# 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。

#### 4.3.2 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。