Linux内核中的GPIO系统之（3）：pin controller driver代码分析

作者：linuxer 发布于：2014-7-22 20:37 分类：GPIO子系统

一、前言

对于一个嵌入式软件工程师，我们的软件模块经常和硬件打交道，pin control subsystem也不例外，被它驱动的硬件叫做pin controller（一般ARM soc的datasheet会把pin controller的内容放入GPIO controller的章节中），主要功能包括：

（1）pin multiplexing。基于ARM core的嵌入式处理器一般会提供丰富的功能，例如camera interface、LCD interface、USB、I2C、SPI等等。虽然处理器有几百个pin，但是这些pin还是不够分配，因此有些pin需要复用。例如：127号GPIO可以做一个普通的GPIO控制LED，也可以配置成I2C的clock信号，也可以配置成SPI的data out信号。当然，这些功能不可能同时存在，因为硬件信号只有一个。

（2）pin configuration。这些配置参数包括：pull-up/down电阻的设定， tri-state设定，drive-strength的设定。

本文主要描述pin control subsystem中的low level driver，也就是驱动pin controller的driver。具体的硬件选用的是S3C2416的硬件平台。既然是代码分析，本文不是非常多的描述框架性的内容，关于整个pin control subsystem软件结构的描述请参考Linux内核中的GPIO系统之（2）。

阅读本文需要device tree的知识，建议首先阅读device tree代码分析。

二、pin controller相关的DTS描述

类似其他的硬件，pin controller这个HW block需要是device tree中的一个节点。此外，各个其他的HW block在驱动之前也需要先配置其引脚复用功能，因此，这些device（我们称pin controller是host，那么这些使用pin controller进行引脚配置的device叫做client device）也需要在它自己的device tree node中描述pin control的相关内容

1、S3C2416 pin controller DTS结构

下面的伪代码描述了S3C2416 pin controller 的DTS结构：

```
pinctrl@56000000 {
        定义S3C2416 pin controller自己的属性

        定义属于S3C2416 pin controller的pin configurations

}
```

每个pin configuration都是pin controller的child node，描述了client device要使用到的一组pin的配置信息。具体如何定义pin configuration是和具体的pin controller相关的。

在pin controller node中定义pin configuration其目的是为了让client device引用。所谓client device其实就是使用pin control subsystem提供服务的那些设备，例如串口设备。在使用之前，我们一般会在初始化代码中配置相关的引脚功能是串口功能。有了device tree，我们可以通过device tree来传递这样的信息。也就是说，各个device可以通过自己节点的属性来指向pin controller的某个child node，也就是pin configuration了。samsung 24xx系列SOC的pin controller的pin configurations包括两类，一类是定义pin bank，另外一类是定义功能复用配置。

2、pin configuration定义

我们举两个简单的例子（当然一个是pin bank，另外一个是定义功能复用配置）来理解pin configuration第一个例子是描述pin bank：

```
pinctrl@56000000 {
        定义S3C2416 pin controller自己的属性
```

站内搜索

功能

留言板
评论列表

随机文章

Linux设备模型(7)_Class
Process Creation（二）
linux kernel内存碎片防治技术
Windows系统结合MinGW搭建软件开发环境
systemd：为何要创建一个新的init系统软件

最新评论

lover713814
@wilson：可以加我微信zhangfan747551 ...
wowo
@franc：博客和论坛的账号不能通用，另外建议大家不要登录...
franc
无法登录
wowo
@Gute：我对BLE也是边理解边记录，大家可以多交流~
wowo
@tinylaker：多谢建议，下次整理的时候，一定填上！

文章分类

Linux内核分析(10)
统一设备模型(12)
电源管理子系统(40)
中断子系统(14)
进程管理(5)
内核同步机制(17)
GPIO子系统(3)
时间子系统(13)
通信类协议(4)
内存管理(5)
图形子系统(1)
Linux应用技巧(11)
软件开发(5)
基础技术(2)
蓝牙(6)
ARMv8A Arch(4)
显示(3)
基础学科(9)
技术漫谈(10)
项目专区(0)
X Project(2)

文章存档

2016年4月(7)
2016年3月(5)
2016年2月(5)
2016年1月(6)

```
……

        gpf {
            gpio-controller;
            #gpio-cells = <0x2>;
            interrupt-controller;
            #interrupt-cells = <0x2>;
            linux,phandle = <0xc>;
            phandle = <0xc>;
        };

……

}
```

其实S3C2416 pin controller定义了gpa到gpm共计11个sub node，每个sub node是描述S3C2416 GPIO controller的各个bank信息。S3C2416有138个I/O 端口（或者叫做pin、finger、pad）这些端口分成了11个bank（这里没有用group这个术语，为了和pin group这个概念区分开，pin group的概念下面会具体描述）：

```
Port A(GPA) : 25-output port
Port B(GPB) : 9-input/output port
Port C(GPC) : 16-input/output port
Port D(GPD) : 16-input/output port
Port E(GPE) : 16-input/output port
Port F(GPF) : 8-input/output port
Port G(GPG) : 8-input/output port
Port H(GPH) : 15-input/output port
Port K(GPK) : 16-input/output port
Port L(GPL) : 7-input/output port
Port M(GPM) : 2-input port
```

之所以分成bank，主要是把特性相同的GPIO进行分组，方便控制。例如：这些bank中，只有GPF和GPG这两个bank上的引脚有中断功能，其他的都没有。interrupt-controller这个属性相信大家已经熟悉，就是说明该node是一个interrupt controller。gpio-controller类似，说明该device node是一个GPIO controller。#gpio-cells属性是一个GPIO controller的必须定义的属性，它描述了需要多少个cell来具体描述一个GPIO（这是和具体的GPIO controller相关的）。#interrupt-cells的概念类似，不再赘述。phandle（linux,phandle这个属性和phandle是一样的，只不过linux,phandle是old-style，多定义一个属性是为了兼容）定义了一个句柄，当其他的device node想要引用这个node的时候就可以使用该句柄。具体的例子参考下节client device的DTS的描述。

另外一个例子是uart的pin configuration，代码如下：

```
pinctrl@56000000 {
        定义S3C2416 pin controller自己的属性

……

uart0-data {
    samsung,pins = "gph-0", "gph-1";
    samsung,pin-function = <0x2>;
    linux,phandle = <0x2>;
    phandle = <0x2>;
};

uart0-fctl {
    samsung,pins = "gph-8", "gph-9";
    samsung,pin-function = <0x2>;
    linux,phandle = <0x3>;
    phandle = <0x3>;
};

……

}
```

samsung,pins这个属性定义了一个pin configuration所涉及到的引脚定义。对于uart0-data这个node，该配置涉及了gph bank中的第一个和第二个GPIO pin。一旦选择了一个功能，那么samsung,pins定义的所有的引脚都需要做相应的功能设定，那么具体设定什么值呢？这就是samsung,pin-function定义的内容了。而具体设定哪个值则需要去查阅datasheet了。对于uart0-data，向gph bank中的第一个和第二个GPIO pin对应的配置寄存器中写入2就可以把这两个pin定义为uart功能。

2015年12月(6)
2015年11月(9)
2015年10月(9)
2015年9月(4)
2015年8月(3)
2015年7月(7)
2015年6月(3)
2015年5月(7)
2015年4月(9)
2015年3月(9)
2015年2月(6)
2015年1月(6)
2014年12月(17)
2014年11月(8)
2014年10月(9)
2014年9月(7)
2014年8月(12)
2014年7月(6)
2014年6月(6)
2014年5月(9)
2014年4月(9)
2014年3月(7)
2014年2月(3)
2014年1月(4)

3.client device的DTS

一个典型的device tree中的外设node定义如下：

```
device-node-name {
        定义该device自己的属性

        pinctrl-names = "sleep", "active"; - - - - - - （1）
        pinctrl-0 = <pin-config-0-a>; - - - - - - - - - - - - - - - （2）
        pinctrl-1 = <pin-config-1-a pin-config-1-b>;
    };
```

（1）pinctrl-names定义了一个state列表。那么什么是state呢？具体说应该是pin state，对于一个client device，它使用了一组pin，这一组pin应该同时处于某种状态，毕竟这些pin是属于一个具体的设备功能。state的定义和电源管理关系比较紧密，例如当设备active的时候，我们需要pin controller将相关的一组pin设定为具体的设备功能，而当设备进入sleep状态的时候，需要pin controller将相关的一组pin设定为普通GPIO，并精确的控制GPIO状态以便节省系统的功耗。state有两种，标识，一种就是pinctrl-names定义的字符串列表，另外一种就是ID。ID从0开始，依次加一。根据例子中的定义，state ID等于0（名字是active）的state对应pinctrl-0属性，state ID等于1（名字是idle）的state对应pinctrl-1属性。具体设备state的定义和各个设备相关，具体参考在自己的device bind。

（2）pinctrl-x的定义。pinctrl-x是一个句柄（phandle）列表，每个句柄指向一个pin configuration。有时候，一个state对应多个pin configure。例如在active的时候，I2C功能有两种配置，一种是从pin ID{7,8}引出，另外一个是从pin ID {69,103}引出。

我们选取samsung串口的dts定义如下：

```
serial@50000000 {
        ……
        pinctrl-names = "default";
        pinctrl-0 = <0x2 0x3>;
    };
```

该serial device只定义了一个state就是default，对应pinctrl-0属性定义。pinctrl-0是一个句柄（phandle）列表，每个句柄指向一个pin configuration。0x2对应上节中的uart0-data节点，0x03对应uart0-fctl 节点，也就是说，这个串口有两种配置，一种是从gph bank中的第一个和第二个GPIO pin引出，另外一个是从gph bank中的第8个和第9个GPIO pin引出。

三、 pin controller driver初始化

1、注册pin control device

旧的内核一般是在machine相关的代码中建立静态的platform device的数据结构，然后在machine初始化的时候，将静态定义的platform device注册到系统。不过在引入device tree之后，事情发生了变化。

根据device tree代码分析，我们知道，在系统初始化的时候，dts描述的device node会形成一个树状结构，在machine初始化的过程中，会scan device node的树状结构，将真正的硬件device node变成一个个的设备模型中的device结构（比如struct platform_device）并加入到系统中。我们看看具体2416描述pin controller的dts code，如下：

```
pinctrl@56000000 {
        reg = <0x56000000 0x1000="">;
        compatible = "samsung,s3c2416-pinctrl";

……省略wakeup的pin configuration

……省略gpb～gpm这些pink bank的pin configuration

……省略Pin groups的相关描述

}
```

reg属性描述pin controller硬件的地址信息，开始地址是0x56000000 ，地址长度是0x1000。compatible属性用来描述pin controller的programming model。该属性的值是string list，定义了一系列的modle（每个string是一个model）。这些字符串列表被操作系统用来选择用哪一个pin controller driver来驱动该设备，后面的代码会更详细的描述。 pin control subsystem要想进行控制，必须首先了解自己控制的对象，也就是说软件需要提供一个方法将各种硬件信息

（total有多少可控的pin，有多少bank，pin的复用情况以及pin的配置情况）注册到pin control subsystem中，这也是pin controller driver的初始化的主要内容。这些信息当然可以通过定义静态的表格（参考linux/drivers/pinctrl目录下的pinctrl-u300.c文件，该文件定义了一个大数组u300_pads来描述每一个pin），也可以通过dts加上静态表格的方式（2416采用的方式）。

2、注册pin controller driver

当然，pinctrl@56000000这个device node也会变成一个platform device加入系统。有了device，那么对应的platform driver是如何注册到系统中的呢？代码如下：

```
static int __init samsung_pinctrl_drv_register(void)
{
   ……

   return platform_driver_register(&samsung_pinctrl_driver);
}
```

系统初始化的时候，该函数会执行，向系统注册了samsung_pinctrl_driver：

```
static struct platform_driver samsung_pinctrl_driver = {
    .probe        = samsung_pinctrl_probe，- - - - 该driver的初始化函数
    .driver = {
        .name    = "samsung-pinctrl",
        .owner    = THIS_MODULE,
        .of_match_table = samsung_pinctrl_dt_match，- - - - 匹配列表
    },
};
```

3、probe过程（driver初始化过程）

在linux kernel引入统一设备模型之后，bus、driver和device形成了设备模型中的铁三角。对于platform这种类型的bus，其铁三角数据是platform_bus_type（表示platform这种类型的bus）、struct platform_device（platform bus上的device）、struct platform_driver（platform bus上的driver）。统一设备模型大大降低了驱动工程师的工作量，驱动工程师只要将driver注册到系统即可，剩余的事情交给统一设备模型来完成。每次系统增加一个platform_driver，platform_bus_type都会启动scan过程，让新加入的driver扫描整个platform bus上的device的链表，看看是否有device让该driver驱动。同样的，每次系统增加一个platform_device，platform_bus_type也会启动scan过程，遍历整个platform bus上的driver的链表，看看是否有适合驱动该device的driver。具体匹配的代码是platform bus上的match函数，如下：

```
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);

    /* Attempt an OF style match first */
    if (of_driver_match_device(dev, drv))
        return 1;

    /* Then try ACPI style match */
    if (acpi_driver_match_device(dev, drv))
        return 1;

    /* Then try to match against the id table */
    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);
}
```

旧的的platform的匹配函数就是简单的比较device和driver的名字，多么简单，多么清晰，真是有点怀念过去单纯而美好的生活。当然，事情没有那么糟糕，我们这里只要关注OF style的匹配过程即可，思路很简单，解铃还需系铃人，把匹配过程推给device tree模块，代码如下：

```
const struct of_device_id *of_match_device(const struct of_device_id *matches,
                    const struct device *dev)
{
    if ((!matches) || (!dev->of_node))
        return NULL;
    return of_match_node(matches, dev->of_node);
}
```

platform driver提供了match table（struct device_driver 中的of_match_table的成员）。platform device提供了device tree node（struct device中的of_node成员）。对于我们这个场景，match table是samsung_pinctrl_dt_match，代码如下：

```
static const struct of_device_id samsung_pinctrl_dt_match[] = {
……
    { .compatible = "samsung,s3c2416-pinctrl",
        .data = s3c2416_pin_ctrl },
……
    {},
};
```

再去看看dts中pin controller的节点compatible属性的定义，你会禁不住感慨：啊，终于遇到对的人。这里还要特别说明的是data成员被设定为s3c2416_pin_ctrl ，它描述了2416的HW pin controller，我们称之samsung pin controller的描述符，初始化的过程中需要这个数据结构，后面还会详细介绍它。一旦pin controller这个device遇到了适当的driver，就会调用probe函数进行具体的driver初始化的动作了，代码如下：

```
static int samsung_pinctrl_probe(struct platform_device *pdev)
{
    struct samsung_pinctrl_drv_data *drvdata;
    struct device *dev = &pdev->dev;
    struct samsung_pin_ctrl *ctrl;
    struct resource *res;
    int ret;

    drvdata = devm_kzalloc(dev, sizeof(*drvdata), GFP_KERNEL); - - - - - - （1）

    ctrl = samsung_pinctrl_get_soc_data(drvdata, pdev); - - - - - - - - - - （2）
    drvdata->ctrl = ctrl;
    drvdata->dev = dev;

    res = platform_get_resource(pdev, IORESOURCE_MEM, 0); - - - - - 分配memory资源
    drvdata->virt_base = devm_ioremap_resource(&pdev->dev, res);
    if (IS_ERR(drvdata->virt_base))
        return PTR_ERR(drvdata->virt_base);

    res = platform_get_resource(pdev, IORESOURCE_IRQ, 0); - - - - - - 分配IRQ资源
    if (res)
        drvdata->irq = res->start;

    ret = samsung_gpiolib_register(pdev, drvdata); - - - - - - - - - - - - - （3）

    ret = samsung_pinctrl_register(pdev, drvdata); - - - - - - - - - - - - - （4）

    if (ctrl->eint_gpio_init) - - - - - - - - - - - - - - - - - - - （5）
        ctrl->eint_gpio_init(drvdata);
    if (ctrl->eint_wkup_init)
        ctrl->eint_wkup_init(drvdata);

    platform_set_drvdata(pdev, drvdata); - 设定platform device的私有数据为
samsung_pinctrl_drv_data

    /* Add to the global list */
    list_add_tail(&drvdata->node, &drvdata_list); - - 挂入全局链表

    return 0;
}
```

（1）devm_kzalloc函数是为struct samsung_pinctrl_drv_data数据结构分配内存。每当driver probe一个具体的device实例的时候，都需要建立一些私有的数据结构来保存该device的一些具体的硬件信息（本场景中，这个数据结构就是struct samsung_pinctrl_drv_data）。在过去，驱动工程师多半使用kmalloc或者kzalloc来分配内存，但这会带来一些潜在的问题。例如：在初始化过程中，有各种各样可能的失败情况，这时候就依靠driver工程师小心的撰写代码，释放之前分配的内存。当然，初始化过程中，除了memory，driver会为probe的device分配各种资源，例如IRQ 号，io memory map、DMA等等。当初始化需要管理这么多的资源分配和释放的时候，很多驱动程序都出现了资源管理的issue。而且，由于这些issue是异常路径上的issue，不是那么容易测试出来，更加重了解决这个issue的必要性。内核解决这个问题的模式（所谓解决一类问题的设计方法就叫做设计模式）是Devres，即device resource management软件模块。更细节的内容就不介绍了，其核心思想就是资源是设备的资源，那么资源的管理归于device，也就是说不需要driver过多的参与。当device和driver detach的时候，device会自动的释放其所有的资源。

（2）分配了struct samsung_pinctrl_drv_data数据结构的内存，当然下一步就是初始化这个数据结构了。我们先看看2416的pin controller driver是如何定义该数据结构的：

```
struct samsung_pinctrl_drv_data {
    struct list_head        node; - - - - - - - - - 多个pin controller的描述符可以形成链表
    void __iomem            *virt_base; - - - - - - - - - 访问硬件寄存器的基地址
    struct device           *dev; - - - - - - - - - - - 和platform device建立联系
    int                 irq; - - - - - - - - irq number，对于2416 pin control硬件而言，不需要irq资源

    struct samsung_pin_ctrl     *ctrl; - - - - samsung pin controller描述符
    struct pinctrl_desc         pctl; - - - - - - 指向pin control subsystem中core driver中抽象的

                                                pin controller描述符。
    struct pinctrl_dev      *pctl_dev; - - - - - - 指向core driver的pin controller class device

    const struct samsung_pin_group    *pin_groups; - 描述samsung pin controller中pin groups的信息
    unsigned int            nr_groups; - - - - - - - - 描述samsung pin controller中pin groups的数目
    const struct samsung_pmx_func    *pmx_functions; - - 描述samsung pin controller中function信息
    unsigned int            nr_functions; - - - - - - - - 描述samsung pin controller中function的数目
};
```

struct pinctrl_desc和struct pinctrl_dev 都是pin control subsystem中core driver的概念。各个具体硬件的pin controller可能会各不相同，但是可以抽取其共同的部分来形成一个HW independent的数据结构，这个数据就是pin controller描述符，在core driver中用struct pinctrl_desc表示，具体该数据结构的定义如下：

```
struct pinctrl_desc {
    const char *name;
    struct pinctrl_pin_desc const *pins; - - - 指向npins个pin描述符，每个描述符描述一个pin
    unsigned int npins; - - - - - - - - - - - - - 该pin controller中有多少个可控的pin
    const struct pinctrl_ops *pctlops; - - - - - - callback函数，是core driver和底层driver的接口
    const struct pinmux_ops *pmxops; - - - - - callback函数，是core driver和底层driver的接口
    const struct pinconf_ops *confops; - - - - - callback函数，是core driver和底层driver的接口
    struct module *owner;
};
```

其实整个初始化过程的核心思想就是low level的driver定义一个pinctrl_desc ，设定pin相关的定义和callback函数，注册到pin control subsystem中。我们用引脚描述符（pin descriptor）来描述一个pin。在pin control subsystem中，struct pinctrl_pin_desc用来描述一个可以控制的引脚，我们称之引脚描述符，代码如下：

```
struct pinctrl_pin_desc {
    unsigned number; - - - - - - - ID，在本pin controller中唯一标识该引脚
    const char *name; - - - - - - - user friedly name
    void *drv_data;
};
```

冰冷的pin ID是给机器用的，而name是给用户使用的，是ascii字符。

struct pinctrl_dev在pin control subsystem的core driver中抽象一个pin control device。其实我们可以通过多个层面来定义一个device。在这个场景下，pin control subsystem的core driver关注的是一类pin controller的硬件设备，具体其底层是什么硬件连接方式，使用什么硬件协议它不关心，它关心的是pin controller这类设备所有的通用特性和功能。当然2416的pin controller是通过platform bus连接的，因此，在low level的层面，需要一个platform device来标识2416的pin controller（需要注意的是：pin controller class device和platform device都是基于一个驱动模型中的device派生而来的，这里struct device是基类，struct pinctrl_dev和struct platform_device都是派生类，当然c本身不支持class，但面向对象的概念是同样的）。为了充分理解class这个概念，我们再举一个例子。对于audio的硬件抽象层，它应该管理所有

的audio设备，因此这个软件模块应该有一个audio class的链表，连接了所有的系统中的audio设备。但这些具体的audio设备可能是PCI接口的audio设备，也可能是usb接口的audio设备，从具体的总线层面来看，也会有PCI或者USB设备来抽象对应的声卡设备。

OK，我们再看看samsung_pinctrl_drv_data底部四个成员，要理解该数据结构底部的四个成员，还要理解什么是pin mux function，什么是pin group。对于SOC而言，其引脚除了配置成普通GPIO之外，若干个引脚还可以组成一个pin group，形成特定的功能。例如pin number是{ 0, 8, 16, 24 }这四个引脚组合形成一个pin group，提供SPI的功能。既然有了pin group的概念，为何又有function这个概念呢？什么是function呢？SPI是function，I2C也是一个function，当然GPIO也是一个function。一个function有可能对应一组或者多组pin。例如：为了设计灵活，芯片内部的SPI0的功能可能引出到pin group { A8, A7, A6, A5 }，也可能引出到另外一个pin group{ G4, G3, G2, G1 }，但毫无疑问，这两个pin group不能同时active，毕竟芯片内部的SPI0的逻辑功能电路只有一个。 从这个角度看，pin control subsystem要进行功能设定的时候必须要给出function以及function的pin group才能确定所有的物理pin的位置。

我们前面已经说过了，struct samsung_pinctrl_drv_data数据结构就是2416的pin controller driver要驱动2416的HW pin controller的私有数据结构。这个数据结构中最重要的就是samsung pin controller描述符了。关于pin controller有两个描述符，一个是struct pinctrl_desc，是具体硬件无关的pin controller的描述符。struct samsung_pin_ctrl描述的具体samsung pin controller硬件相关的信息，比如说：pin bank的信息，不是所有的pin controller都是分bank的，因此pin bank的信息只能封装在low level的samsung pin controller driver中。这个数据结构定义如下：

```
struct samsung_pin_ctrl {
    struct samsung_pin_bank    *pin_banks; - - - - 定义具体的pin bank信息
    u32        nr_banks; - - - - - - - - - - number of pin bank

    u32        base; - - - - 该pin controller的pin ID base。
    u32        nr_pins; - - - - - 总的可以控制的pin的数目

其他成员和本场景无关，和GPIO type的中断控制器驱动代码有关
};
```

关于上面的base可以多说两句。实际上，系统支持多个pin controller设备，这时候，系统要管理多个pin controller控制下的多个pin。每个pin有自己的pin ID，是唯一的，假设系统中有两个pin controller，一个是A，控制32个，另外一个是B，控制64个pin，我们可以统一编号，对A，pin ID从0～31，对于B，pin ID是从32～95。对于B，其pin base就是32。

samsung_pinctrl_probe->samsung_pinctrl_get_soc_data函数中会根据device tree的信息和静态定义的table来初始化该描述符，具体的代码如下：

```
static struct samsung_pin_ctrl *samsung_pinctrl_get_soc_data(
                struct samsung_pinctrl_drv_data *d,
                struct platform_device *pdev)
{
    int id;
    const struct of_device_id *match;
    struct device_node *node = pdev->dev.of_node; - - - 获取device tree中的device node指针
    struct device_node *np;
    struct samsung_pin_ctrl *ctrl;
    struct samsung_pin_bank *bank;
    int i;

    id = of_alias_get_id(node, "pinctrl");
    match = of_match_node(samsung_pinctrl_dt_match, node);
    ctrl = (struct samsung_pin_ctrl *)match->data + id; - - - - - - - - - A

    bank = ctrl->pin_banks;
    for (i = 0; i < ctrl->nr_banks; ++i, ++bank) { - - - - - - - - - - - - - B
        spin_lock_init(&bank->slock);
        bank->drvdata = d;
        bank->pin_base = ctrl->nr_pins; - - - ctrl->nr_pins初始的时候等于0，最后完成bank初始化后，

                                            该值等于total的pin number。
        ctrl->nr_pins += bank->nr_pins;
    }

for_each_child_of_node(node, np) {  - - - - - - - - - - - - - - - - - C
        if (!of_find_property(np, "gpio-controller", NULL))
            continue;
        bank = ctrl->pin_banks;
        for (i = 0; i < ctrl->nr_banks; ++i, ++bank) {
            if (!strcmp(bank->name, np->name)) {
                bank->of_node = np;
                break;
            }
```

```
        }
    }

ctrl->base = pin_base; - - - - - - - - - - - - - - - - - - - - - - - - D
    pin_base += ctrl->nr_pins;

    return ctrl;
}
```

samsung_pinctrl_get_soc_data这个函数名字基本反应了其功能，2416是samsung的一个具体的SOC型号，调用该函数可以返回一个表示2416 SOC的samsung pin controller的描述符。

A：这段代码主要是获取具体的2416的HW pin controller的信息，该数据结构在上文中出现过（具体参考pin controller的device tree match table：samsung_pinctrl_dt_match），就是s3c2416_pin_ctrl这个变量。这个变量定义了2416的pin controller的信息（S3C2416的pin controller的pin bank信息是定义在pin controller driver的静态数据，其实最好在dts中定义）如下：

```
struct samsung_pin_ctrl s3c2416_pin_ctrl[] = {
    {
        .pin_banks    = s3c2416_pin_banks, - - - - - - 静态定义的2416的pin bank的信息
        .nr_banks    = ARRAY_SIZE(s3c2416_pin_banks),
        .eint_wkup_init = s3c24xx_eint_init,
        .label        = "S3C2416-GPIO",
    },
};
```

这个变量中包含了2416的pin bank的信息，包括：有多少个pin bank，每个bank中有多少个pin，pin bank的名字是什么，寄存器的offset是多少。这些信息用来初始化pin controller描述符数据结构。

B：初始化2416 samsung pin controller中各个bank的描述符。

C：device tree中表示pin controller的device node有若干的child node，分别表示gpa～gpl这11个bank，每个bank都是一个gpio controller。下面的代码遍历各个child node，并初始化各个bank描述符中的device tree node成员。 这里需要注意的是静态定义的pin bank的名字要和dts文件中定义的pin bank node的名字一样。

D：系统中有可能有多个pin controller，多个pin controller上的pin ID 应该是系统唯一的，ctrl->base表示本pin controller中的pin ID的起始值。

（3）本来pin control subsystem和GPIO subsystem应该是无关的两个子系统，应该各自进行自己的初始化过程。但实际中，由于硬件的复杂性，这两个子系统耦合性非常高。这里samsung_gpiolib_register函数就是把各个bank代表的gpio chip注册到GPIO subsystem中。更具体的信息请参考GPIO subsystem软件框架文档。

（4）samsung_pinctrl_register函数的主要功能是将本pin controller注册到pin control subsystem。代码如下：

```
static int samsung_pinctrl_register(struct platform_device *pdev,
                struct samsung_pinctrl_drv_data *drvdata)
{
    struct pinctrl_desc *ctrldesc = &drvdata->pctl;
    struct pinctrl_pin_desc *pindesc, *pdesc;
    struct samsung_pin_bank *pin_bank;
    char *pin_names;
    int pin, bank, ret;

    ctrldesc->name = "samsung-pinctrl"; - - - - - - - - - A
    ctrldesc->owner = THIS_MODULE;
    ctrldesc->pctlops = &samsung_pctrl_ops; - - - call 函数，具体参考第四章的内容
    ctrldesc->pmxops = &samsung_pinmux_ops;
    ctrldesc->confops = &samsung_pinconf_ops;

    pindesc = devm_kzalloc(&pdev->dev, sizeof(*pindesc) * - - - - - - - B
          drvdata->ctrl->nr_pins, GFP_KERNEL);
    ctrldesc->pins = pindesc;
    ctrldesc->npins = drvdata->ctrl->nr_pins;
    for (pin = 0, pdesc = pindesc; pin < ctrldesc->npins; pin++, pdesc++) - - - C
        pdesc->number = pin + drvdata->ctrl->base;


    pin_names = devm_kzalloc(&pdev->dev, sizeof(char) * PIN_NAME_LENGTH * - - - B
                drvdata->ctrl->nr_pins, GFP_KERNEL);
```

```
    for (bank = 0; bank < drvdata->ctrl->nr_banks; bank++) { - - - - - - - - - C
        pin_bank = &drvdata->ctrl->pin_banks[bank];
        for (pin = 0; pin < pin_bank->nr_pins; pin++) {
            sprintf(pin_names, "%s-%d", pin_bank->name, pin);
            pdesc = pindesc + pin_bank->pin_base + pin;
            pdesc->name = pin_names;
            pin_names += PIN_NAME_LENGTH;
        }
    }

    ret = samsung_pinctrl_parse_dt(pdev, drvdata); - - - - - - D

    drvdata->pctl_dev = pinctrl_register(ctrldesc, &pdev->dev, drvdata); - - - E

    for (bank = 0; bank < drvdata->ctrl->nr_banks; ++bank) { - - - - - F
        pin_bank = &drvdata->ctrl->pin_banks[bank];
        pin_bank->grange.name = pin_bank->name;
        pin_bank->grange.id = bank;
        pin_bank->grange.pin_base = pin_bank->pin_base;
        pin_bank->grange.base = pin_bank->gpio_chip.base;
        pin_bank->grange.npins = pin_bank->gpio_chip.ngpio;
        pin_bank->grange.gc = &pin_bank->gpio_chip;
        pinctrl_add_gpio_range(drvdata->pctl_dev, &pin_bank->grange);
    }

    return 0;
}
```

A：初始化硬件无关的pin controller描述符（struct samsung_pinctrl_drv_data中的pctl成员）。该数据结构中还包含了所有pin的描述符的信息，这些pin descriptor所需要的内存在步骤B中分配

B：初始化过程中涉及不少内存分配，这些内存主要用于描述每一个pin（术语叫做pin descriptor）以及pin name。

C：初始化每一个pin 描述符的名字和ID。对于samsung的pin描述符，其名字使用pin-bank name + pin ID的形式。 ID的分配是从该pin controller的pin base开始分配ID的，逐个加一。

D：初始化pin group和function（具体内容在下节描述）

E：调用pinctrl_register注册到pin control subsystem 。这是pin control subsystem的核心函数，可以参考GPIO系统之2的描述。

F：在这里又不得不进行pin control subsystem和GPIO系统的耦合了。每个bank都是一个GPIO controller，但是pin bank使用的ID是pin control space中的ID，GPIO 子系统中使用的是GPIO space的ID，对于pin control subsystem而言，它需要建立这两个ID的映射关系。pinctrl_add_gpio_range就是起这个作用的。更具体的内容请参考pin control subsystem软件结构文档。 需要注意的是直接在pin controller driver中调用pinctrl_add_gpio_range是不推荐的，建议使用dts的方式在GPIO controller设备节点中描述。

（5）这里的代码是向kernel中的中断子系统注册interrupt controller。对于2416，有两个bank有中断功能，gpf和gpg，本质上gpf和gpg就是两个interrupt controller，挂接在2416真正的那个interrupt contrller之下，形成树状结构。具体的代码就不分析了，请参考GPIO类型的中断控制器代码分析。

4、pin control subsystem如何获取pin group的信息

具体的代码如下：

```
static int samsung_pinctrl_parse_dt(struct platform_device *pdev,
                    struct samsung_pinctrl_drv_data *drvdata)
{
    struct device *dev = &pdev->dev;
    struct device_node *dev_np = dev->of_node;
    struct device_node *cfg_np;
    struct samsung_pin_group *groups, *grp;
    struct samsung_pmx_func *functions, *func;
    unsigned *pin_list;
    unsigned int npins, grp_cnt, func_idx = 0;
    char *gname, *fname;
    int ret;

    grp_cnt = of_get_child_count(dev_np); - - - - - - （1）
```

```
    groups = devm_kzalloc(dev, grp_cnt * sizeof(*groups), GFP_KERNEL); - - - - （2）
    grp = groups;

    functions = devm_kzalloc(dev, grp_cnt * sizeof(*functions), GFP_KERNEL); - - - （2）
    func = functions;


    for_each_child_of_node(dev_np, cfg_np) { - - - - 遍历pin control的所有的child node
        u32 function;
      if (!of_find_property(cfg_np, "samsung,pins", NULL)) - 忽略掉那些没有samsung,pins属性的node
            continue;

        ret = samsung_pinctrl_parse_dt_pins(pdev, cfg_np, - - - - - - - - - （3）
                &drvdata->pctl,    &pin_list, &npins);
        if (ret)
            return ret;

        /* derive pin group name from the node name */
    gname = devm_kzalloc(dev, strlen(cfg_np->name) + GSUFFIX_LEN, - 分配pin group名字需要的内
存
                GFP_KERNEL);

    sprintf(gname, "%s%s", cfg_np->name, GROUP_SUFFIX); - - 添加“-grp”的后缀

        grp->name = gname; - - - - - - - - - - - - - - - - - - （4）
        grp->pins = pin_list;
        grp->num_pins = npins;
        of_property_read_u32(cfg_np, "samsung,pin-function", &function);
        grp->func = function;
        grp++;

        if (!of_find_property(cfg_np, "samsung,pin-function", NULL))
            continue; - - - - 忽略掉那些没有samsung,pin-function属性的node

        /* derive function name from the node name */
        fname = devm_kzalloc(dev, strlen(cfg_np->name) + FSUFFIX_LEN,
                GFP_KERNEL);
        sprintf(fname, "%s%s", cfg_np->name, FUNCTION_SUFFIX); - - - - - （5）

        func->name = fname;
        func->groups = devm_kzalloc(dev, sizeof(char *), GFP_KERNEL); - - - - （6）
        if (!func->groups) {
            dev_err(dev, "failed to alloc memory for group list "
                    "in pin function");
            return -ENOMEM;
        }
        func->groups[0] = gname;
        func->num_groups = 1;
        func++;
        func_idx++;
    }

    drvdata->pin_groups = groups; - - - - 最终，pin group和function的信息被copy到pin controller
                                          driver的私有数据结构struct samsung_pinctrl_drv_data 中
    drvdata->nr_groups = grp_cnt;
    drvdata->pmx_functions = functions;
    drvdata->nr_functions = func_idx;

    return 0;
}
```

（1）pin controller的device node有若干个child node，每个child node都描述了一个pin configuration。of_get_child_count函数可以获取pin configuration的数目。

（2）根据pin configuration的数目分配内存。在这里共计分配了两片内存，一片保存了所有pin group的信息（struct samsung_pin_group ），一片保存了pin mux function的信息（struct samsung_pmx_func）。实际上，分配pin configuration的数目的内存有些浪费，因为不是每一个pin control的child node都是和pin group相关（例如pin bank node就是和pin group无关）。对于function，就更浪费了，因为有可能多个pin group对应一个function。

（3）samsung_pinctrl_parse_dt_pins函数主要分析samsung,pins这个属性，并根据属性值返回一个pin list，该list中每个entry是一个pin ID。

（4）初始化samsung pin group的描述符。具体的数据结构解释如下：

```
struct samsung_pin_group {
    const char      *name; - - - - - - - - - pin group的名字，名字是device tree node name + -grp
    const unsigned int    *pins; - - - - - - - pin list的信息
    u8          num_pins; - - - - - - - - - - pin list中的数目
    u8          func; - - - - - - - - - - - - 对应samsung,pin-function属性的值，用来配置pin list中各
个pin的功能设定寄存器
};
```

（5）一个pin configuration的device tree node被解析成两个描述符，一个是samsung pin group的描述符，另外一个是samsung pin mux function描述符。这两个描述符的名字都是根据dts file中的pin configuration的device node name生成，只不过pin group的名字附加-grp的后缀，而function描述符的名字后面附加-mux的后缀。

（6）对于samsung pin mux function描述符解释如下：

```
struct samsung_pmx_func {
    const char      *name; - - - - - - pin function的名字，名字是device tree node name + -mux

    const char      **groups; - - - - - 指向pin groups的指针数组
    u8          num_groups; - - - - - - 属于该function的pin group的个数
};
```

在具体的代码实现中num_groups总是等于1。

四、S3C2416 pin controller driver的操作函数

1、操作函数概述

pin controller描述符中包括了三类操作函数：pctlops是一些全局的控制函数，pmxops是复用引脚相关的操作函数，confops操作函数是用来配置引脚的特性（例如：pull-up/down）。这些callback函数都是和具体的底层pin controller的操作相关。

本章节主要描述这些call back函数的逻辑，这些callback的调用时机不会在这里描述，那些内容请参考pin control subsystem的描述。

2、struct pinctrl_ops中各个callback函数的具体的解释如下：

（1）samsung_get_group_count

该函数的代码如下：

```
static int samsung_get_group_count(struct pinctrl_dev *pctldev)
{
    struct samsung_pinctrl_drv_data *drvdata;

    drvdata = pinctrl_dev_get_drvdata(pctldev);
    return drvdata->nr_groups;
}
```

该函数主要是用来获取指定pin control device的pin group的数目。逻辑很简单，通过pin control的class device的driver_data成员可以获得samsung pin control driver的私有数据（struct samsung_pinctrl_drv_data），可以nr_groups成员返回group的数目。

（2）samsung_get_group_name

该函数的代码如下：

```
static const char *samsung_get_group_name(struct pinctrl_dev *pctldev,
                      unsigned selector)
{
    struct samsung_pinctrl_drv_data *drvdata;

    drvdata = pinctrl_dev_get_drvdata(pctldev);
    return drvdata->pin_groups[selector].name;
}
```

该函数主要用来获取指定group selector的pin group信息。

（3）samsung_get_group_pins

该函数的代码如下：

```
static int samsung_get_group_pins(struct pinctrl_dev *pctldev,
        unsigned selector, const unsigned **pins, unsigned *num_pins)
{
    struct samsung_pinctrl_drv_data *drvdata;

    drvdata = pinctrl_dev_get_drvdata(pctldev);
    *pins = drvdata->pin_groups[selector].pins;
    *num_pins = drvdata->pin_groups[selector].num_pins;
    return 0;
}
```

该函数的主要功能是给定一个group selector（index），获取该pin group中pin的信息（该pin group包括多少个pin，每个pin的ID是什么）。

（4）samsung_dt_node_to_map

该函数的代码如下：

```
static int samsung_dt_node_to_map(struct pinctrl_dev *pctldev,
            struct device_node *np, struct pinctrl_map **maps,
            unsigned *nmaps)
{
    struct device *dev = pctldev->dev;
    struct pinctrl_map *map;
    unsigned long *cfg = NULL;
    char *gname, *fname;
    int cfg_cnt = 0, map_cnt = 0, idx = 0;

    /* count the number of config options specfied in the node */
    for (idx = 0; idx < ARRAY_SIZE(pcfgs); idx++) {
        if (of_find_property(np, pcfgs[idx].prop_cfg, NULL))
            cfg_cnt++;
    }

    /*
     * Find out the number of map entries to create. All the config options
     * can be accomadated into a single config map entry.
     */
    if (cfg_cnt)
        map_cnt = 1;
    if (of_find_property(np, "samsung,pin-function", NULL))
        map_cnt++;
    if (!map_cnt) {
        dev_err(dev, "node %s does not have either config or function "
                "configurations\n", np->name);
        return -EINVAL;
    }

    /* Allocate memory for pin-map entries */
    map = kzalloc(sizeof(*map) * map_cnt, GFP_KERNEL);
    if (!map) {
        dev_err(dev, "could not alloc memory for pin-maps\n");
        return -ENOMEM;
    }
    *nmaps = 0;

    /*
     * Allocate memory for pin group name. The pin group name is derived
     * from the node name from which these map entries are be created.
     */
    gname = kzalloc(strlen(np->name) + GSUFFIX_LEN, GFP_KERNEL);
    if (!gname) {
```

```
        dev_err(dev, "failed to alloc memory for group name\n");
        goto free_map;
    }
    sprintf(gname, "%s%s", np->name, GROUP_SUFFIX);

    /*
     * don't have config options? then skip over to creating function
     * map entries.
     */
    if (!cfg_cnt)
        goto skip_cfgs;

    /* Allocate memory for config entries */
    cfg = kzalloc(sizeof(*cfg) * cfg_cnt, GFP_KERNEL);
    if (!cfg) {
        dev_err(dev, "failed to alloc memory for configs\n");
        goto free_gname;
    }

    /* Prepare a list of config settings */
    for (idx = 0, cfg_cnt = 0; idx < ARRAY_SIZE(pcfgs); idx++) {
        u32 value;
        if (!of_property_read_u32(np, pcfgs[idx].prop_cfg, &value))
            cfg[cfg_cnt++] =
                PINCFG_PACK(pcfgs[idx].cfg_type, value);
    }

    /* create the config map entry */
    map[*nmaps].data.configs.group_or_pin = gname;
    map[*nmaps].data.configs.configs = cfg;
    map[*nmaps].data.configs.num_configs = cfg_cnt;
    map[*nmaps].type = PIN_MAP_TYPE_CONFIGS_GROUP;
    *nmaps += 1;

skip_cfgs:
    /* create the function map entry */
    if (of_find_property(np, "samsung,pin-function", NULL)) {
        fname = kzalloc(strlen(np->name) + FSUFFIX_LEN,    GFP_KERNEL);
        if (!fname) {
            dev_err(dev, "failed to alloc memory for func name\n");
            goto free_cfg;
        }
        sprintf(fname, "%s%s", np->name, FUNCTION_SUFFIX);

        map[*nmaps].data.mux.group = gname;
        map[*nmaps].data.mux.function = fname;
        map[*nmaps].type = PIN_MAP_TYPE_MUX_GROUP;
        *nmaps += 1;
    }

    *maps = map;
    return 0;

free_cfg:
    kfree(cfg);
free_gname:
    kfree(gname);
free_map:
    kfree(map);
    return -ENOMEM;
}
```

具体分析TODO

（5）samsung_dt_free_map

该函数的代码如下：

```
static void samsung_dt_free_map(struct pinctrl_dev *pctldev,
                struct pinctrl_map *map, unsigned num_maps)
{
    int idx;
```

```
    for (idx = 0; idx < num_maps; idx++) {
        if (map[idx].type == PIN_MAP_TYPE_MUX_GROUP) {
            kfree(map[idx].data.mux.function);
            if (!idx)
                kfree(map[idx].data.mux.group);
        } else if (map->type == PIN_MAP_TYPE_CONFIGS_GROUP) {
            kfree(map[idx].data.configs.configs);
            if (!idx)
                kfree(map[idx].data.configs.group_or_pin);
        }
    };

    kfree(map);
}
```

具体分析TODO

3、复用引脚相关的操作函数struct pinmux_ops的具体解释如下：

（1）samsung_get_functions_count

该函数的代码如下：

```
static int samsung_get_functions_count(struct pinctrl_dev *pctldev)
{
    struct samsung_pinctrl_drv_data *drvdata;

    drvdata = pinctrl_dev_get_drvdata(pctldev);
    return drvdata->nr_functions;
}
```

该函数的主要功能是就是返回pin controller device支持的function的数目

（2）samsung_pinmux_get_fname

该函数的代码如下：

```
static const char *samsung_pinmux_get_fname(struct pinctrl_dev *pctldev,
                        unsigned selector)
{
    struct samsung_pinctrl_drv_data *drvdata;

    drvdata = pinctrl_dev_get_drvdata(pctldev);
    return drvdata->pmx_functions[selector].name;
}
```

该函数的主要功能是就是：给定一个function selector（index），获取指定function的name

（3）samsung_pinmux_get_groups

该函数的代码如下：

```
static int samsung_pinmux_get_groups(struct pinctrl_dev *pctldev,
        unsigned selector, const char * const **groups,
        unsigned * const num_groups)
{
    struct samsung_pinctrl_drv_data *drvdata;

    drvdata = pinctrl_dev_get_drvdata(pctldev);
    *groups = drvdata->pmx_functions[selector].groups;
    *num_groups = drvdata->pmx_functions[selector].num_groups;
    return 0;
}
```

该函数的主要功能是就是：给定一个function selector（index），获取指定function的pin groups信息

（4）samsung_pinmux_enable和samsung_pinmux_disable

这个两个callback函数都是通过samsung_pinmux_setup实现，该函数的代码如下：

```
static void samsung_pinmux_setup(struct pinctrl_dev *pctldev, unsigned selector,
                 unsigned group, bool enable)
{
    struct samsung_pinctrl_drv_data *drvdata;
    const unsigned int *pins;
    struct samsung_pin_bank *bank;
    void __iomem *reg;
    u32 mask, shift, data, pin_offset, cnt;
    unsigned long flags;

    drvdata = pinctrl_dev_get_drvdata(pctldev);
    pins = drvdata->pin_groups[group].pins;

    /*
     * for each pin in the pin group selected, program the correspoding pin
     * pin function number in the config register.
     */
    for (cnt = 0; cnt < drvdata->pin_groups[group].num_pins; cnt++) {
        struct samsung_pin_bank_type *type;

        pin_to_reg_bank(drvdata, pins[cnt] - drvdata->ctrl->base,
                ®, &pin_offset, &bank);
        type = bank->type;
        mask = (1 << type->fld_width[PINCFG_TYPE_FUNC]) - 1;
        shift = pin_offset * type->fld_width[PINCFG_TYPE_FUNC];
        if (shift >= 32) {
            /* Some banks have two config registers */
            shift -= 32;
            reg += 4;
        }

        spin_lock_irqsave(&bank->slock, flags);

        data = readl(reg + type->reg_offset[PINCFG_TYPE_FUNC]);
        data &= ~(mask << shift);
        if (enable)
            data |= drvdata->pin_groups[group].func << shift;
        writel(data, reg + type->reg_offset[PINCFG_TYPE_FUNC]);

        spin_unlock_irqrestore(&bank->slock, flags);
    }
}
```

该函数主要用来enable一个指定function。具体指定function的时候要给出function selector和pin group的selector 。具体的操作涉及很多底层的寄存器操作（TODO）。

（5）samsung_pinmux_gpio_set_direction

该函数的代码如下：

```
static int samsung_pinmux_gpio_set_direction(struct pinctrl_dev *pctldev,
        struct pinctrl_gpio_range *range, unsigned offset, bool input)
{
    struct samsung_pin_bank_type *type;
    struct samsung_pin_bank *bank;
    struct samsung_pinctrl_drv_data *drvdata;
    void __iomem *reg;
    u32 data, pin_offset, mask, shift;
    unsigned long flags;

    bank = gc_to_pin_bank(range->gc);
    type = bank->type;
    drvdata = pinctrl_dev_get_drvdata(pctldev);
```

```
    pin_offset = offset - bank->pin_base;
    reg = drvdata->virt_base + bank->pctl_offset +
                type->reg_offset[PINCFG_TYPE_FUNC];

    mask = (1 << type->fld_width[PINCFG_TYPE_FUNC]) - 1;
    shift = pin_offset * type->fld_width[PINCFG_TYPE_FUNC];
    if (shift >= 32) {
        /* Some banks have two config registers */
        shift -= 32;
        reg += 4;
    }

    spin_lock_irqsave(&bank->slock, flags);

    data = readl(reg);
    data &= ~(mask << shift);
    if (!input)
        data |= FUNC_OUTPUT << shift;
    writel(data, reg);

    spin_unlock_irqrestore(&bank->slock, flags);

    return 0;
}
```

该函数用来设定GPIO的方向。

4、配置引脚的特性的struct pinconf_ops数据结构的各个成员定义如下：

（1）samsung_pinconf_get
（2）samsung_pinconf_set
（3）samsung_pinconf_group_get
（4）samsung_pinconf_group_set

（1）和（2）是对单个pin的配置进行读取或者设定，（3）和（4）是对pin group中的所有pin进行配置进行读取或者设定。这些函数的底层都是samsung_pinconf_rw，该函数代码如下：

```
static int samsung_pinconf_rw(struct pinctrl_dev *pctldev, unsigned int pin,
                unsigned long *config, bool set)
{
    struct samsung_pinctrl_drv_data *drvdata;
    struct samsung_pin_bank_type *type;
    struct samsung_pin_bank *bank;
    void __iomem *reg_base;
    enum pincfg_type cfg_type = PINCFG_UNPACK_TYPE(*config);
    u32 data, width, pin_offset, mask, shift;
    u32 cfg_value, cfg_reg;
    unsigned long flags;

    drvdata = pinctrl_dev_get_drvdata(pctldev);
    pin_to_reg_bank(drvdata, pin - drvdata->ctrl->base, &reg_base,
                &pin_offset, &bank);
    type = bank->type;

    if (cfg_type >= PINCFG_TYPE_NUM || !type->fld_width[cfg_type])
        return -EINVAL;

    width = type->fld_width[cfg_type];
    cfg_reg = type->reg_offset[cfg_type];

    spin_lock_irqsave(&bank->slock, flags);

    mask = (1 << width) - 1;
    shift = pin_offset * width;
    data = readl(reg_base + cfg_reg);

    if (set) {
        cfg_value = PINCFG_UNPACK_VALUE(*config);
        data &= ~(mask << shift);
        data |= (cfg_value << shift);
        writel(data, reg_base + cfg_reg);
    } else {
```

```
            data >>= shift;
            data &= mask;
            *config = PINCFG_PACK(cfg_type, data);
        }

        spin_unlock_irqrestore(&bank->slock, flags);

        return 0;
    }
```

具体分析TODO

*原创文章，转发请注明出处。蜗窝科技。*http://www.wowotech.net/linux_kenrel/pin-controller-driver.html

标签: driver pin controller

« linux内核中的GPIO系统之（2）： pin control subsystem | linux内核中的GPIO系统之（1）： 软件框架»

评论:

emeralddream
2016-04-06 15:38

有一点不是很明白，比如你在DTS中把某个 PIN组 配置了内部上拉电阻屏蔽。那么屏蔽这个动作在什么时候有效？是在系统解析设备树后就直接有效或者pinctrl控制器注册之后。还是说在驱动中使用了特定的api获取了设备树资源才开始有效。

还有如下把一个PIN既单独组成了一个引脚组，在设备节点中又用了2种方法去引用它，这里会有什么问题吗？

```
      gpio2_bq20745{
          bq20745_d:bq20745_d{
              rockchip,pins =<GPIO2_B1> ;
              rockchip,pull = <VALUE_PULL_DISABLE>;
          };
      };
2.rk3128-fireprime.dts中加入：
&bq20745 {
    status = "okay";
    compatible = "duwa,bq20745";
    irq_charge_plugin = <&gpio0 GPIO_D6 IRQ_TYPE_EDGE_BOTH>;
    irq_gpio_off = <& gpio2 GPIO_B1 IRQ_TYPE_EDGE_FALLING>;
    irq_gpio_charge_level = <&gpio3 GPIO_C1 IRQ_TYPE_EDGE_FALLING>;

    pinctrl-names = "default";
    pinctrl-0 = <&bq20745_d>;
};
```

回复

wowo
2016-04-06 16:03

@emeralddream：第一个问题，是在driver probe的时候：

```
driver_probe_device
      really_probe
            pinctrl_bind_pins
                  pinctrl_select_state
                        ...
```

第二个问题，不是很明白您的意思。

回复

emeralddream
2016-04-06 16:26

@wowo：pinctrl控制器驱动的probe?

第二个问题针对gpio2_b1,在设备节点中有

```
irq_gpio_off = <&gpio2 GPIO_B1 IRQ_TYPE_EDGE_FALLING>;

pinctrl-names = "default";
    pinctrl-0 = <&bq20745_d>;
```

这2个都是对该pin的引用，这样做是不是重复了？

回复

wowo
2016-04-06 19:21

@emeralddream：是各个device driver的probe。
如果有重复，就是后设置的那个生效。

回复

electrlife
2015-07-30 07:22

很好的文章，linux新手，热情很高，感觉这里正是我一直要找的地方！

想提几点建议，对于我样的新手来说：
１，能否在描述每个子系统时，先从应用的角度对其进行概括性的描述
２，能否对子系统中的各个接口进行简单分类的描述，比如：从芯片厂商的角度来讲述子模块，从其它设备驱动角度描述，从用户空间角度描述，因为对于我这样的新手经常迷失在各种struct中，不知其实际的使用场景，所以理解起来有些难度，比如pinctrl pinctrl_dev两个结构，

另，关于gpio子系统中关于pinctrl - single这块有时间博主帮忙解答下，先谢了！

回复

linuxer
2015-07-30 09:21

@electrlife：人的精力有限，我们很难维护那么多的内核子系统，GPIO subsystem被抛弃了，不过还是多谢你的建议，在后续写文档的时候，我们会考虑的

回复

wxia2010
2015-06-30 15:50

ID从0开始，依次加一。根据例子中的定义，state ID等于0（名字是active）的state对应pinctrl-0属性，state ID等于1（名字是idle）的state对应pinctrl-1属性。

和定义似乎有冲突

```
pinctrl-names = "sleep", "active"; - - - - - - （1）
    pinctrl-0 = < pin-config-0-a>; - - - - - - - - - - - - - - - （2）
    pinctrl-1 = < pin-config-1-a pin-config-1-b>;
  };
```

回复

蜗蜗
2015-06-30 21:51

@wxia2010：多谢提醒，这里应该是笔误，我们会尽快改正。谢谢~~~

回复

perr
2014-09-27 22:25

看2和3看的起鸡皮疙瘩...果断是好用难懂的东西.继总线抽象模型后的又一个大坎.

回复

欧丽源
2014-08-18 10:11

学习了 很不错啊 嘿嘿

回复

linuxer
2014-08-18 12:20

@欧丽源：多谢鼓励，其实这个专题的文档写的不是很好。只是GPIO以及pin control属于边缘地带，我不是那么的关注在这个主题上，如果有机会还是要回来再修正一些内容的

回复

常山黄豆
2014-10-24 14:03

@linuxer：我的android4.4版本的kernel里面samsung_pinctrl_parse_dt函数与你文章不一样，看到这里我没法再看下去了，是我的kernel太旧了吗？另外
4、GPIO subsystem
5、GPIO controller driver
这两篇文章还没有发布吧？什么时候发布？谢谢！

回复

linuxer
2014-10-24 19:00

@常山黄豆：我这里的代码分析都是基于3.14内核的，如果不是的话，可能是对不上的。

另外，说来惭愧，因为工作比较忙，实际上我所有的业余时间都用在写文档了，但是，最近对中断子系统和内核同步更感兴趣，写起来比较有快感，因此，GPIO这部分就荒废了，恐怕最近都不会发布这两份文档了，抱歉，抱歉！

回复

Daniel Shieh
2015-11-25 13:34

@linuxer：linuxer大牛，我觉得这个子系统还是比较重要的，麻雀虽小，五脏俱全，等您不忙别的子系统后，是不是回来把这个系列完善一下呢？我觉得这是深入进内核一个相对比较容易的切入点，别的子系统都要复杂一些。
非常感谢你们的劳动成果，受益良多，嘻嘻。

回复

linuxer
2015-11-26 10:25

@Daniel Shieh：多谢你的建议！我应该会在某个时间点再回来的。

回复

常山黄豆
2014-10-24 14:16

@linuxer：楼主的kernel是3.14版本的吧，android4.4的kernel是3.8版本的。

回复

foiron
2014-07-23 10:25

ok，先阅读3！学习了：）

回复

linuxer
2014-07-23 10:35

@foiron：一共五篇：
1、框架
2、pin control subsystem
3、pin controller driver
4、GPIO subsystem
5、GPIO controller driver

其实对于新的内核的机制，我也是边看代码边理解，大家一起学习吧

回复

Daniel Shieh
2015-11-25 09:33

@linuxer：这个系列感觉干货很多，麻雀虽小，五脏俱全，这样的切入点更容易让我们读者理解，建议linuxer有时间了，把这个系列补全吧，期待您的文章，每一系列文章就像一部作品一样。非常感谢有您这样的大牛给我们指路，带我们深入kernel。

回复

foiron
2014-07-23 10:01

hi linuxer 是不是标题要修改一下，没有发表gpio系统2，直接先发表系统之3吗

回复

linuxer
2014-07-23 10:24

@foiron：gpio系统2还在我的计算机里面呢，完成了50%，我是2和3一起写，3先写完成了，2是描述pin control subsystem的软件框架，还没有完成

回复

发表评论：

昵称

邮件地址 (选填)

个人主页 (选填)

发表评论