# Linux驱动\_pinctrl子系统

## 1.pinctrl子系统初始化

### 1.1 dts树中关于pinctrl的定义

pinctrl@56000000 {

compatible = "samsung,s3c2440-pinctrl";

};

pinctrl\_0: pinctrl@56000000 {

reg = <0x56000000 0x1000>;

wakeup-interrupt-controller {

compatible = "samsung,s3c2410-wakeup-eint";

interrupts = <0 0 0 3>,

<0 0 1 3>,

<0 0 2 3>,

<0 0 3 3>,

<0 0 4 4>,

<0 0 5 4>;

};

};

&pinctrl\_0 {

/\*

\* Pin banks

\*/

gpa: gpa {

gpio-controller;

#gpio-cells = <2>;

};

gpb: gpb {

gpio-controller;

#gpio-cells = <2>;

};

gpc: gpc {

gpio-controller;

#gpio-cells = <2>;

};

gpd: gpd {

gpio-controller;

#gpio-cells = <2>;

};

gpe: gpe {

gpio-controller;

#gpio-cells = <2>;

};

gpf: gpf {

gpio-controller;

#gpio-cells = <2>;

interrupt-controller;

#interrupt-cells = <2>;

};

gpg: gpg {

gpio-controller;

#gpio-cells = <2>;

interrupt-controller;

#interrupt-cells = <2>;

};

gph: gph {

gpio-controller;

#gpio-cells = <2>;

};

gpj: gpj {

gpio-controller;

#gpio-cells = <2>;

};

/\*

\* Pin groups

\*/

uart0\_data: uart0-data {

samsung,pins = "gph-0", "gph-1";

samsung,pin-function = <2>;

};

i2c0\_bus: i2c0-bus {

samsung,pins = "gpe-14", "gpe-15";

samsung,pin-function = <EXYNOS\_PIN\_FUNC\_2>;

};

nand\_pinctrl: nand\_pinctrl {

samsung,pins = "gpa-17", "gpa-18", "gpa-19",

"gpa-20", "gpa-22";

samsung,pin-function = <1>;

};

};

### 1.2 2440 pinctrl子系统初始化

samsung,s3c2440-pinctrl的注册代码如下：

static const struct of\_device\_id samsung\_pinctrl\_dt\_match**[]** **=** **{**

**{** **.**compatible **=** "samsung,s3c2440-pinctrl"**,**

**.**data **=** s3c2440\_pin\_ctrl **},**

**};**

static struct platform\_driver samsung\_pinctrl\_driver **=** **{**

**.**probe **=** samsung\_pinctrl\_probe**,**

**.**driver **=** **{**

**.**name **=** "samsung-pinctrl"**,**

**.**of\_match\_table **=** samsung\_pinctrl\_dt\_match**,**

**.**suppress\_bind\_attrs **=** **true,**

**},**

**};**

static int \_\_init samsung\_pinctrl\_drv\_register**(**void**)**

**{**

register\_syscore\_ops**(&**samsung\_pinctrl\_syscore\_ops**);**

**return** platform\_driver\_register**(&**samsung\_pinctrl\_driver**);**

**}**

匹配后samsung\_pinctrl\_probe函数调用。

static int samsung\_pinctrl\_probe**(**struct platform\_device **\***pdev**)**

**{**

struct samsung\_pinctrl\_drv\_data **\***drvdata**;**

const struct samsung\_pin\_ctrl **\***ctrl**;**

struct device **\***dev **=** **&**pdev**->**dev**;**

struct resource **\***res**;**

int ret**;**

//分配samsung\_pinctrl\_drv\_data结构体空间

drvdata **=** devm\_kzalloc**(**dev**,** **sizeof(\***drvdata**),** GFP\_KERNEL**);**

//初始化samsung\_pinctrl\_drv\_data

ctrl **=** samsung\_pinctrl\_get\_soc\_data**(**drvdata**,** pdev**);**

//drvdata关联dev

drvdata**->**dev **=** dev**;**

//pinctrl操作对应的虚拟地址

res **=** platform\_get\_resource**(**pdev**,** IORESOURCE\_MEM**,** 0**);**samsung\_pinctrl\_parse\_dt

drvdata**->**virt\_base **=** devm\_ioremap\_resource**(&**pdev**->**dev**,** res**);**

//获取pinctrl对应的irq号，这里没有

res **=** platform\_get\_resource**(**pdev**,** IORESOURCE\_IRQ**,** 0**);**

**if** **(**res**)**

drvdata**->**irq **=** res**->**start**;**

//注册到gpio子系统中

ret **=** samsung\_gpiolib\_register**(**pdev**,** drvdata**);**

//注册到pinctrl子系统中

ret **=** samsung\_pinctrl\_register**(**pdev**,** drvdata**);**

//gpio相关中断注册到中断子系统中

**if** **(**ctrl**->**eint\_gpio\_init**)**

ctrl**->**eint\_gpio\_init**(**drvdata**);**

**if** **(**ctrl**->**eint\_wkup\_init**)**

ctrl**->**eint\_wkup\_init**(**drvdata**);**

//设置device的私有数据

platform\_set\_drvdata**(**pdev**,** drvdata**);**

/\* Add to the global list \*/

list\_add\_tail**(&**drvdata**->**node**,** **&**drvdata\_list**);**

**return** 0**;**

**}**

probe函数对pinctrl、中断、gpio子系统都进行了操作。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。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。

#### 1.2.1 samsung\_pinctrl\_drv\_data

samsung\_pinctrl\_drv\_data结构体定义如下：

struct samsung\_pinctrl\_drv\_data **{**

//多个pin controller的描述符可以形成链表

struct list\_head node**;**

//访问硬件寄存器的基地址

void \_\_iomem **\***virt\_base**;**

//和platform device建立联系

struct device **\***dev**;**

//irq number,2440不需要irq资源

int irq**;**

//指向pin control subsystem中core driver中抽象的pin controller描述符。

struct pinctrl\_desc pctl**;**

//指向core driver的pin controller class device

struct pinctrl\_dev **\***pctl\_dev**;**

//描述samsung pin controller中pin groups的信息

const struct samsung\_pin\_group **\***pin\_groups**;**

//描述samsung pin controller中pin groups的数目

unsigned int nr\_groups**;**

//描述samsung pin controller中function信息

const struct samsung\_pmx\_func **\***pmx\_functions**;**

//描述samsung pin controller中function的数目

unsigned int nr\_functions**;**

//每个bank的描述

struct samsung\_pin\_bank **\***pin\_banks**;**

//一共有多少个bank

u32 nr\_banks**;**

//pin脚初始值

unsigned int pin\_base**;**

//一共有多少个pin

unsigned int nr\_pins**;**

void **(\***suspend**)(**struct samsung\_pinctrl\_drv\_data **\*);**

void **(\***resume**)(**struct samsung\_pinctrl\_drv\_data **\*);**

**};**

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。

#### 1.2.2 samsung\_pinctrl\_get\_soc\_data

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。

static const 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**;**

struct device\_node **\***np**;**

const struct samsung\_pin\_bank\_data **\***bdata**;**

const struct samsung\_pin\_ctrl **\***ctrl**;**

struct samsung\_pin\_bank **\***bank**;**

int i**;**

//获取device node对应的pinctrl alias id号，这里为0

id **=** of\_alias\_get\_id**(**node**,** "pinctrl"**);**

//match为如下数据

//{ .compatible = "samsung,s3c2440-pinctrl",

// .data = s3c2440\_pin\_ctrl }

match **=** of\_match\_node**(**samsung\_pinctrl\_dt\_match**,** node**);**

//获取对应的samsung\_pin\_ctrl s3c2440\_pin\_ctrl数组中的第一组数据

//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。

ctrl **=** **(**struct samsung\_pin\_ctrl **\*)**match**->**data **+** id**;**

//samsung\_pinctrl\_drv\_data设置对应的变量和函数

d**->**suspend **=** ctrl**->**suspend**;**

d**->**resume **=** ctrl**->**resume**;**

d**->**nr\_banks **=** ctrl**->**nr\_banks**;**

d**->**pin\_banks **=** devm\_kcalloc**(&**pdev**->**dev**,** d**->**nr\_banks**,sizeof(\***d**->**pin\_banks**),** GFP\_KERNEL**);**

//设置pin\_banks信息

bank **=** d**->**pin\_banks**;**

bdata **=** ctrl**->**pin\_banks**;**

//为每个bank设置基础数据

**for** **(**i **=** 0**;** i **<** ctrl**->**nr\_banks**;** **++**i**,** **++**bdata**,** **++**bank**)** **{**

//type按寄存器操作的bit数分类，

//例如BANK A只需一个bit完成pin功能的设置,其他的需要两个bit

bank**->**type **=** bdata**->**type**;**

//寄存器偏移

bank**->**pctl\_offset **=** bdata**->**pctl\_offset**;**

//每个BANK有多少个pin脚

bank**->**nr\_pins **=** bdata**->**nr\_pins**;**

//gpf gpg具备外部中断能力

bank**->**eint\_func **=** bdata**->**eint\_func**;**

//中断类型，这里为EINT\_TYPE\_WKUP

bank**->**eint\_type **=** bdata**->**eint\_type**;**

//那些gpio具备中断能力

bank**->**eint\_mask **=** bdata**->**eint\_mask**;**

//从第几个gpio具有中断能力

bank**->**eint\_offset **=** bdata**->**eint\_offset**;**

//gpio bank的名字,例如GPA

bank**->**name **=** bdata**->**name**;**

spin\_lock\_init**(&**bank**->**slock**);**

//根据bank能找到对应的samsung\_pinctrl\_drv\_data

bank**->**drvdata **=** d**;**

//设置bank的gpio数量偏移

bank**->**pin\_base **=** d**->**nr\_pins**;**

//更新整个2440具有的gpio数量

d**->**nr\_pins **+=** bank**->**nr\_pins**;**

**}**

//遍历device node下的gpio-controller子节点

for\_each\_child\_of\_node**(**node**,** np**)** **{**

//判断子节点是否是gpio-controller

**if** **(!**of\_find\_property**(**np**,** "gpio-controller"**,** **NULL))**

**continue;**

//轮询所有的samsung\_pin\_bank,名字相同的bank和gpio-controller做一个关联

bank **=** d**->**pin\_banks**;**

**for** **(**i **=** 0**;** i **<** d**->**nr\_banks**;** **++**i**,** **++**bank**)** **{**

**if** **(!**strcmp**(**bank**->**name**,** np**->**name**))** **{**

bank**->**of\_node **=** np**;**

**break;**

**}**

**}**

**}**

//更新全局变量pin\_base

d**->**pin\_base **=** pin\_base**;**

pin\_base **+=** d**->**nr\_pins**;**

//返回s3c2440\_pin\_ctrl

**return** ctrl**;**

**}**

struct samsung\_pinctrl\_drv\_data和硬件相关数据来自s3c2440\_pin\_ctrl：

struct samsung\_pin\_ctrl **{**

//定义具体的pin bank信息,例如操作寄存器/bank名称/有多少pin等

const struct samsung\_pin\_bank\_data **\***pin\_banks**;**

//number of pin bank

u32 nr\_banks**;**

//和中断相关的代码

int **(\***eint\_gpio\_init**)(**struct samsung\_pinctrl\_drv\_data **\*);**

int **(\***eint\_wkup\_init**)(**struct samsung\_pinctrl\_drv\_data **\*);**

void **(\***suspend**)(**struct samsung\_pinctrl\_drv\_data **\*);**

void **(\***resume**)(**struct samsung\_pinctrl\_drv\_data **\*);**

**};**

static const struct samsung\_pin\_bank\_data s3c2440\_pin\_banks**[]** \_\_initconst **=** **{**

PIN\_BANK\_A**(**25**,** 0x000**,** "gpa"**),**

PIN\_BANK\_2BIT**(**11**,** 0x010**,** "gpb"**),**

PIN\_BANK\_2BIT**(**16**,** 0x020**,** "gpc"**),**

PIN\_BANK\_2BIT**(**16**,** 0x030**,** "gpd"**),**

PIN\_BANK\_2BIT**(**16**,** 0x040**,** "gpe"**),**

PIN\_BANK\_2BIT\_EINTW**(**8**,** 0x050**,** "gpf"**,** 0**,** 0xff**),**

PIN\_BANK\_2BIT\_EINTW**(**16**,** 0x060**,** "gpg"**,** 8**,** 0xffff00**),**

PIN\_BANK\_2BIT**(**11**,** 0x070**,** "gph"**),**

PIN\_BANK\_2BIT**(**13**,** 0x0d0**,** "gpj"**),**

**};**

const struct samsung\_pin\_ctrl s3c2440\_pin\_ctrl**[]** \_\_initconst **=** **{**

**{**

**.**pin\_banks **=** s3c2440\_pin\_banks**,**

**.**nr\_banks **=** ARRAY\_SIZE**(**s3c2440\_pin\_banks**),**

**.**eint\_wkup\_init **=** s3c24xx\_eint\_init**,**

**},**

**};**

#### 1.2.3 samsung\_gpiolib\_register

samsung\_gpiolib\_register将gpio信息注册到了gpio子系统中。

static const struct gpio\_chip samsung\_gpiolib\_chip **=** **{**

**.**request **=** gpiochip\_generic\_request**,**

**.**free **=** gpiochip\_generic\_free**,**

**.**set **=** samsung\_gpio\_set**,**

**.**get **=** samsung\_gpio\_get**,**

**.**direction\_input **=** samsung\_gpio\_direction\_input**,**

**.**direction\_output **=** samsung\_gpio\_direction\_output**,**

**.**to\_irq **=** samsung\_gpio\_to\_irq**,**

**.**owner **=** THIS\_MODULE**,**

**};**

static int samsung\_gpiolib\_register**(**struct platform\_device **\***pdev**,** struct samsung\_pinctrl\_drv\_data **\***drvdata**)**

**{**

//得到pin\_bank描述结构体

struct samsung\_pin\_bank **\***bank **=** drvdata**->**pin\_banks**;**

//一个gpio\_chip代表一个bank

struct gpio\_chip **\***gc**;**

int ret**;**

int i**;**

**for** **(**i **=** 0**;** i **<** drvdata**->**nr\_banks**;** **++**i**,** **++**bank**)** **{**

//将三星通用的gpio chip先赋值给gpio\_chip

bank**->**gpio\_chip **=** samsung\_gpiolib\_chip**;**

//修改每个bank不同的属性

gc **=** **&**bank**->**gpio\_chip**;**

//每个bank对应的gpio起始值

gc**->**base **=** drvdata**->**pin\_base **+** bank**->**pin\_base**;**

//每个bank有多少个gpio

gc**->**ngpio **=** bank**->**nr\_pins**;**

//该bank的parent device

gc**->**parent **=** **&**pdev**->**dev**;**

//该bank对应的device node

gc**->**of\_node **=** bank**->**of\_node**;**

//该bank的名字

gc**->**label **=** bank**->**name**;**

//将该bank添加到gpio子系统中

ret **=** gpiochip\_add\_data**(**gc**,** bank**);**

**}**

**return** 0**;**

**}**

因为4.x的内核采用了设备树，所以gpio的申请采用设备树方式，下面将详细分析基于设备树的gpio子系统的使用方法。

##### 1.2.3.1 gpiochip\_add\_data

将gpio\_chip添加到gpio子系统的核心函数为：

int gpiochip\_add\_data**(**struct gpio\_chip **\***chip**,** void **\***data**)**

**{**

unsigned long flags**;**

int status **=** 0**;**

unsigned i**;**

//获取该bank的gpio起始序号

int base **=** chip**->**base**;**

//分配一个gpio\_device结构体

//gpio\_device是gpio子系统一个bank的标准定义

struct gpio\_device **\***gdev**;**

gdev **=** kzalloc**(sizeof(\***gdev**),** GFP\_KERNEL**);**

//gpio\_device对应gpio bus

gdev**->**dev**.**bus **=** **&**gpio\_bus\_type**;**

//设置gpio\_device对应的gpio\_chip结构体

gdev**->**chip **=** chip**;**

//设置gpio\_chip对应的gpio\_device结构体

chip**->**gpiodev **=** gdev**;**

//gpio\_chip存在parent device

**if** **(**chip**->**parent**)** **{**

//设置gpio\_device的parent为gpio\_chip的parent

gdev**->**dev**.**parent **=** chip**->**parent**;**

//设置gpio\_device的device node为gpio\_chip parent decvice的device node

//这里为dts samsung,s3c2440-pinctrl节点

//后面还会修改该节点

gdev**->**dev**.**of\_node **=** chip**->**parent**->**of\_node**;**

**}**

//如果支持dts树

#ifdef CONFIG\_OF\_GPIO

//如果gpio\_chip拥有device node

**if** **(**chip**->**of\_node**)**

//设置gpio\_device的device node为dts树中的gpx

gdev**->**dev**.**of\_node **=** chip**->**of\_node**;**

#endif

//得到一个id值

gdev**->**id **=** ida\_simple\_get**(&**gpio\_ida**,** 0**,** 0**,** GFP\_KERNEL**);**

//设置device的名字，创建字符设备的时候需要使用，会生成/dev/gpiochipx

dev\_set\_name**(&**gdev**->**dev**,** "gpiochip%d"**,** gdev**->**id**);**

//初始化gpio\_device对应的device

//创建/sys/bus/gpio/devices/gpiochipx

//创建/sys/devices/platform/samsung-pinctrl/gpiochipx

device\_initialize**(&**gdev**->**dev**);**

//设置device对应的drv\_data为gpio\_device

dev\_set\_drvdata**(&**gdev**->**dev**,** gdev**);**

**if** **(**chip**->**parent **&&** chip**->**parent**->**driver**)**

gdev**->**owner **=** chip**->**parent**->**driver**->**owner**;**

**else** **if** **(**chip**->**owner**)**

gdev**->**owner **=** chip**->**owner**;**

**else**

gdev**->**owner **=** THIS\_MODULE**;**

//分配gpio\_desc用于描述该bank下所有gpio的信息，每个gpio对应一个gpio\_desc结构体

gdev**->**descs **=** kcalloc**(**chip**->**ngpio**,** **sizeof(**gdev**->**descs**[**0**]),** GFP\_KERNEL**);**

//设置gpio\_device的名称

**if** **(**chip**->**label**)**

gdev**->**label **=** kstrdup**(**chip**->**label**,** GFP\_KERNEL**);**

//设置gpio\_device的个数

gdev**->**ngpio **=** chip**->**ngpio**;**

//设置gpio\_device的data数据为samsung\_pin\_bank

gdev**->**data **=** data**;**

spin\_lock\_irqsave**(&**gpio\_lock**,** flags**);**

//设置gpio\_device的gpio起始序号

gdev**->**base **=** base**;**

//将gpio\_device添加到gpio\_devices链表中

//保持链表的排序方式为[base, base + ngpio - 1]

status **=** gpiodev\_add\_to\_list**(**gdev**);**

spin\_unlock\_irqrestore**(&**gpio\_lock**,** flags**);**

//遍历gpio\_desc

**for** **(**i **=** 0**;** i **<** chip**->**ngpio**;** i**++)** **{**

struct gpio\_desc **\***desc **=** **&**gdev**->**descs**[**i**];**

//设置gpio\_desc对应的gpio\_device

desc**->**gdev **=** gdev**;**

//如果get\_direction为输出或者没有direction\_input函数

//认为该gpio只有输出功能，设置标志位FLAG\_IS\_OUT

**if** **(**chip**->**get\_direction**)** **{**

int dir **=** chip**->**get\_direction**(**chip**,** i**);**

**if** **(!**dir**)**

set\_bit**(**FLAG\_IS\_OUT**,** **&**desc**->**flags**);**

**}** **else** **if** **(!**chip**->**direction\_input**)** **{**

set\_bit**(**FLAG\_IS\_OUT**,** **&**desc**->**flags**);**

**}**

**}**

//如果支持pinctrl子系统，初始化pin\_ranges链表

//pin\_ranges是干什么用的？

#ifdef CONFIG\_PINCTRL

INIT\_LIST\_HEAD**(&**gdev**->**pin\_ranges**);**

#endif

//这个函数这里没有作用，直接返回了

status **=** gpiochip\_set\_desc\_names**(**chip**);**

//这个函数这里没有作用，直接返回了

status **=** gpiochip\_irqchip\_init\_valid\_mask**(**chip**);**

//分析dts gpio相关的属性

status **=** of\_gpiochip\_add**(**chip**);**

//为每个bank创建字符设备，gpio\_device对应的id作为次设备号

**if** **(**gpiolib\_initialized**)** **{**

status **=** gpiochip\_setup\_dev**(**gdev**);**

**}**

**return** 0**;**

**}**

GPIO子系统为每个bank分配了一个gpio\_device结构体用于描述该bank，gpio\_device中有gpio\_desc描述每个gpio pin。

struct gpio\_device **{**

//gpio\_device的id，由系统分配

int id**;**

//对应的device

struct device dev**;**

//对应的字符设备结构体

struct cdev chrdev**;**

//防止重复生成/sys/class/gpio/gpiochipx

struct device **\***mockdev**;**

//模块所有者

struct module **\***owner**;**

//对应的gpio\_chip

struct gpio\_chip **\***chip**;**

//每个gpio pin对应的结构体

struct gpio\_desc **\***descs**;**

//起始pin number

int base**;**

//该bank有多少个gpio

u16 ngpio**;**

//bank名字

char **\***label**;**

//这里对应为samsung\_pin\_bank

void **\***data**;**

struct list\_head list**;**

#ifdef CONFIG\_PINCTRL

struct list\_head pin\_ranges**;**

#endif

**};**

##### 1.2.3.2 of\_gpiochip\_add

of\_gpiochip\_add用于分析dts树中和gpio相关节点的信息，方便其他节点调用gpio。

int of\_gpiochip\_add**(**struct gpio\_chip **\***chip**)**

**{**

int status**;**

//每个gpio\_chip有对应的device node

//该步骤跳过

**if** **((!**chip**->**of\_node**)** **&&** **(**chip**->**parent**))**

chip**->**of\_node **=** chip**->**parent**->**of\_node**;**

//如果没有提供of\_xlate，则设置默认的of\_xlate函数

**if** **(!**chip**->**of\_xlate**)** **{**

chip**->**of\_gpio\_n\_cells **=** 2**;**

chip**->**of\_xlate **=** of\_gpio\_simple\_xlate**;**

**}**

//分析gpio-ranges-group-names gpio-ranges等dts信息，这里没有定义

status **=** of\_gpiochip\_add\_pin\_range**(**chip**);**

//分析gpio-line-names dts信息，这里没有定义

**if** **(!**chip**->**names**)**

of\_gpiochip\_set\_names**(**chip**);**

//device node计数增加

of\_node\_get**(**chip**->**of\_node**);**

//分析gpio-hog dts信息，这里没有定义

**return** of\_gpiochip\_scan\_gpios**(**chip**);**

**}**

##### 1.2.3.3 gpiochip\_setup\_dev

gpiochip\_setup\_dev为每个bank创建一个字符设备和相关sys调试文件。

static int gpiochip\_setup\_dev**(**struct gpio\_device **\***gdev**)**

**{**

int status**;**

//创建字符设备

//设置字符设备操作函数集

cdev\_init**(&**gdev**->**chrdev**,** **&**gpio\_fileops**);**

//设置字符设备拥有者

gdev**->**chrdev**.**owner **=** THIS\_MODULE**;**

//设置字符设备对应的parent device

gdev**->**chrdev**.**kobj**.**parent **=** **&**gdev**->**dev**.**kobj**;**

//设置字符设备主次设备号

gdev**->**dev**.**devt **=** MKDEV**(**MAJOR**(**gpio\_devt**),** gdev**->**id**);**

status **=** cdev\_add**(&**gdev**->**chrdev**,** gdev**->**dev**.**devt**,** 1**);**

//添加device到内核中

// /sys/platform/samsung-pinctrl/gpiochipx/dev

// /sys/bus/gpio/devices/gpiochipx/dev

status **=** device\_add**(&**gdev**->**dev**);**

//创建/sys/class/gpio/gpiochipx及相关调试文件

status **=** gpiochip\_sysfs\_register**(**gdev**);**

//当销毁device时调用的函数

gdev**->**dev**.**release **=** gpiodevice\_release**;**

**return** 0**;**

**}**

int gpiochip\_sysfs\_register**(**struct gpio\_device **\***gdev**)**

**{**

struct device **\***dev**;**

struct device **\***parent**;**

struct gpio\_chip **\***chip **=** gdev**->**chip**;**

//判断gpio\_class是否存在

**if** **(!**gpio\_class**.**p**)**

**return** 0**;**

//这里parent = chip->parent

**if** **(**chip**->**parent**)**

parent **=** chip**->**parent**;**

**else**

parent **=** **&**gdev**->**dev**;**

//创建/sys/class/gpio/gpiochipx

//该目录下有base label ngpio调试文件(gpiochip\_groups)

dev **=** device\_create\_with\_groups**(&**gpio\_class**,** parent**,**

MKDEV**(**0**,** 0**),**

chip**,** gpiochip\_groups**,**

"gpiochip%d"**,** chip**->**base**);**

mutex\_lock**(&**sysfs\_lock**);**

gdev**->**mockdev **=** dev**;**

mutex\_unlock**(&**sysfs\_lock**);**

**return** 0**;**

**}**

##### 1.2.3.4 设备树方式使用GPIO

以模拟i2c为例子，说明dts树下使用gpio的方法。

###### 1.2.3.4.1 从设备树中获取gpio对应的pin number

dts树中关于gpio的描述：

i2c\_gpio\_1**:** i2c**-**gpio**-**1 **{**

compatible **=** "i2c-gpio"**;**

#address-cells = <1>;

#size-cells = <0>;

gpios **=** **<&**gpe 15 GPIO\_ACTIVE\_HIGH**>,** // SDA

**<&**gpe 14 GPIO\_ACTIVE\_HIGH**>;** // SCL

i2c**-**gpio**,**delay**-**us **=** **<**5**>;** // 100KHz

status **=** "okay"**;**

**};**

驱动中获取对应gpio的方法：

static int i2c\_gpio\_probe**(**struct platform\_device **\***pdev**)**

**{**

of\_i2c\_gpio\_get\_pins**(**pdev**->**dev**.**of\_node**,&**sda\_pin**,** **&**scl\_pin**);**

devm\_gpio\_request**(&**pdev**->**dev**,** sda\_pin**,** "sda"**);**

devm\_gpio\_request**(&**pdev**->**dev**,** scl\_pin**,** "scl"**);**

**}**

首先需要从dts树中获取gpio相关信息，获取函数为of\_i2c\_gpio\_get\_pins。

static int of\_i2c\_gpio\_get\_pins**(**struct device\_node **\***np**,** unsigned int **\***sda\_pin**,** unsigned int **\***scl\_pin**)**

**{**

//统计节点中gpios属性，判断有几个gpio节点

//因为是i2c，所以至少需要两个gpio

**if** **(**of\_gpio\_count**(**np**)** **<** 2**)**

**return** **-**ENODEV**;**

**\***sda\_pin **=** of\_get\_gpio**(**np**,** 0**);**

**\***scl\_pin **=** of\_get\_gpio**(**np**,** 1**);**

**return** 0**;**

**}**

从上面代码可以看出通过of\_get\_gpio从dts树中的gpio信息获取对应的gpio number。

static inline int of\_get\_gpio**(**struct device\_node **\***np**,** int index**)**

**{**

**return** of\_get\_gpio\_flags**(**np**,** index**,** **NULL);**

**}**

static inline int of\_get\_gpio\_flags**(**struct device\_node **\***np**,** int index**,** enum of\_gpio\_flags **\***flags**)**

**{**

**return** of\_get\_named\_gpio\_flags**(**np**,** "gpios"**,** index**,** flags**);**

**}**

int of\_get\_named\_gpio\_flags**(**struct device\_node **\***np**,** const char **\***list\_name**,** int index**,** enum of\_gpio\_flags **\***flags**)**

**{**

struct gpio\_desc **\***desc**;**

//根据<&gpe 15 GPIO\_ACTIVE\_HIGH>或<&gpe 14 GPIO\_ACTIVE\_HIGH>信息

//获取对应的gpio\_desc结构体

desc **=** of\_get\_named\_gpiod\_flags**(**np**,** list\_name**,** index**,** flags**);**

//获得对应的的gpio number

**return** desc\_to\_gpio**(**desc**);**

**}**

of\_get\_named\_gpiod\_flags函数代码分析如下：

struct gpio\_desc **\***of\_get\_named\_gpiod\_flags**(**struct device\_node **\***np**,** const char **\***propname**,** int index**,** enum of\_gpio\_flags **\***flags**)**

**{**

struct of\_phandle\_args gpiospec**;**

struct gpio\_chip **\***chip**;**

struct gpio\_desc **\***desc**;**

int ret**;**

//分析gpios的选项

//gpiospec保存了解析的相关参数

//device\_node为gpe, args\_count为2, args为15 GPIO\_ACTIVE\_HIGH 或者14 GPIO\_ACTIVE\_HIGH

ret **=** of\_parse\_phandle\_with\_args**(**np**,** propname**,** "#gpio-cells"**,** index**,** **&**gpiospec**);**

//遍历gpio\_devices链表,比对gpio\_devices和gpiospec中的device node

//如果相同，则找到了需要的gpio\_devices，则返回对应的gpio\_chip结构体

chip **=** of\_find\_gpiochip\_by\_xlate**(&**gpiospec**);**

//根据gpio\_chip信息和pin number信息，得到对应的gpio\_desc结构体

desc **=** of\_xlate\_and\_get\_gpiod\_flags**(**chip**,** **&**gpiospec**,** flags**);**

**return** desc**;**

**}**

static struct gpio\_chip **\***of\_find\_gpiochip\_by\_xlate**(**struct of\_phandle\_args **\***gpiospec**)**

**{**

**return** gpiochip\_find**(**gpiospec**,** of\_gpiochip\_match\_node\_and\_xlate**);**

**}**

struct gpio\_chip **\***gpiochip\_find**(**void **\***data**,**int **(\***match**)(**struct gpio\_chip **\***chip**,** void **\***data**))**

**{**

struct gpio\_device **\***gdev**;**

struct gpio\_chip **\***chip **=** **NULL;**

unsigned long flags**;**

spin\_lock\_irqsave**(&**gpio\_lock**,** flags**);**

//轮询gpio\_devices链表,获取gpio\_device结构体

//并调用match函数，找到对应的gpio\_chip结构体

list\_for\_each\_entry**(**gdev**,** **&**gpio\_devices**,** list**)**

**if** **(**gdev**->**chip **&&** match**(**gdev**->**chip**,** data**))** **{**

chip **=** gdev**->**chip**;**

**break;**

**}**

spin\_unlock\_irqrestore**(&**gpio\_lock**,** flags**);**

**return** chip**;**

**}**

static int of\_gpiochip\_match\_node\_and\_xlate**(**struct gpio\_chip **\***chip**,** void **\***data**)**

**{**

struct of\_phandle\_args **\***gpiospec **=** data**;**

//两者的device node相同，并且参数有数值

**return** chip**->**gpiodev**->**dev**.**of\_node **==** gpiospec**->**np **&&**

chip**->**of\_xlate**(**chip**,** gpiospec**,** **NULL)** **>=** 0**;**

**}**

在添加到gpio子系统的时候添加了of\_xlate的对应函数：

//args[0]存储的是pin number

//args[1]存储的是GPIO\_ACTIVE\_HIGH等信息。

int of\_gpio\_simple\_xlate**(**struct gpio\_chip **\***gc**,** const struct of\_phandle\_args **\***gpiospec**,** u32 **\***flags**)**

**{**

**if** **(**gc**->**of\_gpio\_n\_cells **<** 2**)** **{**

WARN\_ON**(**1**);**

**return** **-**EINVAL**;**

**}**

**if** **(**WARN\_ON**(**gpiospec**->**args\_count **<** gc**->**of\_gpio\_n\_cells**))**

**return** **-**EINVAL**;**

**if** **(**gpiospec**->**args**[**0**]** **>=** gc**->**ngpio**)**

**return** **-**EINVAL**;**

**if** **(**flags**)**

**\***flags **=** gpiospec**->**args**[**1**];**

**return** gpiospec**->**args**[**0**];**

**}**

###### 1.2.3.4.2 申请gpio

现在已经拿到了gpio对应的pin number了，那么下面就是通过devm\_gpio\_request申请gpio了。

devm\_gpio\_request**(&**pdev**->**dev**,** sda\_pin**,** "sda"**);**

devm\_gpio\_request**(&**pdev**->**dev**,** scl\_pin**,** "scl"**);**

那么gpio\_request完成了什么功能呢？

int gpio\_request**(**unsigned gpio**,** const char **\***label**)**

**{**

//通过gpio number获取对应的gpio\_desc结构体

struct gpio\_desc **\***desc **=** gpio\_to\_desc**(**gpio**);**

**return** gpiod\_request**(**desc**,** label**);**

**}**

内核已经将gpio\_request函数放到gpiolib-legacy中了，是否表示已经过时了呢？

继续分析\_\_gpiod\_request。

static int \_\_gpiod\_request**(**struct gpio\_desc **\***desc**,** const char **\***label**)**

**{**

struct gpio\_chip **\***chip **=** desc**->**gdev**->**chip**;**

int status**;**

unsigned long flags**;**

spin\_lock\_irqsave**(&**gpio\_lock**,** flags**);**

//检查FLAG\_REQUESTED标志，看是否被申请了

**if** **(**test\_and\_set\_bit**(**FLAG\_REQUESTED**,** **&**desc**->**flags**)** **==** 0**)** **{**

//如果没有被申请，设置gpio\_desc的label属性，这里为scl或sda

desc\_set\_label**(**desc**,** label **?** **:** "?"**);**

status **=** 0**;**

**}**

//检查gpio\_chip是否有request函数

//这里会调用gpiochip\_generic\_request

**if** **(**chip**->**request**)** **{**

spin\_unlock\_irqrestore**(&**gpio\_lock**,** flags**);**

status **=** chip**->**request**(**chip**,** gpio\_chip\_hwgpio**(**desc**));**

spin\_lock\_irqsave**(&**gpio\_lock**,** flags**);**

**}**

//调用get\_direction,这里未定义

**if** **(**chip**->**get\_direction**)** **{**

spin\_unlock\_irqrestore**(&**gpio\_lock**,** flags**);**

gpiod\_get\_direction**(**desc**);**

spin\_lock\_irqsave**(&**gpio\_lock**,** flags**);**

**}**

**}**

由于gpio\_chip定义了request回调函数gpiochip\_generic\_request，进一步分析：

int gpiochip\_generic\_request**(**struct gpio\_chip **\***chip**,** unsigned offset**)**

**{**

//申请一个pin作为gpio

**return** pinctrl\_request\_gpio**(**chip**->**gpiodev**->**base **+** offset**);**

**}**

到这里是否发现pinctrl已经和gpio有关联了。

虽然pinctrl提供了pinctrl\_request\_gpio()这样的API，但在代码中不可以直接调用pinctrl\_request\_gpio，当设备驱动申请一个gpio时，仍然需要调用gpio\_request()，这里会调用pinctrl\_request\_gpio()，调用过程如下：

gpio\_request**()**

gpiod\_request**()**

chip**->**request**(**chip**,**gpio\_chip\_hwgpio**(**desc**));**

同样地，对于pinctrl\_free\_gpio()、pinctrl\_gpio\_direction\_input()和pinctrl\_gpio\_direction\_output()也有类似说明。因此在clientdevice驱动中，申请和释放gpio仍然要调gpio\_request()、gpio\_free()；设置gpio为input/output仍然要调gpio\_direction\_input()和gpio\_direction\_output()。向pinctrl请求pin用作GPIO，成功后，该Pin仅能被GPIO驱动所使用，不能被其他驱动所混用。

由于pinctrl子系统代码还未分析，这里不详细分析pinctrl\_request\_gpio代码，该函数会根据gpio的pin number找到对应的pin control device和gpio range，并标记该pin已经用作gpio了，如果后面有模块request该资源，则会失败。

###### 1.2.3.4.3 gpio的操作

(1) direction

static void i2c\_gpio\_setsda\_dir**(**void **\***data**,** int state**)**

**{**

struct i2c\_gpio\_platform\_data **\***pdata **=** data**;**

**if** **(**state**)**

gpio\_direction\_input**(**pdata**->**sda\_pin**);**

**else**

gpio\_direction\_output**(**pdata**->**sda\_pin**,** 0**);**

**}**

static void i2c\_gpio\_setscl\_dir**(**void **\***data**,** int state**)**

**{**

struct i2c\_gpio\_platform\_data **\***pdata **=** data**;**

**if** **(**state**)**

gpio\_direction\_input**(**pdata**->**scl\_pin**);**

**else**

gpio\_direction\_output**(**pdata**->**scl\_pin**,** 0**);**

**}**

gpio\_direction\_input和gpio\_direction\_output代码比较简单，调用gpio注册的回调函数samsung\_gpio\_direction\_input和samsung\_gpio\_direction\_output完成输入输出的设置。

(2)value

static void i2c\_gpio\_setsda\_val**(**void **\***data**,** int state**)**

**{**

struct i2c\_gpio\_platform\_data **\***pdata **=** data**;**

gpio\_set\_value**(**pdata**->**sda\_pin**,** state**);**

**}**

static void i2c\_gpio\_setscl\_val**(**void **\***data**,** int state**)**

**{**

struct i2c\_gpio\_platform\_data **\***pdata **=** data**;**

gpio\_set\_value**(**pdata**->**scl\_pin**,** state**);**

**}**

static int i2c\_gpio\_getsda**(**void **\***data**)**

**{**

struct i2c\_gpio\_platform\_data **\***pdata **=** data**;**

**return** gpio\_get\_value**(**pdata**->**sda\_pin**);**

**}**

static int i2c\_gpio\_getscl**(**void **\***data**)**

**{**

struct i2c\_gpio\_platform\_data **\***pdata **=** data**;**

**return** gpio\_get\_value**(**pdata**->**scl\_pin**);**

**}**

gpio\_set\_value和gpio\_get\_value也是调用gpio注册时的回调函数samsung\_gpio\_set和samsung\_gpio\_get完成相关寄存器的设置。

(4)irq

gpio\_to\_irq最终也会调用samsung\_gpio\_to\_irq得到对应的irq number，该函数和中断相关，后面会进行分析。

#### 1.2.4 samsung\_pinctrl\_register

samsung\_pinctrl\_register将2440的pin信息注册到pinctrl子系统。

static int samsung\_pinctrl\_register**(**struct platform\_device **\***pdev**,** struct samsung\_pinctrl\_drv\_data **\***drvdata**)**

**{**

//获取pinctrl\_desc结构体对应的指针

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**;**

//初始化pinctrl\_desc

ctrldesc**->**name **=** "samsung-pinctrl"**;**

ctrldesc**->**owner **=** THIS\_MODULE**;**

ctrldesc**->**pctlops **=** **&**samsung\_pctrl\_ops**;**

ctrldesc**->**pmxops **=** **&**samsung\_pinmux\_ops**;**

ctrldesc**->**confops **=** **&**samsung\_pinconf\_ops**;**

//为每个pin创建pinctrl\_pin\_desc结构体

pindesc **=** devm\_kzalloc**(&**pdev**->**dev**,** **sizeof(\***pindesc**)** **\*** drvdata**->**nr\_pins**,** GFP\_KERNEL**);**

//再次设置pinctrl\_desc

ctrldesc**->**pins **=** pindesc**;**

ctrldesc**->**npins **=** drvdata**->**nr\_pins**;**

//设置每个pinctrl\_pin\_desc对应的pin number

**for** **(**pin **=** 0**,** pdesc **=** pindesc**;** pin **<** ctrldesc**->**npins**;** pin**++,** pdesc**++)**

pdesc**->**number **=** pin **+** drvdata**->**pin\_base**;**

//分配空间用于存储pinctrl\_pin\_desc对应的name

pin\_names **=** devm\_kzalloc**(&**pdev**->**dev**,** **sizeof(**char**)** **\*** PIN\_NAME\_LENGTH **\*** drvdata**->**nr\_pins**,** GFP\_KERNEL**);**

//为每个pinctrl\_pin\_desc设置name(pin-bank name + pin number)

**for** **(**bank **=** 0**;** bank **<** drvdata**->**nr\_banks**;** bank**++)** **{**

pin\_bank **=** **&**drvdata**->**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**;**

**}**

**}**

//分析device node下的i2c/spi等function信息

ret **=** samsung\_pinctrl\_parse\_dt**(**pdev**,** drvdata**);**

//根据pinctrl\_desc和samsung\_pinctrl\_drv\_data注册到pinctrl子系统中,并返回pinctrl\_dev

drvdata**->**pctl\_dev **=** devm\_pinctrl\_register**(&**pdev**->**dev**,** ctrldesc**,** drvdata**);**

//创建pinctrl\_gpio\_range结构体，并添加到gpio\_ranges链表中

//pinctrl\_gpio\_range记录了每个bank的pin信息

**for** **(**bank **=** 0**;** bank **<** drvdata**->**nr\_banks**;** **++**bank**)** **{**

pin\_bank **=** **&**drvdata**->**pin\_banks**[**bank**];**

pin\_bank**->**grange**.**name **=** pin\_bank**->**name**;**

pin\_bank**->**grange**.**id **=** bank**;**

pin\_bank**->**grange**.**pin\_base **=** drvdata**->**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**;**

**}**

##### 1.2.4.1 pinctrl\_desc

首先注册函数完成了pinctrl\_desc结构体的构建，pinctrl\_desc描述了pin的相关信息：有多少个pin，这些pin的相关操作函数集合，每个pin都有一个pinctrl\_pin\_desc结构体对应。

struct pinctrl\_desc **{**

//这里为samsung-pinctrl

const char **\***name**;**

//每一个pin的信息

const struct pinctrl\_pin\_desc **\***pins**;**

//有多少个pin

unsigned int npins**;**

//全局的控制函数，例如获取pin个数，node\_to\_map

const struct pinctrl\_ops **\***pctlops**;**

//复用引脚相关的操作函数

const struct pinmux\_ops **\***pmxops**;**

//配置引脚的特性（例如：pull-up/down）

const struct pinconf\_ops **\***confops**;**

//拥有者

struct module **\***owner**;**

**};**

struct pinctrl\_pin\_desc **{**

unsigned number**;**

const char **\***name**;**

void **\***drv\_data**;**

**};**

##### 1.2.4.2 samsung\_pinctrl\_parse\_dt

samsung\_pinctrl\_parse\_dt会分析pinctrl dts树中的相关信息，并将解析出来的信息存放在samsung\_pinctrl\_drv\_data中，供后面使用。

static int samsung\_pinctrl\_parse\_dt**(**struct platform\_device **\***pdev**,** struct samsung\_pinctrl\_drv\_data **\***drvdata**)**

**{**

struct device **\***dev **=** **&**pdev**->**dev**;**

//描述samsung pin controller中pin groups的信息

struct samsung\_pin\_group **\***groups**;**

//描述samsung pin controller中function信息

struct samsung\_pmx\_func **\***functions**;**

unsigned int grp\_cnt **=** 0**,** func\_cnt **=** 0**;**

//创建samsung\_pin\_group空间，用于保存group信息，每个group对应一个pin

groups **=** samsung\_pinctrl\_create\_groups**(**dev**,** drvdata**,** **&**grp\_cnt**);**

//创建samsung\_pmx\_func空间，例如一组i2s对应一个samsung\_pmx\_func

functions **=** samsung\_pinctrl\_create\_functions**(**dev**,** drvdata**,** **&**func\_cnt**);**

drvdata**->**pin\_groups **=** groups**;**

drvdata**->**nr\_groups **=** grp\_cnt**;**

drvdata**->**pmx\_functions **=** functions**;**

drvdata**->**nr\_functions **=** func\_cnt**;**

**return** 0**;**

**}**

###### 1.2.4.2.1 samsung\_pinctrl\_create\_groups

samsung\_pin\_group也包含了每个pin的描述，一个group对应一个pin。

struct samsung\_pin\_group **{**

//group的名字

const char **\***name**;**

//pin name，这里只有一个pin

const unsigned int **\***pins**;**

//该group包含多少个pin，这里为1

u8 num\_pins**;**

//这里无作用

u8 func**;**

**};**

samsung\_pinctrl\_create\_groups创建了npins个samsung\_pin\_group结构体，并从pinctrl\_desc获取了pin的相关信息。

static struct samsung\_pin\_group **\***samsung\_pinctrl\_create\_groups**(** struct device **\***dev**,**struct samsung\_pinctrl\_drv\_data **\***drvdata**,**unsigned int **\***cnt**)**

**{**

struct pinctrl\_desc **\***ctrldesc **=** **&**drvdata**->**pctl**;**

struct samsung\_pin\_group **\***groups**,** **\***grp**;**

const struct pinctrl\_pin\_desc **\***pdesc**;**

int i**;**

//分配了npins个samsung\_pin\_group结构体

groups **=** devm\_kzalloc**(**dev**,** ctrldesc**->**npins **\*** **sizeof(\***groups**),** GFP\_KERNEL**);**

grp **=** groups**;**

//pinctrl\_pin\_desc

pdesc **=** ctrldesc**->**pins**;**

//初始化samsung\_pin\_group，每个group对应一个pin

**for** **(**i **=** 0**;** i **<** ctrldesc**->**npins**;** **++**i**,** **++**pdesc**,** **++**grp**)** **{**

grp**->**name **=** pdesc**->**name**;**

grp**->**pins **=** **&**pdesc**->**number**;**

grp**->**num\_pins **=** 1**;**

**}**

**\***cnt **=** ctrldesc**->**npins**;**

**return** groups**;**

**}**

###### 1.2.4.2.2 samsung\_pinctrl\_create\_functions

struct samsung\_pmx\_func **{**

//pin function的名字

const char **\***name**;**

//对应的 pin 的名字

const char **\*\***groups**;**

//属于该function的pin的个数

u8 num\_groups**;**

//该function对应的值，samsung,pin-function属性对应的值

u32 val**;**

**};**

samsung\_pmx\_func保存了如下dts树的信息：

i2c0\_bus**:** i2c0**-**bus **{**

samsung**,**pins **=** "gpe-14"**,** "gpe-15"**;**

samsung**,**pin**-**function **=** **<**EXYNOS\_PIN\_FUNC\_2**>;**

**};**

samsung\_pinctrl\_create\_functions分析dts树中pinctrl节点中的信息，解析i2c,spi,nand等配置信息，存入到samsung\_pmx\_func中。

static struct samsung\_pmx\_func **\***samsung\_pinctrl\_create\_functions**(**struct device **\***dev**,**struct samsung\_pinctrl\_drv\_data **\***drvdata**,**unsigned int **\***cnt**)**

**{**

struct samsung\_pmx\_func **\***functions**,** **\***func**;**

//获取对应的device node

struct device\_node **\***dev\_np **=** dev**->**of\_node**;**

struct device\_node **\***cfg\_np**;**

unsigned int func\_cnt **=** 0**;**

int ret**;**

//遍历device node下的子节点，找对子节点下的samsung,pin-function属性，统计个数

for\_each\_child\_of\_node**(**dev\_np**,** cfg\_np**)** **{**

struct device\_node **\***func\_np**;**

//如果没有子节点，检查samsung,pin-function属性

**if** **(!**of\_get\_child\_count**(**cfg\_np**))** **{**

**if** **(!**of\_find\_property**(**cfg\_np**,**"samsung,pin-function"**,** **NULL))**

**continue;**

**++**func\_cnt**;**

**continue;**

**}**

//如果有子节点，继续遍历子节点下的节点，并检查samsung,pin-function属性

for\_each\_child\_of\_node**(**cfg\_np**,** func\_np**)** **{**

**if** **(!**of\_find\_property**(**func\_np**,**"samsung,pin-function"**,** **NULL))**

**continue;**

**++**func\_cnt**;**

**}**

**}**

//创建func\_cnt个samsung\_pmx\_func结构体

functions **=** devm\_kzalloc**(**dev**,** func\_cnt **\*** **sizeof(\***functions**),**GFP\_KERNEL**);**

func **=** functions**;**

func\_cnt **=** 0**;**

//重新遍历子节点

for\_each\_child\_of\_node**(**dev\_np**,** cfg\_np**)** **{**

struct device\_node **\***func\_np**;**

//如果子节点下没有节点

**if** **(!**of\_get\_child\_count**(**cfg\_np**))** **{**

//初始化一个samsung\_pmx\_func

ret **=** samsung\_pinctrl\_create\_function**(**dev**,** drvdata**,**cfg\_np**,** func**);**

**if** **(**ret **>** 0**)** **{**

**++**func**;**

**++**func\_cnt**;**

**}**

**continue;**

**}**

//如果子节点下有节点，继续遍历

for\_each\_child\_of\_node**(**cfg\_np**,** func\_np**)** **{**

//初始化一个samsung\_pmx\_func

ret **=** samsung\_pinctrl\_create\_function**(**dev**,** drvdata**,**func\_np**,** func**);**

**if** **(**ret **>** 0**)** **{**

**++**func**;**

**++**func\_cnt**;**

**}**

**}**

**}**

//一共有多少个fun

**\***cnt **=** func\_cnt**;**

**return** functions**;**

**}**

samsung\_pinctrl\_create\_function为实际的节点信息解析函数：

static int samsung\_pinctrl\_create\_function**(**struct device **\***dev**,**struct samsung\_pinctrl\_drv\_data **\***drvdata**,**struct device\_node **\***func\_np**,**struct

samsung\_pmx\_func **\***func**)**

**{**

int npins**;**

int ret**;**

int i**;**

//获取samsung,pin-function对应的值，并存储到samsung\_pmx\_func的val中

**if** **(**of\_property\_read\_u32**(**func\_np**,** "samsung,pin-function"**,** **&**func**->**val**))**

**return** 0**;**

//samsung,pins属性下有几个字符串

npins **=** of\_property\_count\_strings**(**func\_np**,** "samsung,pins"**);**

//samsung\_pmx\_func的名字为从，表示该node的full path

func**->**name **=** func\_np**->**full\_name**;**

//创建内存空间存储group名字

func**->**groups **=** devm\_kzalloc**(**dev**,** npins **\*** **sizeof(**char **\*),** GFP\_KERNEL**);**

//从samsung,pins读取group名字，并设置到groups中

**for** **(**i **=** 0**;** i **<** npins**;** **++**i**)** **{**

const char **\***gname**;**

ret **=** of\_property\_read\_string\_index**(**func\_np**,** "samsung,pins"**,** i**,** **&**gname**);**

func**->**groups**[**i**]** **=** gname**;**

**}**

//该function对应哪几个pin

func**->**num\_groups **=** npins**;**

**return** 1**;**

**}**

##### 1.2.4.3 pinctrl\_register

pinctrl\_register将pin信息添加到pinctrl子系统中。

struct pinctrl\_dev **\***pinctrl\_register**(**struct pinctrl\_desc **\***pctldesc**,** struct device **\***dev**,** void **\***driver\_data**)**

**{**

struct pinctrl\_dev **\***pctldev**;**

int ret**;**

//创建pinctrl\_dev结构体

pctldev **=** kzalloc**(sizeof(\***pctldev**),** GFP\_KERNEL**);**

//初始化pinctrl\_dev结构体

pctldev**->**owner **=** pctldesc**->**owner**;**

pctldev**->**desc **=** pctldesc**;**

pctldev**->**driver\_data **=** driver\_data**;**

INIT\_RADIX\_TREE**(&**pctldev**->**pin\_desc\_tree**,** GFP\_KERNEL**);**

INIT\_LIST\_HEAD**(&**pctldev**->**gpio\_ranges**);**

pctldev**->**dev **=** dev**;**

mutex\_init**(&**pctldev**->**mutex**);**

//检查samsung\_pctrl\_ops的回调函数是否设置合理

ret **=** pinctrl\_check\_ops**(**pctldev**);**

//检查samsung\_pinmux\_ops的回调函数是否设置合理

**if** **(**pctldesc**->**pmxops**)** **{**

ret **=** pinmux\_check\_ops**(**pctldev**);**

**}**

//检查samsung\_pinconf\_ops的回调函数是否设置合理

**if** **(**pctldesc**->**confops**)** **{**

ret **=** pinconf\_check\_ops**(**pctldev**);**

**}**

//为每个pin分配pin\_ctrl结构体，并添加到radix树中

ret **=** pinctrl\_register\_pins**(**pctldev**,** pctldesc**->**pins**,** pctldesc**->**npins**);**

//将pinctrl\_dev添加到pinctrldev\_list链表中

mutex\_lock**(&**pinctrldev\_list\_mutex**);**

list\_add\_tail**(&**pctldev**->**node**,** **&**pinctrldev\_list**);**

mutex\_unlock**(&**pinctrldev\_list\_mutex**);**

//根据pctldev->dev创建pinctrl结构体

pctldev**->**p **=** pinctrl\_get**(**pctldev**->**dev**);**

**if** **(!**IS\_ERR**(**pctldev**->**p**))**

**{**

//寻找default配置

pctldev**->**hog\_default **=** pinctrl\_lookup\_state**(**pctldev**->**p**,** PINCTRL\_STATE\_DEFAULT**);**

//设置default配置

pinctrl\_select\_state**(**pctldev**->**p**,** pctldev**->**hog\_default**);**

//寻找sleep配置

pctldev**->**hog\_sleep **=** pinctrl\_lookup\_state**(**pctldev**->**p**,** PINCTRL\_STATE\_SLEEP**);**

**}**

//pinctrl debugfs调试接口

pinctrl\_init\_device\_debugfs**(**pctldev**);**

**return** pctldev**;**

**}**

pinctrl\_register围绕pinctrl\_dev结构体来处理，函数在开始先初始化了pinctrl\_dev的一些成员并检查了一些成员函数的合法性。

###### 1.2.4.3.1 pinctrl\_register\_pins

pinctrl\_register\_pins为每个pin分配了一个pin\_desc结构体，描述该pin对应的复用功能是那个，如果已经被复用了，那么其他驱动便不可再申请这个pin。这些pin\_desc通过radix树进行管理。

struct pin\_desc **{**

//对应的 pinctrl\_dev结构体

struct pinctrl\_dev **\***pctldev**;**

//pin name

const char **\***name**;**

bool dynamic\_name**;**

//samsung\_pinctrl\_drv\_data

void **\***drv\_data**;**

#ifdef CONFIG\_PINMUX

//如果操作该pin的复用功能，mux\_usecount++

unsigned mux\_usecount**;**

const char **\***mux\_owner**;**

//描述pin的做了什么复用

const struct pinctrl\_setting\_mux **\***mux\_setting**;**

const char **\***gpio\_owner**;**

#endif

**};**

pinctrl\_register\_pins函数如下：

static int pinctrl\_register\_pins**(**struct pinctrl\_dev **\***pctldev**,** struct pinctrl\_pin\_desc const **\***pins**,** unsigned num\_descs**)**

**{**

unsigned i**;**

int ret **=** 0**;**

**for** **(**i **=** 0**;** i **<** num\_descs**;** i**++)** **{**

ret **=** pinctrl\_register\_one\_pin**(**pctldev**,** **&**pins**[**i**]);**

**}**

**return** 0**;**

**}**

static int pinctrl\_register\_one\_pin**(**struct pinctrl\_dev **\***pctldev**,** const struct pinctrl\_pin\_desc **\***pin**)**

**{**

struct pin\_desc **\***pindesc**;**

//从pin\_desc\_tree树中寻找pin number对应的pin\_desc结构体

//如果存在则退出

pindesc **=** pin\_desc\_get**(**pctldev**,** pin**->**number**);**

**if** **(**pindesc **!=** **NULL)** **{**

**return** **-**EINVAL**;**

**}**

//创建pin\_desc结构体

pindesc **=** kzalloc**(sizeof(\***pindesc**),** GFP\_KERNEL**);**

//设置pin\_desc对应的pinctrl\_dev

pindesc**->**pctldev **=** pctldev**;**

//根据pinctrl\_pin\_desc的name，设置pin\_desc的name

**if** **(**pin**->**name**)** **{**

pindesc**->**name **=** pin**->**name**;**

**}**

//设置pin\_desc的drv\_data

pindesc**->**drv\_data **=** pin**->**drv\_data**;**

//将pin\_desc插入到radix树中

radix\_tree\_insert**(&**pctldev**->**pin\_desc\_tree**,** pin**->**number**,** pindesc**);**

**return** 0**;**

**}**

###### 1.2.4.3.2 pinctrl\_get

如果pinctrl节点下定义了类型属性：

pinctrl**-**names **=** "default"**;**

pinctrl**-**0 **=** **<&**xxxxx**>;**

则会解析该属性对应的节点的相关信息，但是这里pinctrl节点下并没有定义这些节点，但是还是要分析，后面其他设备会使用到。

pinctrl\_get对应pinctrl结构体，pinctrl对应设备，用于管理该设备的pin信息，如有哪些状态、每个状态有哪些设置。

struct pinctrl **{**

//pinctrl\_list对应的node

struct list\_head node**;**

//设备对应的dev

struct device **\***dev**;**

//该设备pinctrl\_state链表，对应default等状态

struct list\_head states**;**

//当前选中的state

struct pinctrl\_state **\***state**;**

//pinctrl\_dt\_map链表

struct list\_head dt\_maps**;**

struct kref users**;**

**};**

pinctrl\_get函数为：

struct pinctrl **\***pinctrl\_get**(**struct device **\***dev**)**

**{**

struct pinctrl **\***p**;**

//从pinctrl\_list链表中寻找dev对应的pinctrl结构体

//如果找到了对应的pinctrl结构体则返回

p **=** find\_pinctrl**(**dev**);**

**if** **(**p **!=** **NULL)** **{**

kref\_get**(&**p**->**users**);**

**return** p**;**

**}**

//根据dev创建pinctrl结构体

**return** create\_pinctrl**(**dev**);**

**}**

create\_pinctrl为pinctrl\_get的核心函数，下面将对这个函数进行分析。

static struct pinctrl **\***create\_pinctrl**(**struct device **\***dev**)**

**{**

struct pinctrl **\*** **;**

const char **\***devname**;**

struct pinctrl\_maps **\***maps\_node**;**

int i**;**

struct pinctrl\_map const **\***map**;**

int ret**;**

//创建pinctrl结构体

p **=** kzalloc**(sizeof(\***p**),** GFP\_KERNEL**);**

//设置pinctrl对应的dev

p**->**dev **=** dev**;**

//初始化链表

INIT\_LIST\_HEAD**(&**p**->**states**);**

INIT\_LIST\_HEAD**(&**p**->**dt\_maps**);**

//建立该device需要的pinctrl\_map

//pinctrl\_map包含了该device使用了哪些pin，这些pin对应的配置信息

//并生成了pinctrl\_maps结构体

ret **=** pinctrl\_dt\_to\_map**(**p**);**

//这里为pinctrl节点的name

devname **=** dev\_name**(**dev**);**

mutex\_lock**(&**pinctrl\_maps\_mutex**);**

//遍历pinctrl\_maps

for\_each\_maps**(**maps\_node**,** i**,** map**)** **{**

//map必须属于这个device

**if** **(**strcmp**(**map**->**dev\_name**,** devname**))**

**continue;**

//将该device对应的pinctrl\_map转换成pinctrl\_setting并添加到pinctrl\_state链表中

ret **=** add\_setting**(**p**,** map**);**

**}**

mutex\_unlock**(&**pinctrl\_maps\_mutex**);**

kref\_init**(&**p**->**users**);**

//pinctrl添加到pinctrl\_list链表中

mutex\_lock**(&**pinctrl\_list\_mutex**);**

list\_add\_tail**(&**p**->**node**,** **&**pinctrl\_list**);**

mutex\_unlock**(&**pinctrl\_list\_mutex**);**

**return** p**;**

**}**

create\_pinctrl创建了pinctrl结构体，并初始化其中的对象。

**(1)pinctrl\_dt\_to\_map**

pinctrl\_dt\_to\_map解析如下节点信息，并转换成pinctrl\_map结构体。

pinctrl**-**names **=** "default"**;**

pinctrl**-**0 **=** **<&**xxxxx**>;**

pinctrl\_dt\_to\_map函数解析如下：

int pinctrl\_dt\_to\_map**(**struct pinctrl **\***p**)**

**{**

//dts树中pinctrl节点对应的device node

struct device\_node **\***np **=** p**->**dev**->**of\_node**;**

int state**,** ret**;**

char **\***propname**;**

struct property **\***prop**;**

const char **\***statename**;**

const \_\_be32 **\***list**;**

int size**,** config**;**

phandle phandle**;**

struct device\_node **\***np\_config**;**

of\_node\_get**(**np**);**

**for** **(**state **=** 0**;** **;** state**++)** **{**

//获取pinctrl-\*属性

propname **=** kasprintf**(**GFP\_KERNEL**,** "pinctrl-%d"**,** state**);**

//寻找pinctrl节点下的pinctrl-\*属性的节点

prop **=** of\_find\_property**(**np**,** propname**,** **&**size**);**

kfree**(**propname**);**

//获取pinctrl-\*属性对应的pin configuration phandle列表

list **=** prop**->**value**;**

//pinctrl-\*属性对应的pin configuration phandle数目

size **/=** **sizeof(\***list**);**

//获取pinctrl-names对应index的字符内容

//statename为default或者flow-control

of\_property\_read\_string\_index**(**np**,** "pinctrl-names"**,** state**,** **&**statename**);**

**for** **(**config **=** 0**;** config **<** size**;** config**++)** **{**

//获取phandle

phandle **=** be32\_to\_cpup**(**list**++);**

//寻找phandle对应的device node

np\_config **=** of\_find\_node\_by\_phandle**(**phandle**);**

//分析一个pin configuration，并生成pinctrl\_map

ret **=** dt\_to\_map\_one\_config**(**p**,** statename**,** np\_config**);**

of\_node\_put**(**np\_config**);**

**}**

//如果该设备下没有pinctrl-\*属性，创建一个pinctrl\_map

//type为PIN\_MAP\_TYPE\_DUMMY\_STATE

**if** **(!**size**)** **{**

ret **=** dt\_remember\_dummy\_state**(**p**,** statename**);**

**}**

**}**

**return** 0**;**

**}**

**(2) dt\_to\_map\_one\_config**

dt\_to\_map\_one\_config解析一个pinctrl配置的节点属性，例如<&xxxxx>中的属性。

pinctrl**-**names **=** "default"**;**

pinctrl**-**0 **=** **<&**xxxxx**>;**

<&xxxxx>中会定义该配置使用了多少个pin和这些pin的配置，例如：

xxxxxx {

samsung,pins = "gpe-14", "gpe-15";

samsung,pin-function = <2>;

};

dt\_to\_map\_one\_config会将上述信息转换成pinctrl\_map结构体，并组成pinctrl\_maps结构体并添加到pinctrl\_maps链表中。

static int dt\_to\_map\_one\_config**(**struct pinctrl **\***p**,** const char **\***statename**,** struct device\_node **\***np\_config**)**

**{**

struct device\_node **\***np\_pctldev**;**

struct pinctrl\_dev **\***pctldev**;**

const struct pinctrl\_ops **\***ops**;**

int ret**;**

struct pinctrl\_map **\***map**;**

unsigned num\_maps**;**

//device node计数增加

//np\_config为pin configuration对应的device\_node

np\_pctldev **=** of\_node\_get**(**np\_config**);**

**for** **(;;)** **{**

//获取parent device node

np\_pctldev **=** of\_get\_next\_parent**(**np\_pctldev**);**

//如果找不到或者是root node，则进入出错处理。

**if** **(!**np\_pctldev **||** of\_node\_is\_root**(**np\_pctldev**))** **{**

of\_node\_put**(**np\_pctldev**);**

**return** **-**EPROBE\_DEFER**;**

**}**

//找到了一个pinctrl节点对应的device node

//从pinctrldev\_list链表中找到device node相同的pinctrl\_dev结构体

//pinctrl节点对应的pinctrl\_dev

pctldev **=** get\_pinctrl\_dev\_from\_of\_node**(**np\_pctldev**);**

//一旦找到pinctrl\_dev结构体跳出循环

**if** **(**pctldev**)**

**break;**

//假如device node没有对应的pinctrl\_dev结构体

//并且两者的device node相同，则进入出错处理

//意思是如果pinctrl节点下的pin configuration配置在自身的节点下

//如果自身的pinctrl节点都没有加入到pinctrl子系统中

//那么就无法进行解析

**if** **(**np\_pctldev **==** p**->**dev**->**of\_node**)** **{**

of\_node\_put**(**np\_pctldev**);**

**return** **-**ENODEV**;**

**}**

**}**

of\_node\_put**(**np\_pctldev**);**

//pin configuration

// -->parent handle

// -->device node

// -->parent device node

// -->pinctrl\_dev

ops **=** pctldev**->**desc**->**pctlops**;**

//调用samsung\_pctrl\_ops中的dt\_node\_to\_map函数

//samsung\_dt\_node\_to\_map解析出这个配置节点,毕竟格式只有对应的pinctrl driver最清楚

//节点类似如下

/\*

uart0\_data: uart0-data {

samsung,pins = "gph-0", "gph-1";

samsung,pin-function = <2>;

};

\*/

//将配置节点的信息转换成pinctrl\_map,

//每个pin对应pinctrl\_map\_configs或者pinctrl\_map\_mux两种类型pinctrl\_map

ret **=** ops**->**dt\_node\_to\_map**(**pctldev**,** np\_config**,** **&**map**,** **&**num\_maps**);**

//将多个pinctrl\_map组合成pinctrl\_maps结构体，并添加到pinctrl\_maps链表中

**return** dt\_remember\_or\_free\_map**(**p**,** statename**,** pctldev**,** map**,** num\_maps**);**

**}**

**(3)** **dt\_node\_to\_map**

dt\_node\_to\_map回调为samsung\_dt\_node\_to\_map函数，该函数完成了pinctrl\_map的实际转换。

pinctrl\_map结构体定义如下：

struct pinctrl\_map **{**

//该pinctrl\_map对应的device name

const char **\***dev\_name**;**

//pinctrl状态名，类似default

const char **\***name**;**

//pinctrl\_map类别，有复用和配置两种类型

enum pinctrl\_map\_type type**;**

//pinctrl节点的device name

const char **\***ctrl\_dev\_name**;**

//存储mux或config类型

union **{**

struct pinctrl\_map\_mux mux**;**

struct pinctrl\_map\_configs configs**;**

**}** data**;**

**};**

每个pin一般来说会对应两个pinctrl\_map的结构体，不同的是一个代表mux，一个代表config。

static int samsung\_dt\_node\_to\_map**(**struct pinctrl\_dev **\***pctldev**,** struct device\_node **\***np\_config**,** struct pinctrl\_map **\*\***map**,** unsigned **\***num\_maps**)**

**{**

struct samsung\_pinctrl\_drv\_data **\***drvdata**;**

unsigned reserved\_maps**;**

struct device\_node **\***np**;**

int ret**;**

drvdata **=** pinctrl\_dev\_get\_drvdata**(**pctldev**);**

reserved\_maps **=** 0**;**

**\***map **=** **NULL;**

**\***num\_maps **=** 0**;**

//判断配置节点是否有子节点，如果没有则调用samsung\_dt\_subnode\_to\_map

**if** **(!**of\_get\_child\_count**(**np\_config**))**

**return** samsung\_dt\_subnode\_to\_map**(**drvdata**,** pctldev**->**dev**,** np\_config**,** map**,** **&**reserved\_maps**,** num\_maps**);**

for\_each\_child\_of\_node**(**np\_config**,** np**)** **{**

ret **=** samsung\_dt\_subnode\_to\_map**(**drvdata**,** pctldev**->**dev**,** np**,** map**,&**reserved\_maps**,** num\_maps**);**

**}**

**return** 0**;**

**}**

继续分析samsung\_dt\_subnode\_to\_map：

static int samsung\_dt\_subnode\_to\_map**(**struct samsung\_pinctrl\_drv\_data **\***drvdata**,** struct device **\***dev**,** struct device\_node **\***np**,** struct

pinctrl\_map **\*\***map**,** unsigned **\***reserved\_maps**,** unsigned **\***num\_maps**)**

**{**

int ret**,** i**;**

u32 val**;**

unsigned long config**;**

unsigned long **\***configs **=** **NULL;**

unsigned num\_configs **=** 0**;**

unsigned reserve**;**

struct property **\***prop**;**

const char **\***group**;**

bool has\_func **=** **false;**

//读取节点samsung,pin-function属性，值保存在val

ret **=** of\_property\_read\_u32**(**np**,** "samsung,pin-function"**,** **&**val**);**

**if** **(!**ret**)**

has\_func **=** **true;**

//解析samsung,pin-pud，samsung,pin-drv等属性

**for** **(**i **=** 0**;** i **<** ARRAY\_SIZE**(**cfg\_params**);** i**++)** **{**

ret **=** of\_property\_read\_u32**(**np**,** cfg\_params**[**i**].**property**,** **&**val**);**

**if** **(!**ret**)** **{**

//将上/下拉或驱动能力信息组成16bit数据

config **=** PINCFG\_PACK**(**cfg\_params**[**i**].**param**,** val**);**

//将生成的16bit数据添加到configs中

ret **=** add\_config**(**dev**,** **&**configs**,** **&**num\_configs**,** config**);**

**}**

**}**

reserve **=** 0**;**

**if** **(**has\_func**)**

reserve**++;**

**if** **(**num\_configs**)**

reserve**++;**

//读取samsung,pins属性

ret **=** of\_property\_count\_strings**(**np**,** "samsung,pins"**);**

reserve **\*=** ret**;**

//预分配一些空间，用于存储新增的pinctrl\_map

ret **=** reserve\_map**(**dev**,** map**,** reserved\_maps**,** num\_maps**,** reserve**);**

//遍历samsung,pins对应的string

of\_property\_for\_each\_string**(**np**,** "samsung,pins"**,** prop**,** group**)** **{**

//如果有samsung,pin-function属性

//添加pinctrl\_map\_mux类型pinctrl\_map,group类似为"gph-0",np->full\_name为节点路径

**if** **(**has\_func**)** **{**

ret **=** add\_map\_mux**(**map**,** reserved\_maps**,** num\_maps**,** group**,** np**->**full\_name**);**

**}**

//如果有samsung,pin-pud，samsung,pin-drv等属性

//添加pinctrl\_map\_configs类型，group类似为"gph-0",configs为配置信息,num\_configs有几项配置

**if** **(**num\_configs**)** **{**

ret **=** add\_map\_configs**(**dev**,** map**,** reserved\_maps**,** num\_maps**,** group**,** configs**,** num\_configs**);**

**}**

**}**

ret **=** 0**;**

**}**

上面函数才是真正分析如下属性的函数：

xxxxxx {

samsung,pins = "gpe-14", "gpe-15";

samsung,pin-function = <2>;

samsung**,**pin**-**pud **=** **<**EXYNOS\_PIN\_PULL\_UP**>;**

samsung**,**pin**-**drv **=** **<**EXYNOS4\_PIN\_DRV\_LV4**>;**

};

经过samsung\_dt\_subnode\_to\_map函数后，上面示例属性会生成4个pinctrl\_map结构体，分别表示gpe-14/gpe-15的mux信息和config信息。

struct pinctrl\_map\_mux **{**

//pin名字,类似gpe-14,gpe-15

const char **\***group**;**

//该属性节点的dts路径名

//也就四xxxxx1的dts路径名

//方便后续找到samsung,pin-function值

const char **\***function**;**

**};**

struct pinctrl\_map\_configs **{**

//pin名，类似gpe-14,gpe-15

const char **\***group\_or\_pin**;**

//配置属性，上/下拉和驱动能力合在一起

unsigned long **\***configs**;**

//有几个配置

unsigned num\_configs**;**

**};**

**(4) dt\_remember\_or\_free\_map**

dt\_remember\_or\_free\_map将生成的pinctrl\_map结构体添加到pinctrl\_dt\_map结构体中，并将pinctrl\_dt\_map添加到pinctrl的dt\_maps链表中。

每个如下类似节点对应一个pinctrl\_dt\_map结构体。

xxxxxx {

samsung,pins = "gpe-14", "gpe-15";

samsung,pin-function = <2>;

samsung**,**pin**-**pud **=** **<**EXYNOS\_PIN\_PULL\_UP**>;**

samsung**,**pin**-**drv **=** **<**EXYNOS4\_PIN\_DRV\_LV4**>;**

};

pinctrl\_dt\_map结构体定义：

struct pinctrl\_dt\_map **{**

//dt\_maps链表节点

struct list\_head node**;**

//对应的pinctrl\_dev

struct pinctrl\_dev **\***pctldev**;**

//该节点所有的pinctrl\_map数组

struct pinctrl\_map **\***map**;**

//有多少个map

unsigned num\_maps**;**

**};**

dt\_remember\_or\_free\_map函数如下：

static int dt\_remember\_or\_free\_map**(**struct pinctrl **\***p**,** const char **\***statename**,** struct pinctrl\_dev **\***pctldev**,** struct pinctrl\_map **\***map**,**

unsigned num\_maps**)**

**{**

int i**;**

struct pinctrl\_dt\_map **\***dt\_map**;**

//重新设置pinctl\_map的参数

**for** **(**i **=** 0**;** i **<** num\_maps**;** i**++)** **{**

//设置pinctrl\_map对应的device name,device name使用pin的设备名

map**[**i**].**dev\_name **=** dev\_name**(**p**->**dev**);**

//该pin配置对应的状态，例如default等

map**[**i**].**name **=** statename**;**

//pinctrl节点对应的名称

**if** **(**pctldev**)**

map**[**i**].**ctrl\_dev\_name **=** dev\_name**(**pctldev**->**dev**);**

**}**

//创建pinctrl\_dt\_map结构体

dt\_map **=** kzalloc**(sizeof(\***dt\_map**),** GFP\_KERNEL**);**

//dt\_map对应的pinctrl节点对应的pinctrl\_dev结构体

dt\_map**->**pctldev **=** pctldev**;**

//将pinctrl\_map添加到dt\_map中

dt\_map**->**map **=** map**;**

dt\_map**->**num\_maps **=** num\_maps**;**

//将该dt\_map添加到pinctrl节点的dt\_maps链表

list\_add\_tail**(&**dt\_map**->**node**,** **&**p**->**dt\_maps**);**

//将pinctrl\_map数组构造一个pinctrl\_maps结构体，并添加到pinctrl\_maps链表中

**return** pinctrl\_register\_map**(**map**,** num\_maps**,** **false);**

**}**

pinctrl\_register\_map函数分析：

struct pinctrl\_maps **{**

struct list\_head node**;**

struct pinctrl\_map const **\***maps**;**

unsigned num\_maps**;**

**};**

int pinctrl\_register\_map**(**struct pinctrl\_map const **\***maps**,** unsigned num\_maps**,** bool dup**)**

**{**

int i**,** ret**;**

struct pinctrl\_maps **\***maps\_node**;**

//遍历maps

**for** **(**i **=** 0**;** i **<** num\_maps**;** i**++)** **{**

//检查map是否有设备名

**if** **(!**maps**[**i**].**dev\_name**)** **{**

**return** **-**EINVAL**;**

**}**

//检查map是否有statename

**if** **(!**maps**[**i**].**name**)** **{**

**return** **-**EINVAL**;**

**}**

//检查map是否有pinctrl dev name并且type不为dummy

**if** **(**maps**[**i**].**type **!=** PIN\_MAP\_TYPE\_DUMMY\_STATE **&&** **!**maps**[**i**].**ctrl\_dev\_name**)** **{**

**return** **-**EINVAL**;**

**}**

//继续检查不同类型pinctrl\_map的合理性

**switch** **(**maps**[**i**].**type**)** **{**

**case** PIN\_MAP\_TYPE\_MUX\_GROUP**:**

ret **=** pinmux\_validate\_map**(&**maps**[**i**],** i**);**

**break;**

**case** PIN\_MAP\_TYPE\_CONFIGS\_PIN**:**

**case** PIN\_MAP\_TYPE\_CONFIGS\_GROUP**:**

ret **=** pinconf\_validate\_map**(&**maps**[**i**],** i**);**

**break;**

**}**

**}**

//创建pinctrl\_maps结构体

maps\_node **=** kzalloc**(sizeof(\***maps\_node**),** GFP\_KERNEL**);**

//pinctrl\_maps保存pinctrl\_map信息

maps\_node**->**num\_maps **=** num\_maps**;**

**if** **(**dup**)** **{**

maps\_node**->**maps **=** kmemdup**(**maps**,** **sizeof(\***maps**)** **\*** num\_maps**,** GFP\_KERNEL**);**

**}** **else** **{**

maps\_node**->**maps **=** maps**;**

**}**

//pinctrl\_maps添加到pinctrl\_maps链表中

mutex\_lock**(&**pinctrl\_maps\_mutex**);**

list\_add\_tail**(&**maps\_node**->**node**,** **&**pinctrl\_maps**);**

mutex\_unlock**(&**pinctrl\_maps\_mutex**);**

**return** 0**;**

**}**

###### 1.2.4.3.3 add\_setting

device的state，例如default等，对应的是pinctrl\_state结构体，也就是每个state状态都会有一个pinctrl\_state结构体。

struct pinctrl\_state **{**

//pinctrl->states链表节点

struct list\_head node**;**

//state name，例如default

const char **\***name**;**

//保存pinctrl\_setting链表

struct list\_head settings**;**

**};**

pinctrl\_state下的settings链表存储了许多pinctrl\_setting，pinctrl\_setting表示每个pin的mux或config属性。

struct pinctrl\_setting **{**

//settings链表节点

struct list\_head node**;**

//type分为mux/config两种类型

enum pinctrl\_map\_type type**;**

//对应的pinctrl\_dev结构体

struct pinctrl\_dev **\***pctldev**;**

//使用device的名字

const char **\***dev\_name**;**

//mux config结构体

union **{**

struct pinctrl\_setting\_mux mux**;**

struct pinctrl\_setting\_configs configs**;**

**}** data**;**

**};**

pinctrl\_setting下的pinctrl\_setting\_mux和pinctrl\_setting\_configs结构体pinctrl\_map\_mux和pinctrl\_map\_configs比较类似。

struct pinctrl\_setting\_mux **{**

//pin number

unsigned group**;**

//samsung\_pmx\_func结构体

//在pmx\_functions数组的id

unsigned func**;**

**};**

struct pinctrl\_setting\_configs **{**

//pin number

unsigned group\_or\_pin**;**

//config配置信息

unsigned long **\***configs**;**

//有几个配置 上下拉/驱动能力

unsigned num\_configs**;**

**};**

add\_setting函数将该device对应的pinctrl\_map转换成pinctrl\_setting并添加到对应的pinctrl\_state链表中，并且add\_setting会创建pinctrl\_state结构体。

static int add\_setting**(**struct pinctrl **\***p**,** struct pinctrl\_map const **\***map**)**

**{**

struct pinctrl\_state **\***state**;**

struct pinctrl\_setting **\***setting**;**

int ret**;**

//从pinctrl中的states链表中寻找name相等的pinctrl\_state结构体

//这里的name指的是类似default的statename

state **=** find\_state**(**p**,** map**->**name**);**

//不存在则创建pinctrl\_state结构体，并添加到pinctrl的states链表中

**if** **(!**state**)**

state **=** create\_state**(**p**,** map**->**name**);**

//dummy map直接返回

**if** **(**map**->**type **==** PIN\_MAP\_TYPE\_DUMMY\_STATE**)**

**return** 0**;**

//创建pinctrl\_setting结构体

setting **=** kzalloc**(sizeof(\***setting**),** GFP\_KERNEL**);**

//设置setting的type

setting**->**type **=** map**->**type**;**

//根据pinctrl节点名称从pinctrldev\_list链表中找到对应的pinctrl\_dev结构体

setting**->**pctldev **=** get\_pinctrl\_dev\_from\_devname**(**map**->**ctrl\_dev\_name**);**

//设置setting的dev\_name，该setting被哪个device使用

setting**->**dev\_name **=** map**->**dev\_name**;**

**switch** **(**map**->**type**)** **{**

**case** PIN\_MAP\_TYPE\_MUX\_GROUP**:**

//将mux类型的pinctrl\_map转换成pinctrl\_setting

ret **=** pinmux\_map\_to\_setting**(**map**,** setting**);**

**break;**

**case** PIN\_MAP\_TYPE\_CONFIGS\_GROUP**:**

//将conf类型的pinctrl\_map转换成pinctrl\_setting

ret **=** pinconf\_map\_to\_setting**(**map**,** setting**);**

**break;**

**}**

//将pinctrl\_setting添加到pinctrl\_state的settings链表中

list\_add\_tail**(&**setting**->**node**,** **&**state**->**settings**);**

**return** 0**;**

**}**

创建pinctrl\_state的函数如下：

static struct pinctrl\_state **\***create\_state**(**struct pinctrl **\***p**,** const char **\***name**)**

**{**

struct pinctrl\_state **\***state**;**

//创建pinctrl\_state

state **=** kzalloc**(sizeof(\***state**),** GFP\_KERNEL**);**

//设置state名称，例如default等

state**->**name **=** name**;**

//初始化setttings链表

INIT\_LIST\_HEAD**(&**state**->**settings**);**

//将pinctrl\_state添加到pinctrl中的states链表

list\_add\_tail**(&**state**->**node**,** **&**p**->**states**);**

**return** state**;**

**}**

mux类型的pinctrl\_map转换成pinctrl\_setting的函数如下：

int pinmux\_map\_to\_setting**(**struct pinctrl\_map const **\***map**,** struct pinctrl\_setting **\***setting**)**

**{**

struct pinctrl\_dev **\***pctldev **=** setting**->**pctldev**;**

//获取pinctrl节点的mux操作函数

const struct pinmux\_ops **\***pmxops **=** pctldev**->**desc**->**pmxops**;**

char const **\*** const **\***groups**;**

unsigned num\_groups**;**

int ret**;**

const char **\***group**;**

//返回samsung\_pinctrl\_drv\_data中对应的samsung\_pmx\_func结构体在pmx\_functions数组的id

ret **=** pinmux\_func\_name\_to\_selector**(**pctldev**,** map**->**data**.**mux**.**function**);**

//设置setting.mux对应的function id

setting**->**data**.**mux**.**func **=** ret**;**

//samsung\_pmx\_func对应的groups和num\_groups信息，也就是对应的pin名字

ret **=** pmxops**->**get\_function\_groups**(**pctldev**,** setting**->**data**.**mux**.**func**,** **&**groups**,** **&**num\_groups**);**

**if** **(**map**->**data**.**mux**.**group**)** **{**

//找到map group在groups中的id

group **=** map**->**data**.**mux**.**group**;**

ret **=** match\_string**(**groups**,** num\_groups**,** group**);**

**}**

//返回samsung\_pinctrl\_drv\_data中pin\_groups对应的id

ret **=** pinctrl\_get\_group\_selector**(**pctldev**,** group**);**

//设置setting下的group id

setting**->**data**.**mux**.**group **=** ret**;**

**return** 0**;**

**}**

static int pinmux\_func\_name\_to\_selector**(**struct pinctrl\_dev **\***pctldev**,** const char **\***function**)**

**{**

//这里的function存储了描述该属性的device node路径

const struct pinmux\_ops **\***ops **=** pctldev**->**desc**->**pmxops**;**

//samsung\_pinctrl\_drv\_data对应的nr\_functions

unsigned nfuncs **=** ops**->**get\_functions\_count**(**pctldev**);**

unsigned selector **=** 0**;**

/\* See if this pctldev has this function \*/

**while** **(**selector **<** nfuncs**)** **{**

//samsung\_pinctrl\_drv\_data对应的samsung\_pmx\_func的名字

const char **\***fname **=** ops**->**get\_function\_name**(**pctldev**,** selector**);**

//如果名称相同，返回pmx\_functions数组的id

**if** **(!**strcmp**(**function**,** fname**))**

**return** selector**;**

selector**++;**

**}**

//没有找到，则返回出错

dev\_err**(**pctldev**->**dev**,** "function '%s' not supported\n"**,** function**);**

**return** **-**EINVAL**;**

**}**

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**;**

**}**

int pinctrl\_get\_group\_selector**(**struct pinctrl\_dev **\***pctldev**,** const char **\***pin\_group**)**

**{**

//这里的group指的是每个pin脚

const struct pinctrl\_ops **\***pctlops **=** pctldev**->**desc**->**pctlops**;**

//samsung\_get\_group\_name获取samsung\_pinctrl\_drv\_data中的nr\_groups信息

unsigned ngroups **=** pctlops**->**get\_groups\_count**(**pctldev**);**

unsigned group\_selector **=** 0**;**

**while** **(**group\_selector **<** ngroups**)** **{**

//获取samsung\_pinctrl\_drv\_data中pin\_groups数组对应id的name

const char **\***gname **=** pctlops**->**get\_group\_name**(**pctldev**,** group\_selector**);**

//如果name相同则退出，返回id

**if** **(!**strcmp**(**gname**,** pin\_group**))** **{**

**return** group\_selector**;**

**}**

group\_selector**++;**

**}**

dev\_err**(**pctldev**->**dev**,** "does not have pin group %s\n"**,** pin\_group**);**

**return** **-**EINVAL**;**

**}**

conf类型的pinctrl\_map转换成pinctrl\_setting函数如下：

int pinconf\_map\_to\_setting**(**struct pinctrl\_map const **\***map**,** struct pinctrl\_setting **\***setting**)**

**{**

struct pinctrl\_dev **\***pctldev **=** setting**->**pctldev**;**

int pin**;**

**switch** **(**setting**->**type**)** **{**

**case** PIN\_MAP\_TYPE\_CONFIGS\_GROUP**:**

//获取samsung\_pinctrl\_drv\_data中pin\_groups数组对应id的name

pin **=** pinctrl\_get\_group\_selector**(**pctldev**,** map**->**data**.**configs**.**group\_or\_pin**);**

setting**->**data**.**configs**.**group\_or\_pin **=** pin**;**

**break;**

**}**

setting**->**data**.**configs**.**num\_configs **=** map**->**data**.**configs**.**num\_configs**;**

setting**->**data**.**configs**.**configs **=** map**->**data**.**configs**.**configs**;**

**return** 0**;**

**}**

##### 1.2.4.4 pinctrl\_lookup\_state

上面的步骤已经创建好了pinctrl结构体，那么可以从该结构体中寻找对应的pinctrl\_state结构体。

struct pinctrl\_state **\***pinctrl\_lookup\_state**(**struct pinctrl **\***p**,** const char **\***name**)**

**{**

struct pinctrl\_state **\***state**;**

state **=** find\_state**(**p**,** name**);**

**return** state**;**

**}**

static struct pinctrl\_state **\***find\_state**(**struct pinctrl **\***p**,** const char **\***name**)**

**{**

struct pinctrl\_state **\***state**;**

//从pinctrl中的states链表中找到名字相同的pinctrl\_state节点

list\_for\_each\_entry**(**state**,** **&**p**->**states**,** node**)**

**if** **(!**strcmp**(**state**->**name**,** name**))**

**return** state**;**

**return** **NULL;**

**}**

##### 1.2.4.5 pinctrl\_select\_state

通过pinctrl\_lookup\_state获取到了对应的pinctrl\_state结构体，那么设置对应的pin状态函数为pinctrl\_select\_state。

int pinctrl\_select\_state**(**struct pinctrl **\***p**,** struct pinctrl\_state **\***state**)**

**{**

struct pinctrl\_setting **\***setting**,** **\***setting2**;**

struct pinctrl\_state **\***old\_state **=** p**->**state**;**

int ret**;**

//如果当前需要设置的pinctrl\_state和当前的一致，直接退出

**if** **(**p**->**state **==** state**)**

**return** 0**;**

//如果以前该设备设置过pinctrl\_state，遍历该state下的所有settings

//如果setting类型为mux，则先将该setting对应pin的相关属性清空

**if** **(**p**->**state**)** **{**

list\_for\_each\_entry**(**setting**,** **&**p**->**state**->**settings**,** node**)** **{**

**if** **(**setting**->**type **!=** PIN\_MAP\_TYPE\_MUX\_GROUP**)**

**continue;**

pinmux\_disable\_setting**(**setting**);**

**}**

**}**

//该设备对应的state也清空

p**->**state **=** **NULL;**

//轮询state下的settings链表，重新设置pin对应的功能

list\_for\_each\_entry**(**setting**,** **&**state**->**settings**,** node**)** **{**

**switch** **(**setting**->**type**)** **{**

**case** PIN\_MAP\_TYPE\_MUX\_GROUP**:**

//设置gpio复用功能

ret **=** pinmux\_enable\_setting**(**setting**);**

**break;**

**case** PIN\_MAP\_TYPE\_CONFIGS\_GROUP**:**

//设置上下拉,驱动能力等属性

ret **=** pinconf\_apply\_setting**(**setting**);**

**break;**

**}**

**}**

//设置该device对应的pinctrl\_state结构体

p**->**state **=** state**;**

**return** 0**;**

**}**

pinctrl\_select\_state遍历其下面的pinctrl\_setting并做对应的设置(pin复用和配置)。

如果device以前已经设置过state，那么首先需要清除原来的设置。

void pinmux\_disable\_setting**(**struct pinctrl\_setting const **\***setting**)**

**{**

struct pinctrl\_dev **\***pctldev **=** setting**->**pctldev**;**

const struct pinctrl\_ops **\***pctlops **=** pctldev**->**desc**->**pctlops**;**

int ret **=** 0**;**

const unsigned **\***pins **=** **NULL;**

unsigned num\_pins **=** 0**;**

int i**;**

struct pin\_desc **\***desc**;**

//获取samsung\_pinctrl\_drv\_data对应group的pins和num\_pins信息

//这里的pins为该setting对应的pin名，num\_pins为1

**if** **(**pctlops**->**get\_group\_pins**)**

ret **=** pctlops**->**get\_group\_pins**(**pctldev**,** setting**->**data**.**mux**.**group**,** **&**pins**,** **&**num\_pins**);**

**for** **(**i **=** 0**;** i **<** num\_pins**;** i**++)** **{**

//获取pin对应的pin\_desc结构体

desc **=** pin\_desc\_get**(**pctldev**,** pins**[**i**]);**

**if** **(**desc**->**mux\_setting **==** **&(**setting**->**data**.**mux**))** **{**

//设置mux\_setting为空

desc**->**mux\_setting **=** **NULL;**

//设置pin对应的mux\_usecount--，gpio\_owner，mux\_setting设置为null

pin\_free**(**pctldev**,** pins**[**i**],** **NULL);**

**}**

**}**

**}**

设置pin复用状态的函数为：

int pinmux\_enable\_setting**(**struct pinctrl\_setting const **\***setting**)**

**{**

struct pinctrl\_dev **\***pctldev **=** setting**->**pctldev**;**

const struct pinctrl\_ops **\***pctlops **=** pctldev**->**desc**->**pctlops**;**

const struct pinmux\_ops **\***ops **=** pctldev**->**desc**->**pmxops**;**

int ret **=** 0**;**

const unsigned **\***pins **=** **NULL;**

unsigned num\_pins **=** 0**;**

int i**;**

struct pin\_desc **\***desc**;**

//获取samsung\_pinctrl\_drv\_data对应group的pins和num\_pins信息

//这里的pins为该setting对应的pin名，num\_pins为1

**if** **(**pctlops**->**get\_group\_pins**)**

ret **=** pctlops**->**get\_group\_pins**(**pctldev**,** setting**->**data**.**mux**.**group**,&**pins**,** **&**num\_pins**);**

**for** **(**i **=** 0**;** i **<** num\_pins**;** i**++)** **{**

//申请该pin，判断该pin是否被人使用

ret **=** pin\_request**(**pctldev**,** pins**[**i**],** setting**->**dev\_name**,** **NULL);**

**}**

//设置pin\_desc的mux\_setting

**for** **(**i **=** 0**;** i **<** num\_pins**;** i**++)** **{**

desc **=** pin\_desc\_get**(**pctldev**,** pins**[**i**]);**

desc**->**mux\_setting **=** **&(**setting**->**data**.**mux**);**

**}**

//samsung\_pinmux\_set\_mux设置pin的复用属性

ret **=** ops**->**set\_mux**(**pctldev**,** setting**->**data**.**mux**.**func**,** setting**->**data**.**mux**.**group**);**

**return** 0**;**

**}**

设置pin复用分两个步骤，先申请该pin，看是否被使用过。

static int pin\_request**(**struct pinctrl\_dev **\***pctldev**,** int pin**,** const char **\***owner**,** struct pinctrl\_gpio\_range **\***gpio\_range**)**

**{**

struct pin\_desc **\***desc**;**

const struct pinmux\_ops **\***ops **=** pctldev**->**desc**->**pmxops**;**

int status **=** **-**EINVAL**;**

//获取pin对应的pin\_desc结构体

desc **=** pin\_desc\_get**(**pctldev**,** pin**);**

//如果该pin已经被使用了，则退出

**if** **(**desc**->**mux\_usecount **&&** strcmp**(**desc**->**mux\_owner**,** owner**))** **{**

**goto** out**;**

**}**

//该pin的使用计数+1

desc**->**mux\_usecount**++;**

//如果大于1,直接返回

**if** **(**desc**->**mux\_usecount **>** 1**)**

**return** 0**;**

//设置该pin使用的device name

desc**->**mux\_owner **=** owner**;**

/\* Let each pin increase references to this module \*/

**if** **(!**try\_module\_get**(**pctldev**->**owner**))** **{**

**goto** out\_free\_pin**;**

**}**

**if** **(**gpio\_range **&&** ops**->**gpio\_request\_enable**)**

/\* This requests and enables a single GPIO pin \*/

status **=** ops**->**gpio\_request\_enable**(**pctldev**,** gpio\_range**,** pin**);**

**else** **if** **(**ops**->**request**)**

status **=** ops**->**request**(**pctldev**,** pin**);**

**else**

status **=** 0**;**

**return** status**;**

**}**

第二个步骤就是寄存器的设置了

static int samsung\_pinmux\_set\_mux**(**struct pinctrl\_dev **\***pctldev**,** unsigned selector**,** unsigned group**)**

**{**

samsung\_pinmux\_setup**(**pctldev**,** selector**,** group**,** **true);**

**return** 0**;**

**}**

static void samsung\_pinmux\_setup**(**struct pinctrl\_dev **\***pctldev**,** unsigned selector**,** unsigned group**,** bool enable**)**

**{**

struct samsung\_pinctrl\_drv\_data **\***drvdata**;**

const struct samsung\_pin\_bank\_type **\***type**;**

struct samsung\_pin\_bank **\***bank**;**

void \_\_iomem **\***reg**;**

u32 mask**,** shift**,** data**,** pin\_offset**;**

unsigned long flags**;**

const struct samsung\_pmx\_func **\***func**;**

const struct samsung\_pin\_group **\***grp**;**

//获得samsung\_pinctrl\_drv\_data结构体

drvdata **=** pinctrl\_dev\_get\_drvdata**(**pctldev**);**

//pmx\_functions中保存了一个配置信息中包含的pin和number以及复用寄存器需要设置的val

func **=** **&**drvdata**->**pmx\_functions**[**selector**];**

//pin\_groups中包含了一个pin的相关信息

grp **=** **&**drvdata**->**pin\_groups**[**group**];**

//给定一个pin number 获取到寄存器,偏移,samsung\_pin\_bank信息

pin\_to\_reg\_bank**(**drvdata**,** grp**->**pins**[**0**]** **-** drvdata**->**pin\_base**,** **&**reg**,** **&**pin\_offset**,** **&**bank**);**

//下面就是寄存器设置步骤了

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 **|=** func**->**val **<<** shift**;**

writel**(**data**,** reg **+** type**->**reg\_offset**[**PINCFG\_TYPE\_FUNC**]);**

spin\_unlock\_irqrestore**(&**bank**->**slock**,** flags**);**

**}**

同样的配置上下拉，驱动能力等属性的函数为pinconf\_apply\_setting，这里就不详细分析了。

### 1.3 pinctrl子系统和gpio子系统的联系

前面有描述到：

pinctrl提供了pinctrl\_request\_gpio()这样的API，但在代码中不可以直接调用pinctrl\_request\_gpio，当设备驱动申请一个gpio时，仍然需要调用gpio\_request()，这里会调用pinctrl\_request\_gpio()。

pinctrl\_request\_gpio的作用就是在pinctrl子系统中找到对应的pinctrl\_dev并标记对应的pin被使用。

int pinctrl\_request\_gpio**(**unsigned gpio**)**

**{**

struct pinctrl\_dev **\***pctldev**;**

struct pinctrl\_gpio\_range **\***range**;**

int ret**;**

int pin**;**

//根据gpio number获取对应的pinctrl\_dev和pinctrl\_gpio\_range结构体

ret **=** pinctrl\_get\_device\_gpio\_range**(**gpio**,** **&**pctldev**,** **&**range**);**

mutex\_lock**(&**pctldev**->**mutex**);**

//转换为pin controllers number

pin **=** gpio\_to\_pin**(**range**,** gpio**);**

//在pinctrl子系统中申请pin

ret **=** pinmux\_request\_gpio**(**pctldev**,** range**,** pin**,** gpio**);**

mutex\_unlock**(&**pctldev**->**mutex**);**

**return** ret**;**

**}**

int pinmux\_request\_gpio**(**struct pinctrl\_dev **\***pctldev**,**struct pinctrl\_gpio\_range **\***range**,**unsigned pin**,** unsigned gpio**)**

**{**

const char **\***owner**;**

int ret**;**

owner **=** kasprintf**(**GFP\_KERNEL**,** "%s:%d"**,** range**->**name**,** gpio**);**

//在pinctrl子系统中申请pin

ret **=** pin\_request**(**pctldev**,** pin**,** owner**,** range**);**

**return** ret**;**

**}**

### 1.4 probe时候的pinctrl设置

当platform\_drvier和platform\_device匹配时，会自动设置default或init的pin设置。

really\_probe

pinctrl\_bind\_pins

int pinctrl\_bind\_pins**(**struct device **\***dev**)**

**{**

int ret**;**

dev**->**pins **=** devm\_kzalloc**(**dev**,** **sizeof(\*(**dev**->**pins**)),** GFP\_KERNEL**);**

dev**->**pins**->**p **=** devm\_pinctrl\_get**(**dev**);**

dev**->**pins**->**default\_state **=** pinctrl\_lookup\_state**(**dev**->**pins**->**p**,** PINCTRL\_STATE\_DEFAULT**);**

dev**->**pins**->**init\_state **=** pinctrl\_lookup\_state**(**dev**->**pins**->**p**,** PINCTRL\_STATE\_INIT**);**

**return** 0**;**

**}**