努力成为 linux kernel hacker 的人李万鹏原创作品，为梦而战。转载请标明出处 http://blog.csdn.net/woshixingaaa/archive/2011/05/21/6436172.aspx

首先看一下我的系统中都有什么设备挂在了 platform 虚拟总线上：

```
hacker@hacker:~/linux-2.6.30.4$ cd /sys/bus/platform/
hacker@hacker:/sys/bus/platform$ tree
.
|-- devices
|   |-- Fixed MDIO bus.0 -> ../../../devices/platform/Fixed MDIO bus.0
|   |-- eisa.0 -> ../../../devices/platform/eisa.0
|   |-- i8042 -> ../../../devices/platform/i8042
|   |-- pcspkr -> ../../../devices/platform/pcspkr
|   |-- rtc_cmos -> ../../../devices/platform/rtc_cmos
|   `-- serial8250 -> ../../../devices/platform/serial8250
|-- drivers
|   |-- dsa
|   |   |-- bind
|   |   |-- uevent
|   |   `-- unbind
|   |-- i8042
|   |   |-- bind
|   |   |-- i8042 -> ../../../../devices/platform/i8042
|   |   |-- uevent
|   |   `-- unbind
|   |-- mdio-gpio
|   |   |-- bind
|   |   |-- uevent
|   |   `-- unbind
|   |-- parport_pc
|   |   |-- bind
|   |   |-- module -> ../../../../module/parport_pc
|   |   |-- uevent
|   |   `-- unbind
|   |-- rtc_cmos
|   |   |-- bind
|   |   |-- rtc_cmos -> ../../../../devices/platform/rtc_cmos
|   |   |-- uevent
|   |   `-- unbind
|   |-- serial8250
|   |   |-- bind
|   |   |-- serial8250 -> ../../../../devices/platform/serial8250
|   |   |-- uevent
|   |   `-- unbind
|   `-- twl4030_reg
|       |-- bind
|       |-- uevent
|       `-- unbind
|-- drivers_autoprobe
|-- drivers_probe
`-- uevent

19 directories, 24 files
```

platform 的初始化：首先系统启动的时候会调用 platform_bus_init 来初始化这个虚拟总线，让后向虚拟总线注册即将挂载这条总线上的设备。platform_bus_type 部分是内核为我们实现好的，我们只关系 platform_device 与 platform_driver 就行了。

```
struct bus_type platform_bus_type = {
    .name        = "platform",
    .dev_attrs   = platform_dev_attrs,
    .match       = platform_match,
    .uevent      = platform_uevent,
    .pm      = PLATFORM_PM_OPS_PTR,
};
EXPORT_SYMBOL_GPL(platform_bus_type);

int __init platform_bus_init(void)
{
    int error;

    early_platform_cleanup();

    error = device_register(&platform_bus);
    if (error)
        return error;
    error =  bus_register(&platform_bus_type);
    if (error)
        device_unregister(&platform_bus);
    return error;
}
```

记住总线也是一种设备，所以首先注册总线设备，然后注册总线。

```
static struct platform_device *smdk2410_devices[] __initdata = {
    &s3c_device_usb,
    &s3c_device_lcd,
    &s3c_device_wdt,
    &s3c_device_i2c0,
    &s3c_device_iis,
};
```

把设备挂到 platform 总线上：

```
static void __init smdk2410_init(void)
{
    s3c_i2c0_set_platdata(NULL);
    platform_add_devices(smdk2410_devices, ARRAY_SIZE(smdk2410_devices));
    smdk_machine_init();
}
```

首先来看一个重要的数据结构：

```
struct resource {
    resource_size_t start;    /*资源的起始物理地址*/
    resource_size_t end;      /*资源的结束物理地址*/
    const char *name;         /*资源的名称*/
    unsigned long flags;      /*资源的类型*/
    struct resource *parent, *sibling, *child; /*资源的链表指针*/
};

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 platform_device_id   *id_entry;
};

struct platform_driver {
    int (*probe)(struct platform_device *);
    int (*remove)(struct platform_device *);
    void (*shutdown)(struct platform_device *);
    int (*suspend)(struct platform_device *, pm_message_t state);
    int (*suspend_late)(struct platform_device *, pm_message_t state);
    int (*resume_early)(struct platform_device *);
    int (*resume)(struct platform_device *);
    struct device_driver driver;
    struct platform_device_id *id_table;
};
```

设备的分配：

```
struct platform_device *platform_device_alloc(const char *name, int id);  //name:设备名，id:设备id，一般为-1
```

设备的注册：

```
int platform_device_add(struct platform_device *pdev);
```

获取资源:

```
struct resource *platform_get_resource(struct platform_device *dev, unsigned int type, unsigned int num);
```

/*dev:资源所属的设备，type:获取的资源类型，num:获取的资源数*/

这里详述 platform_device 与 platform_driver 是怎样匹配上的，这里跟踪函数的执行过程，首先是 platform_driver_register：

```
int platform_driver_register(struct platform_driver *drv)
{
    。。。。。。。。。。
    return driver_register(&drv->driver);
}
int driver_register(struct device_driver *drv)
{
    。。。。。。。。。。。
    ret = bus_add_driver(drv);
    。。。。。。。。。。。
}
int bus_add_driver(struct device_driver *drv)
{
    。。。。。。。。。。。。
    if (drv->bus->p->drivers_autoprobe) {
        error = driver_attach(drv);
        if (error)
            goto out_unregister;
    }
    。。。。。。。。。。。。
}
int driver_attach(struct device_driver *drv)
{
    return bus_for_each_dev(drv->bus, NULL, drv, __driver_attach);
}
```

这里来看__driver_attach 这个函数，其中分别调用了 driver_match_device，driver_probe_device 函数。如果匹配成果调用 probe 函数，否则返回。

```
static int __driver_attach(struct device *dev, void *data)
{
    struct device_driver *drv = data;

    /*
     * Lock device and try to bind to it. We drop the error
     * here and always return 0, because we need to keep trying
     * to bind to devices and some drivers will return an error
     * simply if it didn't support the device.
     *
     * driver_probe_device() will spit a warning if there
     * is an error.
     */

    if (!driver_match_device(drv, dev))
        return 0;

    if (dev->parent) /* Needed for USB */
        down(&dev->parent->sem);
    down(&dev->sem);
    if (!dev->driver)
        driver_probe_device(drv, dev);
    up(&dev->sem);
    if (dev->parent)
        up(&dev->parent->sem);

    return 0;
}
```

匹配的时候调用的bus的match函数。

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

找到platform_match:

```
static int platform_match(struct device *dev, struct device_driver *drv)
{
    struct platform_device *pdev = to_platform_device(dev);
    struct platform_driver *pdrv = to_platform_driver(drv);

    /* match against the id table first */
    if (pdrv->id_table)
        return platform_match_id(pdrv->id_table, pdev) != NULL;

    /* fall-back to driver name match */
    return (strcmp(pdev->name, drv->name) == 0);
}
```

最后一行可以看到通过pdev->name与drv->name进行匹配，也就是说是通过设备与驱动的名字进行匹配。匹配成功后调用驱动的probe函数。

```
int driver_probe_device(struct device_driver *drv, struct device *dev)
{
    。。。。。。。。。。
    ret = really_probe(dev, drv);
    。。。。。。。。。
}
static int really_probe(struct device *dev, struct device_driver *drv)
{
    。。。。。。。。。
    if (dev->bus->probe) {
        ret = dev->bus->probe(dev);
        if (ret)
            goto probe_failed;
    } else if (drv->probe) {
        ret = drv->probe(dev);
        if (ret)
            goto probe_failed;
    }
    。。。。。。。。
}
```

由 relly_probe 函数可以看出，如果 bus 定义了 probe 函数，则调用 bus 的 probe 函数；如果 bus，没有定义而 driver 定义了 probe 函数，则调用 driver 的 probe 函数。由上边的 platform_bus_type 可以看出 bus 并没有定义 probe 函数，所以调用 driver 的 probe 函数。

测试程序：

device.c

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/device.h>
#include <linux/string.h>
#include <linux/module.h>
#include <linux/platform_device.h>

static struct platform_device *my_device;

static int __init platform_dev_init(void) {
    int ret;

    //分配结构
    my_device = platform_device_alloc("my_dev", -1);
    //注册设备
    ret = platform_device_add(my_device);

    if(ret)
        printk("platform_device_add failed!/n");

    return ret;
}

static void __exit platform_dev_exit(void) {
    platform_device_unregister(my_device);//卸载设备
}

module_init(platform_dev_init);
```

```
module_exit(platform_dev_exit);
MODULE_LICENSE("GPL");
```

driver.c

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

static int my_probe(struct device *dev) {
    printk("Driver found device!/n");
    return 0;
}

static int my_remove(struct device *dev) {
    printk("Driver found device unpluged!/n");
    return 0;
}
//定义 platform_driver 结构体
static struct platform_driver my_driver = {
    .probe  = my_probe,
    .remove = my_remove,
    .driver = {
        .owner = THIS_MODULE,
        .name = "my_dev",
    },
};

static int __init my_driver_init(void) {
    return platform_driver_register(&my_driver);
}

static void __exit my_driver_exit(void) {
    platform_driver_unregister(&my_driver);
}

module_init(my_driver_init);
module_exit(my_driver_exit);
MODULE_LICENSE("GPL");
```

测试效果：

```
root@hacker:/home/hacker/platform# insmod driver.ko
root@hacker:/home/hacker/platform# insmod device.ko
root@hacker:/home/hacker/platform# dmesg
[ 4499.724439] Driver found device!
root@hacker:/home/hacker/platform# rmmod driver.ko
```

```
root@hacker:/home/hacker/platform# dmesg
[ 4499.724439] Driver found device!
[ 4513.368712] Driver found device unpluged!
root@hacker:/home/hacker/platform# rmmod device.ko

root@hacker:/home/hacker/platform# insmod device.ko
root@hacker:/home/hacker/platform# insmod driver.ko
root@hacker:/home/hacker/platform# dmesg
[ 4540.509227] Driver found device!
root@hacker:/home/hacker/platform# rmmod device.ko
root@hacker:/home/hacker/platform# dmesg
[ 4540.509227] Driver found device!
[ 4545.786076] Driver found device unpluged!
root@hacker:/home/hacker/platform# rmmod driver.ko
root@hacker:/home/hacker/platform# dmesg
[ 4540.509227] Driver found device!
[ 4545.786076] Driver found device unpluged!
```