专题13-总线设备驱动模型

一、总线设备驱动模型

1.1、总线模型概述

随着技术的不断进步，系统的拓扑结构也越来越复杂，对热插拔，跨平台移植性的要求也越来越高，2.4内核已经难以满足这些需求。为适应这种形势的需要，从Linux 2.6内核开始提供了全新的设备模型。

1.2、总线

1.2.1、描述结构

在 Linux 内核中, 总线由 bus_type 结构表示,定义在 <linux/device.h>

```
struct bus_type {
    const char *name; /*总线名称*/
    int (*match) (struct device *dev, struct device_driver *drv); /*驱动与设备的匹配函数*/
    ……
}
```

```
int (*match)(struct device * dev, struct device_driver * drv)
```

当一个新设备或者新驱动被添加到这个总线时，该函数被调用。用于判断指定的驱动程序是否能处理指定的设备。若可以，则返回非零。

1.2.2、注册

总线的注册使用如下函数

```
bus_register(struct bus_type *bus)
```

若成功，新的总线将被添加进系统，并可在 /sys/bus 下看到相应的目录。

1.2.3、注销

总线的注销使用：

```
void bus_unregister(struct bus_type *bus)
```

1.3、驱动

1.3.1、描述结构

在 Linux内核中, 驱动由 device_driver结构表示。

```
struct device_driver {
{
    const char *name; /*驱动名称*/
    struct bus_type *bus; /*驱动程序所在的总线*/
    int (*probe) (struct device *dev);
    ……
}
```

1.3.2、注册

驱动的注册使用如下函数

```
int driver_register(struct device_driver *drv)
```

1.3.3、注销

驱动的注销使用：

```
void driver_unregister(struct device_driver *drv)
```

1.4、设备

1.4.1、描述结构

在 Linux内核中, 设备由struct device结构表示。

```
struct device {
{
    const char *init_name; /*设备的名字*/
    struct bus_type *bus; /*设备所在的总线*/
    ……
}
```

1.4.2、注册

设备的注册使用如下函数

```
int device_register(struct device *dev)
```

1.4.3、注销

设备的注销使用：

```
void device_unregister(struct device *dev)
```

设备和驱动的名字需要一样！

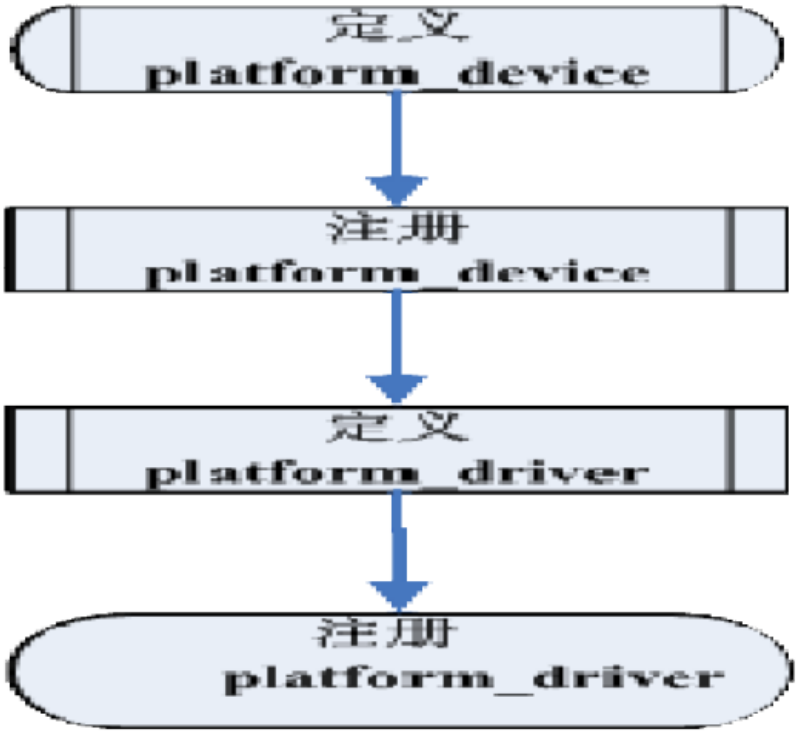
设备和驱动可以不分先后挂载。

二、平台总线设计

2.1、平台总线概述

平台总线(Platform bus)是linux2.6内核 加入的一种虚拟总线，其优势在于采用了 总线的模型对设备与驱动进行了管理，这 样提高了程序的可移植性。

通过平台总线机制开发设备驱动的流程如图




```
struct bus_type platform_bus_type = {
    .name = "platform",
    .dev_attrs = platform_dev_attrs,
    .match = platform_match,
    .uevent = platform_uevent,
    .pm = &platform_dev_pm_ops,
};
```

2.2、平台设备

平台设备使用struct platform_device来描述:

```
struct platform_device {
    const char *name; /*设备名*/
    int id; /*设备编号，配合设备名使用*/
    struct device dev;
    u32 num_resources;
    struct resource *resource; /*设备资源*/
}

struct resource {
    resource_size_t start;
    resource_size_t end;
    const char *name;
    unsigned long flags; /*资源的类型*/
    struct resource *parent, *sibling, *child;
```

```
};
```

注册平台设备，使用函数:

```
int platform_device_register(struct platform_device *pdev)
```

2.3、平台驱动

平台驱动使用struct platform_driver 描述：

```
struct platform_driver {
    int (*probe)(struct platform_device *);
    int (*remove)(struct platform_device *);
    ……
}
```

平台驱动注册使用函数：

```
int platform_driver_register(struct platform_driver *)
```

2.4、将按键驱动修改为平台驱动模型

Makefile：

```
obj-m := key_dev.o key_drv.o

KDIR := /home/S5-driver/lesson7/linux-ok6410

all:
    make -C $(KDIR) M=$(PWD) modules CROSS_COMPILE=arm-linux- ARCH=arm

clean:
    rm -f *.order *.symvers *.mod.o *.o *.ko *.mod.c
```

key_dev.c:

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/interrupt.h>

#define GPNCON 0x7f008830
#define GPNDAT 0x7f008834

struct resource key_resource[] = {
 [0] = {
  .start = GPNCON,
  .end = GPNCON + 8,
  .flags = IORESOURCE_MEM,
 },

 [1] = {
  .start = S3C_EINT(0),
  .end = S3C_EINT(1),
  .flags = IORESOURCE_IRQ,
 },
};

struct platform_device key_device = {
 .name = "my-key",
 .id = 0,
 .num_resources = ARRAY_SIZE(key_resource),
 .resource = key_resource,
};

static int __init keydev_init(void)
{
 platform_device_register(&key_device);

 return 0;
}

static void __exit keydev_exit(void)
```

```
{
 platform_device_unregister(&key_device);
}

module_init(keydev_init);
module_exit(keydev_exit);

MODULE_LICENSE("GPL");
```

key_drv.c:

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/miscdevice.h>
#include <linux/interrupt.h>
#include <linux/fs.h>
#include <linux/io.h>
#include <linux/workqueue.h>
#include <linux/slab.h>
#include <linux/timer.h>
#include <linux/uaccess.h>
#include <linux/wait.h>
#include <linux/sched.h>
#include <linux/platform_device.h>


struct work_struct * work1;

struct timer_list key_timer;

unsigned int key_num = 0;

wait_queue_head_t key_q;

struct resource * res_irq;
struct resource * res_mem;
unsigned int * key_base;

void work1_func(struct work_struct * work)
{
    mod_timer(&key_timer, jiffies + HZ/10);
}

void key_timer_func(unsigned long data)
{
    unsigned int key_val;

    key_val = readl(key_base + 1) & 0b11;

    if (key_val == 0b10) {
        key_num = 1;
    }

    if (key_val == 0b01) {
        key_num = 2;
    }

    wake_up(&key_q);
}

static irqreturn_t key_int(int irq, void *dev_id)
{
    /*Check if a key interrupt has occurred */

    /*Clear key interrupts that have occurred(If it is a CPU internal interrupt (non-peripheral), the system will help clear) */

    /*Submit the bottom half */
        /*queue work*/
    schedule_work(work1);
```

```
    //return 0;
    return IRQ_HANDLED;
}

void key_hw_init(void)
{
    unsigned int data;

    data = readl(key_base);
    data &= ~0b1111;                    //set key1 and key2
    data |= 0b1010;
    writel(data, key_base);

    //gpio_data = ioremap(GPNDAT, 4);
}

int key_open(struct inode * node, struct file * filp)
{
    return 0;
}

ssize_t key_read (struct file * filp, char __user * buf, size_t size, loff_t * pos)
{
    wait_event(key_q, key_num);

    copy_to_user(buf, &key_num, 4);

    key_num = 0;

    return 4;
}

ssize_t key_write (struct file * filp, const char __user * buf, size_t size, loff_t * pos)
{
    return 0;
}

int key_close (struct inode * node, struct file * filp)
{
    return 0;
}

struct file_operations key_fops = {
    .open = key_open,
    .read = key_read,
    .write = key_write,
    .release = key_close,
};


struct miscdevice key_miscdev = {
    .minor = 200,
    .name = "key",
    .fops = &key_fops,
};

static int key_probe(struct platform_device * pdev)
{
    int ret;
    int size;

    ret = misc_register(&key_miscdev);

    if (ret !=0)
        printk(KERN_WARNING"register fail!\n");

    res_irq = platform_get_resource(pdev, IORESOURCE_IRQ, 0);
```

```
    request_irq(res_irq->start, key_int, IRQF_TRIGGER_FALLING, "key", (void *)1);
    request_irq(res_irq->end, key_int, IRQF_TRIGGER_FALLING, "key", (void *)2);

    /*init key*/
    res_mem = platform_get_resource(pdev, IORESOURCE_MEM, 0);
    size = res_mem->end - res_mem->start + 1;
    key_base = ioremap(res_mem->start, size);
    key_hw_init();

    /*init work*/
    work1 = kmalloc(sizeof(struct work_struct), GFP_KERNEL);

    INIT_WORK(work1, work1_func);

    /*init timer */
    init_timer(&key_timer);
    key_timer.function = key_timer_func;

    /*register timer */
    add_timer(&key_timer);

    /*init wait queue*/
    init_waitqueue_head(&key_q);

    return ret;
}

static int key_remove(struct platform_device *pdev)
{
    free_irq(res_irq->start, (void *)1);

    free_irq(res_irq->end, (void *)2);

    iounmap(key_base);

    misc_deregister(&key_miscdev);

    return 0;
}

static struct platform_driver key_driver = {
    .probe    = key_probe,
    .remove    = __devexit_p(key_remove),
    .driver = {
        .name = "my-key",
        .owner = THIS_MODULE,
    },
};

static int __init ok6410_key_init(void)
{
    //printk(KERN_WARNING"key init\n");

    return  platform_driver_register(&key_driver);

}

static void __exit ok6410_key_exit(void)
{
    //printk(KERN_WARNING"key exit\n");

    platform_driver_unregister(&key_driver);
}


module_init(ok6410_key_init);
module_exit(ok6410_key_exit);
MODULE_LICENSE("GPL");
```

key_app.c:

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int fd;
    int key_num;

    /*open device*/
    fd = open("/dev/ok6410key", 0);

    if (fd < 0)
        printf("open device fail!\n");

    /*read device*/
    read(fd, &key_num, 4);
    printf("key is %d\n", key_num);

    /*close device*/
    close(fd);
}
```