Linux Platform驱动模型(二) _驱动方法

在Linux设备树语法详解和Linux Platform驱动模型(一) _设备信息中我们讨论了设备信息的写法，本文主要讨论平台总线中另外一部分-驱动方法，将试图回答下面几个问题：

1. 如何填充platform_driver对象?
2. 如何将驱动方法对象注册到平台总线中?

正文前的一点罗嗦

写驱动也有一段时间了，可以发现，其实驱动本质上只做了两件事：**向上提供接口，向下控制硬件**，当然，这里的**向上**并不是直接提供接口到应用层，而是提供接口给内核再由内核间接的将我们的接口提供给应用层。而写驱动也是有一些套路可寻的，拿到一个硬件，我们大体可以按照下面的流程写一个驱动：

1. **确定驱动架构**：根据硬件连接方式结合分层/分离思想设计驱动的基本结构
2. **确定驱动对象**：内核中的一个驱动/设备就是一个对象，1.定义，2.初始化，3.注册，4.注销
3. **向上提供接口**：根据业务需要确定提供cdev/proc/sysfs哪种接口
4. **向下控制硬件**：1.查看原理图确定引脚和控制逻辑，2.查看芯片手册确定寄存器配置方式，3.进行内存映射，4.实现控制逻辑

认识驱动方法对象

内核用platform_driver结构来表示一个驱动方法对象

```
//include/linux/device.h
173 struct platform_driver {
```

公告
昵称： Abnor
园龄： 5年7个月
粉丝： 294
关注： 0
+加关注

搜索
找找看

随笔分类 (142)
ARM汇编(6)
C上一层楼(9)
Linux环境编程(27)
Linux命令收集(42)
Linux驱动开发(35)
Makefile(6)
Shell脚本(5)
Ubuntu Tricks(5)
设计模式(1)
系统移植(6)

阅读排行榜
1. Linux设备树语法详解(62688)
2. shell 脚本关键字&符号(28886)
3. Linux Platform驱动模型(二) _驱动方法(24903)

```
174         int (*probe)(struct platform_device *);
175         int (*remove)(struct platform_device *);
176         void (*shutdown)(struct platform_device *);
177         int (*suspend)(struct platform_device *, pm_message_t state);
178         int (*resume)(struct platform_device *);
179         struct device_driver driver;
180         const struct platform_device_id *id_table;
181         bool prevent_deferred_probe;
182 };
```

在这个结构中，我们主要关心以下几个成员

> **struct platform_driver**
> --174-->探测函数，如果驱动匹配到了目标设备，总线会自动回调probe函数，必须实现，下面详细讨论。
> --175-->释放函数，如果匹配到的设备从总线移除了，总线会自动回调remove函数，必须实现
> --179-->platform_driver的父类，我们接下来讨论
> --180-->用于C语言写的设备信息，下面详细讨论。

platform_driver里面有些内容需要在父类driver中实现，

```
 //include/linux/device.h
 228 struct device_driver {
 229         const char              *name;
 230         struct bus_type         *bus;
 231
 232         struct module           *owner;
 233         const char              *mod_name;      /* used for built-in
modules */
 234
 235         bool suppress_bind_attrs;       /* disables bind/unbind via
sysfs */
 236
 237         const struct of_device_id       *of_match_table;
```

4. Linux i2c子系统(一) _动手写一个i2c设备驱动(23320)
5. Linux设备管理（一）_kobject, kset,ktype分析(19570)
6. Linux块设备IO子系统(一) _驱动模型(17375)
7. Linux设备管理（二）_从cdev_add说起(16572)
8. 从0移植uboot (一) _配置分析(14214)
9. Linux字符设备驱动框架(13786)
10. 从0移植uboot (二) _uboot启动流程分析(13482)

评论排行榜

1. Linux tcp黏包解决方案(7)
2. Linux设备管理（一）_kobject, kset,ktype分析(6)
3. Linux usb子系统(一) _写一个usb鼠标驱动(2)
4. 跟着内核学框架-从misc子系统到3+2+1设备识别驱动框架(2)
5. Linux驱动技术(八) _并发控制技术(2)
6. Linux驱动技术(四) _异步通知技术(2)
7. Linux设备文件三大结构：inode,file,file_operations(2)
8. Linux 多线程信号量同步(2)
9. 从0移植uboot(六) _实现网络功能(1)
10. Linux input子系统编程、分析与模板(1)

推荐排行榜

1. Linux Platform驱动模型(二) _驱动方法(11)
2. Linux设备树语法详解(11)
3. Linux i2c子系统(一) _动手写一个i2c设备驱动(6)
4. Linux Platform驱动模型(一) _设备信息(6)
5. Linux usb子系统(二) _usb-skeleton.c精析(5)

```
238         const struct acpi_device_id     *acpi_match_table;
239
240         int (*probe) (struct device *dev);
241         int (*remove) (struct device *dev);
242         void (*shutdown) (struct device *dev);
243         int (*suspend) (struct device *dev, pm_message_t state);
244         int (*resume) (struct device *dev);
245         const struct attribute_group **groups;
246
247         const struct dev_pm_ops *pm;
248
249         struct driver_private *p;
250 };
```

下面是我们关心的几个成员

> **struct device_driver**
> --229-->驱动名，如果这个驱动只匹配一个C语言的设备，那么可以通过name相同来匹配
> --230-->总线类型，这个成员由内核填充
> --232-->owner，通常就写THIS_MODULE
> --237-->of_device_id顾名思义就是用来匹配用设备树写的设备信息，下面详细讨论
> --249-->私有数据

driver与device的匹配

设备信息有三种表达方式，而一个驱动是可以匹配多个设备的，平台总线中的驱动要具有三种匹配信息的能力，基于这种需求，platform_driver中使用不同的成员来进行相应的匹配。

of_match_table

对于使用设备树编码的设备信息，我们使用其父类device_driver中的of_match_table就是用来匹配

```
//include/linux/mod_devicetable.h
220 /*
```

```
221  * Struct used for matching a device
222  */
223 struct of_device_id
224 {
225         char    name[32];
226         char    type[32];
227         char    compatible[128];
228         const void *data;
229 };
```

struct of_device_id

--225-->name[32]设备名

--226-->type[32]设备类型

--227-->**重点！**compatible[128]用于与设备树compatible属性值匹配的字符串

--228-->data驱动私有数据

对于一个驱动匹配多个设备的情况，我们使用struct of_device_id tbl[]来表示。

```
struct of_device_id of_tbl[] = {
        {.compatible = "xj4412,demo0",},
        {.compatible = "xj4412,demo1",},
        {},
};
```

id_table

对于使用C语言编码的设备信息，我们用platform_driver对象中的id_table就是用来匹配。我们使用struct platform_device_id ids[]来实现一个驱动匹配多个C语言编码的设备信息。

```
//include/linux/mod_deviceid.h
485 struct platform_device_id {
486         char name[PLATFORM_NAME_SIZE];
```

```
487         kernel_ulong_t driver_data;
488 };
```

> **struct platform_device_id**
> --486-->name就是设备名

下面这个例子就是用一个驱动来匹配两个分别叫**"demo0"和"demo1"**的设备，注意，数组最后的**{}**是一定要的，这个是内核判断数组已经结束的标志。

```
static struct platform_device_id tbl[] = {
        {"demo0"},
        {"demo1"},
        {},
};
```

name

如果platform_driver和C语言编码的platform_device是一一匹配的，我们还可以使用device_driver中的name来进行匹配

注册设备表

填充完platform_driver结构之后，我们应该将其中用到的设备表注册到内核，虽然不注册也可以工作，但是注册可以将我们表加入到相关文件中，便于内核管理设备。

```
MODULE_DEVICE_TABLE(类型, ID表);
设备树ID表
类型:of
C写的platform_device的ID表
类型:platform
C写的i2c设备的ID表
类型:i2c
C写的USB设备的ID表
类型:usb
```

匹配小结

细心的读者可能会发现，这么多方式都写在一个对象中，那如果我同时注册了三种匹配结构内核该用哪种呢？此时就需要我们搬出平台总线的匹配方式：

```
//drivers/base/platform.c
 748 static int platform_match(struct device *dev, struct device_driver
*drv)
 749 {
 750         struct platform_device *pdev = to_platform_device(dev);
 751         struct platform_driver *pdrv = to_platform_driver(drv);
 752
 753         /* Attempt an OF style match first */
 754         if (of_driver_match_device(dev, drv))
 755                 return 1;
 756
 757         /* Then try ACPI style match */
 758         if (acpi_driver_match_device(dev, drv))
 759                 return 1;
 760
 761         /* Then try to match against the id table */
 762         if (pdrv->id_table)
 763                 return platform_match_id(pdrv->id_table, pdev) !=
NULL;
 764
 765         /* fall-back to driver name match */
 766         return (strcmp(pdev->name, drv->name) == 0);
 767 }
```

从中不难看出，这几中形式的匹配是有优先级的：**of_match_table>id_table>name**，了解到这点，我们甚至可以构造出同时适应两种设备信息的平台驱动：

```
static struct platform_driver drv = {
        .probe  = demo_probe,
```

```
        .remove = demo_remove,

        .driver = {
                .name = "demo",
#ifdef CONFIG_OF
                .of_match_table = of_tbl,
#endif
        },

        .id_table = tbl,
};
```

此外，如果你追一下of_driver_match_device()，就会发现**平台总线的最终的匹配是compatible，name,type三个成员，其中一个为NULL或""时表示任意，所以我们使用平台总线时总是使用compatile匹配设备树，而不是节点路径或节点名。**

probe()

probe即探测函数，如果驱动匹配到了目标设备，总线会自动回调probe函数，下面详细讨论。并把匹配到的设备信息封装策划嗯platform_device对象传入，里面主要完成下面三个工作

1. **申请资源**
2. **初始化**
3. **提供接口(cdev/sysfs/proc)**

显然，remove主要完成与probe相反的操作，这两个接口都是我们必须实现的。

在probe的工作中，最常见的就是提取设备信息，虽然总线会将设备信息封装成一个platform_device对象并传入probe函数，我们可以很容易的得到关于这个设备的所有信息，但是更好的方法就是直接使用内核API中相关的函数

```
/**
 * platform_get_resource - 获取资源
 * @dev: 平台总线设备
 * @type:资源类型，include/linux/ioport.h中有定义
```

```
 * @num: 资源索引，即第几个此类型的资源，从0开始
 */
struct resource *platform_get_resource(struct platform_device *dev,unsigned
int type, unsigned int num)
```

注意，通过内核API(eg,上下这两个API)获取的resource如果是中断，那么只能是软中断号，而不是芯片手册/C语言设备信息/设备树设备信息中的硬中断号，但是此时获取的resource的flag是可以正确的反映该中断的触发方式的，只需要 `flag & IRQF_TRIGGER_MASK` 即可获取该中断的触发方式。

```
/**
 * platform_get_irq - 获取一个设备的中断号
 * @dev: 平台总线设备
 * @num: 中断号索引，即想要获取的第几个中断号，从0开始
 */
int platform_get_irq(struct platform_device *dev, unsigned int num)
```

```
/**
 * dev_get_platdata - 获取私有数据
 */
static inline void *dev_get_platdata(const struct device *dev){
        return dev->platform_data;
}
```

注册/注销platform_driver对象

内核提供了两个API来注册/注销platform_driver对象到内核

```
/**
 * platform_driver_register - 注册
 */
int platform_driver_register(struct platform_driver *drv);
```

```
/**
 * platform_driver_unregister - 注销
 */
int platform_driver_unregister(struct platform_driver *drv);
```

在动态编译的情况下，我们往往在模块初始化函数中注册一个驱动方法对象，而在模块卸载函数中注销一个驱动方法对象，所以我们可以使用内核中如下的宏来提高代码复用

```
module_platform_driver(driver_name);
```

实例

这个实例同时使用了设备信息模块和设备树两种设备信息来源，不过最终使用的是设备树，需要注意的是，当我们用设备树的设备信息时，有一个成员platform_device.device.of_node来表示设备的节点，这样就允许我们使用丰富的设备树操作API来操作。

```
//#include "private.h"
/*
/{
        demo{
                compatible = "4412,demo0";
                reg = <0x5000000 0x2 0x5000008 0x2>;
                interrupt-parent = <&gic>;
                interrupts = <0 25 0>, <0 26 0>;
                intpriv = <0x12345678>;
                strpriv = "hello world";
        };
};
*/

struct privatedata {
        int val;
        char str[36];
```

```
};
static void getprivdata(struct device_node *np)
{
        struct property *prop;
        prop = of_find_property(np, "intpriv", NULL);
        if(prop)
                printk("private val: %x\n", *((int *)(prop->value)));

        prop = of_find_property(np, "strpriv", NULL);
        if(prop)
                printk("private str: %s\n", (char *)(prop->value) );
}

static int demo_probe(struct platform_device *pdev)
{
        int irq;
        struct resource *addr;
        struct privatedata *priv;

        printk(KERN_INFO "%s : %s : %d - entry.\n", __FILE__, __func__,
__LINE__);

        priv = dev_get_platdata(&pdev->dev);
        if(priv){
                printk(KERN_INFO "%x : %s \n", priv->val, priv->str);
        }else{
                getprivdata(pdev->dev.of_node);
        }

        addr = platform_get_resource(pdev, IORESOURCE_MEM, 0);
        if(addr){
                printk(KERN_INFO "0: %x : %d \n", addr->start,
resource_size(addr));
        }
        addr = platform_get_resource(pdev, IORESOURCE_MEM, 1);
        if(addr){
```

```
                printk(KERN_INFO "1: %x : %d \n", addr->start,
resource_size(addr));
        }
        addr = platform_get_resource(pdev, IORESOURCE_MEM, 2);
        if(!addr){
                printk(KERN_INFO "No 2 resource\n");
        }

        irq = platform_get_irq(pdev, 0);
        if(0 > irq){
                return irq;
        }else{
                printk(KERN_INFO "irq 0: %d \n", irq);
        }
        irq = platform_get_irq(pdev, 1);
        if(0 > irq){
                return irq;
        }else{
                printk(KERN_INFO "irq 0: %d \n", irq);
        }

        irq = platform_get_irq(pdev, 2);
        if(0 > irq){
                        printk(KERN_INFO "No 2 irq\n");
        }

        return 0;
}

static int demo_remove(struct platform_device *pdev)
{
        return 0;
}

static struct platform_device_id tbl[] = {
        {"demo0"},
```

```
        {"demo1"},
        {},
};
MODULE_DEVICE_TABLE(platform, tbl);

#ifdef CONFIG_OF
struct of_device_id of_tbl[] = {
        {.compatible = "4412,demo0",},
        {.compatible = "4412,demo1",},
        {},
};
#endif

//1. alloc obj
static struct platform_driver drv = {
        .probe  = demo_probe,
        .remove = demo_remove,

        .driver = {
                .name = "demo",
#ifdef CONFIG_OF
                .of_match_table = of_tbl,
#endif
        },

        .id_table = tbl,
};


static int __init drv_init(void)
{
        //get command and pid
        printk(KERN_INFO "(%s:pid=%d), %s : %s : %d - entry.\n",current-
>comm, current->pid, __FILE__, __func__, __LINE__);
        return platform_driver_register(&drv);
}
```

```
static void __exit drv_exit(void)
{
        //get command and pid
        printk(KERN_INFO "(%s:pid=%d), %s : %s : %d - leave.\n",current->comm, current->pid, __FILE__, __func__, __LINE__);

        platform_driver_unregister(&drv);
}
module_init(drv_init);
module_exit(drv_exit);

MODULE_LICENSE("GPL");
```

分类: Linux驱动开发

标签: 平台总线, platform_driver, platfom_match, 获取设备信息api, platform_get_resource

Abnor
关注 - 0
粉丝 - 294
+加关注

11 0

« 上一篇: Linux Platform驱动模型(一)_设备信息
» 下一篇: Linux Platform驱动模型(三)_platform+cdev

posted @ 2017-02-06 08:16 Abnor 阅读(24908) 评论(0) 编辑 收藏 举报