Linux 设备模型浅析之设备篇

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_device 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 为例。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。

一、**S3C2410 RTC** 的 platform device 定义在 arch/arm/plat-s3c24xx/devs.c 中，如下：

```
static struct resource s3c_rtc_resource[] = {
        [0] = {
                .start = S3C24XX_PA_RTC,
                .end   = S3C24XX_PA_RTC + 0xff,
                .flags = IORESOURCE_MEM,
        },
        [1] = {
                .start = IRQ_RTC,
                .end   = IRQ_RTC,
                .flags = IORESOURCE_IRQ,
        },
        [2] = {
                .start = IRQ_TICK,
                .end   = IRQ_TICK,
                .flags = IORESOURCE_IRQ
        }
};

struct platform_device s3c_device_rtc = {
        .name             = "s3c2410-rtc",
        .id               = -1,
        .num_resources          = ARRAY_SIZE(s3c_rtc_resource),
        .resource         = s3c_rtc_resource,
};
```

把它们添加在 arch/arm/mach-s3c2440/ mach- smdk2440.c 中，如下：

```
static struct platform_device *smdk2440_devices[] __initdata = {
        &s3c_device_usb,
        &s3c_device_lcd,
        &s3c_device_wdt,
```

```
        &s3c_device_i2c0,
        &s3c_device_iis,
        & s3c_device_rtc
};
```

系统初始化的时候会调用 drivers/base/platform.c 里的 platform_add_devices(smdk2440_devices, ARRAY_SIZE(smdk2440_devices))将其注册到 platform_bus_type，最终被添加到 device hierarchy 。platform_add_devices()调用了 platform_device_register()，而后者又先后调用了 device_initialize()和 platform_device_add()。有必要对 platform_bus_type 的定义作一番注释，其定义如下：

```
struct bus_type platform_bus_type = {
        .name           = "platform",           // bus 的名字，将会生成/sys/bus/platform 目录

        /* 该属性文件将产生在所有 platform_bus_type 类型的设备目录下，文件名为"modalias" */
        .dev_attrs      = platform_dev_attrs,
        .match          = platform_match,     // 用于 drive 与 device 匹配的例程
        .uevent         = platform_uevent,    // 用于输出环境变量，与属性文件“uevent”相关
        .pm             = PLATFORM_PM_OPS_PTR, // 电源管理方面
};
```

代码中，

1. 通过 **bus_register**(&platform_bus_type)将 platform_bus_type 注册到总线模块。本例中，当 cat /sys/device/platform/s3c2410-rtc/modalias 时将会打印出"platform:s3c2410-wdt"，从 platform_dev_attrs 的具体实现中你就能看出来。当 cat /sys/device/platform/s3c2410-rtc/uevent 时将会打印出”DRIVER=s3c2410-wdt MODALIAS=platform:s3c2410-wdt”，从 platform_uevent 的具体实现中你就能看出来。

二、下面解析 **device_initialize()**和 **platform_device_add()**两个例程，它们分别定义在 drivers/base/core.c 和 drivers/base/platform.c 中。

device_initialize()的代码如下：

```
void device_initialize(struct device *dev)
{
        dev->kobj.kset = devices_kset;       // 设置其指向的 kset 容器
        kobject_init(&dev->kobj, &device_ktype); // 初始化 kobj，将 device_ktype 传递给它
        klist_init(&dev->klist_children, klist_children_get,
                   klist_children_put);         // 初试化 klist
        INIT_LIST_HEAD(&dev->dma_pools);
        init_MUTEX(&dev->sem);
        spin_lock_init(&dev->devres_lock);
        INIT_LIST_HEAD(&dev->devres_head);
        device_init_wakeup(dev, 0);
        device_pm_init(dev);            // 初试化电源管理
        set_dev_node(dev, -1);
}
```

代码中，

1. devices_kset 是所有 dev 的 kset，也就是所有 dev 都被链接在该 kset 下，其在初试化例程 devices_init()中通过调用 kset_create_and_add("devices", &device_uevent_ops, NULL)来创建。由于参数 parent=NULL ，所以生成/sys/devices 目录。这里说明下 kobj，kset 结构体中包含有一个

kobj，一个 kobj 生成一个目录，在这里就是”**devices** "目录，通过调用 kobject_add_internal()例程生成。所以从 dev->kobj.kset = devices_kset 可以看出，该 dev.kobj 添加到了 devices_kset 容器中，所的 kobj 都归属于一个特定的 kset。关于 kset，kobj，ktype，kref 的关系可以参考书 LDD3 的第十四章，在第 370 页有一张说明 kobj 和 kset 关系的图（英文版）。

2. kobject_init(&dev->kobj, &device_ktype)用于初始化 dev->kobj 中变量的参数，如 ktype、kref、entry 和 state*等。初试化例程 devices_init()还会调用 kobject_create_and_add()例程生成/sys/dev、/sys/dev/block 和/sys/dev/char 目录。

3. 其他初始化。

platform_device_add 代码如下：

```
int platform_device_add(struct platform_device *pdev)
{
        int i, ret = 0;

        if (!pdev)
                return -EINVAL;

        if (!pdev->dev.parent)
                pdev->dev.parent = &platform_bus;  // 设置为 platform_bus  device

        pdev->dev.bus = &platform_bus_type;         // 设置为 platform_bus_type

        // 拷贝 pdev name 到 device bus_id
        if (pdev->id != -1)
                dev_set_name(&pdev->dev, "%s.%d", pdev->name,  pdev->id);
        else
                dev_set_name(&pdev->dev, pdev->name);

        // 将所有 resources 添加到列表
        for (i = 0; i < pdev->num_resources; i++) {
                struct resource *p, *r = &pdev->resource[i];

                if (r->name == NULL)
                        r->name = dev_name(&pdev->dev);

                p = r->parent;
                if (!p) {
                        if (resource_type(r) == IORESOURCE_MEM)
                                p = &iomem_resource;
                        else if (resource_type(r) == IORESOURCE_IO)
                                p = &ioport_resource;
                }

                if (p && insert_resource(p, r)) {
                        printk(KERN_ERR
                               "%s: failed to claim resource %d\n",
                               dev_name(&pdev->dev), i);
                        ret = -EBUSY;
```

```
                goto failed;
            }
        }

        pr_debug("Registering platform device '%s'. Parent at %s\n",
             dev_name(&pdev->dev), dev_name(pdev->dev.parent));

        // 添加 pdev->dev 到 device hierarchy，后面详细分析
        ret = device_add(&pdev->dev);
        if (ret == 0)
            return ret;

// 失败后的清除工作
 failed:
        while (--i >= 0) {
            struct resource *r = &pdev->resource[i];
            unsigned long type = resource_type(r);

            if (type == IORESOURCE_MEM || type == IORESOURCE_IO)
                release_resource(r);
        }

        return ret;
}
```

代码中，

1. 如果 dev 没有设置 parent，显然本例中没有设置，则执行 pdev->dev.parent = &platform_bus，这个 platform_bus 的定义如下：

```
struct device platform_bus = {
        .init_name      = "platform",  // 这个名字将“变成”/sys/devices/platform 目录
};
```

其在

```
int __init platform_bus_init(void)
{
        int error;

        error = device_register(&platform_bus);      // 注册 platform_bus device
        if (error)
            return error;
        error =  bus_register(&platform_bus_type);  //注册 platform_bus_type  bus
        if (error)
            device_unregister(&platform_bus);
        return error;
}
```

中被调用，显然 platform_bus_init ()例程在内核初始化的时候会被调用。 platform_bus 通过 device_register（先后调用 device_initialize()和 device_add()例程）添加到 device hierarchy，由于其没有 class 和 parent，所以 platform_bus .kobj->parent = **devices_kset.kobj**，故生成/sys/devices/platform 目录。由于 dev.parent = &platform_bus，所以注册的 dev.kobj 生成的目录在/sys/devices/

platform/下 。在本例中将生成/sys/devices/platform/s3c2410-rtc 目录，后面会讲到。

2. pdev->dev.bus = &platform_bus_type 表明该 dev 是 platform_bus_type 类型的，并且该 dev 也会被添加到 platform_bus_type 的 devices_kset 容器中和 klist_devices 列表中。

3. 如果 pdev->id 不等于-1，那么说明其指定了序号，会被添加到 pdev-name 名字后面再拷贝给 dev->bus_id，等于-1 则直接拷贝。

4. 把所有的 resources 添加到列表中，iomem_resource 和 ioport_resource 为父节点。

5. 最后调用 device_add()，把该 dev 添加到 device hierarchy，下面就研究它。

三、**device_add()**定义在 drivers/base/core.c 中，其调用很多例程完成 dev 的添加工作，这个例程调用完后则完成添加工作，代码如下：

```
int device_add(struct device *dev)
{
	struct device *parent = NULL;
	struct class_interface *class_intf;
	int error = -EINVAL;

	dev = get_device(dev);  // 增加对 dev 的引用，最终通过增加 kobj->kref 的引用来实现
	if (!dev)		// 判断是否成功
		goto done;

	/* Temporarily support init_name if it is set.
	 * It will override bus_id for now */
	if (dev->init_name)   // 如英文解释所言
		dev_set_name(dev, "%s", dev->init_name);

	if (!strlen(dev->bus_id))		// 判断是否设置了名字
		goto done;

	pr_debug("device: '%s': %s\n", dev_name(dev), __func__);

	parent = get_device(dev->parent);     // 增加对 dev->parent 的引用

	/* 设置 kobj.parent，由于 s3c_device_rtc 的 dev 不存在 class，所以 dev->kobj.parent =
	dev->parent.kobj，也就是等于 platform_bus.kobj ，后面会详细分析
	*/
	setup_parent(dev, parent);

	/* use parent numa_node */
	if (parent)
		set_dev_node(dev, dev_to_node(parent));

	// 添加 kobj 到 kobj->kset(devices_kset )中，生成/sys/devices/platform/s3c2410-rtc 目录
	/* first, register with generic layer. */
	error = kobject_add(&dev->kobj, dev->kobj.parent, "%s", dev_name(dev));
	if (error)
		goto Error;

	/* notify platform of device entry */
```

```
if (platform_notify)    // NULL
        platform_notify(dev);

// 在 s3c2410-rtc 目录下生成名为“uevent” 的属性文件，所有注册的 dev 都会生成这个文
件
error = device_create_file(dev, &uevent_attr);
if (error)
        goto attrError;

// 如果有设备号，则生成相应的文件， s3c_device_rtc 没有。
if (MAJOR(dev->devt)) {
        /* 如果有，则在 s3c2410-rtc 目录下生成名为“dev"的属性文件，这样 udev 就能读
        取该属性文件获得设备号，从而在/dev 目录下创建设备节点 */
        error = device_create_file(dev, &devt_attr);
        if (error)
                goto ueventattrError;

        /*
        如果有，根据设备的类型，在 sys/dev/char 或 block 或其所属 class 指定的 dev_kobj
        目录下生成链接文件，其名字的样式为 major:minor，如 80:8，指向 s3c2410-rtc 目
        录。
        */
        error = device_create_sys_dev_entry(dev);
        if (error)
                goto devtattrError;
}

/* 由于 s3c_device_rtc 没有指定 class，所以不会生成相应的链接，后面会详细分析。*/
error = device_add_class_symlinks(dev);
if (error)
        goto SymlinkError;
error = device_add_attrs(dev); // 生成类的属性文件和其他的属性文件
if (error)
        goto AttrsError;

/*
 将 device 加入到 bus 的 kset 中。
在 s3c2410-rtc 目录下生成一个链接文件，其名为”subsystem"，指向其所属的
platform_bus_type 的目录/sys/bus/platform，以及在/sys/bus/platform/devices 目录生成一个
名为”s3c2410-rtc"的链接文件，指向/sys/devices/platform/s3c2410-rtc。后面会分析
*/
error = bus_add_device(dev);
if (error)
        goto BusError;
error = dpm_sysfs_add(dev); // 在 s3c2410-rtc 目录下生成 power 节点
if (error)
        goto DPMError;
device_pm_add(dev); // 添加到活跃设备的链表里
```

```
        /* Notify clients of device addition.  This call must come
         * after dpm_sysf_add() and before kobject_uevent().
         */
        // 是个回调机制，通知 platform_bus_type 上所有的设备 ADD_DEVICE。
        if (dev->bus)
                blocking_notifier_call_chain(&dev->bus->p->bus_notifier,
                                             BUS_NOTIFY_ADD_DEVICE, dev);
        // 通过 uevents 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(&dev->kobj, KOBJ_ADD);

        // 将设备添加到 platform_bus_type 列表中，并寻找与其匹配的 driver，后面会详细分析
        bus_attach_device(dev);

        // 将该设备链接到其父设备，本例中就是 platform_bus
        if (parent)
                klist_add_tail(&dev->knode_parent, &parent->klist_children);

        // 没有设置 class ，所以不调用。
        if (dev->class) {
                mutex_lock(&dev->class->p->class_mutex);
                /* tie the class to the device */
                klist_add_tail(&dev->knode_class,  // 如英文解释所言
                               &dev->class->p->class_devices);

                /* notify any interfaces that the device is here */
                list_for_each_entry(class_intf,
                                    &dev->class->p->class_interfaces, node)
                        if (class_intf->add_dev)
                                class_intf->add_dev(dev, class_intf);
                mutex_unlock(&dev->class->p->class_mutex);
        }
done:
        put_device(dev);
        return error;
 DPMError:
        bus_remove_device(dev);
 BusError:
        device_remove_attrs(dev);
 AttrsError:
        device_remove_class_symlinks(dev);
 SymlinkError:
        if (MAJOR(dev->devt))
                device_remove_sys_dev_entry(dev);
 devtattrError:
        if (MAJOR(dev->devt))
                device_remove_file(dev, &devt_attr);
 ueventattrError:
        device_remove_file(dev, &uevent_attr);
```

```
attrError:
        kobject_uevent(&dev->kobj, KOBJ_REMOVE);
        kobject_del(&dev->kobj);
Error:
        cleanup_device_parent(dev);
        if (parent)
                put_device(parent);
        goto done;
}
```

代码中，

1. kobject_add ()例程会调用 kobject_add_internal()例程来完成 kobj 的添加工作，下面简要说明下 kobject_add_internal()例程，其中有一小段代码：

```
        parent = kobject_get(kobj->parent);
        if (kobj->kset) {
                if (!parent)
                        parent = kobject_get(&kobj->kset->kobj);
                kobj_kset_join(kobj);
                kobj->parent = parent;
        }
```

很明显在我们的例子里 if (kobj->kset)里的代码会被执行，因为 kobj->kset = devices_kset。但由于 dev->kobj-> parent = dev->parent.kobj（即 platform_bus.kobj），所以 if (!parent)面的代码不执行，故 dev->kobj 的目录（"s3c2410-rtc"）产生在 dev->parent.kobj 目录（"platform"）里，即/sys/devices/platform/s3c2410-rtc。

2. 我们可以使用 cat /sys/devices/platform/s3c2410-rtc/uevent 来查看其内容，具体的过程是这样的，cat()-->.....-->(dev.kobj->ktype->sysfs_ops->show())-->(uevent_attr->show())，其中 ktype 就是在 device_initialize()中通过 kobject_init(&dev->kobj, &device_ktype)例程调用初始化的 device_ktype，在 sysfs_ops->show() 中是通过 to_dev_attr(attr)（也就是 container_of()）得到注册的 uevent_attr。对所有的属性文件的读取和写入都是这个过程，都是使用其所在目录的 kobj->ktype 来完成的。

四、 **setup_parent()**定义在 drivers/base/core.c 中，用于找到并设置 dev->kobj.parent ，代码如下：

```
static void setup_parent(struct device *dev, struct device *parent)
{
        struct kobject *kobj;
        kobj = get_device_parent(dev, parent);        // 后面分析
        if (kobj)
                dev->kobj.parent = kobj;      // 显然，如果找到了 parent，就赋给 dev->kobj.parent
}
```

get_device_parent()代码如下：

```
static struct kobject *get_device_parent(struct device *dev,
                                         struct device *parent)
{
        int retval;

        if (dev->class) {       // 先判断是否设置了 class，本例中没有设置
```

```
struct kobject *kobj = NULL;
struct kobject *parent_kobj;
struct kobject *k;

/*
 * If we have no parent, we live in "virtual".
 * Class-devices with a non class-device as parent, live
 * in a "glue" directory to prevent namespace collisions.
 */
if (parent == NULL)
        /*如果 parent 为 NULL，那么会生成一个 kobj 作为 parent_kobj 和相应
        的/sys/devices/virtual 目录
        */
        parent_kobj = virtual_device_parent(dev);
else if (parent->class)
        /*如果  parent->class 存在，则返回 parent->kobj
        */
        return &parent->kobj;
else
        parent_kobj = &parent->kobj;

/* find our class-directory at the parent and reference it */
spin_lock(&dev->class->p->class_dirs.list_lock);
list_for_each_entry(k, &dev->class->p->class_dirs.list, entry)
        if (k->parent == parent_kobj) { // 从 class_dirs.list 查找
                kobj = kobject_get(k); // 当之前已经添加时则能找到
                break;
        }
spin_unlock(&dev->class->p->class_dirs.list_lock);
if (kobj)        // 如果找到就直接返回，
        return kobj;

/* or create a new class-directory at the parent device */
k = kobject_create();  // 动态生成一个 kobj
if (!k)
        return NULL;
k->kset = &dev->class->p->class_dirs;  // 指向 class_dirs  kset
/* 添加 k->kobj 到 k->kset，并生成/sys/devices/platform/s3c2410-wdt/xx 目录，其中
xxx 为 dev->class->name*/
retval = kobject_add(k, parent_kobj, "%s", dev->class->name);
if (retval < 0) {
        kobject_put(k);
        return NULL;
}
/* 返回新生成的 kobj，故后面的 dev->parent.kobj  =  kobj，会在 k 的目录下生成
dev.kobj 的目录*/
/* do not emit an uevent for this simple "glue" directory */
return k;
```

```
        }

        if (parent)
                return &parent->kobj;
        return NULL;
}
```

代码中，

1. 举个例子说明这个例程的作用，在 rtc-s3c.c 中会调用 rtc_device_register("s3c", &pdev->dev, &s3c_rtcops, THIS_MODULE)注册一个 rtc_device，部分代码如下：

```
struct rtc_device *rtc_device_register(const char *name, struct device *dev,
                                       const struct rtc_class_ops *ops,
                                       struct module *owner)
{
        rtc = kzalloc(sizeof(struct rtc_device), GFP_KERNEL);

        rtc->dev.parent = dev;
        rtc->dev.class = rtc_class;

        dev_set_name(&rtc->dev, "rtc%d", id);

        err = device_register(&rtc->dev);
}
```

pdev->dev 是 ＆s3c_device_rtc，所以 rtc->dev.parent = s3c_device_rtc，class 是 rtc_class，dev->bus_id 是“rtc+id"，这里假设是”rtc0"。这样在 get_device_parent()中新创建的 kobj(k)对应的目录是/sys/devices/platform/rtc，而 rtc->dev.kobj 生成的目录是/sys/devices/platform/rtc/rtc0。

五、**device_add_class_symlinks()**定义在 drivers/base/core.c 中，去掉了 DEPRECATED 部分，代码如下：

```
static int device_add_class_symlinks(struct device *dev)
{
        int error;

        if (!dev->class)  // 如果没有设置 class，那么就直接返回了，但不报错
                return 0;

        /* 在 dev->kobj 目录下生成一个名为”subsystem"链接文件，指向其所属的 class 在 sys 的
        目录/sys/class/xxx
        */
        error = sysfs_create_link(&dev->kobj,
                                  &dev->class->p->class_subsys.kobj,
                                  "subsystem");
        if (error)
                goto out;

        /* 在/sys/class/xxx 目录生成一个名为“dev_name(dev)"的链接文件，指
        向/sys/devices/platform/dev_name(dev)。dev_name(dev)用于获取 dev 的 name 字符串，xxx
        代表 class 的目录。
```

```
        */
        /* link in the class directory pointing to the device */
        error = sysfs_create_link(&dev->class->p->class_subsys.kobj,
                                  &dev->kobj, dev_name(dev));
        if (error)
                goto out_subsys;

        /*在/sys/devices/platform/dev_name(dev)目录中生成一个名为“device”的链接文件，指向
        dev->parent.kobj 目录，
        */
        if (dev->parent && device_is_not_partition(dev)) {
                error = sysfs_create_link(&dev->kobj, &dev->parent->kobj,
                                          "device");
                if (error)
                        goto out_busid;
        }
        return 0;

out_busid:
        sysfs_remove_link(&dev->class->p->class_subsys.kobj, dev_name(dev));

out_subsys:
        sysfs_remove_link(&dev->kobj, "subsystem");
out:
        return error;
}
```

代码中

1. 以上面介绍过的 rtc-s3c.c 为例，则生成链接/sys/devices/platform/s3c2410-rtc/rtc/rtc0/subsystem-->/sys/class/rtc，并且生成链接/sys/class/rtc/rtc0-->/sys/devices/platform/s3c2410-rtc/rtc/rtc0，以及生成链接/sys/devices/platform/s3c2410-rtc/rtc/rtc0/device-->/sys/devices/platform/s3c2410-rtc。

六、接着执行 **bus_attach_device()**例程，定义在 drivers/base/bus.c 中，代码如下：

```
void bus_attach_device(struct device *dev)
{
        struct bus_type *bus = dev->bus; // 在这是 platform_bus_type
        int ret = 0;

        if (bus) {
                if (bus->p->drivers_autoprobe)   // 在 bus_register()例程中已经设置为 1 了
                        ret = device_attach(dev);  // 从 drver 列表中找到与 dev 相匹配的 driver，后面
                                                   // 分析
                WARN_ON(ret < 0);
                // 添加到 klist_devices 列表，后续方便使用，比如匹配驱动程序什么的
                if (ret >= 0)
                        klist_add_tail(&dev->knode_bus, &bus->p->klist_devices);
        }
```

```
}
```

七、**device_attach()**定义在 drivers/base/dd.c 中，代码如下：

```
int device_attach(struct device *dev)
{
        int ret = 0;

        down(&dev->sem);   // 进入临界区
        if (dev->driver) {
                /*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*/
                ret = device_bind_driver(dev);
                if (ret == 0)
                        ret = 1;
                else {
                        dev->driver = NULL;
                        ret = 0;
                }
        } else {
                /*没有关联则需要从 klist_drivers 找到关联的 driver，一旦找到，则调用
                __device_attach()回调函数*/
                ret = bus_for_each_drv(dev->bus, NULL, dev, __device_attach);
        }
        up(&dev->sem);       //出临界区
        return ret;
}
```

代码中，

1. 以上面介绍过的 rtc-s3c.c 为例，则生成链接/sys/devices/platform/s3c2410-rtc/driver-->/sys/bus/platform/drivers/s3c2410-rtc，以及生成链接/sys/bus/platform/drivers/s3c2410-rtc/s3c2410-rtc-->/sys/devices/platform/s3c2410-rtc。rtc-s3c.c driver 生成的目录是/sys/bus/platform/drivers/s3c2410-rtc。显然从程序中的 platform_driver 定义能看出：

```
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",        // 目录就是这个名字
                .owner = THIS_MODULE,
        },
}。
```

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

```
static int __device_attach(struct device_driver *drv, void *data)
{
```

```
        struct device *dev = data;
        return driver_probe_device(drv, dev);          // 这里开始 probe
}
```

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()
        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 porbe()例程
        ret = really_probe(dev, drv);

done:
        return ret;
}
```

八、 platform_match()定义在 drivers/base/platform.c 中，代码如下：

```
static int platform_match(struct device *dev, struct device_driver *drv)
{
        struct platform_device *pdev;

        pdev = container_of(dev, struct platform_device, dev);      // 获取 platform_device
        return (strcmp(pdev->name, drv->name) == 0);      // 仅仅是比较名字的字符串相同否
}
```

九、 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
```

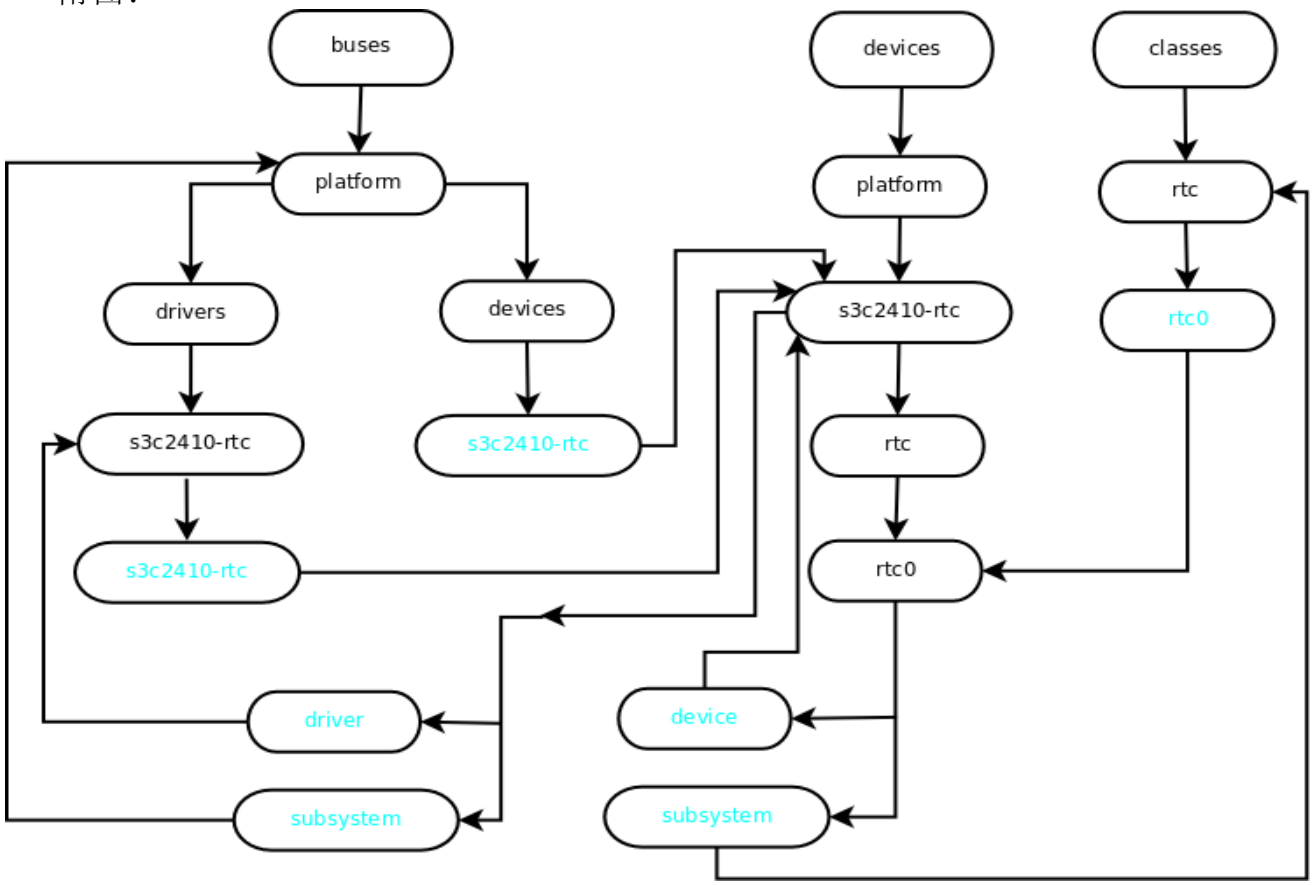
```
        为"driver"的链接文件，指向其关联的驱动 dev->driver 的 sys 目录，并且在 dev->
        driver 的 sys 目录下生成链接文件，名字和 dev 的名字一样，即"3c2410-wdt "，指
        向/sys/devices/platform/s3c2410-rtc 目录
        */
        if (driver_sysfs_add(dev)) {
                printk(KERN_ERR "%s: driver_sysfs_add(%s) failed\n",
                        __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，在后续对 platform_driver 的分析文章中会讲到。
        */
        } 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;
}
```

至此，platform_add_devices()例程完成调用。由于 s3c_device_rtc 是在系统初始化的时候注册的，所以不会找到匹配的 driver。当 rtc-s3c.c 驱动程序中使用 platform_driver_register()例程注册 platform_driver 时也同样会寻找匹配的 device，过程类似于寻找匹配的 driver。所以 rtc-s3c.c 驱动程序驱动中注册的 platform_drive 的 name 必须和 s3c_device_rtc 中的相符。 后续将以 rtc-s3c.c 为例对 platform_driver 进行分析。

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()和 driver_register()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。