# Linux驱动\_中断子系统

这篇文章将分析设备树下的中断子系统。

## 1.中断子系统初始化

设备树中关于中断的描述

intc:interrupt-controller@4a000000 {

compatible = "samsung,s3c2410-irq";

reg = <0x4a000000 0x100>;

interrupt-controller;

#interrupt-cells = <4>;

};

对应的初始化代码如下：

int \_\_init s3c2410\_init\_intc\_of**(**struct device\_node **\***np**,** struct device\_node **\***interrupt\_parent**)**

**{**

**return** s3c\_init\_intc\_of**(**np**,** interrupt\_parent**,**

s3c2410\_ctrl**,** ARRAY\_SIZE**(**s3c2410\_ctrl**));**

**}**

IRQCHIP\_DECLARE**(**s3c2410\_irq**,** "samsung,s3c2410-irq"**,** s3c2410\_init\_intc\_of**);**

IRQCHIP\_DECLARE这个宏的作用是什么呢？

#define IRQCHIP\_DECLARE(name, compat, fn) OF\_DECLARE\_2(irqchip, name, compat, fn)

#define \_OF\_DECLARE(table, name, compat, fn, fn\_type) \

static const struct of\_device\_id \_\_of\_table\_##name \

\_\_used \_\_section(\_\_##table##\_of\_table) \

= { .compatible = compat, \

.data = (fn == (fn\_type)NULL) ? fn : fn }

那么初始化代码利用宏展开为：

static const struct of\_device\_id \_\_of\_table\_s3c2410\_irq \

\_\_used \_\_section**(**\_\_irqchip\_of\_table**)** \

**=** **{** **.**compatible **=** "samsung,s3c2410-irq"**,** \

**.**data **=** s3c2410\_init\_intc\_of**}**

整个linux驱动中断的初始化代码为：

extern struct of\_device\_id \_\_irqchip\_of\_table**[];**

void \_\_init irqchip\_init**(**void**)**

**{**

of\_irq\_init**(**\_\_irqchip\_of\_table**);**

**}**

所有的interrupt-controller通过IRQCHIP\_DECLARE宏添加到\_\_irqchip\_of\_table中。

继续分析of\_irq\_init：

struct of\_intc\_desc **{**

struct list\_head list**;**

of\_irq\_init\_cb\_t irq\_init\_cb**;**

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

struct device\_node **\***interrupt\_parent**;**

**};**

void \_\_init of\_irq\_init**(**const struct of\_device\_id **\***matches**)**

**{**

const struct of\_device\_id **\***match**;**

struct device\_node **\***np**,** **\***parent **=** **NULL;**

struct of\_intc\_desc **\***desc**,** **\***temp\_desc**;**

struct list\_head intc\_desc\_list**,** intc\_parent\_list**;**

INIT\_LIST\_HEAD**(&**intc\_desc\_list**);**

INIT\_LIST\_HEAD**(&**intc\_parent\_list**);**

//遍历所有的node，从matches中找到合适的match和node

//在这里\_\_of\_table\_s3c2410\_irq中有compatible = "samsung,s3c2410-irq"

//dts树中有compatible = "samsung,s3c2410-irq"

//两者匹配后就可以得到对应的match和node信息

for\_each\_matching\_node\_and\_match**(**np**,** matches**,** **&**match**)** **{**

//判断该node是否有interrupt-controllers属性

**if** **(!**of\_find\_property**(**np**,** "interrupt-controller"**,** **NULL)** **||** **!**of\_device\_is\_available**(**np**))**

**continue;**

//判断是否有irq初始化函数

**if** **(**WARN**(!**match**->**data**,** "of\_irq\_init: no init function for %s\n"**,** match**->**compatible**))**

**continue;**

//创建结构体of\_intc\_desc

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

//设置结构体of\_intc\_desc

//初始化函数

desc**->**irq\_init\_cb **=** match**->**data**;**

//对应的device\_node

desc**->**dev **=** of\_node\_get**(**np**);**

//寻找interrupt-parent的父节点

//对于interrupt controller，它也可能是一个树状的结构

desc**->**interrupt\_parent **=** of\_irq\_find\_parent**(**np**);**

**if** **(**desc**->**interrupt\_parent **==** np**)**

desc**->**interrupt\_parent **=** **NULL;**

//添加到intc\_desc\_list链表

list\_add\_tail**(&**desc**->**list**,** **&**intc\_desc\_list**);**

**}**

//正因为interrupt controller被组织成树状的结构，因此初始化的顺序就需要控制

//应该从根节点开始，依次递进到下一个level的interrupt controller

//intc\_desc\_list链表中的节点会被一个个的处理，每处理完一个节点就会将该节点删除，当所有的节点被删除，整个处理过程也就是结束了。

**while** **(!**list\_empty**(&**intc\_desc\_list**))** **{**

list\_for\_each\_entry\_safe**(**desc**,** temp\_desc**,** **&**intc\_desc\_list**,** list**)** **{**

int ret**;**

//最开始的时候parent变量是NULL，确保第一个被处理的是root interrupt controller。

//在处理完root node之后，parent变量被设定为root interrupt controller

//因此，第二个循环中处理的是所有parent是root interrupt controller的child interrupt controller。

//也就是level 1（如果root是level 0的话）的节点。

**if** **(**desc**->**interrupt\_parent **!=** parent**)**

**continue;**

//从链表中删除

list\_del**(&**desc**->**list**);**

//device already created for the node

of\_node\_set\_flag**(**desc**->**dev**,** OF\_POPULATED**);**

//调用注册的初始化函数

ret **=** desc**->**irq\_init\_cb**(**desc**->**dev**,** desc**->**interrupt\_parent**);**

//处理完的节点放入intc\_parent\_list链表，后面会用到

list\_add\_tail**(&**desc**->**list**,** **&**intc\_parent\_list**);**

**}**

//对于level 0，只有一个root interrupt controller，

//对于level 1，可能有若干个interrupt controller，

//因此要遍历这些parent interrupt controller，以便处理下一个level的child node。

desc **=** list\_first\_entry\_or\_null**(&**intc\_parent\_list**,** typeof**(\***desc**),** list**);**

list\_del**(&**desc**->**list**);**

parent **=** desc**->**dev**;**

kfree**(**desc**);**

**}**

list\_for\_each\_entry\_safe**(**desc**,** temp\_desc**,** **&**intc\_parent\_list**,** list**)** **{**

list\_del**(&**desc**->**list**);**

kfree**(**desc**);**

**}**

**}**

只有该node中有interrupt-controller这个属性定义，那么linux kernel就会分配一个interrupt controller的描述符（struct intc\_desc）并挂入队列。通过interrupt-parent属性，可以确定各个interrupt controller的层次关系。

在scan了所有的Device Tree中的interrupt controller的定义之后，系统开始匹配过程。一旦匹配到了interrupt chip列表中的项次后，就会调用相应的初始化函数。

## 2.irq\_domain

### 2.1 概述

在linux kernel中，我们使用下面两个ID来标识一个来自外设的中断：

**IRQ number**：CPU需要为每一个外设中断编号，我们称之IRQ Number。这个IRQ number是一个虚拟的interrupt ID，和硬件无关，仅仅是被CPU用来标识一个外设中断。

**HW interrupt ID**：对于interrupt controller而言，它收集了多个外设的interrupt request line并向上传递，因此，interrupt controller需要对外设中断进行编码。Interrupt controller用HW interrupt ID来标识外设的中断。在interrupt controller级联的情况下，仅仅用HW interrupt ID已经不能唯一标识一个外设中断，还需要知道该HW interrupt ID所属的interrupt controller（HW interrupt ID在不同的Interrupt controller上是会重复编码的）。

这样，CPU和interrupt controller在标识中断上就有了一些不同的概念，但是，对于驱动工程师而言，我们和CPU视角是一样的，我们只希望得到一个IRQ number，而不关系具体是那个interrupt controller上的那个HW interrupt ID。这样一个好处是在中断相关的硬件发生变化的时候，驱动软件不需要修改。因此，linux kernel中的中断子系统需要提供一个将HW interrupt ID映射到IRQ number上来的机制。

### 2.2 历史

关于HW interrupt ID映射到IRQ number上 这事，在过去系统只有一个interrupt controller的时候还是很简单的，中断控制器上实际的HW interrupt line的编号可以直接变成IRQ number。例如我们大家都熟悉的SOC内嵌的interrupt controller，这种controller多半有中断状态寄存器，这个寄存器可能有64个bit（也可能更多），每个bit就是一个IRQ number，可以直接进行映射。这时候，GPIO的中断在中断控制器的状态寄存器中只有一个bit，因此所有的GPIO中断只有一个IRQ number，在该通用GPIO中断的irq handler中进行deduplex，将各个具体的GPIO中断映射到其相应的IRQ number上。如果你是一个足够老的工程师，应该是经历过这个阶段的。

随着linux kernel的发展，将interrupt controller抽象成irqchip这个概念越来越流行，甚至GPIO controller也可以被看出一个interrupt controller chip，这样，系统中至少有两个中断控制器了，一个传统意义的中断控制器，一个是GPIO controller type的中断控制器。随着系统复杂度加大，外设中断数据增加，实际上系统可以需要多个中断控制器进行级联，面对这样的趋势，linux kernel工程师如何应对？答案就是irq domain这个概念。

我们听说过很多的domain，power domain，clock domain等等，所谓domain，就是领域，范围的意思，也就是说，任何的定义出了这个范围就没有意义了。系统中所有的interrupt controller会形成树状结构，对于每个interrupt controller都可以连接若干个外设的中断请求（我们称之interrupt source），interrupt controller会对连接其上的interrupt source（根据其在Interrupt controller中物理特性）进行编号（也就是HW interrupt ID了）。但这个编号仅仅限制在本interrupt controller范围内。

### 2.3 接口

#### 2.3.1 向系统注册irq domain

具体如何进行映射是interrupt controller自己的事情，不过，有软件架构思想的工程师更愿意对形形色色的interrupt controller进行抽象，对如何进行HW interrupt ID到IRQ number映射关系上进行进一步的抽象。因此，通用中断处理模块中有一个irq domain的子模块，该模块将这种映射关系分成了三类：

**线性映射**。其实就是一个lookup table，HW interrupt ID作为index，通过查表可以获取对应的IRQ number。对于Linear map而言，interrupt controller对其HW interrupt ID进行编码的时候要满足一定的条件：hw ID不能过大，而且ID排列最好是紧密的。对于线性映射，其接口API如下：

static inline struct irq\_domain **\***irq\_domain\_add\_linear**(**struct device\_node **\***of\_node**,** unsigned int size**,** const struct irq\_domain\_ops **\***ops**,** void **\***host\_data**)**

**{**

**return** \_\_irq\_domain\_add**(**of\_node\_to\_fwnode**(**of\_node**),** size**,** size**,** 0**,** ops**,** host\_data**);**

**}**

**Radix Tree map**。建立一个Radix Tree来维护HW interrupt ID到IRQ number映射关系。HW interrupt ID作为lookup key，在Radix Tree检索到IRQ number。如果的确不能满足线性映射的条件，可以考虑Radix Tree map。实际上，内核中使用Radix Tree map的只有powerPC和MIPS的硬件平台。对于Radix Tree map，其接口API如下：

static inline struct irq\_domain **\***irq\_domain\_add\_tree**(**struct device\_node **\***of\_node**,** const struct irq\_domain\_ops **\***ops**,** void

**\***host\_data**)**

**{**

**return** \_\_irq\_domain\_add**(**of\_node**,** 0**,** **~**0**,** 0**,** ops**,** host\_data**);**

**}**

**no map**。有些中断控制器很强，可以通过寄存器配置HW interrupt ID而不是由物理连接决定的。例如PowerPC 系统使用的MPIC (Multi-Processor Interrupt Controller)。在这种情况下，不需要进行映射，我们直接把IRQ number写入HW interrupt ID配置寄存器就OK了，这时候，生成的HW interrupt ID就是IRQ number，也就不需要进行mapping了。对于这种类型的映射，其接口API如下：

static inline struct irq\_domain **\***irq\_domain\_add\_nomap**(**struct device\_node **\***of\_node**,** unsigned int max\_irq**,** const struct

irq\_domain\_ops **\***ops**,** void **\***host\_data**)**

**{**

**return** \_\_irq\_domain\_add**(**of\_node**,** 0**,** max\_irq**,** max\_irq**,** ops**,** host\_data**);**

**}**

#### **2.3.2** 为irq domain创建映射

上节的内容主要是向系统注册一个irq domain，具体HW interrupt ID和IRQ number的映射关系都是空的，因此，具体各个irq domain如何管理映射所需要的database还是需要建立的。例如：对于线性映射的irq domain，我们需要建立线性映射的lookup table，对于Radix Tree map，我们要把那个反应IRQ number和HW interrupt ID的Radix tree建立起来。创建映射有四个接口函数：

(1)调用irq\_create\_mapping函数建立HW interrupt ID和IRQ number的映射关系。该接口函数以irq domain和HW interrupt ID为参数，返回IRQ number（这个IRQ number是动态分配的）。该函数的原型定义如下：

extern unsigned int irq\_create\_mapping**(**struct irq\_domain **\***host**,** irq\_hw\_number\_t hwirq**);**

驱动调用该函数的时候必须提供HW interrupt ID，也就是意味着driver知道自己使用的HW interrupt ID，而一般情况下，HW interrupt ID其实对具体的driver应该是不可见的，不过有些场景比较特殊，例如GPIO类型的中断，它的HW interrupt ID和GPIO有着特定的关系，driver知道自己使用那个GPIO，也就是知道使用哪一个HW interrupt ID了。

(2) irq\_create\_strict\_mappings。这个接口函数用来为一组HW interrupt ID建立映射。具体函数的原型定义如下：

extern int irq\_create\_strict\_mappings**(**struct irq\_domain **\***domain**,** unsigned int irq\_base**,** irq\_hw\_number\_t hwirq\_base**,** int count**);**

(3) irq\_create\_of\_mapping。看到函数名字中的of（open firmware），我想你也可以猜到了几分，这个接口当然是利用device tree进行映射关系的建立。具体函数的原型定义如下：

extern unsigned int irq\_create\_of\_mapping**(**struct of\_phandle\_args **\***irq\_data**);**

通常，一个普通设备的device tree node已经描述了足够的中断信息，在这种情况下，该设备的驱动在初始化的时候可以调用irq\_of\_parse\_and\_map这个接口函数进行该device node中和中断相关的内容（interrupts和interrupt-parent属性）进行分析，并建立映射关系，具体代码如下：

unsigned int irq\_of\_parse\_and\_map**(**struct device\_node **\***dev**,** int index**)**

**{**

    struct of\_phandle\_args oirq**;**

//分析device node中的interrupt相关属性

**if** **(**of\_irq\_parse\_one**(**dev**,** index**,** **&**oirq**))**

**return** 0**;**

//创建映射，并返回对应的IRQ number

**return** irq\_create\_of\_mapping**(&**oirq**);**

**}**

对于一个使用Device tree的普通驱动程序（我们推荐这样做），基本上初始化需要调用irq\_of\_parse\_and\_map获取IRQ number，然后调用request\_threaded\_irq申请中断handler。

(4) irq\_create\_direct\_mapping。这是给no map那种类型的interrupt controller使用的，这里不再赘述。

#### 2.3.3 数据结构描述

##### 2.3.3.1 irq domain的callback接口

struct irq\_domain\_ops **{**

int **(\***match**)(**struct irq\_domain **\***d**,** struct device\_node **\***node**,**

enum irq\_domain\_bus\_token bus\_token**);**

int **(\***select**)(**struct irq\_domain **\***d**,** struct irq\_fwspec **\***fwspec**,**

enum irq\_domain\_bus\_token bus\_token**);**

int **(\***map**)(**struct irq\_domain **\***d**,** unsigned int virq**,** irq\_hw\_number\_t hw**);**

void **(\***unmap**)(**struct irq\_domain **\***d**,** unsigned int virq**);**

int **(\***xlate**)(**struct irq\_domain **\***d**,** struct device\_node **\***node**,**

const u32 **\***intspec**,** unsigned int intsize**,**

unsigned long **\***out\_hwirq**,** unsigned int **\***out\_type**);**

**};**

我们先看xlate函数，语义是翻译（translate）的意思，那么到底翻译什么呢？在DTS文件中，各个使用中断的device node会通过一些属性（例如interrupts和interrupt-parent属性）来提供中断信息给kernel以便kernel可以正确的进行driver的初始化动作。这里，interrupts属性所表示的interrupt specifier只能由具体的interrupt controller（也就是irq domain）来解析。而xlate函数就是将指定的设备（node参数）上若干个（intsize参数）中断属性（intspec参数）翻译成HW interrupt ID（out\_hwirq参数）和trigger类型（out\_type）。

match是判断一个指定的interrupt controller（node参数）是否和一个irq domain匹配（d参数），如果匹配的话，返回1。实际上，内核中很少定义这个callback函数，实际上struct irq\_domain中有一个of\_node指向了对应的interrupt controller的device node，因此，如果不提供该函数，那么default的匹配函数其实就是判断irq domain的of\_node成员是否等于传入的node参数。

map和unmap是操作相反的函数，我们描述其中之一就OK了。调用map函数的时机是在创建（或者更新）HW interrupt ID（hw参数）和IRQ number（virq参数）关系的时候。其实，从发生一个中断到调用该中断的handler仅仅调用一个request\_threaded\_irq是不够的，还需要针对该irq number设定：

(1)设定该IRQ number对应的中断描述符（struct irq\_desc）的irq chip

(2)设定该IRQ number对应的中断描述符的highlevel irq-events handler

(3)设定该IRQ number对应的中断描述符的 irq chip data

这些设定不适合由具体的硬件驱动来设定，因此在Interrupt controller，也就是irq domain的callback函数中设定。

##### 2.3.3.2 irq domain

struct irq\_domain **{**

struct list\_head link**;**

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

//callback函数

const struct irq\_domain\_ops **\***ops**;**

void **\***host\_data**;**

unsigned int flags**;**

/\* Optional data \*/

struct fwnode\_handle **\***fwnode**;**

enum irq\_domain\_bus\_token bus\_token**;**

struct irq\_domain\_chip\_generic **\***gc**;**

/\* reverse map data. The linear map gets appended to the irq\_domain \*/

//该domain中最大的那个HW interrupt ID

irq\_hw\_number\_t hwirq\_max**;**

unsigned int revmap\_direct\_max\_irq**;**

//线性映射的size，for Radix Tree map和no map，该值等于0

unsigned int revmap\_size**;**

//Radix Tree map使用到的radix tree root node

struct radix\_tree\_root revmap\_tree**;**

//线性映射使用的lookup table

unsigned int linear\_revmap**[];**

**};**

linux内核中，所有的irq domain被挂入一个全局链表，链表头定义如下：

static LIST\_HEAD**(**irq\_domain\_list**);**

struct irq\_domain中的link成员就是挂入这个队列的节点。通过irq\_domain\_list这个指针，可以获取整个系统中HW interrupt ID和IRQ number的mapping DB。host\_data定义了底层interrupt controller使用的私有数据，和具体的interrupt controller相关。

对于线性映射：

（1）linear\_revmap保存了一个线性的lookup table，index是HW interrupt ID，table中保存了IRQ number值

（2）revmap\_size等于线性的lookup table的size。

（3）hwirq\_max保存了最大的HW interrupt ID

（4）revmap\_direct\_max\_irq没有用，设定为0。revmap\_tree没有用。

对于Radix Tree map：

（1）linear\_revmap没有用，revmap\_size等于0。

（2）hwirq\_max没有用，设定为一个最大值。

（3）revmap\_direct\_max\_irq没有用，设定为0。

（4）revmap\_tree指向Radix tree的root node。

## 3.s3c\_init\_intc\_of

s3c\_init\_intc\_of创建了两个irq\_domain和s3c\_irq\_intc结构体。

s3c\_irq\_intc结构体存储了和中断制器相关的寄存器数据。

/\*

\* Sructure holding the controller data

\* @reg\_pending register holding pending irqs

\* @reg\_intpnd special register intpnd in main intc

\* @reg\_mask mask register

\* @domain irq\_domain of the controller

\* @parent parent controller for ext and sub irqs

\* @irqs irq-data, always s3c\_irq\_data[32]

\*/

struct s3c\_irq\_intc **{**

void \_\_iomem **\***reg\_pending**;**

void \_\_iomem **\***reg\_intpnd**;**

void \_\_iomem **\***reg\_mask**;**

struct irq\_domain **\***domain**;**

struct s3c\_irq\_intc **\***parent**;**

struct s3c\_irq\_data **\***irqs**;**

**};**

代码中使用了一个静态的全局变量s3c\_intc保存s3c\_irq\_intc结构体。

static struct s3c\_irq\_intc **\***s3c\_intc**[**3**];**

s3c\_init\_intc\_of根据s3c2410\_ctrl信息创建s3c\_irq\_intc

static struct s3c24xx\_irq\_of\_ctrl s3c2410\_ctrl**[]** **=** **{**

**{**

**.**name **=** "intc"**,**

**.**offset **=** 0**,**

**},** **{**

**.**name **=** "subintc"**,**

**.**offset **=** 0x18**,**

**.**parent **=** **&**s3c\_intc**[**0**],**

**}**

**};**

s3c\_init\_intc\_of**(**np**,** interrupt\_parent**,** s3c2410\_ctrl**,** ARRAY\_SIZE**(**s3c2410\_ctrl**));**

s3c\_init\_intc\_of函数代码如下：

static int \_\_init s3c\_init\_intc\_of**(**struct device\_node **\***np**,** struct device\_node **\***interrupt\_parent**,** struct s3c24xx\_irq\_of\_ctrl **\***s3c\_ctrl**,**

int num\_ctrl**)**

**{**

struct s3c\_irq\_intc **\***intc**;**

struct s3c24xx\_irq\_of\_ctrl **\***ctrl**;**

struct irq\_domain **\***domain**;**

void \_\_iomem **\***reg\_base**;**

int i**;**

//获取操作寄存器地址reg = <0x4a000000 0x100>

reg\_base **=** of\_iomap**(**np**,** 0**);**

//分配irq\_domain,类型为线性映射

domain **=** irq\_domain\_add\_linear**(**np**,** num\_ctrl **\*** 32**,** **&**s3c24xx\_irq\_ops\_of**,** **NULL);**

//分配并初始化s3c\_irq\_intc结构体

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

ctrl **=** **&**s3c\_ctrl**[**i**];**

intc **=** kzalloc**(sizeof(**struct s3c\_irq\_intc**),** GFP\_KERNEL**);**

//保存对应的irq\_domain

intc**->**domain **=** domain**;**

//irqs用于保存中断的数据

intc**->**irqs **=** kzalloc**(sizeof(**struct s3c\_irq\_data**)** **\*** 32**,** GFP\_KERNEL**);**

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

//子中断

//Sub Source Pending寄存器

intc**->**reg\_pending **=** reg\_base **+** ctrl**->**offset**;**

//Interrupt Sub Mask寄存器

intc**->**reg\_mask **=** reg\_base **+** ctrl**->**offset **+** 0x4**;**

//设置s3c\_irq\_intc

**if** **(\*(**ctrl**->**parent**))** **{**

intc**->**parent **=** **\*(**ctrl**->**parent**);**

**}**

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

//Source Pending寄存器

intc**->**reg\_pending **=** reg\_base **+** ctrl**->**offset**;**

//Interrupt Mask寄存器

intc**->**reg\_mask **=** reg\_base **+** ctrl**->**offset **+** 0x08**;**

//Interrupt Pending寄存器

intc**->**reg\_intpnd **=** reg\_base **+** ctrl**->**offset **+** 0x10**;**

**}**

//清空INTPND SRCPND寄存器

s3c24xx\_clear\_intc**(**intc**);**

//设置s3c\_intc数组

s3c\_intc**[**i**]** **=** intc**;**

**}**

//设置handle\_arch\_irq

set\_handle\_irq**(**s3c24xx\_handle\_irq**);**

**return** 0**;**

**}**

irq\_domain对应的回调函数为：

static const struct irq\_domain\_ops s3c24xx\_irq\_ops\_of **=** **{**

**.**map **=** s3c24xx\_irq\_map\_of**,**

**.**xlate **=** s3c24xx\_irq\_xlate\_of**,**

**};**

下面分析一下irq\_domain\_add\_linear的实现：

/\*\*

\* irq\_domain\_add\_linear() - Allocate and register a linear revmap irq\_domain.

\* @of\_node: pointer to interrupt controller's device tree node.

\* @size: Number of interrupts in the domain.

\* @ops: map/unmap domain callbacks

\* @host\_data: Controller private data pointer

\*/

static inline struct irq\_domain **\***irq\_domain\_add\_linear**(**struct device\_node **\***of\_node**,** unsigned int size**,** const struct irq\_domain\_ops

**\***ops**,** void **\***host\_data**)**

**{**

**return** \_\_irq\_domain\_add**(**of\_node\_to\_fwnode**(**of\_node**),** size**,** size**,** 0**,** ops**,** host\_data**);**

**}**

/\*\*

\* \_\_irq\_domain\_add() - Allocate a new irq\_domain data structure

\* @fwnode: firmware node for the interrupt controller

\* @size: Size of linear map; 0 for radix mapping only

\* @hwirq\_max: Maximum number of interrupts supported by controller

\* @direct\_max: Maximum value of direct maps; Use ~0 for no limit; 0 for no

\* direct mapping

\* @ops: domain callbacks

\* @host\_data: Controller private data pointer

\*

\* Allocates and initialize and irq\_domain structure.

\* Returns pointer to IRQ domain, or NULL on failure.

\*/

struct irq\_domain **\***\_\_irq\_domain\_add**(**struct fwnode\_handle **\***fwnode**,** int size**,** irq\_hw\_number\_t hwirq\_max**,** int direct\_max**,** const struct irq\_domain\_ops **\***ops**,** void **\***host\_data**)**

**{**

struct device\_node **\***of\_node **=** to\_of\_node**(**fwnode**);**

struct irq\_domain **\***domain**;**

//根据内存节点分配内存

//空间大小为domain结构体大小和linear map大小

domain **=** kzalloc\_node**(sizeof(\***domain**)** **+** **(sizeof(**unsigned int**)** **\*** size**),** GFP\_KERNEL**,** of\_node\_to\_nid**(**of\_node**));**

//device node使用+1

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

//初始化irq\_domain结构体

//初始化revmap\_tree

INIT\_RADIX\_TREE**(&**domain**->**revmap\_tree**,** GFP\_KERNEL**);**

//初始化回调函数

domain**->**ops **=** ops**;**

//初始化私有数据

domain**->**host\_data **=** host\_data**;**

//设置服务fwnode

domain**->**fwnode **=** fwnode**;**

//最大的的HW interrupt ID

domain**->**hwirq\_max **=** hwirq\_max**;**

//linear map大小

domain**->**revmap\_size **=** size**;**

//没有用。设置为0

domain**->**revmap\_direct\_max\_irq **=** direct\_max**;**

//判断是否有ops->alloc回调

irq\_domain\_check\_hierarchy**(**domain**);**

//添加到irq\_domain\_list链表

mutex\_lock**(&**irq\_domain\_mutex**);**

list\_add**(&**domain**->**link**,** **&**irq\_domain\_list**);**

mutex\_unlock**(&**irq\_domain\_mutex**);**

**return** domain**;**

**}**

## 4.Mapping DB的建立

### 4.1中断相关的Device Tree知识

对于那些产生中断的外设，我们需要定义interrupt-parent和interrupts属性：

**interrupt-parent**。表明该外设的interrupt request line物理的连接到了哪一个中断控制器上

**interrupts**。这个属性描述了具体该外设产生的interrupt的细节信息（也就是传说中的interrupt specifier）。例如：HW interrupt ID（由该外设的device node中的interrupt**-**parent指向的interrupt controller解析）、interrupt触发类型等。

对于Interrupt controller，我们需要定义interrupt-controller和#interrupt-cells的属性：

**interrupt-controller**。表明该device node就是一个中断控制器

**interrupt-cells**。该中断控制器用多少个cell（一个cell就是一个32**-**bit的单元）描述一个外设的interrupt request line。？具体每个cell表示什么样的含义由interrupt controller自己定义。

**interrupts和interrupt-parent**。对于那些不是root 的interrupt controller，其本身也是作为一个产生中断的外设连接到其他的interrupt controller上，因此也需要定义interrupts和interrupt**-**parent的属性。

### 4.2 Mapping DB的建立流程概述

系统中HW interrupt ID和IRQ number的mapping DB是在整个系统初始化的过程中建立起来的，过程如下：

(1)DTS文件描述了系统中的interrupt controller以及外设IRQ的拓扑结构，在linux kernel启动的时候，由bootloader传递给kernel（实际传递的是DTB）。

(2)在Device Tree初始化的时候，形成了系统内所有的device node的树状结构，当然其中包括所有和中断拓扑相关的数据结构（所有的interrupt controller的node和使用中断的外设node）。

(3)在machine driver初始化的时候会调用of\_irq\_init函数，在该函数中会扫描所有interrupt controller的节点，并调用适合的interrupt controller driver进行初始化。毫无疑问，初始化需要注意顺序，首先初始化root，然后first level，second level，最好是leaf node。在初始化的过程中，一般会调用上节中的接口函数向系统增加irq domain。有些interrupt controller会在其driver初始化的过程中创建映射。

(4)在各个driver初始化的过程中，创建映射。

在第一节和第三节中，已经完成了(1)(2)(3)步骤，下面需要分析步骤(4)。

### 4.3硬件外设的驱动初始化过程中创建HW interrupt ID和IRQ number的映射关系

#### 4.3.1 device node转化为platform\_device

从device node转换为platform\_device的过程中从of\_platform\_populate函数开始。

of\_platform\_populate

**--->** of\_platform\_bus\_create

**--->** of\_platform\_device\_create\_pdata

**--->** of\_device\_alloc：

of\_device\_alloc分析了和interrupts相关的属性。

struct platform\_device **\***of\_device\_alloc**(**struct device\_node **\***np**,** const char **\***bus\_id**,** struct device **\***parent**)**

**{**

struct platform\_device **\***dev**;**

int rc**,** i**,** num\_reg **=** 0**,** num\_irq**;**

struct resource **\***res**,** temp\_res**;**

dev **=** platform\_device\_alloc**(**""**,** **-**1**);**

/\* count the io and irq resources \*/

**...** **...**

num\_irq **=** of\_irq\_count**(**np**);** // 统计这个节点的interrupts属性中描述了几个中断

/\* Populate the resource table \*/

**if** **(**num\_irq **||** num\_reg**)** **{**

res **=** kzalloc**(sizeof(\***res**)** **\*** **(**num\_irq **+** num\_reg**),** GFP\_KERNEL**);**

**...** **...**

dev**->**num\_resources **=** num\_reg **+** num\_irq**;**

dev**->**resource **=** res**;**

**...** **...**

of\_irq\_to\_resource\_table**(**np**,** res**,** num\_irq**)** // 解析interrupts属性，将每一个中断转化为resource结构体

**}**

**...** **...**

**}**

##### 4.3.1.1 of\_irq\_count

of\_irq\_count这个函数会解析interrupts属性，并统计其中描述了几个中断。

int of\_irq\_count**(**struct device\_node **\***dev**)**

**{**

struct of\_phandle\_args irq**;**

int nr **=** 0**;**

**while** **(**of\_irq\_parse\_one**(**dev**,** nr**,** **&**irq**)** **==** 0**)**

nr**++;**

**return** nr**;**

**}**

nr表示的是index，of\_irq\_parse\_one每次成功返回，都表示成功从interrupts属性中解析到了第nr个中断，同时将关于这个中断的信息存放到irq中，struct of\_phandle\_args的含义如下：

#define MAX\_PHANDLE\_ARGS 16

struct of\_phandle\_args **{**

struct device\_node **\***np**;** // 用于存放赋值处理这个中断的中断控制器的节点

int args\_count**;** // 就是interrupt-controller的#interrupt-cells的值

uint32\_t args**[**MAX\_PHANDLE\_ARGS**];** // 用于存放具体描述某一个中断的参数的值

**};**

最后将解析到的中断个数返回。

##### 4.3.1.2 of\_irq\_to\_resource\_table

知道interrupts中描述了几个中断后，这个函数开始将这些中断转换为resource，这个是由of\_irq\_to\_resource函数完成。

**for** **(**i **=** 0**;** i **<** nr\_irqs**;** i**++,** res**++)**

**if** **(!**of\_irq\_to\_resource**(**dev**,** i**,** res**))**

**break;**

第二个参数i表示的是index，即interrupts属性中的第i个中断。

int of\_irq\_to\_resource**(**struct device\_node **\***dev**,** int index**,** struct resource **\***r**)**

**{**

int irq **=** irq\_of\_parse\_and\_map**(**dev**,** index**);** // 返回interrupts中第index个hwirq中断映射到的virq

**if** **(**r **&&** irq**)** **{** // 将这个irq封装成resource

const char **\***name **=** **NULL;**

memset**(**r**,** 0**,** **sizeof(\***r**));**

of\_property\_read\_string\_index**(**dev**,** "interrupt-names"**,** index**,** **&**name**);** // 一般这个"interrupt-names"属性是可选的

r**->**start **=** r**->**end **=** irq**;** // 全局唯一的virq

r**->**flags **=** IORESOURCE\_IRQ **|** irqd\_get\_trigger\_type**(**irq\_get\_irq\_data**(**irq**));** // 这个中断的属性，如上升沿还是下降沿触发

r**->**name **=** name **?** name **:** of\_node\_full\_name**(**dev**);**

**}**

**return** irq**;**

**}**

这个函数的分析重点为irq\_of\_parse\_and\_map。

##### 4.3.1.3 irq\_of\_parse\_and\_map

unsigned int irq\_of\_parse\_and\_map**(**struct device\_node **\***dev**,** int index**)**

**{**

struct of\_phandle\_args oirq**;**

of\_irq\_parse\_one**(**dev**,** index**,** **&**oirq**);** // 获得interrupts的第index个中断参数，并封装到oirq中

**return** irq\_create\_of\_mapping**(&**oirq**);** //返回映射到的virq

**}**

以i2c的device node为例：

i2c**:**i2c@54000000 **{**

compatible **=** "samsung,s3c2410-i2c"**;**

reg **=** **<**0x54000000 0x100**>;**

interrupts **=** **<**0 0 27 3**>;**

#address-cells = <1>;

#size-cells = <0>;

status **=** "disabled"**;**

**}**

irq\_of\_parse\_and\_map这个函数会获得interrupts属性的第index个中断的参数(of\_irq\_parse\_one)。然后获得该中断所隶属的interrupt-controller的irq domain，利用该domain的of\_xlate函数从前面的第index个中断的参数中解析出hwirq和中断类型，最后从系统中为该hwriq分配一个全局唯一的virq，并将映射关系存放到中断控制器的irq domain中。

irq\_create\_of\_mapping代码如下：

unsigned int irq\_create\_of\_mapping**(**struct of\_phandle\_args **\***irq\_data**)**

**{**

struct irq\_fwspec fwspec**;**

//将irq\_data中的数据转存到fwspec，没必要分析

of\_phandle\_args\_to\_fwspec**(**irq\_data**,** **&**fwspec**);**

**return** irq\_create\_fwspec\_mapping**(&**fwspec**);**

**}**

继续分析irq\_create\_fwspec\_mapping。

##### 4.3.1.4 irq\_create\_fwspec\_mapping

irq\_create\_fwspec\_mapping的功能在上面已经描述了。

unsigned int irq\_create\_fwspec\_mapping**(**struct irq\_fwspec **\***fwspec**)**

**{**

struct irq\_domain **\***domain**;**

struct irq\_data **\***irq\_data**;**

irq\_hw\_number\_t hwirq**;**

unsigned int type **=** IRQ\_TYPE\_NONE**;**

int virq**;**

// 根据中断控制器的device\_node找到所对应的irq domain

**if** **(**fwspec**->**fwnode**)** **{**

domain **=** irq\_find\_matching\_fwspec**(**fwspec**,** DOMAIN\_BUS\_WIRED**);**

**}**

//调用ops->xlate函数，也就是s3c24xx\_irq\_xlate\_of

//对dts出现的中断信息类似<0 0 27 3>进行分析得到硬件中断号和中断类型

irq\_domain\_translate**(**domain**,** fwspec**,** **&**hwirq**,** **&**type**);**

**...** **...**

//从这个irq domain查询看该hwirq之前是否已经映射过，一般情况下都没有

//check hwirq is in the linear revmap

virq **=** irq\_find\_mapping**(**domain**,** hwirq**);**

**if** **(**virq**)** **{**

**...** **...**

**return** virq**;**

**...** **...**

**}**

//定义了alloc的domain来说，走这个分支

**if** **(**irq\_domain\_is\_hierarchy**(**domain**))**

**{**

virq **=** irq\_domain\_alloc\_irqs**(**domain**,** 1**,** NUMA\_NO\_NODE**,** fwspec**);**

**}**

// 其他没有定义irq\_domain->ops->alloc的domain，走这个分支

**else**

**{**

//创建映射

virq **=** irq\_create\_mapping**(**domain**,** hwirq**);**

**}**

//获取irq\_desc结构体对应的irq\_data

irq\_data **=** irq\_get\_irq\_data**(**virq**);**

//irq\_data存储trigger type

irqd\_set\_trigger\_type**(**irq\_data**,** type**);**

**return** virq**;**

**}**

##### 4.3.1.5 irq\_domain\_translate

irq\_domain\_translate解析dts中中断信息类似<0 0 27 3>进行分析得到硬件中断号和中断类型。

static int irq\_domain\_translate**(**struct irq\_domain **\***d**,** struct irq\_fwspec **\***fwspec**,** irq\_hw\_number\_t **\***hwirq**,** unsigned int **\***type**)**

**{**

**if** **(**d**->**ops**->**xlate**)**

**return** d**->**ops**->**xlate**(**d**,** to\_of\_node**(**fwspec**->**fwnode**),** fwspec**->**param**,** fwspec**->**param\_count**,** hwirq**,** type**);**

**}**

//输出硬件中断号和trigger type

static int s3c24xx\_irq\_xlate\_of**(**struct irq\_domain **\***d**,** struct device\_node **\***n**,** const u32 **\***intspec**,** unsigned int intsize**,** irq\_hw\_number\_t **\***out\_hwirq**,** unsigned int **\***out\_type**)**

**{**

struct s3c\_irq\_intc **\***intc**;**

struct s3c\_irq\_intc **\***parent\_intc**;**

struct s3c\_irq\_data **\***irq\_data**;**

struct s3c\_irq\_data **\***parent\_irq\_data**;**

int irqno**;**

//中断参数为4

**if** **(**WARN\_ON**(**intsize **<** 4**))**

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

//intspec[0]存储的是ctrl\_num，不能大于2

**if** **(**intspec**[**0**]** **>** 2 **||** **!**s3c\_intc**[**intspec**[**0**]])** **{**

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

**}**

//获得对应的s3c\_intc

intc **=** s3c\_intc**[**intspec**[**0**]];**

//以i2c为例interrupts = <0 0 27 3>

//intspec[0] = 0 intspec[2]= 27 27为i2c中断在中断寄存器的偏移bit

//intspec[3] = 3 这里的trigger type对于i2c没有作用

**\***out\_hwirq **=** intspec**[**0**]** **\*** 32 **+** intspec**[**2**];**

**\***out\_type **=** intspec**[**3**]** **&** IRQ\_TYPE\_SENSE\_MASK**;**

//如果是子中断，需要获取总中断

parent\_intc **=** intc**->**parent**;**

**if** **(**parent\_intc**)** **{**

//以uart0中断分析子中断的处理情况

//interrupts = <1 28 0 4>, <1 28 1 4>;

//uart有tx,rx两个脚，分析第一组参数

//intspec[0] = 1 表示是子中断

//intspec[1] = 28 表示子中断对应的总中断在中断寄存器的偏移bit

//intspec[2] = 0 表示子中断在子中断寄存器的偏移bit

//获取s3c\_intc[1]对应的s3c\_irq\_data结构体

irq\_data **=** **&**intc**->**irqs**[**intspec**[**2**]];**

//记录串口rx子中断对应的父中断号

irq\_data**->**parent\_irq **=** intspec**[**1**];**

//获取s3c\_intc[0]对应的父中断号的s3c\_irq\_data结构体

parent\_irq\_data **=** **&**parent\_intc**->**irqs**[**irq\_data**->**parent\_irq**];**

//设置uart父中断对应的子中断s3c\_irq\_intc结构体

parent\_irq\_data**->**sub\_intc **=** intc**;**

//标记哪些子中断申请了

parent\_irq\_data**->**sub\_bits **|=** **(**1UL **<<** intspec**[**2**]);**

//为父中断号创建映射

irqno **=** irq\_create\_mapping**(**parent\_intc**->**domain**,** intspec**[**1**]);**

//设置串行中断处理函数，也就是一个中断号中还有其他子中断号，

//需要进一步处理。设置串行时，会开启该中断源

//也就是串口中断还需要继续判断其子中断才能确定发生了什么

irq\_set\_chained\_handler**(**irqno**,** s3c\_irq\_demux**);**

**}**

**return** 0**;**

**}**

##### 4.3.1.6 irq\_create\_mapping

unsigned int irq\_create\_mapping**(**struct irq\_domain **\***domain**,** irq\_hw\_number\_t hwirq**)**

**{**

struct device\_node **\***of\_node**;**

int virq**;**

//irq\_domain对应的device node

of\_node **=** irq\_domain\_get\_of\_node**(**domain**);**

//从linear\_revmap检查该映射是否存在

virq **=** irq\_find\_mapping**(**domain**,** hwirq**);**

**if** **(**virq**)** **{**

pr\_debug**(**"-> existing mapping on virq %d\n"**,** virq**);**

**return** virq**;**

**}**

//从系统中得到一个唯一的virq

//其实就是全局变量allocated\_irqs从低位到高位第一个为0的位的位号

//然后将allocated\_irqs的第virq位置为1

//然后会为这个virq分配一个irq\_desc, virq会存放到irq\_desc的irq\_data.irq中

//最后将这个irq\_desc存放到irq\_desc\_tree中，以virq为key

//函数irq\_to\_desc就是以virq为key，查询irq\_desc\_tree迅速定位到irq\_desc

//irq\_desc在内核中有两种组织方式radix-tree方式和数组方式

virq **=** irq\_domain\_alloc\_descs**(-**1**,** 1**,** hwirq**,** of\_node\_to\_nid**(**of\_node**),** **NULL);**

//将hwirq跟virq的映射关系存放到irq domain中

//virq对应的irq\_desc的irq\_data的irq、hwirq、domain分别传入的virq、hwirq和domain

irq\_domain\_associate**(**domain**,** virq**,** hwirq**);**

**return** virq**;**

**}**

##### 4.3.1.7 irq\_domain\_associate

int irq\_domain\_associate**(**struct irq\_domain **\***domain**,** unsigned int virq**,** irq\_hw\_number\_t hwirq**)**

**{**

//获取irq\_desc中的irq\_data

struct irq\_data **\***irq\_data **=** irq\_get\_irq\_data**(**virq**);**

int ret**;**

mutex\_lock**(&**irq\_domain\_mutex**);**

//设置对应的物理中断号

irq\_data**->**hwirq **=** hwirq**;**

//设置对应的irq\_domain

irq\_data**->**domain **=** domain**;**

**if** **(**domain**->**ops**->**map**)** **{**

//调用s3c24xx\_irq\_map\_of

ret **=** domain**->**ops**->**map**(**domain**,** virq**,** hwirq**);**

/\* If not already assigned, give the domain the chip's name \*/

**if** **(!**domain**->**name **&&** irq\_data**->**chip**)**

domain**->**name **=** irq\_data**->**chip**->**name**;**

**}**

//设置硬件中断号和虚拟中断号之间的关系

**if** **(**hwirq **<** domain**->**revmap\_size**)** **{**

domain**->**linear\_revmap**[**hwirq**]** **=** virq**;**

**}**

mutex\_unlock**(&**irq\_domain\_mutex**);**

irq\_clear\_status\_flags**(**virq**,** IRQ\_NOREQUEST**);**

**return** 0**;**

**}**

static int s3c24xx\_irq\_map\_of**(**struct irq\_domain **\***h**,** unsigned int virq**,** irq\_hw\_number\_t hw**)**

**{**

unsigned int ctrl\_num **=** hw **/** 32**;**

unsigned int intc\_hw **=** hw **%** 32**;**

struct s3c\_irq\_intc **\***intc **=** s3c\_intc**[**ctrl\_num**];**

struct s3c\_irq\_intc **\***parent\_intc **=** intc**->**parent**;**

struct s3c\_irq\_data **\***irq\_data **=** **&**intc**->**irqs**[**intc\_hw**];**

//s3c\_irq\_data对应的中断控制器

irq\_data**->**intc **=** intc**;**

//s3c\_irq\_data对应是物理中断号

irq\_data**->**offset **=** intc\_hw**;**

//设置虚拟中断号对应的chip和回调函数

**if** **(!**parent\_intc**)**

irq\_set\_chip\_and\_handler**(**virq**,** **&**s3c\_irq\_chip**,** handle\_edge\_irq**);**

**else**

irq\_set\_chip\_and\_handler**(**virq**,** **&**s3c\_irq\_level\_chip**,** handle\_edge\_irq**);**

//irq\_desc->irq\_data->chip\_data设置为s3c\_irq\_data

irq\_set\_chip\_data**(**virq**,** irq\_data**);**

**return** 0**;**

**}**

#### 4.3.2 中断处理函数的设置

对于中断处理函数，在驱动中都有通用的处理函数，用户自定义的处理函数通过request\_irq进行添加。

在上面使用了如下两个函数添加中断的通用处理函数：

irq\_set\_chip\_and\_handler

irq\_set\_chained\_handler

irq\_set\_chip\_and\_handler代码如下：

static inline void irq\_set\_chip\_and\_handler**(**unsigned int irq**,** struct irq\_chip **\***chip**,**irq\_flow\_handler\_t handle**)**

**{**

irq\_set\_chip\_and\_handler\_name**(**irq**,** chip**,** handle**,** **NULL);**

**}**

void irq\_set\_chip\_and\_handler\_name**(**unsigned int irq**,** struct irq\_chip **\***chip**,** irq\_flow\_handler\_t handle**,** const char **\***name**)**

**{**

irq\_set\_chip**(**irq**,** chip**);**

\_\_irq\_set\_handler**(**irq**,** handle**,** 0**,** name**);**

**}**

irq\_set\_chained\_handler代码如下：

static inline void irq\_set\_chained\_handler**(**unsigned int irq**,** irq\_flow\_handler\_t handle**)**

**{**

\_\_irq\_set\_handler**(**irq**,** handle**,** 1**,** **NULL);**

**}**

设置中断handler调用的都是\_\_irq\_set\_handler函数。

irq\_set\_chip函数代码如下：

int irq\_set\_chip**(**unsigned int irq**,** struct irq\_chip **\***chip**)**

**{**

unsigned long flags**;**

struct irq\_desc **\***desc **=** irq\_get\_desc\_lock**(**irq**,** **&**flags**,** 0**);**

desc**->**irq\_data**.**chip **=** chip**;**

irq\_put\_desc\_unlock**(**desc**,** flags**);**

//在allocated\_irqs中标记被使用了

irq\_mark\_irq**(**irq**);**

**return** 0**;**

**}**

irq\_chip在这里的定义如下：

//定义了一些中断操作的一些常用的函数

//s3c\_irq\_ack用于响应中断，通常是清除当前中断使得可以接收下一个中断。

//s3c\_irq\_mask通过操作INTMSK用于屏蔽对应的中断。

//s3c\_irq\_unmask操作INTMSK打开中断使能

//s3c\_irq\_wake用于设置硬件唤醒终端源

//s3c\_irq\_type用于设置不同的中断类型对应的通用的中断函数

static struct irq\_chip s3c\_irq\_chip **=** **{**

**.**name **=** "s3c"**,**

**.**irq\_ack **=** s3c\_irq\_ack**,**

**.**irq\_mask **=** s3c\_irq\_mask**,**

**.**irq\_unmask **=** s3c\_irq\_unmask**,**

**.**irq\_set\_type **=** s3c\_irq\_type**,**

**.**irq\_set\_wake **=** s3c\_irq\_wake

**};**

static struct irq\_chip s3c\_irq\_level\_chip **=** **{**

**.**name **=** "s3c-level"**,**

**.**irq\_mask **=** s3c\_irq\_mask**,**

**.**irq\_unmask **=** s3c\_irq\_unmask**,**

**.**irq\_ack **=** s3c\_irq\_ack**,**

**.**irq\_set\_type **=** s3c\_irq\_type**,**

**};**

\_\_irq\_set\_handler设置通用的中断处理函数：

void \_\_irq\_set\_handler**(**unsigned int irq**,** irq\_flow\_handler\_t handle**,** int is\_chained**,** const char **\***name**)**

**{**

unsigned long flags**;**

struct irq\_desc **\***desc **=** irq\_get\_desc\_buslock**(**irq**,** **&**flags**,** 0**);**

\_\_irq\_do\_set\_handler**(**desc**,** handle**,** is\_chained**,** name**);**

irq\_put\_desc\_busunlock**(**desc**,** flags**);**

**}**

static void \_\_irq\_do\_set\_handler**(**struct irq\_desc **\***desc**,** irq\_flow\_handler\_t handle**,** int is\_chained**,** const char **\***name**)**

**{**

struct irq\_data **\***irq\_data **=** **&**desc**->**irq\_data**;**

desc**->**handle\_irq **=** handle**;**

desc**->**name **=** name**;**

//如果该中断下还有子中断，则需要默认打开这个当前这个父中断

**if** **(**handle **!=** handle\_bad\_irq **&&** is\_chained**)** **{**

//从state\_use\_accessors中获取trigger type

unsigned int type **=** irqd\_get\_trigger\_type**(&**desc**->**irq\_data**);**

//调用irq\_set\_type函数，对于非外部中断，这里没有什么作用

//对于外部中断，会设置外部中断的trigger type寄存器

**if** **(**type **!=** IRQ\_TYPE\_NONE**)** **{**

\_\_irq\_set\_trigger**(**desc**,** type**);**

desc**->**handle\_irq **=** handle**;**

**}**

//设置该中断不被发现

irq\_settings\_set\_noprobe**(**desc**);**

irq\_settings\_set\_norequest**(**desc**);**

irq\_settings\_set\_nothread**(**desc**);**

//设置一个默认的action，因为该父中断用户不会去设置处理函数

//设置没有作用，要设置对应的子中断

desc**->**action **=** **&**chained\_action**;**

//调用irq\_umask使能中断

irq\_startup**(**desc**,** **true);**

**}**

**}**

## 5.中断的处理流程

### 5.1 中断的入口处理函数

从具体CPU的中断处理到machine相关的处理模块，代码如下：

**.**macro    irq\_handler

#ifdef CONFIG\_MULTI\_IRQ\_HANDLER

    ldr    r1**,** **=**handle\_arch\_irq

    mov    r0**,** sp

    adr    lr**,** BSYM**(**9997f**)**

    ldr    pc**,** **[**r1**]**

#else

    arch\_irq\_handler\_default

#endif

在上面的代码中，如果配置了MULTI\_IRQ\_HANDLER的话，ARM中断处理则直接跳转到一个叫做handle\_arch\_irq函数。

在前面的s3c\_init\_intc\_of代码中，对handle\_arch\_irq进行设置。

set\_handle\_irq**(**s3c24xx\_handle\_irq**);**

void \_\_init set\_handle\_irq**(**void **(\***handle\_irq**)(**struct pt\_regs **\*))**

**{**

**if** **(**handle\_arch\_irq**)**

**return;**

handle\_arch\_irq **=** handle\_irq**;**

**}**

s3c24xx\_handle\_irq的处理流程如下：

s3c24xx\_handle\_irq

s3c24xx\_handle\_intc

handle\_domain\_irq

\_\_handle\_domain\_irq

generic\_handle\_irq

generic\_handle\_irq\_desc

最终调用的是虚拟irq号对应的irq\_desc的handle\_irq回调。

static inline void generic\_handle\_irq\_desc**(**struct irq\_desc **\***desc**)**

**{**

desc**->**handle\_irq**(**desc**);**

**}**

### 5.2 platform\_get\_irq 和request\_irq

用户在dts中需要使用中断时，需要先调用platform\_get\_irq得到虚拟irq号，再调用request\_irq申请中断并添加中断函数。

platform\_get\_irq代码如下：

//得到第num块flags为 IORESOURCE\_IRQ 的资源

//从resource->start得到对应的虚拟irq号

int platform\_get\_irq**(**struct platform\_device **\***dev**,** unsigned int num**)**

**{**

struct resource **\***r**;**

r **=** platform\_get\_resource**(**dev**,** IORESOURCE\_IRQ**,** num**);**

**return** r **?** r**->**start **:** **-**ENXIO**;**

**}**

request\_irq会对trigger type进行设置，打开中断，设置中断回调。

static inline int \_\_must\_check request\_irq**(**unsigned int irq**,** irq\_handler\_t handler**,** unsigned long flags**,** const char **\***name**,** void **\***dev**)**

**{**

**return** request\_threaded\_irq**(**irq**,** handler**,** **NULL,** flags**,** name**,** dev**);**

**}**

可以看到request\_irq实际操作函数为request\_threaded\_irq，那么thread的意思是什么呢？

在linux里，中断处理分为顶半（top half），底半(bottom half)，在顶半里处理优先级比较高的事情，要求占用中断时间尽量的短，在处理完成后，就激活底半，有底半处理其余任务。底半的处理方式主要有soft\_irq, tasklet, workqueue三种，他们在使用方式和适用情况上各有不同。soft\_irq用在对底半执行时间要求比较紧急或者非常重要的场合，主要为一些subsystem用，一般driver基本上用不上。 tasklet和work queue在普通的driver里用的相对较多，主要区别是tasklet是在中断上下文执行,而work queue是在process上下文，因此可以执行可能sleep的操作。

request\_threaded\_irq()是Linux kernel 2.6.30之后新加的irq handler API，从realtime tree移植而来，为了减少kernel 因为要等待每一个硬件中断处理的时间 ，就另外交给kernel thread 处理中断后续工作。

优点： 1 减少 kernel 延迟时间

         2 避免处理中断时要分辨是在硬体中断或软体中断？

         3 更容易为kernel 中断处理除错，可能可完全取代tasklet

跟传统top/bottom havles的差异是threaded\_irq受Linux kernel system的process scheduling控制，不会因为写错的bottom half代码造成整个系统延迟的问题。

request\_threaded\_irq函数如下：

int request\_threaded\_irq**(**unsigned int irq**,** irq\_handler\_t handler**,** irq\_handler\_t thread\_fn**,** unsigned long irqflags**,**const char **\***devname**,** void **\***dev\_id**)**

**{**

struct irqaction **\***action**;**

struct irq\_desc **\***desc**;**

//获取对应的irq\_desc结构体

desc **=** irq\_to\_desc**(**irq**);**

//判断该irq是否能被request

irq\_settings\_can\_request**(**desc**);**

action **=** kzalloc**(sizeof(**struct irqaction**),** GFP\_KERNEL**);**

action**->**handler **=** handler**;**

action**->**thread\_fn **=** thread\_fn**;**

action**->**flags **=** irqflags**;**

action**->**name **=** devname**;**

action**->**dev\_id **=** dev\_id**;**

//设置action

\_\_setup\_irq**(**irq**,** desc**,** action**);**

**}**

\_\_setup\_irq**(**unsigned int irq**,** struct irq\_desc **\***desc**,** struct irqaction **\*new)**

**{**

**new->**irq **=** irq**;**

**if** **(!(new->**flags **&** IRQF\_TRIGGER\_MASK**))**

**new->**flags **|=** irqd\_get\_trigger\_type**(&**desc**->**irq\_data**);**

//如果一个中断描述符是nested thread type的，

//说明这个中断描述符应该设定threaded interrupt handler

nested **=** irq\_settings\_is\_nested\_thread**(**desc**);**

**if** **(**nested**)**

**{**

**if** **(!new->**thread\_fn**)** **{**

         ret **=** **-**EINVAL**;**

**goto** out\_mput**;**

**}**

**new->**handler **=** irq\_nested\_primary\_handler**;**

**}**

//创建中断线程

//new->thread=t

**if** **(new->**thread\_fn **&&** **!**nested**)** **{**

ret **=** setup\_irq\_thread**(new,** irq**,** **false);**

//如果以前已经设置了中断处理函数

struct irqaction **\***old**,** **\*\***old\_ptr**;**

old\_ptr **=** **&**desc**->**action**;**

old **=** **\***old\_ptr**;**

**if(**old**)**

**{**

//old指向注册之前的action list，如果不是NULL，

//那么说明需要共享interrupt line。但是如果要共享

//需要每一个irqaction都同意共享（IRQF\_SHARED）

//每一个irqaction的触发方式相同（都是level trigger或者都是edge trigger）

//相同的oneshot类型的中断（都是one shot或者都不是）

**if** **(!((**old**->**flags **&** **new->**flags**)** **&** IRQF\_SHARED**)** **||**

**((**old**->**flags **^** **new->**flags**)** **&** IRQF\_TRIGGER\_MASK**)** **||**

**((**old**->**flags **^** **new->**flags**)** **&** IRQF\_ONESHOT**))**

**goto** mismatch**;**

//找到action队列的最后一个数据

**do**

**{**

thread\_mask **|=** old**->**thread\_mask**;**

old\_ptr **=** **&**old**->**next**;**

old **=** **\***old\_ptr**;**

**}** **while** **(**old**);**

shared **=** 1**;**

**}**

//one shot本身的意思的只有一次的，结合到中断这个场景，则表示中断是一次性触发的，不能嵌套。对于primary handler，当然是不会嵌套，但是对于threaded interrupt handler我们有两种选择，一种是mask该interrupt source，另外一种是unmask该interrupt source。一旦mask住该interrupt source，那么该interrupt source的中断在整个threaded interrupt handler处理过程中都是不会再次触发的，也就是one shot了。这种handler不需要考虑重入问题。

//具体是否要设定one shot的flag是和硬件系统有关的，我们举一个例子，比如电池驱动，电池里面有一个电量计，是使用HDQ协议进行通信的，电池驱动会注册一个threaded interrupt handler，在这个handler中，会通过HDQ协议和电量计进行通信。对于这个handler，通过HDQ进行通信是需要一个完整的HDQ交互过程，如果中间被打断，整个通信过程会出问题，因此，这个handler就必须是one shot的。

**if** **(new->**flags **&** IRQF\_ONESHOT**)**

**{**

**if** **(**thread\_mask **==** **~**0UL**)** **{**

ret **=** **-**EBUSY**;**

**goto** out\_mask**;**

**}**

**new->**thread\_mask **=** 1 **<<** ffz**(**thread\_mask**);**

**}**

**if** **(!**shared**)**

**{**

//调用irq\_desc->irq\_chip->irq\_request\_resources函数，这里为空

irq\_request\_resources**(**desc**);**

init\_waitqueue\_head**(&**desc**->**wait\_for\_threads**);**

//设置中断trigger type

**if** **(new->**flags **&** IRQF\_TRIGGER\_MASK**)** **{**

ret **=** \_\_irq\_set\_trigger**(**desc**,** **new->**flags **&** IRQF\_TRIGGER\_MASK**);**

**}**

//打开中断

**if** **(**irq\_settings\_can\_autoenable**(**desc**))**

irq\_startup**(**desc**,** **true);**

**}**

**...**

**}**

hardirq和thread\_fn同时出现时，处理thread\_fn时该中断是打开的。

request\_threaded\_irq**(**gpio\_irq**.**irq**,**gpio\_irqhandler**,**gpio\_threadhandler**,**gpio\_irq**.**flags**,**gpio\_irq**.**name**,(**void**\*)**0**);**

但hardirq和thread\_fn只有一个存在时，处理thread\_fn时，中断是关闭的。

request\_threaded\_irq**(**gpio\_irq**.**irq**,NULL,**gpio\_threadhandler**,**gpio\_irq**.**flags**,**gpio\_irq**.**name**,(**void**\*)**0**);**

关于IRQF\_ONESHOT, 直到线程函数执行完毕才会开启该中断。

### 5.3 中断处理流程

这里举两个例子说明级联中断和非级联中断的处理流程。

#### 5.3.1 非级联中断

以i2c中断为例interrupts = <0 0 27 3>。

当i2c中断发生时，其虚拟中断号对应的irq\_desc结构体的handle\_irq函数调用

desc**->**handle\_irq**(**desc**);**

对于非级联中断，handle\_irq为handle\_edge\_irq，handle\_edge\_irq中断分析见2.6文章。

handle\_edge\_irq中会调用request\_irq注册的回调。

#### 5.3.2 级联中断

以uart中断为例interrupts = <1 28 0 4>, <1 28 1 4>。

级联中断意味着当中断发生时，需要从父中断开始，再从子中断寻找到底是哪个子中断发生了。

当串口的rx中断产生时，其父中断串口中断产生，父中断的处理函数为s3c\_irq\_demux。

static void s3c\_irq\_demux**(**struct irq\_desc **\***desc**)**

**{**

struct irq\_chip **\***chip **=** irq\_desc\_get\_chip**(**desc**);**

struct s3c\_irq\_data **\***irq\_data **=** irq\_desc\_get\_chip\_data**(**desc**);**

struct s3c\_irq\_intc **\***intc **=** irq\_data**->**intc**;**

struct s3c\_irq\_intc **\***sub\_intc **=** irq\_data**->**sub\_intc**;**

unsigned int n**,** offset**,** irq**;**

unsigned long src**,** msk**;**

//这里为32

offset **=** irq\_domain\_get\_of\_node**(**intc**->**domain**)** **?** 32 **:** 0**;**

chained\_irq\_enter**(**chip**,** desc**);**

//读取Sub Source Pending寄存器判断哪些子中断发生了

src **=** readl\_relaxed**(**sub\_intc**->**reg\_pending**);**

//读取Interrupt Sub Mask寄存器

msk **=** readl\_relaxed**(**sub\_intc**->**reg\_mask**);**

//屏蔽的子中断不需要处理

src **&=** **~**msk**;**

src **&=** irq\_data**->**sub\_bits**;**

//转到子中断处理函数中

**while** **(**src**)** **{**

//获取子中断号

n **=** \_\_ffs**(**src**);**

src **&=** **~(**1 **<<** n**);**

//从子中断控制器中获取对应的虚拟中断号

irq **=** irq\_find\_mapping**(**sub\_intc**->**domain**,** offset **+** n**);**

//调用irq\_desc->handle\_irq

generic\_handle\_irq**(**irq**);**

**}**

chained\_irq\_exit**(**chip**,** desc**);**

**}**

父中断处理完后，继续调用子中断的handle\_edge\_irq函数。

## 6.外部中断处理

gpf和gpg的gpio拥有用外部中断的能力，并且外部中断由主中断和外部子中断两部分共同处理，所以需要为gpf和gpg两个外部中断控制器创建irq\_domain。

### 6.1 外部中断DTS描述

pinctrl\_0**:** pinctrl@56000000 **{**

reg **=** **<**0x56000000 0x1000**>;**

wakeup**-**interrupt**-**controller **{**

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

//6个外部中断的信息描述

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**>;**

**};**

**};**

### 6.2 s3c24xx\_eint\_init

在pinctrl子系统的初始化过程中调用了eint\_wkup\_init函数。

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**,**

**},**

**};**

eint\_wkup\_init也就是s3c24xx\_eint\_init。

//所有的外部中断数据

struct s3c24xx\_eint\_data **{**

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

//总共有24个外部中断，记录每个外部中断对应的irq\_domain

struct irq\_domain **\***domains**[**NUM\_EINT**];**

//外部中断在主中断的虚拟中断号

int parents**[**NUM\_EINT\_IRQ**];**

**};**

//每个bank对应一个该结构体

struct s3c24xx\_eint\_domain\_data **{**

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

struct s3c24xx\_eint\_data **\***eint\_data**;**

bool eint0\_3\_parent\_only**;**

**};**

static const struct of\_device\_id s3c24xx\_eint\_irq\_ids**[]** **=** **{**

**{** **.**compatible **=** "samsung,s3c2410-wakeup-eint"**,** **.**data **=** **(**void **\*)**1 **},**

**{** **.**compatible **=** "samsung,s3c2412-wakeup-eint"**,** **.**data **=** **(**void **\*)**0 **},**

**{** **}**

**};**

static irq\_flow\_handler\_t s3c2410\_eint\_handlers**[**NUM\_EINT\_IRQ**]** **=** **{**

s3c2410\_demux\_eint0\_3**,**

s3c2410\_demux\_eint0\_3**,**

s3c2410\_demux\_eint0\_3**,**

s3c2410\_demux\_eint0\_3**,**

s3c24xx\_demux\_eint4\_7**,**

s3c24xx\_demux\_eint8\_23**,**

**};**

static int s3c24xx\_eint\_init**(**struct samsung\_pinctrl\_drv\_data **\***d**)**

**{**

struct device **\***dev **=** d**->**dev**;**

const struct of\_device\_id **\***match**;**

struct device\_node **\***eint\_np **=** **NULL;**

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

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

struct s3c24xx\_eint\_data **\***eint\_data**;**

const struct irq\_domain\_ops **\***ops**;**

unsigned int i**;**

bool eint0\_3\_parent\_only**;**

irq\_flow\_handler\_t **\***handlers**;**

//遍历pinctrl下的子节点

for\_each\_child\_of\_node**(**dev**->**of\_node**,** np**)** **{**

//寻找到samsung,s3c2410-wakeup-eint

match **=** of\_match\_node**(**s3c24xx\_eint\_irq\_ids**,** np**);**

**if** **(**match**)** **{**

//获取对应的device\_node

eint\_np **=** np**;**

//表示eint0/1/2/3是非级联的

eint0\_3\_parent\_only **=** **(**bool**)**match**->**data**;**

**break;**

**}**

**}**

//分配空间创建s3c24xx\_eint\_data结构体

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

//设置drvdata

eint\_data**->**drvdata **=** d**;**

//设置中断处理函数

handlers **=** eint0\_3\_parent\_only **?** s3c2410\_eint\_handlers**:** s3c2412\_eint\_handlers**;**

**for** **(**i **=** 0**;** i **<** NUM\_EINT\_IRQ**;** **++**i**)** **{**

unsigned int irq**;**

//得到外部中断在主中断主的虚拟irq号

irq **=** irq\_of\_parse\_and\_map**(**eint\_np**,** i**);**

eint\_data**->**parents**[**i**]** **=** irq**;**

//设置外部中断都为级联，并设置处理函数

//设置对应的irq\_desc->irq\_common\_data.handler\_data

irq\_set\_chained\_handler\_and\_data**(**irq**,** handlers**[**i**],** eint\_data**);**

**}**

//遍历pin bank

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

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

struct s3c24xx\_eint\_domain\_data **\***ddata**;**

unsigned int mask**;**

unsigned int irq**;**

unsigned int pin**;**

//只有gpf gpg才可以作为外部中断使用

**if** **(**bank**->**eint\_type **!=** EINT\_TYPE\_WKUP**)**

**continue;**

//分配并创建s3c24xx\_eint\_domain\_data结构体

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

//设置s3c24xx\_eint\_domain\_data

ddata**->**bank **=** bank**;**

ddata**->**eint\_data **=** eint\_data**;**

ddata**->**eint0\_3\_parent\_only **=** eint0\_3\_parent\_only**;**

//设置irq\_domain的callback函数

//根据eint\_offset判断是gpf还是gpg

//gpf的eint\_offset为0 gpg的eint\_offset为8，因为gpg从EINT8开始

ops **=** **(**bank**->**eint\_offset **==** 0**)** **?** **&**s3c24xx\_gpf\_irq\_ops **:** **&**s3c24xx\_gpg\_irq\_ops**;**

//由于gpf gpg算两个中断控制器，所以需要创建两个irq\_domain。

bank**->**irq\_domain **=** irq\_domain\_add\_linear**(**bank**->**of\_node**,** bank**->**nr\_pins**,** ops**,** ddata**);**

//每个bank的irq起始值

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

//哪些gpio具备外部中断能力

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

**for** **(**pin **=** 0**;** mask**;** **++**pin**,** mask **>>=** 1**)** **{**

**if** **(**irq **>=** NUM\_EINT**)**

**break;**

**if** **(!(**mask **&** 1**))**

**continue;**

//为每个外部中断记录对应的irq\_domain，也就是中断控制器信息

eint\_data**->**domains**[**irq**]** **=** bank**->**irq\_domain**;**

**++**irq**;**

**}**

**}**

**return** 0**;**

**}**

### 6.3 如何使用外部中断

在dts中描述某个外部中断：

interrupt**-**parent **=** **<&**gpf**>;**

interrupts **=** **<**7 IRQ\_TYPE\_EDGE\_RISING**>;**

同样的上面信息的转换需要经过4.3的步骤，部分关键且差异化的步骤如下：

(1)首先将dts中的信息转换为of\_phandle\_args结构体。

#define MAX\_PHANDLE\_ARGS 16

struct of\_phandle\_args **{**

struct device\_node **\***np**;** // 用于存放赋值处理这个中断的中断控制器的节点

int args\_count**;** // 就是interrupt-controller的#interrupt-cells的值

uint32\_t args**[**MAX\_PHANDLE\_ARGS**];** // 用于存放具体描述某一个中断的参数的值

**};**

(2)然后调用irq\_domain\_translate对of\_phandle\_args进行转换，得到硬件中断号和中断类型。

gpf和gpg对应的转换函数如下：

static const struct irq\_domain\_ops s3c24xx\_gpf\_irq\_ops **=** **{**

**.**map **=** s3c24xx\_gpf\_irq\_map**,**

**.**xlate **=** irq\_domain\_xlate\_twocell**,**

**};**

static const struct irq\_domain\_ops s3c24xx\_gpg\_irq\_ops **=** **{**

**.**map **=** s3c24xx\_gpg\_irq\_map**,**

**.**xlate **=** irq\_domain\_xlate\_twocell**,**

**};**

irq\_domain\_xlate\_twocell函数定义如下：

int irq\_domain\_xlate\_twocell**(**struct irq\_domain **\***d**,** struct device\_node **\***ctrlr**,**const u32 **\***intspec**,** unsigned int intsize**,** irq\_hw\_number\_t **\***out\_hwirq**,** unsigned int **\***out\_type**)**

**{**

**if** **(**WARN\_ON**(**intsize **<** 2**))**

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

**\***out\_hwirq **=** intspec**[**0**];**

**\***out\_type **=** intspec**[**1**]** **&** IRQ\_TYPE\_SENSE\_MASK**;**

**return** 0**;**

**}**

(3)接下来irq\_create\_mapping会根据硬件中断号分配虚拟中断号。

(4)调用domain->ops->map设置虚拟中断号对应的处理函数，这里以s3c24xx\_gpf\_irq\_map为例。

static int s3c24xx\_gpf\_irq\_map**(**struct irq\_domain **\***h**,** unsigned int virq**,** irq\_hw\_number\_t hw**)**

**{**

struct s3c24xx\_eint\_domain\_data **\***ddata **=** h**->**host\_data**;**

struct samsung\_pin\_bank **\***bank **=** ddata**->**bank**;**

//当中断号<3时

**if** **(**hw **<=** 3**)** **{**

**if** **(**ddata**->**eint0\_3\_parent\_only**)**

irq\_set\_chip\_and\_handler**(**virq**,** **&**s3c2410\_eint0\_3\_chip**,** handle\_edge\_irq**);**

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

irq\_set\_chip\_and\_handler**(**virq**,** **&**s3c24xx\_eint\_chip**,** handle\_edge\_irq**);**

**}**

irq\_set\_chip\_data**(**virq**,** bank**);**

**return** 0**;**

**}**

### 6.4 外部中断处理流程

当一个外部中断发生时，我们分级联和非级联两种方式进行分析。

#### 6.4.1 非级联的外部中断EINT0~3

当EINT0~3中断发生时，不需要去子中断继续确认信息。

当EINT0~3中断发生时，会执行到s3c2410\_demux\_eint0\_3中。

static void s3c2410\_demux\_eint0\_3**(**struct irq\_desc **\***desc**)**

**{**

struct irq\_data **\***data **=** irq\_desc\_get\_irq\_data**(**desc**);**

struct s3c24xx\_eint\_data **\***eint\_data **=** irq\_desc\_get\_handler\_data**(**desc**);**

unsigned int virq**;**

//获取外部物理中断号(例如0~3)对应的虚拟中断号

virq **=** irq\_linear\_revmap**(**eint\_data**->**domains**[**data**->**hwirq**],** data**->**hwirq**);**

//调用handle\_edge\_irq

generic\_handle\_irq**(**virq**);**

**}**

#### 6.4.2 级联的外部中断EINT4~7/EINT8~23

static void s3c24xx\_demux\_eint4\_7**(**struct irq\_desc **\***desc**)**

**{**

s3c24xx\_demux\_eint**(**desc**,** 0**,** 0xf0**);**

**}**

static void s3c24xx\_demux\_eint8\_23**(**struct irq\_desc **\***desc**)**

**{**

s3c24xx\_demux\_eint**(**desc**,** 8**,** 0xffff00**);**

**}**

static inline void s3c24xx\_demux\_eint**(**struct irq\_desc **\***desc**,** u32 offset**,** u32 range**)**

**{**

struct s3c24xx\_eint\_data **\***data **=** irq\_desc\_get\_handler\_data**(**desc**);**

struct irq\_chip **\***chip **=** irq\_desc\_get\_chip**(**desc**);**

struct samsung\_pinctrl\_drv\_data **\***d **=** data**->**drvdata**;**

unsigned int pend**,** mask**;**

chained\_irq\_enter**(**chip**,** desc**);**

//获取外部子中断寄存器哪些中断发生了

pend **=** readl**(**d**->**virt\_base **+** EINTPEND\_REG**);**

mask **=** readl**(**d**->**virt\_base **+** EINTMASK\_REG**);**

pend **&=** **~**mask**;**

pend **&=** range**;**

//逐一查找发生的外部中断号

**while** **(**pend**)** **{**

unsigned int virq**,** irq**;**

irq **=** \_\_ffs**(**pend**);**

pend **&=** **~(**1 **<<** irq**);**

//从gpf gpg的irq\_domain中获取对应的外部中断虚拟中断号

//gpf管理的是0~7.gpg管理的是8~23

//所以gpg的offset为8

virq **=** irq\_linear\_revmap**(**data**->**domains**[**irq**],** irq **-** offset**);**

//调用handle\_edge\_irq

generic\_handle\_irq**(**virq**);**

**}**

chained\_irq\_exit**(**chip**,** desc**);**

**}**

### 6.5 gpio\_to\_irq

有时候在gpio子系统中，需要通过gpio number获取对应的irq中断号，这个函数就是gpio\_to\_irq。

static inline int \_\_gpio\_to\_irq**(**unsigned gpio**)**

**{**

**return** gpiod\_to\_irq**(**gpio\_to\_desc**(**gpio**));**

**}**

int gpiod\_to\_irq**(**const struct gpio\_desc **\***desc**)**

**{**

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

int offset**;**

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

offset **=** gpio\_chip\_hwgpio**(**desc**);**

**if** **(**chip**->**to\_irq**)** **{**

int retirq **=** chip**->**to\_irq**(**chip**,** offset**);**

**return** retirq**;**

**}**

**}**

static int samsung\_gpio\_to\_irq**(**struct gpio\_chip **\***gc**,** unsigned offset**)**

**{**

struct samsung\_pin\_bank **\***bank **=** gpiochip\_get\_data**(**gc**);**

unsigned int virq**;**

virq **=** irq\_create\_mapping**(**bank**->**irq\_domain**,** offset**);**

**return** **(**virq**)** **?** **:** **-**ENXIO**;**

**}**