert.h>
#include <fcntl.h>
#include <sys/ioctl.h>

#define MY_GPIO_NUMBER      4

// 4个设备节点
char gpio_name[MY_GPIO_NUMBER][16] = {
    "/dev/mygpio0",
    "/dev/mygpio1",
    "/dev/mygpio2",
    "/dev/mygpio3"
};

int main(int argc, char *argv[])
{
    int fd, gpio_no, val;

    // 参数个数检查
    if (3 != argc)
    {
        printf("Usage: ./app_gpio gpio_no value \n");
        return -1;
    }

    gpio_no = atoi(argv[1]);
    val = atoi(argv[2]);

    // 参数合法性检查
```

```
assert(gpio_no < MY_GPIO_NUMBER);
assert(0 == val || 1 == val);

// 打开 GPIO 设备
if((fd = open(gpio_name[gpio_no], O_RDWR | O_NDELAY)) < 0){
    printf("%s: open failed! \n", gpio_name[gpio_no]);
    return -1;
}

printf("%s: open success! \n", gpio_name[gpio_no]);

// 控制 GPIO 设备状态
ioctl(fd, val, gpio_no);

// 关闭设备
close(fd);
}
```

以上代码也不需要过多解释，只要注意参数的顺序即可。

接下来就是编译和测试了：

```
$ gcc app_mygpio.c -o app_mygpio
```

执行应用程序的时候，需要携带 2 个参数：GPIO 设备编号(0 ~ 3)，设置的状态值(0 或者 1)。

这里设置一下 /dev/mygpio0 这个设备，状态设置为 1：

```
$ sudo ./app_mygpio 0 1
[sudo] password for xxx: <输入用户密码>
/dev/mygpio0: open success!
```

如何确认 /dev/mygpio0 这个 GPIO 的状态确实被设置为 1 了呢？当然是看 dmesg 指令的打印信息：

```
$ dmesg
```

```
[ 5406.792554] gpio_open is called.  
[ 5406.792621] gpio_ioctl is called.  
[ 5406.792622] set GPIO: 0 to 1.  
[ 5406.792623] gpio_hw_set is called. gpio_no = 0, val = 1.
```

通过打印信息可以看到：确实执行了【设置 mygpio0 的状态为 1】的动作。

再继续测试一下：设置 mygpio0 的状态为 0：

```
$ sudo ./app_mypio 0 0
```

```
[ 5779.393809] gpio_open is called.  
[ 5779.393878] gpio_ioctl is called.  
[ 5779.393879] set GPIO: 0 to 0.  
[ 5779.393879] gpio_hw_set is called. gpio_no = 0, val = 0.
```

当然了，设置其他几个 GPIO 口的状态，都是可以正确执行的！

卸载驱动模块

卸载指令：

```
$ sudo rmmod mygpio
```

此时， /proc/devices 下主设备号 244 的 mygpio 已经不存在了。

```
180 usb
189 usb_device
204 ttyMAX
245 media
246 bsg
247 hmm_device
248 watchdog
249 rtc
250 dax
```

知乎 @道哥

再来看一下 dmesg 的打印信息：

```
[ 5940.682257] gpio_driver_exit is called.
[ 5940.684298] gpio_hw_release is called: 0.
[ 5940.684299] gpio_hw_release is called: 1.
[ 5940.684300] gpio_hw_release is called: 2.
[ 5940.684302] gpio_hw_release is called: 3.
```

知乎 @道哥

可以看到：驱动程序中的 `gpio_driver_exit()` 被调用执行了。

并且，`/dev` 目录下的 4 个设备节点，也被函数 `device_destroy()` 自动删除了！

----- End -----