Platform驱动

驱动的分离与分隔：主机控制器驱动（半导体厂家提供）和设备驱动（我们需要在linux驱动框架下编写设备驱动）；中间的联系就是核心层

驱动-总线-设备:根据驱动的分离与分层衍生出了总线（bus）、驱动（driver）、设备（deverce）

1. 总线数据类型为bus type.向内核注册总线使用bus_register()

```
struct bus_type {
    const char      *name;
    struct bus_attribute    *bus_attrs;
    struct device_attribute *dev_attrs;
    struct driver_attribute *drv_attrs;

    int (*match)(struct device *dev, struct device_driver *drv);
    int (*uevent)(struct device *dev, struct kobj_uevent_env *env);
    int (*probe)(struct device *dev);
    int (*remove)(struct device *dev);
    void (*shutdown)(struct device *dev);

    int (*suspend)(struct device *dev, pm_message_t state);
    int (*resume)(struct device *dev);

    const struct dev_pm_ops *pm;

    struct subsys_private *p;
};
```

2. 驱动：

- 驱动的数据类型是device_driver;
- 向总线注册驱动的时候，会检查当前总线下的所有设备，有没有与此驱动匹配的设备，如果有的话就执行驱动里面的probe函数

```
struct device_driver {
    const char      *name;
    struct bus_type     *bus;

    struct module       *owner;
    const char      *mod_name;  /* used for built-in modules */

    bool suppress_bind_attrs;   /* disables bind/unbind via sysfs */

    const struct of_device_id   *of_match_table;

    int (*probe) (struct device *dev);//驱动和设备匹配之后就会执行这个函数
            //先开始调用driver_register函数进行驱动注册->bus_add_driver->
            driver_attach():查找总线下与其匹配的设备->bus_for_each_dev()->
            __driver_attach():每个设备都会调用此函数，查看每个设备是否与驱动匹配->
            driver_match_device():检查是否匹配->bus-match
            driver_probe_device()->really_probe()->drv->probe(dev);执行probe
    int (*remove) (struct device *dev);
    void (*shutdown) (struct device *dev);
    int (*suspend) (struct device *dev, pm_message_t state);
    int (*resume) (struct device *dev);
    const struct attribute_group **groups;

    const struct dev_pm_ops *pm;

    struct driver_private *p;
};

```

3. 设备;

设备数据类型为device，通过device_register向内核注册设备

- device 类型

```
struct device {
    struct device       *parent;

    struct device_private   *p;

    struct kobject kobj;
    const char      *init_name; /* initial name of the device */
    const struct device_type *type;

    struct mutex        mutex;  /* mutex to synchronize calls to
                     * its driver.
                     */

    struct bus_type *bus;       /* type of bus device is on */
```

```c
    struct device_driver *driver;   /* which driver has allocated this
                       device */
    void        *platform_data; /* Platform specific data, device
                       core doesn't touch it */
    struct dev_pm_info  power;
    struct dev_power_domain *pwr_domain;

#ifdef CONFIG_NUMA
    int     numa_node;  /* NUMA node this device is close to */
#endif
    u64     *dma_mask;  /* dma mask (if dma'able device) */
    u64     coherent_dma_mask;/* Like dma_mask, but for
                         alloc_coherent mappings as
                         not all hardware supports
                         64 bit addresses for consistent
                         allocations such descriptors. */

    struct device_dma_parameters *dma_parms;

    struct list_head    dma_pools;  /* dma pools (if dma'ble) */

    struct dma_coherent_mem *dma_mem; /* internal for coherent mem
                         override */
    /* arch specific additions */
    struct dev_archdata archdata;

    struct device_node  *of_node; /* associated device tree node */

    dev_t           devt;   /* dev_t, creates the sysfs "dev" */

    spinlock_t      devres_lock;
    struct list_head    devres_head;

    struct klist_node   knode_class;
    struct class        *class;
    const struct attribute_group **groups;  /* optional groups */

    void    (*release)(struct device *dev);
};
```

- device_register

```
 int device_register(struct device *dev)
{
    device_initialize(dev);
    return device_add(dev);
}
device_add()
    ->bus_add_device()
    ->bus_probe_device()
        ->int device_attach()
            ->bus_for_each_drv(dev->bus, NULL, dev, __device_attach)
            ->__device_attach()
                ->driver_match_device()//匹配驱动
                    ->drv->bus->match
                    ->driver_probe_device()//然后和注册驱动过程一样了
```

-
- 设备和驱动匹配之后probe函数就会执行，probe函数就是驱动编写人员去编写的
- 设备和驱动匹配过程是在注册驱动以及注册设备的时候进行的
-

4. platform平台驱动模型

对于soc内部的RTC、timer等等不好归结的总线，我们都把它归结到platform总线上

4.1 platform总线注册

```
platform_bus_init//platfoem.c
    ->bus_register()
        注册的内容：
        struct bus_type platform_bus_type = {
        .name       = "platform",
        .dev_attrs  = platform_dev_attrs,
        .match      = platform_match,
        .uevent     = platform_uevent,
        .pm     = &platform_dev_pm_ops,
        };

        struct bus_type {
        const char      *name;
        struct bus_attribute    *bus_attrs;
        struct device_attribute *dev_attrs;
        struct driver_attribute *drv_attrs;

        int (*match)(struct device *dev, struct device_driver *drv);
        int (*uevent)(struct device *dev, struct kobj_uevent_env *env);
        int (*probe)(struct device *dev);
        int (*remove)(struct device *dev);
        void (*shutdown)(struct device *dev);

        int (*suspend)(struct device *dev, pm_message_t state);
        int (*resume)(struct device *dev);

        const struct dev_pm_ops *pm;

        struct subsys_private *p;
    };
        重点函数 int (*match)(struct device *dev, struct device_driver *drv);
         驱动和设备进行匹配 platform_match())

```

4.2 platform 驱动

结构体为platform_driver,内容为

```
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 (*resume)(struct platform_device *);
    struct device_driver driver;
            //device_driver结构体中const struct of_device_id   *of_match_table;这两个来进
            ->const char      *name;
    const struct platform_device_id *id_table;
};
```

- 向总线上注册一个platform驱动的函数是platform_driver_register()

```
int platform_driver_register(struct platform_driver *drv)//platform.c
{
    drv->driver.bus = &platform_bus_type;
    if (drv->probe)
```

```
        drv->driver.probe = platform_drv_probe;
    if (drv->remove)
        drv->driver.remove = platform_drv_remove;
    if (drv->shutdown)
        drv->driver.shutdown = platform_drv_shutdown;

    return driver_register(&drv->driver);
}
```

- 注册的流程

```
platform_driver_register
    ->drv->driver.probe = platform_drv_probe;//设置platform_driver结构体下的成员driver为pl
    ->driver_register()//驱动注册，这就到了之前介绍的往内核中注册一个驱动
        ->匹配到设备之后会执行结构体device_driver->probe,然后执行platform_drv_probe()
         而platform_drv_probe会执行platform_driver结构体下的probe函数
```

结论：向内核注册platform驱动的时候，如果驱动和设备匹配成功最终会执行platform_driver的probe函数

4.3 platform设备

结构体为 platform_device

```
struct platform_device {
    const char  * name;
    int     id;
    struct device   dev;
    u32     num_resources;
    struct resource * resource;

    const struct platform_device_id *id_entry;

    /* MFD cell pointer */
    struct mfd_cell *mfd_cell;

    /* arch specific additions */
    struct pdev_archdata    archdata;
};
```

平台设备会分两种情况

- 无设备树，需要自己写注册设备的过程,此时需要驱动开发人员编写设备注册文件

```
platform_device_register()
```

- 有设备树，修改设备树的设备节点即可

5. platform匹配过程

根据前面的分析，驱动和设备匹配是通过bus->match函数，platform总线下的match函数就是platform_match

```
if (of_driver_match_device(dev, drv))//关于设备树
return (strcmp(pdev->name, drv->name) == 0);//通过字符串对driver和device进行匹配
platform_match_id(pdrv->id_table, pdev) != NULL;
```

有设备树的时候

```
of_driver_match_device()
        ->of_match_device(drv->of_match_table,dev)//of_match_table非常重要，里面 是支持设别
               ->of_match_node
```

6.

7.