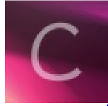
Linux misc设备驱动理解

原创

armwind 于 2016-08-11 21:17:31 发布 20869 收藏 26
分类专栏： Linux驱动
版权声明：本文为博主原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接和本声明。
本文链接：https://blog.csdn.net/armwind/article/details/52166139
版权

Linux驱动 专栏收录该内容
9 篇文章 1 订阅
订阅专栏

感兴趣可以加QQ群85486140，大家一起交流相互学习下！

Linux里面的misc杂项设备是主设备号为10的驱动设备，它的注册跟使用比较的简单，所以比较适用于功能简单的设备。正因为简单，所以它通常嵌套在platform 总线驱动中，配合总线驱动达到更复杂，多功能的效果。下面我们一起来分析它的过程。博客中如果有问题，欢迎大家指正，我们共同努力，进步。

一.杂项设备数据结构分析

杂项设备驱动结构还是很简单的，他可以夹杂到其它结构体当中，以丰富驱动的血肉。一般情况下，我们是将它嵌套在其它结构当中的。

```
struct miscdevice  {
 int minor; //次设备号，主设备号已经敲定是10了，后面我们跟进代码看一下。
 const char *name;  //驱动名字，最终会反映在设备节点名字上。
 const struct file_operations *fops; //设备操作方法集合
 struct list_head list; //链接到所有杂项设备链表当中。
 struct device *parent; //父设备，这个一般为NULL
 struct device *this_device;//当前设备的devices结构。
 const char *nodename;
 umode_t mode;
};
```

上面：

parent:这个指针决定了在/sys文件系统里面，它是创建在哪个目录下。如果为空就在/sys/class根目录下创建，如果不为空都是在/sys/class/misc 文件下面创建的一些属性文件。
this_device：这个就代表当前设备的设备结构体，这个在查找扩充数据结构时，非常有用。

```
typedef struct led{
 struct miscdevice *led_dev;
 struct mutex led_lock;
 spinlock_t io_lock;
 int flset;
 int flen;
 uint8_t regs[4+1];
}rt8547_dev_t;
```

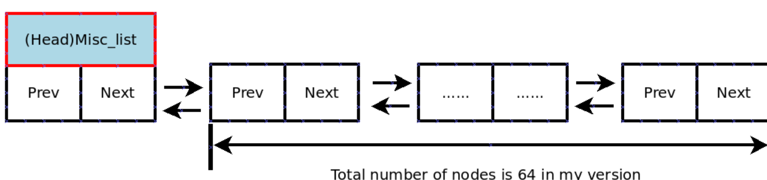
上面就是我定义的数据结构，我就是把struct miscdevice 嵌套在我自己定义的头文件中。

二、杂项设备的注册过程

在介绍注册过程时，我们先来了解一些非常重要的全局变量。

```
static LIST_HEAD(misc_list);
/*************************************************
*#define LIST_HEAD_INIT(name) { &(name), &(name) }
*
*#define LIST_HEAD(name) \
*   struct list_head name = LIST_HEAD_INIT(name) //可以看到这里偷偷的定义了一个为name的struct list_head结构
*************************************************/
 static DEFINE_MUTEX(misc_mtx);
/************************************************************
* #define DEFINE_MUTEX(mutexname) \
*    struct mutex mutexname = __MUTEX_INITIALIZER(mutexname) //同样这里也定义了一个互斥锁。
************************************************************/
 /*
 * Assigned numbers, used for dynamic minors
 */
#define DYNAMIC_MINORS 64 /* like dynamic majors */
static DECLARE_BITMAP(misc_minors, DYNAMIC_MINORS);
/***********************************************
*#define DECLARE_BITMAP(name,bits) \
*    unsigned long name[BITS_TO_LONGS(bits)] //这里这个宏BITS_TO_LONGS(64) 我在机器上验证结果是2
*所以，上面相当与直接定义了 unsigned long misc_minors[2];//而我所验证的平台上unsigned long大小为4个字节，这样的话可
************************************************/
```

misc_list:这是所有misc设备的头指针，打个比方说，所有misc设备结构体都挂在它上面。我画了一个简图如下所示。在我的内核版本中，我打印log发现，它只定义了64个标志位，来标示次设备号的使用情况。网上有人说次设备号最多有255个，但不管它有多少个吧，原理都是一样的。

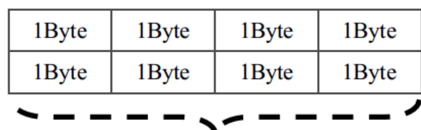


注册的过程是用下面这个函数来实现的，具体的注释我都打在代码中了，自己理解吧^_^

```
int misc_register(struct miscdevice * misc)
{
 dev_t dev;
 int err = 0;

 INIT_LIST_HEAD(&misc->list); //struct list_head 初始化，这个当我们有这个结构的话，要记着用这个宏初始化一下。

 mutex_lock(&misc_mtx); //同样互斥锁也要初始化，当然我们如果用到的话，也要这样初始化。
```

```
 if (misc->minor == MISC_DYNAMIC_MINOR) { //我们在驱动中配置的就是MISC_DYNAMIC_MINOR(动态分配)
  int i = find_first_zero_bit(misc_minors, DYNAMIC_MINORS); //其中的原理我们不深究，这里就是找一个没有使用的次设备
  if (i >= DYNAMIC_MINORS) {
   mutex_unlock(&misc_mtx);
   return -EBUSY;
  }
  misc->minor = DYNAMIC_MINORS - i - 1;
  set_bit(i, misc_minors); //请看下面详细代码。
 } else {
  struct miscdevice *c;

  list_for_each_entry(c, &misc_list, list) { //如果上面不是动态注册，就遍历整个misc_list，确定是否已经注册过了。
   if (c->minor == misc->minor) {
    mutex_unlock(&misc_mtx);
    return -EBUSY;
   }
  }
 }

 dev = MKDEV(MISC_MAJOR, misc->minor);//主设备好和次设备号，加工成真正的设备号。

 misc->this_device = device_create(misc_class, misc->parent, dev, //这个函数是非常重要的。
       misc, "%s", misc->name);
 if (IS_ERR(misc->this_device)) {
  int i = DYNAMIC_MINORS - misc->minor - 1;
  if (i < DYNAMIC_MINORS && i >= 0)
   clear_bit(i, misc_minors);
  err = PTR_ERR(misc->this_device);
  goto out;
 }

 /*
  * Add it to the front, so that later devices can "override"
  * earlier defaults
  */
 list_add(&misc->list, &misc_list); //最后将该设备加入全局misc设备链表中
 out:
 mutex_unlock(&misc_mtx); //解锁
 return err;
}

//下面一段的函数的意思就是把数组中的对应的标志位置位，表示这个次设备号已经使用了。
static inline void set_bit(int nr, volatile unsigned long *addr)
{
    //#define BIT_MASK(nr)        (1UL << ((nr) % BITS_PER_LONG))
    unsigned long mask = BIT_MASK(nr); //找到次设备号偏移，如上宏定义
    unsigned long *p = ((unsigned long *)addr) + BIT_WORD(nr);//这里如果nr>=32,这里就会+1,(类型是long，所以实际
    unsigned long flags;

     //#define BIT_MASK(nr)        (1UL << ((nr) % BITS_PER_LONG)) 即1<<(nr % 32)

     _atomic_spin_lock_irqsave(p, flags); //加了保护
    *p  |= mask; //对应标志为置1.
    _atomic_spin_unlock_irqrestore(p, flags);
}
```

假如现在我们创建的次设备号是**20**，那么标志位偏移量就是**1<<20**，那么就会有上面的**misc_minors[0]** =**misc_minors[0] | 1<<20**;同样假如现在我们创建的次设备号是**40**，那么就会有**mask = 1<<(40%32)**,即**mask = 1<<8**,就有**misc_minors[1] = misc_minors[1] | 1<<8**。下面这幅图是根据我所使用的内核版本，我使用的内核里面只定义了64个标志位，这里也就画了8个字节。但是不管有多少个字节吧，原理还是一样的。



每一行 32 个标志位，记录 32 个次设备标志
第一行：0 ~ 31，第二行 32~63

可以看看/dev目录下的设备节点主设备是10，次设备号是42.这样的话，标志位就放在第二排了。

```
crw------- root     root      10,  42 2012-01-02 01:19 rt8547_led
```

三、引人深思的**misc_class**

misc_class就是一个引子，创建了这样一个class之后，所有的misc设备的class设备文件都会在/sys/class/misc目录下创建。

```
static struct class *misc_class; //全局的变量

static int __init misc_init(void)
{
 int err;

#ifdef CONFIG_PROC_FS
 proc_create("misc", 0, NULL, &misc_proc_fops); //如果允许proc文件系统，这里也会创建proc文件夹。
#endif
 misc_class = class_create(THIS_MODULE, "misc");//这里就看到了，创建了全局的class文件。
 err = PTR_ERR(misc_class);
 if (IS_ERR(misc_class))
  goto fail_remove;

 err = -EIO;
 if (register_chrdev(MISC_MAJOR,"misc",&misc_fops)) //这里还是调用字符设备的注册接口，只不过这里主设备号始终就是10
  goto fail_printk;
 misc_class->devnode = misc_devnode;
 return 0;

fail_printk:
 printk("unable to get major %d for misc devices\n", MISC_MAJOR);
 class_destroy(misc_class);
fail_remove:
 remove_proc_entry("misc", NULL);
 return err;
}
subsys_initcall(misc_init); //这里预示系统刚起来时，就会调用这里的初始化函数，确保在所有misc设备初始化之前，misc_cla
```

只要有了misc_class对象，所有misc设备就有地方放了。

四.杂项设备的注销过程

杂项设备的注销过程，其实就是注册过程的逆向过程。只是说这里代码非常简单，其实主要工作都在device_destroy()中。

```
int misc_deregister(struct miscdevice *misc)
{
 int i = DYNAMIC_MINORS - misc->minor - 1;

 if (WARN_ON(list_empty(&misc->list)))
  return -EINVAL;

 mutex_lock(&misc_mtx);
 list_del(&misc->list);
 device_destroy(misc_class, MKDEV(MISC_MAJOR, misc->minor));
 if (i < DYNAMIC_MINORS && i >= 0)
  clear_bit(i, misc_minors);
 mutex_unlock(&misc_mtx);
 return 0;
}

void device_destroy(struct class *class, dev_t devt)
{
 struct device *dev;

 dev = class_find_device(class, NULL, &devt, __match_devt);
 if (dev) {
  put_device(dev);
  device_unregister(dev);
 }
}
```

上面就是根据misc_class，加上设备的次设备好，就可以找到代表这个设备的struct device结构，然后就可以将它从misc_list中移除。这样就完成了设备注销。当然注销过程也包含了一些属性文件的删除，只是这里没有体现出来。

五.Rt8547闪光灯misc设备驱动的分析

下面的这个例子是我在添加camera 闪光灯rt8547设备时，编写的一个闪光灯驱动的初版文件。初步的功能我已经验证过了，是可以工作的。

1）open函数中的共享精神

下面限于篇幅问题，我不会贴出所有代码，只是加上自己的一些分析，和在编码过程中遇到的问题。

```
rt8547_dev_t *led_dev = NULL; //全局的闪光灯设备结构体

static int led_open(struct inode *node, struct file *file){

 struct my_platfrom_struct *my_data = kmalloc(sizeof(struct my_platfrom_struct), GFP_KERNEL);
 if (!my_data){
  RT8547_ERR("malloc my_data falied!");
 }
 my_data->grade = 1;
 my_data->age = 22;
 my_data->phone = 110;
 file->private_data = my_data;
 RT8547_INFO("open led devices success!!!\n");
 return 0;
}
```

上面代码中，我最想强调的就是file->private_data = my_data.这行代码鬼使神功的将我们需要共享的变量，放到了其它函数都能看得到的地方。因为在其它的read，write，ioctl，release等函数都可以通过file->private_data拿到这个共享的变量。一定要学会共享，理解共享的精神。

2）ioctl函数中的多元化

```
static long led_ioctl(struct file *file, unsigned int cmd,unsigned long arg){
 struct my_platfrom_struct *para = (struct my_platfrom_struct *)arg;
 struct my_platfrom_struct *my_dat = (struct my_platfrom_struct *)file->private_data; //这里我们就拿到了共享变

 switch(cmd){
  case LED_IO_GET_MY_DATA:
   copy_to_user(para,my_dat,sizeof(struct my_platfrom_struct));
   break;
  case LED_IO_OPEN_TORCH:
   {
    int level;
    copy_from_user(&level,(void*)arg,sizeof(int));
    rt8547_send_data(led_dev,RT8547_CONTROL3,RT8547_MODE_MASK,1);//Torch Mode
    rt8547_send_data(led_dev,RT8547_CONTROL3,0x0f,level&0x0f);
    RT8547_INFO("Open Torch, level:%d",level);
    rt8547_set_led_on(led_dev);
   }
   break;
  case LED_IO_OPEN_FLASH:
   {
    int level;
    copy_from_user(&level,(void*)arg,sizeof(int));
    rt8547_send_data(led_dev,RT8547_CONTROL3,RT8547_MODE_MASK,0);//flash Mode
    rt8547_send_data(led_dev,RT8547_CONTROL2,0x1f,level&0x1f);
    RT8547_INFO("Open Flash, level:%d",level);
    rt8547_set_led_on(led_dev);
   }
   break;
  case LED_IO_CLOSE_LED:
   rt8547_set_led_off(led_dev);
   RT8547_INFO("rt8547 close device!\n");
   break;
  }
 return 0;
}
```

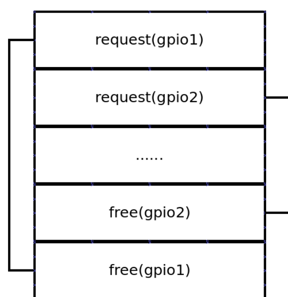
ioctl函数在驱动中用到最多，在实际开发中一般都是通过ioctl和kernel交互的。

3）Rt8547 gpio 初始化和释放

gpio 初始化相当重要，不初始化就不能用了。gpio的初始化和释放所有kernel都是一样的过程，大家要熟记这些方法。

```
static void rt8547_gpio_deinit(const rt8547_dev_t *dev)
{
 gpio_direction_input(dev->flen);
 gpio_free(dev->flen);

 gpio_direction_input(dev->flset);
 gpio_free(dev->flset);
}

static int rt8547_gpio_init(const rt8547_dev_t *dev)
{
 int ret;
 gpio_request(dev->flen,NULL); //请求gpio
 if (ret) {
  RT8547_INFO("sensor: flen already request %d.\n",dev->flen);
  goto request_flen_fail;
 }
 gpio_direction_output(dev->flen, 1); //设置gpio的输出输出方向
 gpio_set_value(dev->flen,0); //设置为高电平还是低电平

 gpio_request(dev->flset,NULL); //以下同理
 if (ret) {
  printk("sensor: flset already request %d.\n",dev->flen);
  goto request_flset_fail;
 }
 gpio_direction_output(dev->flset, 1);//out
 gpio_set_value(dev->flset,0);
 return 0;

request_flset_fail:
 gpio_free(dev->flset);
request_flen_fail:
 gpio_free(dev->flen);
request_fail:
 return ret;
}
```

注意：大家有没有注意到上面函数gpio口的释放，有什么异样没有。也许你已经发现，谁先注册的，就放到最后在释放，这也是为了kernel的稳定性才这样写的。当有一个gpio口请求失败，那么之前注册成功的gpio资源都要释放。要不然也许有其它模块在请求这个gpio口资源呢。记住注册过程和释放过程是相对的。如下所示。



4）led_fops和misdevice设备填充

```
static const struct file_operations led_fops = {
 .owner = THIS_MODULE, //一般都是THIS_MODULE
 .open = led_open,
 .read = led_read,
 .write = led_write,
 .unlocked_ioctl = led_ioctl,
 .release = led_release,
};//上面都是填充一些关键的函数指针，只要知道怎么用就行了。

static struct miscdevice led_device = {
 .minor = LED_MINOR, //次设备号
 .name = LED_DEVICE_NAME, //设备名字宏定义
 .fops = &led_fops, //文件操作指针集合，如上面的结构
};
```

上面两个结构是字符设备必不可少的两个结构体。大家有没有发现这两个结构一般都放在靠近**驱动模块末尾的位置**。不知道大家有没有考虑过为什么会有这样的情况。这是由于驱动中一般没有声明open,read,write,ioctl等接口函数，如果你不想将file_operations安排在驱动末尾。你也可以声明这些驱动函数，这样就不会受限制了。这个要看自己了，一般不这样来，这也是一种习惯。

5）rt8547驱动初始化函数和注销函数

```
int led_init(){
 int ret;
 RT8547_INFO("led_init!\n");
 led_dev =  kzalloc(sizeof(rt8547_dev_t), GFP_KERNEL);
 if(!led_dev){
  RT8547_ERR("alloc mem failed");
 }

 led_dev->led_dev = &led_device;
 led_dev->flset= GPIO_CAM_FLASH_SET;
 led_dev->flen = GPIO_CAM_FLASH_EN;
 led_dev->regs[RT8547_DUMMY] = 0;
 led_dev->regs[RT8547_CONTROL1] = 0x06; //这里模拟寄存器
 led_dev->regs[RT8547_CONTROL2] = 0x12;
 led_dev->regs[RT8547_CONTROL3] = 0x02;
 led_dev->regs[RT8547_CONTROL4] = 0x0f;

 //rt8547_parse_dt(led_dev->led_dev->this_device->of_node,led_dev);
 rt8547_gpio_init(led_dev);
 mutex_init(&led_dev->led_lock);
 ret = misc_register(led_dev->led_dev);
 if (unlikely(ret)) {
  RT8547_ERR("failed to register misc device'%s'!\n",
    led_dev->led_dev->name);
  goto out_free_buffer;
 }

 ret = device_create_file(led_dev->led_dev->this_device, &dev_attr_flash); //创建属性文件
 if (ret < 0) {
  printk(KERN_ALERT"Failed to create attribute flash.");
  goto err_register_flash;
 }

 RT8547_INFO("create device attribute file flash!\n");
 return 0;
err_register_flash:

out_free_buffer:
 kfree(led_dev);
 return ret;
}

void led_exit(void)
{
 RT8547_INFO("rt8547 exit!\n");
 misc_deregister(led_dev->led_dev);
 kfree(led_dev);
}
```

驱动初始化函数中一般进行全局设备结构题的初始化，资源的请求，设备树的解析。当然如果是platform设备，这些也可以在probe函数中做。大家有没有发现我在上面的初始化函数中使用device_create_file()创建了一个设备属性文件。后面我会专门写一篇博文专门来分析这些和创建设备文件相关的函数。

6）应用程序

```
#include <fcntl.h>
#include <errno.h>
#include <sys/ioctl.h>
#include <stdio.h>
#include "led_rt8547.h"

#define FILE_PATH "/dev/rt8547_led"

int main(){
 int ret;
 int mode,level;
 my_data my_dat;

 int fd = open(FILE_PATH,O_RDWR);
 if ( -1 == fd){
  printf("open led_flash node failed!\n");
  return -1;
 }
 printf("unsigned size:%d\n",sizeof(unsigned long));

 printf("\n1:GET_MY_DATA\n \
  2:LED_OPEN_TORCH\n \
  3:LED_OPEN_FLASH\n \
  4:LED_CLOSE\n");
 printf("mode:");
 scanf("%d",&mode);
 if(2 == mode || 3 == mode){
  printf("level:");
  scanf("%d",&level);
 }
 printf("Rt8547 mode:%d,level:%d\n",mode,level);
 switch(mode){
  case 1:
   ret = ioctl(fd,LED_IO_GET_MY_DATA, &my_dat);
   printf("Rt8547 Get my data success!\n");
   printf("Rt8547 grade:%d,age:%d,phone:%d",my_dat.grade,my_dat.age,my_dat.phone);
   if(0 != ret){
    printf("Rt8547 Get my data failed!\n");
    ret = -1;
   }
   break;
  case 2:
   ret = ioctl(fd,LED_IO_OPEN_TORCH, &level);
   printf("Rt8547 open torch success!\n");
   if(0 != ret){
    printf("Rt8547 open torch failed!\n");
    ret = -1;
   }
   break;
  case 3:
   ret = ioctl(fd,LED_IO_OPEN_FLASH, &level);
   printf("Rt8547 open flash success!\n");
   if(0 != ret){
    printf("Rt8547 open flash failed!\n");
    ret = -1;
   }
   break;
```

```
  case 4:
   ret = ioctl(fd,LED_IO_CLOSE_LED, &level);
   printf("Rt8547 close led success!\n");
   if(0 != ret){
    printf("Rt8547 close led failed!\n");
    ret = -1;
   }
   break;
 }
 return ret;
}
```

上面调用系统调用，一切都在代码中，read the fucking code!!!!!!

五.调试过程中遇到的问题及解决办法和思路

1.Android.mk编写问题

在Anrdroid大环境下，最主要的问题是找不到头文件的问题。一开始的编译的时候，找不到头文件，后来参考其它工程的Android.mk发现他们一般都包含了$(**TARGET_OUT_INTERMEDIATES**)/KERNEL/source/include/video(如下所示)，大家一看都明白，这里就是存放头文件的地方。其实在编译的过程中，kernel的头文件都会拷贝到out目录对应工程的/KERNEL/source/include目录下，这些头文件和kernel源码中是一样的。大家一定要牢记。我在编译led_flash测试程序时，用的就是下面这个Android.mk

其中LOCAL_MODULE_TAGS := optional意思就是所有版本都编译该模块。

知识扩展：
user: 指该模块只在user版本下才编译
eng: 指该模块只在eng版本下才编译
tests: 指该模块只在tests版本下才编译
optional:指该模块在 所有版本下都编译，默认是optional

```
LOCAL_PATH:= $(call my-dir)

include $(CLEAR_VARS)//清除环境变量
#LOCAL_ADDITIONAL_DEPENDENCIES := $(common_deps)

#头文件路径
LOCAL_C_INCLUDES := \
  $(TARGET_OUT_INTERMEDIATES)/KERNEL/source/include/misc \
  $(TARGET_OUT_INTERMEDIATES)/KERNEL/source/include/video

LOCAL_SRC_FILES := led_flash.c  #源文件
LOCAL_MODULE := led_flash #应用程序，编译成可执行程序时的名字
LOCAL_MODULE_TAGS := optional

include $(BUILD_EXECUTABLE)
```

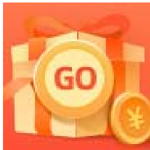
2）Kernel头文件编写问题

在上面Android.mk中包含了头文件路径时，有时候还是会提示找不到一些头文件。比如我在led_rt8547.h头文件中添加了一个#include <linux/mutex.h>。编译的时候找不到mutex.h这个头文件。这是因为我们在Android.mk中根本就没有包含mutex.h头文件所在的路径。为此我有以下几点意见：

1.应用程序一般都是使用IOCTL命令和一些宏信息，所以在驱动模块的头文件中，不要添加一些和kernel相关的结构体，尽量使用我们自己定义的数据类型。

2.头文件尽量少包含与kernel紧密联系的头文件(有联系的就要将头文件路径加到android.mk中)，要不然也找不到头文件(除非已经包含了头文件路径)，我们自己定义的其它头文件是可以添加进来。

3.模块中和内核紧密联系的结构体，**尽量放在驱动源码中**，避免编译报错。应用层其实也用不到这些结构体，那么就不要放到头文件中了。



创作打卡挑战赛

赢取流量/现金/CSDN周边激励大奖