# 1 内核对设备树的处理

bootloader启动内核时,会设置r0,r1,r2三个寄存器,

r0一般设置为0;

r1一般设置为machine id (在使用设备树时该参数没有被使用);

r2一般设置ATAGS或DTB的开始地址

/\*把bootloader传来的r1值, 赋给了C变量: \_\_machine\_arch\_type

把bootloader传来的r2值, 赋给了C变量: \_\_atags\_pointer // dtb首地址\*/

bootloader给内核传递的参数时有2种方法：ATAGS 或 DTB

## 1.1内核参数处理流程DTB方法

head.S

a. \_\_lookup\_processor\_type : 使用汇编指令读取CPU ID, 根据该ID找到对应的proc\_info\_list结构体(里面含有这类CPU的初始化函数、信息)

b. \_\_vet\_atags : 判断是否存在可用的ATAGS或DTB

c. \_\_create\_page\_tables : 创建页表, 即创建虚拟地址和物理地址的映射关系

d. \_\_enable\_mmu : 使能MMU, 以后就要使用虚拟地址了

e. \_\_mmap\_switched : 上述函数里将会调用\_\_mmap\_switched

f. 把bootloader传入的r2参数, 保存到变量\_\_atags\_pointer中

g. 调用C函数start\_kernel

### **1.1.1 设备树中平台信息的处理**

//imx6q-arm2.dts

/ {

model = "Freescale i.MX6 Quad Armadillo2 Board";

compatible = "fsl,imx6q-arm2", "fsl,imx6q";//表示支持的板子

chosen {//bootargs和uboot中命令行参数作用一样

bootargs = "console=ttymxc0,115200 root=/dev/mmcblk3p3 rootwait";

};

/\*mach-imx6q.c

内核中有多个machine\_desc,其中有dt\_compat成员, 它指向一个字符串数组, 里面表示该machine\_desc支持哪些单板\*/

static const char \*imx6q\_dt\_compat[] \_\_initdata = {

"fsl,imx6q-arm2",//优先被选中

"fsl,imx6q-sabrelite",

"fsl,imx6q",

NULL,

};

DT\_MACHINE\_START(IMX6Q, "Freescale i.MX6 Quad (Device Tree)")

.init\_machine = imx6q\_init\_machine,

.dt\_compat = imx6q\_dt\_compat,

MACHINE\_END

/\*使用compatile属性的值, 跟每一个machine\_desc.dt\_compat比较, compatile属性值匹配，对应的machine\_desc即被选中，多个匹配则dt\_compat数组下标最低的优先被选中\*/

/\*main.c linux-3.4.2\init \*/

start\_kernel

setup\_arch(&command\_line); // arch/arm/kernel/setup.c

mdesc = setup\_machine\_fdt(\_\_atags\_pointer); // arch/arm/kernel/devtree.c

devtree = phys\_to\_virt(dt\_phys);//物理地址转换为虚拟地址

initial\_boot\_params = devtree;

// 找到最匹配的DT\_MACHINE\_START ，即machine\_desc, drivers/of/ftd.c

for\_each\_machine\_desc(mdesc) { //匹配的

score = of\_flat\_dt\_match(dt\_root, mdesc->dt\_compat)

return of\_fdt\_match(initial\_boot\_params, node, compat);

if (score > 0 && score < mdesc\_score) {

mdesc\_best = mdesc;

mdesc\_score = score;

}

}

of\_scan\_flat\_dt(early\_init\_dt\_scan\_chosen, boot\_command\_line);

of\_scan\_flat\_dt(early\_init\_dt\_scan\_root, NULL);

of\_scan\_flat\_dt(early\_init\_dt\_scan\_memory, NULL);

return mdesc\_best

machine\_desc = mdesc;//都是machine\_desc类型指针

### 1.1.2设备树中运行时配置信息的处理

/\*chosen节点中bootargs属性的值, 存入全局变量： boot\_command\_line\*/

of\_scan\_flat\_dt(early\_init\_dt\_scan\_chosen, boot\_command\_line);

/\*确定根节点的这2个属性的值: #address-cells, #size-cells存入全局变量: dt\_root\_addr\_cells, dt\_root\_size\_cells\*/

of\_scan\_flat\_dt(early\_init\_dt\_scan\_root, NULL);

/\*解析/memory中的reg属性, 提取出"base, size", 最终调用memblock\_add(base, size);\*/

of\_scan\_flat\_dt(early\_init\_dt\_scan\_memory, NULL);

**device\_node转换为platform\_device**

device\_node构成一棵树, 根节点为: of\_root

dts -> dtb -> device\_node -> platform\_device

**jz2440 最小的dts文件**

#define S3C2410\_GPA(\_nr) ((0<<16) + (\_nr))

#define S3C2410\_GPB(\_nr) ((1<<16) + (\_nr))

#define S3C2410\_GPC(\_nr) ((2<<16) + (\_nr))

#define S3C2410\_GPD(\_nr) ((3<<16) + (\_nr))

#define S3C2410\_GPE(\_nr) ((4<<16) + (\_nr))

#define S3C2410\_GPF(\_nr) ((5<<16) + (\_nr))

#define S3C2410\_GPG(\_nr) ((6<<16) + (\_nr))

#define S3C2410\_GPH(\_nr) ((7<<16) + (\_nr))

#define S3C2410\_GPJ(\_nr) ((8<<16) + (\_nr))

#define S3C2410\_GPK(\_nr) ((9<<16) + (\_nr))

#define S3C2410\_GPL(\_nr) ((10<<16) + (\_nr))

#define S3C2410\_GPM(\_nr) ((11<<16) + (\_nr))

/dts-v1/;

/ {

model = "SMDK24440";

compatible = "samsung,smdk2440";

#address-cells = <1>;

#size-cells = <1>;

memory@30000000 {

device\_type = "memory";

reg = <0x30000000 0x4000000>;

};

/\*

cpus {

cpu {

compatible = "arm,arm926ej-s";

};

};

\*/

chosen {

bootargs = "noinitrd root=/dev/mtdblock4 rw init=/linuxrc console=ttySAC0,115200";

};

led {

compatible = "jz2440\_led";

pin = <S3C2410\_GPF(6)>;

};

};

**在根文件系统中查看设备树**

1、// 以目录结构程现的dtb文件, 根节点对应base目录, 每一个节点对应一个目录, 每一个属性对应一个文件

cd /sys/firmware/devicetree

ls

2、// /proc/device-tree 是链接文件, 指向 /sys/firmware/devicetree/base

cd /proc/device-tree

ls

3、/\* /sys/devices/platform 系统中所有的platform\_device, 有来自设备树的, 也有来有.c文件中注册的对于来自设备树的platform\_device,可以进入 /sys/devices/platform/<设备名>/of\_node 查看它的设备树属性\*/

cd /sys/devices/platform/

cd led/of\_nod

ls 显示led节点信息:

compatible name pin

4、/sys/firmware/fdt // 原始dtb文件

hexdump -C /sys/firmware/fdt

# 4 u-boot对设备树的支持

## 4.1传递dtb给内核

### 4.1.1 u-boot中内核启动命令:

bootm <uImage\_addr> // 无设备树,bootm 0x30007FC0

bootm <uImage\_addr> <initrd\_addr> <dtb\_addr> // 有设备树

比如 :

nand read.jffs2 0x30007FC0 kernel; // 读内核uImage到内存0x30007FC0

nand read.jffs2 32000000 device\_tree; // 读dtb到内存32000000

bootm 0x30007FC0 - 0x32000000 // 启动, 没有initrd时对应参数写为"-"

Linux初始RAM磁盘（initrd）是在系统引导过程中挂载的一个临时根文件系统，用来支持两阶段的引导过程。initrd文件中包含了各种可执行程序和驱动程序，它们可以用来挂载实际的根文件系统，然后再将这个 initrd RAM磁盘卸载，并释放内存。在很多嵌入式Linux系统中，initrd 就是最终的根文件系统。

bootargs=noinitrd root=/dev/mtdblock4 rw init=/linuxrc console=ttySAC0,115200

bootcmd=nand read.jffs2 0x30007FC0 kernel; nand read.jffs2 32000000 device\_tree; bootm 0x30007FC0 - 0x32000000

mtdparts=mtdparts=nandflash0:256k@0(bootloader),128k(device\_tree),128k(params),4m(kernel),-(root)

### 4.1.2bootm命令怎么把dtb\_addr写入r2寄存器传给内核?

ARM程序调用规则(ATPCS)

c\_function(p0, p1, p2) // p0 => r0, p1 => r1, p2 => r2

定义函数指针 the\_kernel, 指向内核的启动地址,

然后执行: the\_kernel(0, machine\_id, 0x32000000);

### 4.1.3 dtb\_addr 可以随便选吗?

c.1 不要破坏u-boot本身

c.2 不要挡内核的路: 内核本身的空间不能占用, 内核要用到的内存区域也不能占用

内核启动时一般会在它所处位置的下边放置页表, 这块空间(一般是0x4000即16K字节)不能被占用

## 4.2 JZ2440内存使用情况:

------------------------------

0x33f80000 ->| u-boot |

------------------------------

| u-boot所使用的内存(栈等)|

------------------------------

| |

| |

| 空闲区域 |

| |

| |

| |

| |

------------------------------

0x30008000 ->| zImage |

------------------------------ uImage = 64字节的头部+zImage

0x30007FC0 ->| uImage头部 |

------------------------------

0x30004000 ->| 内核创建的页表 | head.S

------------------------------

| |

| |

-----> ------------------------------

|

|

--- (内存基址 0x30000000)

# 5 中断系统中的设备树

## 5.1 中断号的演变与irq\_domain

以前中断号(virq)跟硬件密切相关,现在的趋势是中断号跟硬件无关, 仅仅是一个标号而已

以前, 对于每一个硬件中断(hwirq)都预先确定它的中断号(virq),这些中断号一般都写在一个头文件里, 比如arch\arm\mach-s3c24xx\include\mach\irqs.h

使用时,

1、 执行 request\_irq(virq, my\_handler) :

内核根据virq可以知道对应的硬件中断, 然后去设置、使能中断等

2、发生硬件中断时,

内核读取硬件信息, 确定hwirq, 反算出virq,然后调用 irq\_desc[virq].handle\_irq, 最终会用到my\_handle

**怎么根据hwirq计算出virq？**

硬件上有多个intc(中断控制器), 对于同一个hwirq数值, 会对应不同的virq

所以在讲hwirq时，应该强调"是哪一个intc的hwirq",在描述hwirq转换为virq时, 引入一个概念: irq\_domain, 域, 在这个域里hwirq转换为某一个virq

**当中断控制器越来越多、当中断越来越多，上述方法(virq和hwirq固定绑定)有缺陷:**

a. 增加工作量, 你需要给每一个中断确定它的中断号, 写出对应的宏, 可能有成百上千个

b. 你要确保每一个硬件中断对应的中断号互不重复

**有什么方法改进?**

a. hwirq跟virq之间不再绑定 。virq还是和以前的virq一个意思，但是不需要宏定义绑定virq和hwirq

b. 要使用某个hwirq时, 先在irq\_desc数组中找到一个空闲项, 它的位置就是virq，再在irq\_desc[virq]中放置处理函数

新中断体系中, 怎么使用中断:

a.以前是request\_irq发起,

现在是先在设备树文件中声明想使用哪一个中断(哪一个中断控制器下的哪一个中断)

b. 内核解析设备树时,

会根据"中断控制器"确定irq\_domain,根据"哪一个中断"确定hwirq,

然后在irq\_desc数组中找出一个空闲项, 它的位置就是virq

并且把virq和hwirq的关系保存在irq\_domain中: irq\_domain.linear\_revmap[hwirq] = virq;

c. 驱动程序 request\_irq(virq, my\_handler)

d. 发生硬件中断时,

内核读取硬件信息, 确定hwirq, 确定 virq = irq\_domain.linear\_revmap[hwirq];

然后调用 irq\_desc[virq].handle\_irq, 最终会用到my\_handler

假设要使用子中断控制器(subintc)的n号中断, 它发生时会导致父中断控制器(intc)的m号中断:

a. 设备树表明要使用<subintc n>

subintc表示要使用<intc m>

b. 解析设备树时,

会为<subintc n>找到空闲项 irq\_desc[virq'], sub irq\_domain.linear\_revmap[n] = virq';

会为<intc m> 找到空闲项 irq\_desc[virq], irq\_domain.linear\_revmap[m] = virq;

并且设置它的handle\_irq为某个分析函数demux\_func

c. 驱动程序 request\_irq(virq', my\_handler)

d. 发生硬件中断时,

内核读取intc硬件信息, 确定hwirq = m, 确定 virq = irq\_domain.linear\_revmap[m];

然后调用 irq\_desc[m].handle\_irq, 即demux\_func

e. demux\_func:

读取sub intc硬件信息, 确定hwirq = n, 确定 virq' = sub irq\_domain.linear\_revmap[n];

然后调用 irq\_desc[n].handle\_irq, 即my\_handler

## 5.2在S3C2440上使用设备树描述中断体验

tar xzf linux-4.19-rc3.tar.gz

cd linux-4.19-rc3

patch -p1 < ../linux-4.19-rc3\_device\_tree\_for\_irq\_jz2440.patch

cp config\_ok .config

make uImage // 生成 arch/arm/boot/uImage

make dtbs // 生成 arch/arm/boot/dts/jz2440\_irq.dtb

nfs 30000000 192.168.1.124:/work/nfs\_root/uImage;

nfs 32000000 192.168.1.124:/work/nfs\_root/jz2440\_irq.dtb;

bootm 30000000 - 32000000

cat /proc/interrupts 查看注册的中断

a. 某个设备要使用中断, 需要在设备树中描述中断, 如何?

它要用哪一个中断? 这个中断连接到哪一个中断控制器去?

即: 使用哪一个中断控制器的哪一个中断?

至少有有2个属性:

interrupts // 表示要使用哪一个中断, 中断的触发类型等等

interrupt-parent // 这个中断要接到哪一个设备去? 即父中断控制器是谁

b. 上述的interrupts属性用多少个u32来表示?

这应该由它的父中断控制器来描述,

在父中断控制器中, 至少有2个属性:

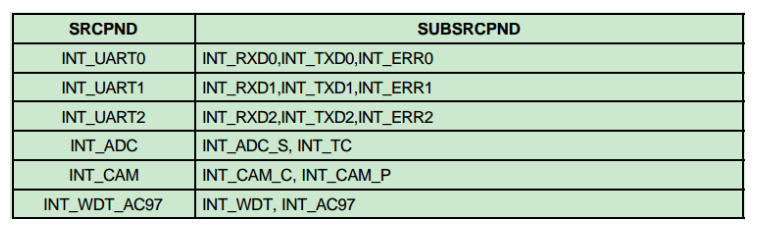
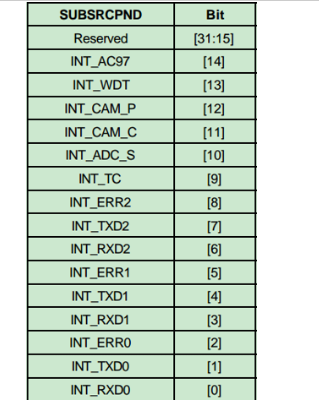
interrupt-controller; // 表示自己是一个中断控制器

#interrupt-cells // 表示自己的子设备里应该有几个U32的数据来描述中断

## 5.3 使用设备树描述按键中断

参考博客写的非常好：https://www.cnblogs.com/pengdonglin137/p/6847685.html

在设备树的设备节点中描述"中断的硬件信息",表明使用了"哪一个中断控制器里的哪一个中断, 及中断触发方式",设备节点会被转换为 platform\_device, "中断的硬件信息" 会转换为"中断号", 保存在platform\_device的"中断资源"里,驱动程序从platform\_device的"中断资源"取出中断号, 就可以request\_irq了。



gpf: gpf {

gpio-controller;

#gpio-cells = <2>;

interrupt-controller;

#interrupt-cells = <2>;

};

gpg: gpg {

gpio-controller;

#gpio-cells = <2>;

interrupt-controller;

#interrupt-cells = <2>;

};

修改jz2440\_irq.dts,添加按键信息：

buttons {

compatible = "jz2440\_button";

eint-pins = <&gpf 0 0>, <&gpf 2 0>, <&gpg 3 0>, <&gpg 11 0>;

interrupts-extended = <&intc 0 0 0 3>,

<&intc 0 0 2 3>,

<&gpg 3 3>,

<&gpg 11 3>;

};

把"002th\_buttons\_drv/jz2440\_irq.dts" 放入内核 arch/arm/boot/dts目录,

在内核根目录下执行:

make dtbs // 得到 arch/arm/boot/dts/jz2440\_irq.dtb

使用上节视频的uImage或这个jz2440\_irq.dtb启动内核;

编译、测试驱动:

把 002th\_buttons\_drv 上传到ubuntu编译驱动

cd 002th\_buttons\_drv

make // 得到 buttons.ko

b.3 编译测试程序:使用arm-linux-gcc 4.4.3

cd 002th\_buttons\_drv

arm-linux-gcc -o buttons\_test buttons\_test.c

b.4 测试:

insmod buttons.ko

./buttons\_test &

然后按键

使用dtc工具编译dts

内核配置make menuconfig中选择支持设备树

make menuconfig --->

Boot options --->

Flattened Device Tree support

DTC的源代码位于内核scripts/dtc目录，在linux内核使能设备树时，编译内核dtc会自动被编译出来。

在Linux内核的arch/arm/boot/dts/Makefile中，描述了当某种soc被选中后，哪些dtb文件会被编译出来，没有发现Makfefile 需要自己写？

dtb-$(CONFIG\_ARCH\_EXYNOS)+=exynos4210-origen.dtb+exynos4210-smdkv310.dtb+exynos4412-origen.dtb

我们可以单独编译Device Tree文件，当我们在linux内核下运行make dtbs时，若我们之前选择了ARCH-EXYNOS,上述dtb都会由对应的.dts编译出来。

1、cp arch/arm/boot/dts/exynos4412-origen.dts arch/arm/boot/dts/exynos4412-fs4412.dts

2、vim arch/arm/boot/dts/Makefile 在exynos4412-origne.dts\下面添加如下内容

exynos4412-fs4412.dtb

3、make dtbs

4、cp arch/arm/boot/dts/exynos4412-fs4412.dtb /tftpboot/

5、设置启动参数

set bootargs=console=ttySAC2,115200 init=/linuxrc root=/dev/nfs nfsroot=192.1681.111:/home/chao/work/rootfs ip=192.168.1.112 clk\_ignore\_unused

set bootcmd tftp 0x41000000 uImag\;tftp 0x42000000 exynos4412-fs4412.dtb\;bootm 0x41000000 - 0x42000000

从网络下载内核到内存地址0x41000000，下载exynos4412-fs4412.dtb到内存地址0x42000000，并且从0x41000000启动，将dtb地址0x42000000传给内核。

**dtb使用流程**

1. uboot加载内核
2. uboot加载dtb
3. uboot将dtb地址传送给内核后启动内核
4. 内核会到dtb地址处读取dtb并且展开为结构体

**名词**

DT: Device Tree

FDT: flattened Device Tree 展开设备树

OF: Open Firmware 开源协会

DTS: Device Tree source 设备树源文件

DTSI: Device Tree source include 设备树头文件

DTB: Device Tree blob 设备树二进制文件

DTC: Device Tree compiler 编译设备树的工具

**设备树节点和属性**

**节点 node：**

格式:<名称>[@<设备地址>]；

名称就是ascii字符串，应该体现的是什么样的设备，比如3com以太网适配器的节点名称应该为ethernet,而不应该是3com509。

设备地址用来访问该设备的主地址，并且该地址页在节点reg属性中列出，同级节点必须唯一，但是只要主地址不同，也可以使用一样的名称。设备地址是可选的。

**属性property**

键值对格式；它的值可以为空或着包含一个任意字节流。键其实就是属性名，可以自己命名。空属性一般是作为一个标志位，代表支持什么功能。

下面是设备树源文件支持的几个基本的数据表示形式：

文本字符串（无结束符）可以用双引号表示：string-property=”a string”

cells是32位无符号整数，用尖括号限定：cell-property=<0xbeeef 123 0xabcd1234>

二进制数据用方括号限定：binary-property=[01 23 45 87];

不同表示形式的数据可以使用都会连在一起：mixed-property=”a string”, [01 23 45 87], <0xbeeef 123>

字符串列表： string-list=”red fish”,”blue fish”

**常见属性**

compatible:指定系统名称。是一个字符串列表，实际代码中用于匹配。标识当前选择的是哪个机器。它包含了一个“<制造商><型号>”形式的字符串。重要的是要指定一个确切的设备，并且包括制造商名字，以避免命名冲突。

compatible = "fsl,imx6q-arm2", "fsl,imx6q";

#adress-cell 属性：

#adress-cell =<1>表示子节点address字段的长度为1

#size-cells 属性

#size-cells=<1>;表示子节点length字段的长度为1

reg属性：

格式：reg=<address1 length1 [address2 length2]>其中每一组address length表明了设备使用的一个地址范围。

中断属性

中断和soc的连接：外部中断---->通用中断控制器GIC---->SOC

GIC作用：对外设中断进行屏蔽，优先级设置。

描述中断连接需要的四个属性：

interrupt-controller :一个空的属性，定义该节点作为一个接收中断信号的设备。

#interrupt-cells 这是一个中断控制器节点的属性，它声明了该中断控制器的中断指示符中cell的个数（类似于#address-cells和size-cells）

interrupt-parent 这是一个设备节点的属性，包含一个指向该设备连接的中断控制器的phandle(指向或者可以引用&)那些没有interrupt-parent的节点则从它们的父节点中继承该属性。

interrupts:一个设备节点属性，包含个终端指示符的列表，对应于该设备上每个中断输出信号。如果是3个cell描述，依次是中断类型，中断号，触发方

对于arm架构

中断类型:0 ：spi interrupts共享中断，1 ： ppi

中断号：spi中断范围[0-987],ppi中断范围[0-15]

中断标志： 1=低电平到高电平边沿触发；2=高电平到低电平边沿触发

4=高电平触发； 8=低电平触发。

intc: interrupt-controller@00a01000 {

compatible = "arm,cortex-a9-gic";

#interrupt-cells = <3>;

#address-cells = <1>;

#size-cells = <1>;

interrupt-controller;

reg = <0x00a01000 0x1000>,

<0x00a00100 0x100>;

};

soc {

#address-cells = <1>;//指示子节点地址占一个cell

#size-cells = <1>;//指示子节点地址长度占一个cell

compatible = "simple-bus";

interrupt-parent = <&intc>; //指示了所有子节点中断的interrupt-parent

ranges;

timer@00a00600 {//soc子节点

compatible = "arm,cortex-a9-twd-timer";//这个timer名称

reg = <0x00a00600 0x20>;

interrupts = <1 13 0xf01>;//中断控制器的中断指示符中cell的个数3

};

}

解析dts节点的API

test\_nod@12345678{

compatible=”test,farsight”;

reg=<0xa2345678 0x24

0xb345678 0x24>;

testprop,mytest;

test\_list\_string=”red fish”,”blue fish”;

interrupt-parent=<&gpx1> //中断控制器gpx1

interrupts=<1 2>;//中断号1，触发类型2

}

**/\***

**中断源**🡪interrupt parent:**GPIO**🡪interrupt parent:**GIC1**🡪**CPU**

**gpx1就是gpio控制器1，gpx2 gpio控制器2，gpx1\_2 接到gpx1控制器的第2引脚**

**\*/**

系统启动后查看设备树是否解析成功：

cd proc/device-tree/

ls 可以查看到添加的所有设备节点文件夹，文件夹里面包含节点的所有属性。可以cat打印属性值

常用的OF API：

//根据路径查找节点,给出绝对路径

struct device\_node \*of\_find\_node\_by\_path(const char \*path)

//根据名字查找节点，from 从哪个节点开始查找，可以是NULL

struct device\_node \*of\_find\_node\_by\_name(struct device\_node \*from,const char \*name)

//根据property结构的name参数,在指定的device node中查找合适的propertys,

//lenp可以是NULL

struct property \*of\_find\_property(const struct device\_node \*np,const char \*name,int \*lenp)

//根据compat参数与device node的compatible匹配，返回匹配度

int of\_device\_is\_compatible(const struct device\_node \*device,const char \*compat)

//获得父节点的devece node

struct device\_node \*of\_get\_parent(const struct device\_node \*node)

//根据属性名Propname,读出该属性的数组中sz个属性值给out\_values。这个函数可以

//用来读取reg属性的地址和长度，可以依次读取多个属性值。

int of\_property\_read\_u32\_array(const struct device\_node \*np,const char \*propname, u32 \*out\_values,size\_t sz)

//读取该设备的第index个irq号

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

**在代码中使用API**

//linux/of.h

irqreturn\_t key\_irq\_handler(int irqno,void \*devid)

{

printk(“key has preesed”);

return IRQ\_HANDLED;

}

static int \_\_init dt\_drv\_init(void)

{

struct device\_node \*np=NULL

np=of\_find\_node\_by\_path(“/test\_nod@12345678”);

if(np) printk(“find node%s\n”,np->name);

struct property \*prop=NULL;

prop=of\_find\_property(np,”compatible”,NULL);

if(prop) printk(“find compatible = %s\n”,prop->value);

if(of\_device\_is\_compatible(np, ”test,farsight”))

printk(“find compatible ok\n”);

//获取数组

u32 regdata[4];

int ret;

ret=of\_propterty\_read\_u32\_array(np,”reg”, regdata,4);

if(ret==0)

{

printk( “get reg ok”\n);

for(i=0;i<4;i++)

printk(“regdata[%d]=%x\n”,i,regdata[i]);

}

//获取字符串列表

const char \*pstr[3];

int i;

for(i=0;i<3,i++)

{

ret= of\_property\_read\_string\_index(np,”test\_list\_string” ,i,&pstr[i]);

if(!ret) printk(“pstr[%d]=%s\n”,i,pstr[i]);

}

//有空属性作为标志位

if(of\_find\_property(np,”testprop,mytest”,NULL))

{

is\_good=1;

}

//中断获取< linux/of\_irq.h>

int irqno=irq\_of\_parse\_and\_map(np, 0);//获取该节点的第0个中断，得到的是中断号

printk(“irqnum=%d”,irqno);

ret=request\_irq(irqno,key\_irq\_handler,IRQF\_TRIGGER\_FALLING|IRQF\_TRIGGER\_RISING,”KEY\_IRQ”,NULL);

if(ret) printk(“request\_irq error\n”);

}

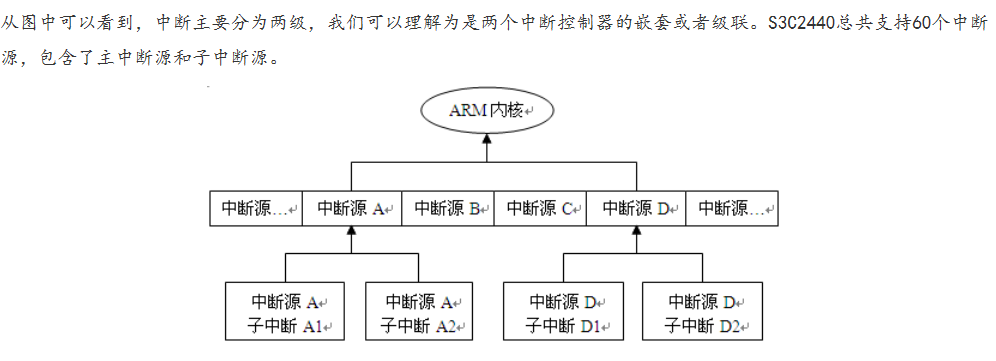
static void \_\_exit dt\_drv\_exit(void)

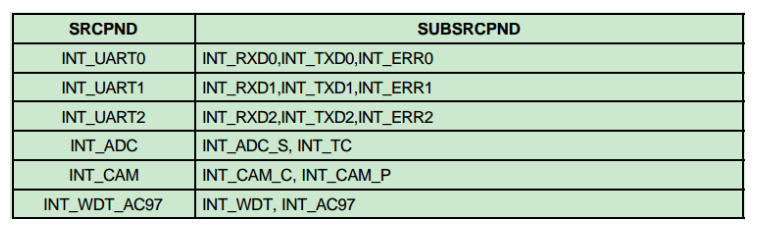
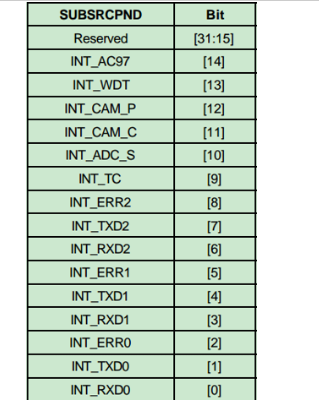
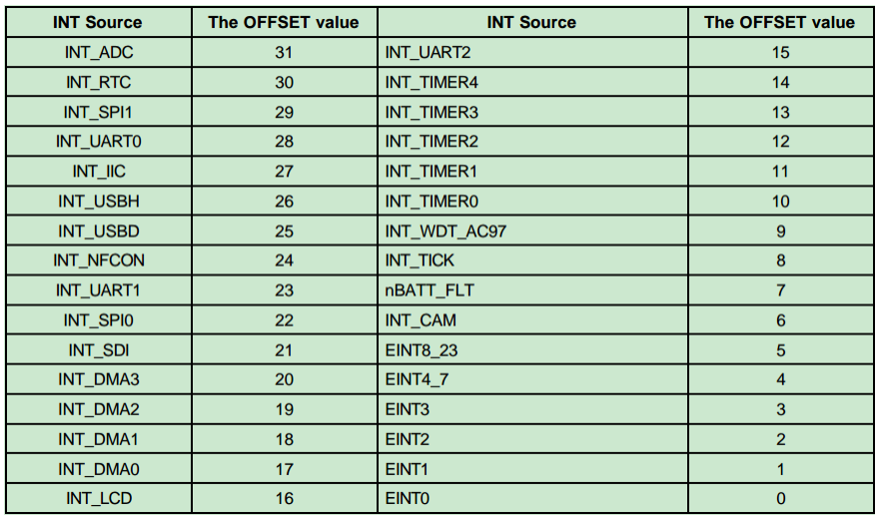
{

free(irqno,NUL);

}

[基于设备树的TQ2440的中断（1）](https://www.cnblogs.com/pengdonglin137/p/6847685.html)





s3c2440的中断控制器intc,在s3c24xx.dtsi中定义。

intc:interrupt-controller@4a000000

{ compatible = "samsung,s3c2410-irq";

reg = <0x4a000000 0x100>;

interrupt-controller;

#interrupt-cells = <4>;

};

//中断例子

serial@50000000

{ compatible = "samsung,s3c2440-uart";

reg = <0x50000000 0x4000>;

interrupts = <1 28 0 4>, <1 28 1 4>; //RXD0中断和TXD0中断

status = "okay";

clock-names = "uart";

clocks = <&clock PCLK\_UART0>;

pinctrl-names = "default";

pinctrl-0 = <0x3>;

};

i2c:i2c@54000000

{ compatible = "samsung,s3c2410-i2c";

reg = <0x54000000 0x100>;

interrupts = <0 0 27 3>;

#address-cells = <1>;

#size-cells = <0>; };

四个参数的意义：

从中断控制器的#interrupt-cells属性知道，要描述一个中断需要四个参数，每一个参数的含义需要由中断控制器的驱动来解释，具体是有中断控制器的irq\_domain\_ops中的xlate来解释，对于s3c2440就是drivers/irqchip/irq-s3c24xx.c中的s3c24xx\_irq\_xlate\_of。

分析s3c24xx\_irq\_xlate\_of函数可知道

第一个参数不能大于2，只能是0和1， 0表示主中断，1表示子中断

对于子中断，第二个参数才有意义，表示该子中断所隶属的主中断的硬件中断号。s3c2440有主中断和子中断之分。比如主中断INT\_UART0就有三个子中断INT\_RXD0,INT\_TXD0,INT\_ERR0

第三个参数表示的是硬件中断号，从这里知道，主中断的硬件中断是0~31，子中断的硬件中断是32及其以上

第四个参数表示的是中断类型，可以查看IRQ\_TYPE\_SENSE\_MASK定义，就知道含义： 27 // 1表示上升沿触发，2表示下降沿触发，3表示双边沿触发，4表示高电平触发，8表示低电平触发，12表示高低电平触发

以串口的<1 28 0 4>和i2c的<0 0 27 3>为例：

<0 0 27 3>：第1个0表示的是主中断，第2个数字0没啥用，第3个数字27表示硬件中断号，第4个数字3表示双边沿触发。从上面的图一中可以看到I2C的硬件中断号是27

<1 28 0 4>：第1个数字1表示的是子中断，第2个数字28表示的是uart0的主中断，从上面的图一中可以看到uart0的主中断是28，第3个数字0表示的是子中断的硬件中断号，也就是图二中INT\_RXD0的位号0， 第4个数字4表示的是高电平触发