Linux 设备模型浅析之驱动篇

Linux 设备模型，仅仅看理论介绍，比如 LDD3 的第十四章，会感觉太抽象不易理解，而通过阅读内核代码就更具体更易理解，所以结合理论介绍和内核代码阅读能够更快速的理解掌握 linux 设备模型。这一序列的文章的目的就是在于此，看这些文章之前最好能够仔细阅读 LDD3 的第十四章。大部分 device 和 driver 都被包含在一个特定 bus 中，platform_device 和 platform_driver 就是如此，包含在 platform_bus_type 中。这里就以对 platform_bus_type 的调用为主线，浅析 platform_driver 的注册过程，从而理解 linux 设备模型。platform_bus_type 用于关联 SOC 的 platform device 和 platform driver，比如在内核 linux-2.6.29 中所有 S3C2410 中的 platform device 都保存在 devs.c 中。这里就以 S3C2410 RTC 的驱动程序 rtc-s3c.c 为例来分析 platform_driver_register()例程的调用过程。在文章的最后贴有一张针对本例的 device model 图片，可在阅读本文章的时候作为参照。阅读这篇文章之前，最好先阅读文章《Linux 设备模型浅析之设备篇》。

一、**S3C2410 RTC** 的 **platform_driver** 定义在 drivers/rtc/rtc-s3c.c 中，代码如下：

```
static struct platform_driver s3c2410_rtc_driver = {
        .probe          = s3c_rtc_probe,
        .remove         = __devexit_p(s3c_rtc_remove),
        .suspend        = s3c_rtc_suspend,
        .resume         = s3c_rtc_resume,
        .driver         = {
                .name   = "s3c2410-rtc",
                .owner = THIS_MODULE,
        },
};
```

由于 name = "s3c2410-rtc"，后面会分析到，它将成为一个目录的名字，即/sys/bus/platform/s3c2410-rtc。s3c_rtc_probe ()的调用后面也会讲到，其实在之前《Linux 设备模型浅析之设备篇》中已经分析过了。使用 platform_driver_register()注册 s3c2410_rtc_driver，下面就分析它。

二、**platform_driver_register()**例程定义在 drivers/base/ platform.c 中，代码如下：

```
int platform_driver_register(struct platform_driver *drv)
{
        drv->driver.bus = &platform_bus_type;       // 设置为 platform_bus_type
        if (drv->probe)
                drv->driver.probe = platform_drv_probe;  // 转存到 device_driver 中
        if (drv->remove)
                drv->driver.remove = platform_drv_remove;
        if (drv->shutdown)
                drv->driver.shutdown = platform_drv_shutdown;
        if (drv->suspend)
                drv->driver.suspend = platform_drv_suspend;
        if (drv->resume)
```

```
        drv->driver.resume = platform_drv_resume;
    return driver_register(&drv->driver); // 该例程将 device_driver 注册到 bus，后面将分析
}
```

代码中，

1. 设置成 platform_bus_type 这个很重要，因为 driver 和 device 是通过 bus 联系在一起的，具体在本例中是通过 platform_bus_type 中注册的回调例程和属性来是实现的，在《Linux 设备模型浅析之设备篇》中讲到的 driver 与 device 的匹配就是通过 platform_bus_type 注册的回到例程 mach()来完成的。关于 platform_bus_type 的分析请参考《Linux 设备模型浅析之设备篇》。下面就分析 driver_register()。

三、 **driver_register()**例程定义在 drivers/base/ driver.c 中，代码如下：

```
int driver_register(struct device_driver *drv)
{
    int ret;
    struct device_driver *other;

    // 做些判断
    if ((drv->bus->probe && drv->probe) ||
        (drv->bus->remove && drv->remove) ||
        (drv->bus->shutdown && drv->shutdown))
            printk(KERN_WARNING "Driver '%s' needs updating - please use "
                    "bus_type methods\n", drv->name);

    /*通过驱动的名字查找 driver，如果找到了，说明已经注册过，返回错误代码，后面会分析*/
    other = driver_find(drv->name, drv->bus);
    if (other) {
            put_driver(other);
            printk(KERN_ERR "Error: Driver '%s' is already registered, "
                    "aborting...\n", drv->name);
            return -EEXIST;
    }

    /* 将 driver 加入到 bus 的 kset，并生成些文件夹和链接文件，后面会分析 */
    ret = bus_add_driver(drv);
    if (ret)
            return ret;
    /* 添加 attribute_group，本例中没有设置 drv->groups */
    ret = driver_add_groups(drv, drv->groups);
    if (ret)
            bus_remove_driver(drv);
    return ret;
}
```

代码中，

1. 正如该例程的英文注释所言，大部分工作在 bus_add_driver()例程中完成。bus_add_driver()例程的作用类似于《Linux 设备模型浅析之设备篇》分析过的 bus_add_device()，只不过后者将 device 添得到 bus 中。下面分析 driver_find()例程。

2. driver_find()例程通过驱动所属 bus 的 driver 容器 drivers_kset 来查找，该例程定义在 drivers/base/ driver.c 中，代码如下：

```
struct device_driver *driver_find(const char *name, struct bus_type *bus)
{
        struct kobject *k = kset_find_obj(bus->p->drivers_kset, name);  // 在 drivers_kset 容器中查找
        struct driver_private *priv;

        if (k) {
                priv = to_driver(k);
                return priv->driver;    // 返回找到的 driver
        }
        return NULL;
}
```

代码中，

2.1. 通过 kset_find_obj(bus->p->drivers_kset, name)查找该 driver 的 kobj，其代码如下，

```
struct kobject *kset_find_obj(struct kset *kset, const char *name)
{
        struct kobject *k;
        struct kobject *ret = NULL;

        spin_lock(&kset->list_lock);
        list_for_each_entry(k, &kset->list, entry) {  // 遍历 kset->list 列表获取 kobj
                if (kobject_name(k) && !strcmp(kobject_name(k), name)) { // 比较 name 字符
                        ret = kobject_get(k);  // 如果找到就增加引用并返回
                        break;
                }
        }
        spin_unlock(&kset->list_lock);
        return ret;
}
```

显然，所有同类型的 driver 都注册到了 一个 bus->p->drivers_kset->list 中，所以可通过其查找已经注册的 driver。下面分析 bus_add_driver()例程。

3. bus_add_driver()例程将 driver 注册到 bus 中，在本例中是 platform_bus_type，该例程定义在 drivers/base/ bus.c 中，代码如下：

```
int bus_add_driver(struct device_driver *drv)
{
        struct bus_type *bus;
        struct driver_private *priv;
        int error = 0;

        bus = bus_get(drv->bus);       // 增加对 bus 的引用
        if (!bus)
                return -EINVAL;

        pr_debug("bus: '%s': add driver %s\n", bus->name, drv->name);
```

```
priv = kzalloc(sizeof(*priv), GFP_KERNEL);        // 这个结构体中存放着 kobj 相关的数据
if (!priv) {
        error = -ENOMEM;
        goto out_put_bus;
}
klist_init(&priv->klist_devices, NULL, NULL);
priv->driver = drv;     // 反向指向包含其的 drv，以便后续使用
drv->p = priv;          // 将 priv 保存到 device_driver

/* 指向 bus 的 drivers_kset 容器，该容器的作用与 device_kset 容器相同，前者是包含所
有注册到该 bus 的 driver，后者是包含所有注册到该 bus 的 device。*/
priv->kobj.kset = bus->p->drivers_kset;

/* 初始化 priv->kobj，并将其添加到 bus->p->drivers_kset 中，在本例中生
成/sys/bus/platform/drivers/s3c2410-rtc 目录，后面会分析 drivers_kset 的初始化
及/sys/bus/platform/drivers/目录的生成 */
error = kobject_init_and_add(&priv->kobj, &driver_ktype, NULL,
                             "%s", drv->name);
if (error)
        goto out_unregister;

if (drv->bus->p->drivers_autoprobe) {       // 在 bus_register()例程中已经设置为 1 了
        error = driver_attach(drv);     // 所以会寻找匹配的 device，后面分析
        if (error)
                goto out_unregister;
}

// 将 driver 链接到 klist_drivers，方便后续快速查找 driver
klist_add_tail(&priv->knode_bus, &bus->p->klist_drivers);

// 将 driver 添加到 module 模块，后面会分析
module_add_driver(drv->owner, drv);

/*  生成/sys//sys/bus/platform/drivers/s3c2410-rtc/uevent 属性文件，其作用与 device 中的
uevent 类似，可参照《Linux 设备模型浅析之设备篇》*/
error = driver_create_file(drv, &driver_attr_uevent);
if (error) {
        printk(KERN_ERR "%s: uevent attr (%s) failed\n",
                __func__, drv->name);
}
/* 添加 bus 的公有属性文件到/sys//sys/bus/platform/drivers/s3c2410-rtc/目录，所有的 driver
都添加 */
error = driver_add_attrs(bus, drv);
if (error) {
        /* How the hell do we get out of this pickle? Give up */
        printk(KERN_ERR "%s: driver_add_attrs(%s) failed\n",
                __func__, drv->name);
}
```

```
        /* 如果配置了"CONFIG_HOTPLUG"，则生成“bind”和"unbind”属性文件，可用于手动匹
        配和移除 device 与 driver 之间的关联 */
        error = add_bind_files(drv);
        if (error) {
                /* Ditto */
                printk(KERN_ERR "%s: add_bind_files(%s) failed\n",
                        __func__, drv->name);
        }

        // 通过 uevent 设置几个环境变量并通知用户空间，以便调用程序来完成相关设置
        kobject_uevent(&priv->kobj, KOBJ_ADD);
        return error;
out_unregister:
        kobject_put(&priv->kobj);
out_put_bus:
        bus_put(bus);
        return error;
}
```

代码中，

3.1. 老一点的 2.6 的内核版本如 2.6.10 直接将 kobj 嵌入在 struct device_driver 中，新版的内核用 struct driver_private 来存放 kobj 相关的数据，将 struct driver_private 嵌入在 struct device_driver 中。struct driver_private 定义如下：

```
struct driver_private {
        struct kobject kobj;    // kobj
        struct klist klist_devices;       // 用于链接关联到该 driver 的 device
        struct klist_node knode_bus; // 用于链接到 bus->p->klist_drivers
        struct module_kobject *mkobj;        // 模块的 kobj，后面会讲到 module
        struct device_driver *driver; // 反向指向包含其的 driver
}
```

3.2. 使用 bus_register()例程注册 bus（本例是 platform_bus_type）的时候，除了生成该 bus 的 kset 容器 subsys，还生成 devices_kset 和 drivers_kset 容器，都包含在 struct bus_type_private 里。当然也先后生成他们的目录/sys/bus/platform、/sys/bus/platform/devices、/sys/bus/platform/drivers。先看看 struct bus_type_private 的定义：

```
struct bus_type_private {
        struct kset subsys;     // 代表该 bus，里面的 kobj 是该 bus 的主 kobj，也就是最顶层
        struct kset *drivers_kset;      // 包含该 bus 所有的 driver
        struct kset *devices_kset;      // 包含该 bus 所有的 device
        struct klist klist_devices;     // 其作用与 devices_kset->list 作用相同
        struct klist klist_drivers;     // 其作用与 drivers _kset->list 作用相同
        struct blocking_notifier_head bus_notifier;  // 通知 bus 相关的模块
        unsigned int drivers_autoprobe:1;    // 设置是否在 driver 注册的时候 probe device
        struct bus_type *bus;           // 回指包含自己的 bus
}
```

其实该结构体的英文注释写的很清楚，可到内核的 drivers/base/ base.h 中查看。

3.3. bus_register()例程定义在 drivers/base/bus.c 中 ，部分代码如下：

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

```
{
        int retval;
        struct bus_type_private *priv;

        priv = kzalloc(sizeof(struct bus_type_private), GFP_KERNEL);
        if (!priv)
                return -ENOMEM;

        priv->bus = bus;        // 反向指向 bus
        bus->p = priv;

        // 将 bus 的名字设置为 kobj 的名字，本例中是"platform"
        retval = kobject_set_name(&priv->subsys.kobj, "%s", bus->name);
        if (retval)
                goto out;

        priv->subsys.kobj.kset = bus_kset;  // 指向其父 kset， bus_kset 在 buses_init()例程中添加
        priv->subsys.kobj.ktype = &bus_ktype;       // 设置读取属性文件的默认方法
        priv->drivers_autoprobe = 1;            // 设置为注册时 probe device

        /* 注册 priv->subsys 容器，初试化完后，调用了 kobject_add_internal()注册其 kobj 到
        bus_kset 父容器里，并生成目录/sys/bus/platfrom */
        retval = kset_register(&priv->subsys);

        /* 生成 devices_kset 容器，命名为"devices" ，由于其父 kobj 是 priv->subsys.kobj，所以生
        成的目录是/sys/bus/platform/devices  */
        priv->devices_kset = kset_create_and_add("devices", NULL,
                                                 &priv->subsys.kobj);

        /* 生成 drivers _kset 容器，命名为"drivers" ，由于其父 kobj 是 priv->subsys.kobj，所以生
        成的目录是/sys/bus/platform/ drivers  */
        priv->drivers_kset = kset_create_and_add("drivers", NULL,
                                                 &priv->subsys.kobj);

}
```

下面该分析 driver_attach()例程了。

3.4. driver_attach()例程从 bus 中查找匹配的 device，该例程定义在 drivers/base/ dd.c 中，代码如下：

```
int driver_attach(struct device_driver *drv)
{
        /* 遍历 bus 的 klist_devices 列表，对每个 device 使用回调函数__driver_attach()来鉴别是否
        和 driver 匹配 */
        return bus_for_each_dev(drv->bus, NULL, drv, __driver_attach);
}
```

代码中，

3.4. 1. bus_for_each_dev()定义在在 drivers/base/ bus.c 中，代码如下：

```
int bus_for_each_dev(struct bus_type *bus, struct device *start,
```

```
                    void *data, int (*fn)(struct device *, void *))
{
        struct klist_iter i;
        struct device *dev;
        int error = 0;

        if (!bus)
                return -EINVAL;

        // 设置 i
        klist_iter_init_node(&bus->p->klist_devices, &i,
                             (start ? &start->knode_bus : NULL));
        // 使用 i 遍历
        while ((dev = next_device(&i)) && !error)
                error = fn(dev, data);  // 使用回调例程处理
        klist_iter_exit(&i);
        return error;
}
```

接着分析回调例程__driver_attach()。

3.4. 2. 回调例程**__driver_attach()**定义在 drivers/base/ dd.c 中，代码如下：

```
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.
         */

        /* 调用 bus 的 match ()，在这里是 platform_bus_type 的 mach()，即 platform_match()例
        程 ，其在《Linux 设备模型浅析之设备篇》中分析过 */
        if (drv->bus->match && !drv->bus->match(dev, drv))
                return 0;

        if (dev->parent)        /* Needed for USB */
                down(&dev->parent->sem);
        down(&dev->sem);
        if (!dev->driver)  // 显然本例中 s3c_device_rtc 在注册时没有找到 driver，所以这里会执行
                driver_probe_device(drv, dev);        // 这里开始 probe
        up(&dev->sem);
        if (dev->parent)
                up(&dev->parent->sem);

        return 0;
```

}

3.4.2.1. driver_probe_device()在之前的《Linux 设备模型浅析之设备篇》文章已经分析，所以这里直接拷贝过来。

driver_probe_device()也是定义在 drivers/base/dd.c 中，代码如下：

```
int driver_probe_device(struct device_driver *drv, struct device *dev)
{
	int ret = 0;

	if (!device_is_registered(dev))		// 判断 dev 是否已经注册
		return -ENODEV;

	/* 调用 bus 的 match ()，在这里是 platform_bus_type 的 mach()，即 platform_match()例
	程 ，其在《Linux 设备模型浅析之设备篇》中分析过 */
	if (drv->bus->match && !drv->bus->match(dev, drv))
		goto done;

	pr_debug("bus: '%s': %s: matched device %s with driver %s\n",
		 drv->bus->name, __func__, dev_name(dev), drv->name);
	// 这里真正开始调用用户在 device_driver 中注册的 probe()例程
	ret = really_probe(dev, drv);

done:
	return ret;
}
```

下面分析 really_probe()例程，顾名思义，其真正开始 probe 了。之前的《Linux 设备模型浅析之设备篇》文章已经分析，所以这里直接拷贝过来。

3.4.2.2. really_probe()定义在 drivers/base/dd.c 中，代码如下：

```
static int really_probe(struct device *dev, struct device_driver *drv)
{
	int ret = 0;

	atomic_inc(&probe_count);
	pr_debug("bus: '%s': %s: probing driver %s with device %s\n",
		 drv->bus->name, __func__, drv->name, dev_name(dev));
	WARN_ON(!list_empty(&dev->devres_head));

	dev->driver = drv;	// 将匹配的 driver 指针关联到 dev，以便后续使用

	/*如果设备和驱动已经关联了，则在 dev 目录下，即 s3c2410-rtc 目录下生成名
	为"driver"的链接文件，指向其关联的驱动 dev->driver 的 sys 目录，并且在 dev->
	driver 的 sys 目录下生成链接文件，名字和 dev 的名字一样，即"3c2410-wdt "，指
	向/sys/devices/platform/s3c2410-rtc 目录
	*/
	if (driver_sysfs_add(dev)) {
		printk(KERN_ERR "%s: driver_sysfs_add(%s) failed\n",
			__func__, dev_name(dev));
		goto probe_failed;
```

```
        }

        // 如果设置了 dev->bus->probe，则调用，在 platform_bus_type 没有设置
        if (dev->bus->probe) {
                ret = dev->bus->probe(dev);
                if (ret)
                        goto probe_failed;
        /* 所以，调用驱动注册在 device_driver 里的 probe，这个很常用，用于获得硬件资源，初
        始化硬件等，在本例中就是调用注册到 driver 的 s3c_rtc_probe ()例程。
        */
        } else if (drv->probe) {
                ret = drv->probe(dev);
                if (ret)
                        goto probe_failed;
        }

        // 将 device 添加到 driver 列表中，并通知 bus 上的设备，表明 BOUND_DRIVER。
        driver_bound(dev);
        ret = 1;
        pr_debug("bus: '%s': %s: bound device %s to driver %s\n",
                 drv->bus->name, __func__, dev_name(dev), drv->name);
        goto done;

probe_failed:
        devres_release_all(dev);
        driver_sysfs_remove(dev);
        dev->driver = NULL;

        if (ret != -ENODEV && ret != -ENXIO) {
                /* driver matched but the probe failed */
                printk(KERN_WARNING
                    "%s: probe of %s failed with error %d\n",
                    drv->name, dev_name(dev), ret);
        }
        /*
         * Ignore errors returned by ->probe so that the next driver can try
         * its luck.
         */
        ret = 0;
done:
        atomic_dec(&probe_count);
        wake_up(&probe_waitqueue);
        return ret;
}
```

代码中，

3.4.2.2.1.在 s3c_rtc_probe ()中获取了硬件资源和注册了 rtc device，所以会产生相关的文件夹和文件，并调用 rtc_device_register("s3c", &pdev->dev, &s3c_rtcops, THIS_MODULE)注册 rtc 类设备，在《Linux 设备模型浅析之设备篇》中做了具体分析，请参照。

3.5. 接着执行 **module_add_driver()**例程，其定义在 drivers/base/module.c 中，代码如下：

```
void module_add_driver(struct module *mod, struct device_driver *drv)
{
        char *driver_name;
        int no_warn;
        struct module_kobject *mk = NULL;

        if (!drv)
                return;

        if (mod)        // 本例中设置为 THIS_MODULE，所以执行
                mk = &mod->mkobj;
        else if (drv->mod_name) {    // 如果设置了模块的名字，则到 module_kset 容器列表中查找
                struct kobject *mkobj;

                /* Lookup built-in module entry in /sys/modules */
                mkobj = kset_find_obj(module_kset, drv->mod_name);     // 根据模块名查找
                if (mkobj) {
                        mk = container_of(mkobj, struct module_kobject, kobj);
                        /* remember our module structure */
                        drv->p->mkobj = mk;
                        /* kset_find_obj took a reference */
                        kobject_put(mkobj);
                }
        }

        if (!mk)
                return;

        /* Don't check return codes; these calls are idempotent */
        /* 本例中，假设 rtc-s3c.c 驱动编译成模块，手工在 shell 中使用 insmod 命令加载。所以，
        会在/sys/bus/platform/drivers/s3c2410-rtc/目录下生成名为"module"的链接文件，指
        向/sys/modules/rtc-s3c 目录，至于/sys/modules/rtc-s3c 目录是如何产生的，稍后将做分析
        */
        no_warn = sysfs_create_link(&drv->p->kobj, &mk->kobj, "module");

        // 本例中，生成的 driver_name 是“platform:s3c2410-rtc”，你看了该例程的实现就会明白
        driver_name = make_driver_name(drv);
        if (driver_name) {
                // 生成/sys/modules/rtc-s3c/drivers 目录
                module_create_drivers_dir(mk);
                /*  本例中，在/sys/modules/rtc-s3c/drivers 目录下生成名为“platform:s3c2410-rtc”的
                链接文件，指向/sys/bus/platform/drivers/s3c2410-rtc/目录 */
                no_warn = sysfs_create_link(mk->drivers_dir, &drv->p->kobj,
                                            driver_name);
                kfree(driver_name);
        }
}
```

代码中，

3.5.1. 看了上面的分析，一定会产生一个疑问，/sys/modules/rtc-s3c 目录是如何产生的呢？下面就说这个问题。首先说说/sys/modules 目录是如何产生的。在 kernel/params.c 中有个初始化例程 param_sysfs_init()在系统初始化的时候会调用，在该例程中调用了 module_kset = kset_create_and_add("module", &module_uevent_ops, NULL)，显然生成了一个 kset 容器，产生了/sys/module 目录，该 kset 容器被赋给了全局指针 module_kset，所以我们所有的驱动模块的 mkobj 都挂在在它的名下。

3.5.2. 再说说/sys/modules/rtc-s3c 目录是如何产生的。rtc-s3c.c 驱动程序被编译成模块 rtc-s3c.ko。insmod 加载时会产生系统调用，调用到的内核入口程序是定义在 kernel/module.c 中的 init_module()例程（其实是 sys_init_module()）。该例程会调用在同一个文件中的 load_module()例程，该例程会生成一个 struct module 并根据记载的模块进行初始化并将其加入到一个链表中，以便以后进行引用。在 init_module()例程中会调用到 mod_sysfs_init()例程，其代码如下：

```
int mod_sysfs_init(struct module *mod)
{
        int err;
        struct kobject *kobj;

        if (!module_sysfs_initialized) {
                printk(KERN_ERR "%s: module sysfs not initialized\n",
                     mod->name);
                err = -EINVAL;
                goto out;
        }

        // 先在 module_kset 容器的列表中查找，看是否该 mod 已经加载
        kobj = kset_find_obj(module_kset, mod->name);
        if (kobj) {
                // 加载过了则打印错误信息并返回
                printk(KERN_ERR "%s: module is already loaded\n", mod->name);
                kobject_put(kobj);
                err = -EINVAL;
                goto out;
        }

        mod->mkobj.mod = mod;
        memset(&mod->mkobj.kobj, 0, sizeof(mod->mkobj.kobj));

        // 将 kobj.kset 指向 module_kset，也就是包含在它的名下
        mod->mkobj.kobj.kset = module_kset;

        /* 因为传入的 parent 参数为 NULL，所以会使用 module_kset.kobj 作为 mkobj.kobj 的
        partent kobj。mod->name 就是模块名，根据生成的模块来获得，本例中模块为 rtc-
        s3c.ko，显然 mod->name = “rtc-s3c”，所以会产生/sys/module/rtc-s3c 目录。*/
        err = kobject_init_and_add(&mod->mkobj.kobj, &module_ktype, NULL,
                                   "%s", mod->name);
        if (err)
                kobject_put(&mod->mkobj.kobj);
```

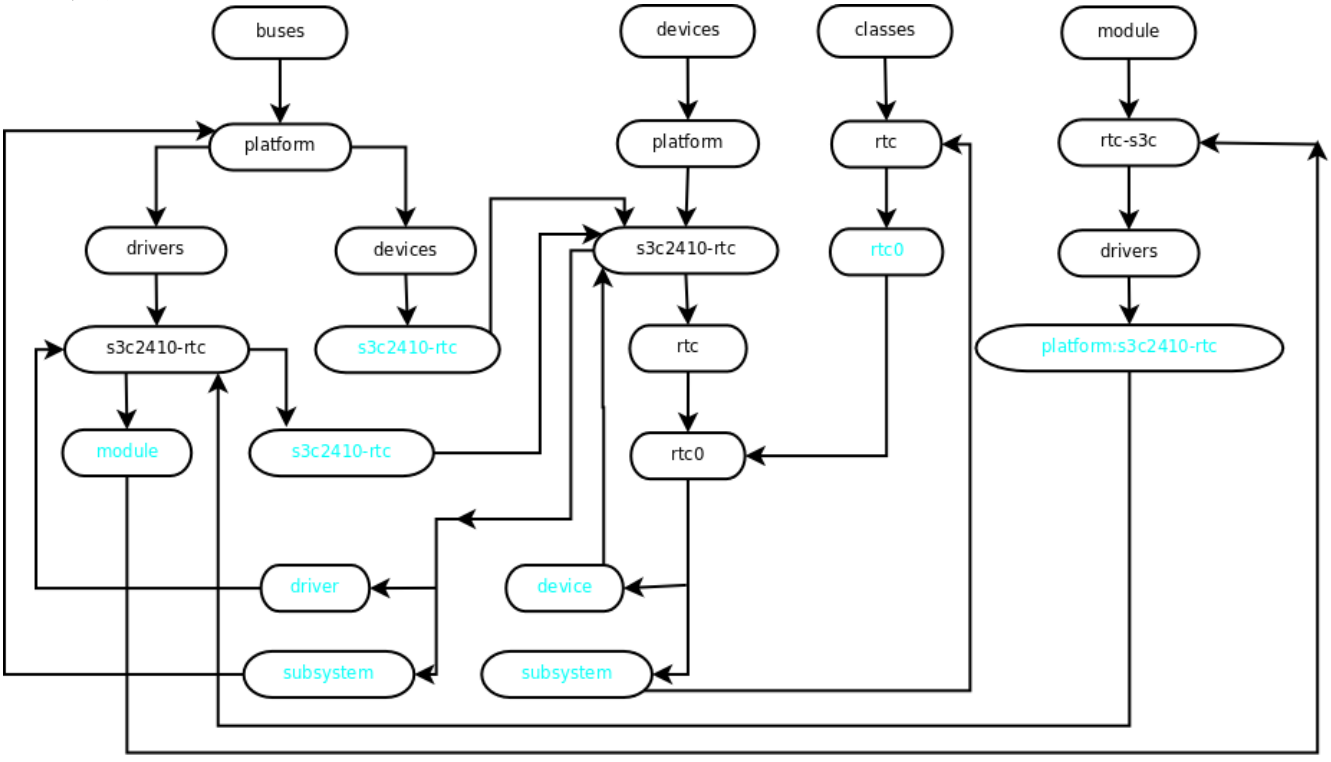
```
	/* delay uevent until full sysfs population */
out:
	return err;
}
```

3.5.3. 顺便说说本例中 platform_driver_register(&s3c2410_rtc_driver)是被如何调用的，该例程被 模块初始化例程 s3c_rtc_init(void)调用。module_init(s3c_rtc_init)使得其会被赋给 __this_module.init，而该 init 函数指针会在 init_module()例程中被调用。这样 platform_driver_register(&s3c2410_rtc_driver)就被调用了，呵呵。

至此，platform_driver_register()例程完成调用。由于 platform_device s3c_device_rtc 是在系统初始化的时候注册的，所以 driver 能够找到匹配的 device，并产生一系列的文件夹和文件，可看附图。

作个小结，从上面的分析可以看出，sys 文件系统中 devices、bus、class 和 dev 目录里的内容之间的关联是通过调用 device_register()、driver_register()和 init_module()例程来完成的。很显然，linux 设备模型就这样建立起来了。

附图：



注：

1. 其中黑色字体的椭圆形表示是个文件夹；
2. 其中青色字体的椭圆形表示是个链接文件；
3. 用箭头表示文件夹之间的隶属关系和链接文件与文件夹之间的链接关系。