# Linux驱动\_设备树

这里仅仅对2440上的设备树进行探索。

## 1. 为什么使用设备树

### 1.1设备树起源

Linus Torvalds在2011年3月17日的ARM Linux邮件列表宣称“this whole ARM thing is a f\*cking pain in the ass”，引发ARM Linux社区的地震，随后ARM社区进行了一系列的重大修正。在过去的ARM Linux中，arch/arm/plat-xxx和arch/arm/mach-xxx中充斥着大量的垃圾代码，相当多数的代码只是在描述板级细节，而这些板级细节对于内核来讲，不过是垃圾，如板上的platform设备、resource、i2c\_board\_info、spi\_board\_info以及各种硬件的platform\_data。读者有兴趣可以统计下常见的s3c2410、s3c6410等板级目录，代码量在数万行。

社区必须改变这种局面，于是PowerPC等其他体系架构下已经使用的Flattened Device Tree（FDT）进入ARM社区的视野。Device Tree是一种描述硬件的数据结构，它起源于 OpenFirmware (OF)。在Linux 2.6中，ARM架构的板极硬件细节过多地被硬编码在arch/arm/plat-xxx和arch/arm/mach-xxx，采用Device Tree后，许多硬件的细节可以直接透过它传递给Linux，而不再需要在kernel中进行大量的冗余编码。  
Device Tree由一系列被命名的结点（node）和属性（property）组成，而结点本身可包含子结点。所谓属性，其实就是成对出现的name和value。在Device Tree中，可描述的信息包括（原先这些信息大多被hard code到kernel中）：

CPU的数量和类别

内存基地址和大小

总线和桥

外设连接

中断控制器和中断使用情况

GPIO控制器和GPIO使用情况

Clock控制器和Clock使用情况

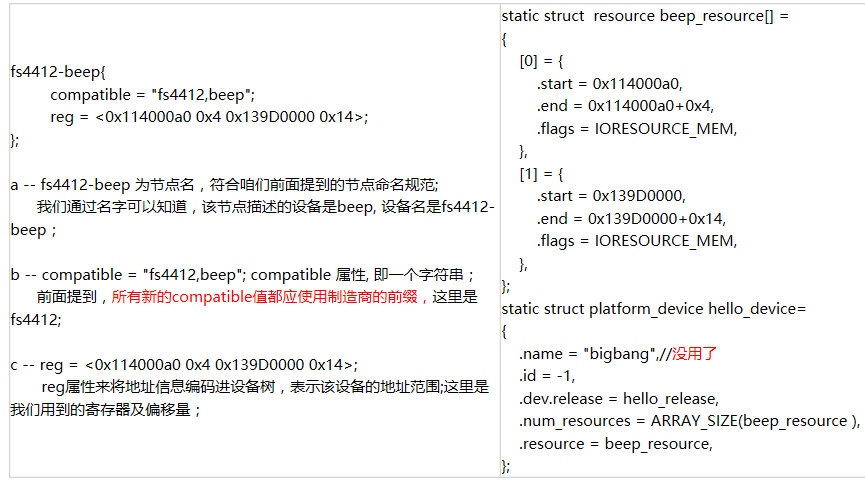
它基本上就是画一棵电路板上CPU、总线、设备组成的树，Bootloader会将这棵树传递给内核，然后内核可以识别这棵树，并根据它展开出Linux内核中的platform\_device、i2c\_client、spi\_device等设备，而这些设备用到的内存、IRQ等资源，也被传递给了内核，内核会将这些资源绑定给展开的相应的设备。

**从设备树导入的历史看，为了缓解不同开发板产生的大量冗余代码，使用一个树的形式将设备/总线等描述放在一个配置文件中。**

### 1.2 **设备树在platform设备驱动开发中的使用解析**

#### 1.2.1 **设备树对platform中platform\_device的替换**

其实我们可以看到，Device Tree 是用来描述设备信息的，每一个设备在设备树中是以节点的形式表现出来；而在上面的 platform 设备中，我们利用platform\_device 来描述一个设备，我们可以看一下二者的对比。



以看到设备树中的设备节点完全可以替代掉platform\_device。

### 1.2.2 **有了设备树，如何实现device与driver的匹配**

以前的platform\_device中，是利用.name 来实现device与driver的匹配的，但现在设备树替换掉了device，那我们将如何实现二者的匹配呢？有了设备树后，platform比较的名字存在哪？

利用"compatible"来匹配，即设备树加载之后，内核会自动把设备树节点转换成 platform\_device这种格式，同时把名字放到of\_node这个地方。

### 1.2.3 **基于设备树的driver的结构体的填充**

匹配的方式发生了改变，那我们的platform\_driver 也要修改了。

基于设备树的driver的结构体的填充：

static struct of\_device\_id beep\_table**[]** **=** **{**

**{.**compatible **=** "fs4412,beep"**},**

**};**

static struct platform\_driver beep\_driver**=**

**{**

**.**probe **=** beep\_probe**,**

**.**remove **=** beep\_remove**,**

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

**.**name **=** "bigbang"**,**

**.**of\_match\_table **=** beep\_table**,**

**},**

**};**

static struct of\_device\_id beep\_table[] = {

{.compatible = "fs4412,beep"},

};

static struct platform\_driver beep\_driver=

{

.probe = beep\_probe,

.remove = beep\_remove,

.driver={

.name = "bigbang",

.of\_match\_table = beep\_table,

},

};

对比原来的driver。

fs4412**-**beep**{**

         compatible **=** "fs4412,beep"**;**

         reg **=** **<**0x114000a0 0x4 0x139D0000 0x14**>;**

**};**

## 2. 在2440上导入设备树

1.编译设备树 make dtbs，生成s3c2440-tq2440-dt.dtb。

2.带设备树linux系统的启动

下載uImage：tftp 0x30008000 uImage**;**

下載ramdisk：tftp 0x31000000 ramdisk**.**img**;**

下載設備樹文件：tftp 0x33000000 s3c2440**-**tq2440**-**dt**.**dtb**;**

啓動：bootm 0x30008000 0x31000000 0x33000000

3.设备树结构

s3c2440**-**tq2440**-**dt**.**dts

**---->** s3c2440**.**dtsi

**---->** s3c24xx**.**dtsi

**---->** skeleton**.**dtsi

**---->** s3c2440**-**pinctrl**.**dtsi

skeleton.dtsi 存放的是一個設備樹必備的一些基本屬性。  
s3c24xx.dtsi 中存放的是整個s3c24xx系列SoC公共的一些屬性，如中斷控制器、串口、看門狗、RTC、I2C控制器等等。  
s3c2440-pinctrl.dtsi 存放的是s3c2440這款SoC中GPIO控制器、外部中斷控制器、引腳複用等信息的配置  
s3c2440.dtsi 存放的是s3c2440這個SoC跟其他s3c24xx系列不同的一些硬件信息，如clock控制器、串口等等。  
s3c2440-tq2440-dt.dts 存放的是tq2440的硬件信息。

4.修改arch/arm/boot/dts/Makefile

dtb**-**$**(**CONFIG\_ARCH\_S3C24XX**)** **+=** \

s3c2440**-**tq2440**-**dt**.**dtb

這樣在make dtbs編譯設備樹的時候就會編譯s3c2440-tq2440-dt.dts，在arch/arm/boot/dts/下生成s3c2440-tq2440-dt.dtb。

5.修改Makefile和Kconfig，添加tq2440板子的信息，以便在kernel啓動的時候能夠用從設備樹鏡像中解析到的信息匹配到tq2440板子。

修改arch/arm/mach-s3c24xx/Kconfig

config MACH\_TQ2440\_DT

bool "TQ2440 development board using device tree"

select CLKSRC\_OF

select USE\_OF

select PINCTRL

select PINCTRL\_S3C24XX

help

Say Y here to select support **for** the TQ2440**.**

修改arch/arm/mach-s3c24xx/Makefile

obj**-**$**(**CONFIG\_MACH\_TQ2440\_DT**)** **+=** mach**-**tq2440**-**dt**.**o

6.添加arch/arm/mach-s3c24xx/mach-tq2440-dt.c

#include <linux/clocksource.h>

#include <linux/irqchip.h>

#include <linux/serial\_s3c.h>

#include <asm/mach/arch.h>

#include <mach/map.h>

#include <plat/cpu.h>

#include <plat/pm.h>

#include "common.h"

static void \_\_init tq2440\_dt\_map\_io**(**void**)**

**{**

s3c24xx\_init\_io**(NULL,** 0**);**

**}**

static void \_\_init tq2440\_dt\_machine\_init**(**void**)**

**{**

s3c\_pm\_init**();**

**}**

static const char **\***const tq2440\_dt\_compat**[]** \_\_initconst **=** **{**

"samsung,s3c2440"**,**

"samsung,tq2440"**,**

**NULL**

**};**

DT\_MACHINE\_START**(**TQ2440\_DT**,** "Samsung S3C2440 (Flattened Device Tree)"**)**

**.**dt\_compat **=** tq2440\_dt\_compat**,**

**.**map\_io **=** tq2440\_dt\_map\_io**,**

**.**init\_irq **=** irqchip\_init**,**

**.**init\_machine **=** tq2440\_dt\_machine\_init**,**

MACHINE\_END

注意dt\_compat數組，其中的值要跟設備樹中的compatible匹配，这里为arch/arm/boot/dts/s3c2440-tq2440-dt.dts。

**/**dts**-**v1**/;**

#include "s3c2440.dtsi"

#include <dt-bindings/interrupt-controller/irq.h>

#include <dt-bindings/clock/s3c2410.h>

**/** **{**

model **=** "TQ2440"**;**

compatible **=** "samsung,s3c2440"**,** "samsung,tq2440"**;**

memory **{**

reg **=** **<**0x30000000 0x34000000**>;**

**};**

7.删除uboot的bootargs

如果uboot中設置了bootargs屬性的話，在boot的之前它會修改設備樹鏡像，覆蓋其中chosen節點中的bootargs屬性，爲了便於調試，我在uboot中執行setenv bootargs命令，這樣就可以刪除uboot中bootargs環境變量。

下面是設備樹（s3c2440-tq2440-dt.dts）中chosen節點的定義：

chosen**{**

bootargs **=** "root=/dev/ram0 rw rootfstype=ext2 console=ttySAC0,115200n8 init=/linuxrc ignore\_loglevel earlyprintk"**;**

**};**

## 3. 设备树的展开

（1）首先在内核入口处将从u-boot传递过来的镜像基地址。

（2）通过调用early\_init\_dt\_scan()函数来获取内核前期初始化所需的bootargs，cmd\_line等系统引导参数。

（3）根据bootargs，cmd\_line等系统引导参数进入start\_kernel()函数，进行内核的第二阶段初始化。

（4）调用unflatten\_device\_tree()函数来解析dtb文件，构建一个由device\_node结构连接而成的单项链表，并使用全局变量of\_allnodes指针来保存这个链表的头指针。

（5）内核调用OF提供的API函数获取of\_allnodes链表信息来初始化内核其他子系统、设备等。

## 4. 设备树文件解析

### 4.1 基本概念

1个dts文件+n个dtsi文件，它们编译而成的dtb文件就是真正的设备树。

soc厂商会把soc公共的特性和多块开发板公用的特性提炼为dtsi，而dts则负责描述某个具体的产品（开发板）的特性。dts直接或间接的包含多个dtsi（类似于c语言的头文件），就体现了一个完整的产品（开发板）所有的特性。

dts/dtsi兼容c语言的一些语法，能使用宏定义，也能包含.h文件。

**基本构造**

{}包围起来的结构称之为节点，dts中最开头的/ {}，称为根节点。节点的标准结构是xxx@yyy{…}，xxx是节点的名字，yyy则不是必须的，其值为节点的地址（寄存器地址或其他地址），比如i2c1: i2c@021a0000中的就是一个i2c控制器的寄存器基地址，rtc: pcf8523@68中的就是这个rtc设备的i2c地址。

**属性：地址**

有关节点的地址，比如i2c@021a0000，虽然它在名字后面跟了地址，但是正式的设置是在reg属性中设置的比如：reg = <0x021a0000 0x4000>; reg的格式通常为<address length>，0x021a0000是寄存器基地址，0x4000是长度。address 和length的个数是可变的，由父节点的属性#address-cells 和#size-cells 决定，比如节点i2c@021a0000的父节点是aips-bus@02000000，其#address-cells 和#size-cells均为1，所以下面的i2c节点的reg属性就有一个address 和length，而i2c节点本身#address-cells 和#size-cells 分别为1和0，所以其下的rtc: pcf8523@68 的reg属性就只有一个0x68（i2c地址）了。

**属性：兼容性**

如果一个节点是设备节点，那么它一定要有compatible（兼容性），因为这将作为驱动和设备（设备节点）的匹配依据，compatible（兼容性）的值可以有不止一个字符串以满足不同的需求，详见下一节。而根节点的compatible也是非常重要的，也就是"fsl,imx6dl"这个字符串，因为系统启动后，将根据根节点的compatible来判断cpu信息，并由此进行初始化。

**属性设置的套路**

一般来说，每一种设备的节点属性设置都会有一些套路，比如可以设置哪些属性？属性值怎么设置？那怎么知道这些套路呢，有两种思路。

第一种是抄类似的dts，比如我们自己项目的平台是4412，那么就可以抄exynos4412-tiny4412.dts、exynos4412-smdk4412.dts这类相近的dts。

第二种是查询内核中的文档，比如Documentation/devicetree/bindings/i2c/i2c-imx.txt就描述了imx平台的i2c属性设置方法；Documentation/devicetree/bindings/fb就描述了lcd、lvds这类属性设置方法。

**节点之间的联系**

节点与节点之间的关联，通常通过“标号引用”和“包含”来实现。

所谓标号引用，就是在节点名称前加上标号，这样设备树的其他位置就能够通过&符号来调用/访问该节点。例如下面代码ir\_recv节点中的gpio属性，就引用了gpio1标号处的节点。

ir\_recv**:** ir**-**receiver **{**

compatible **=** "gpio-ir-receiver"**;**

gpios **=** **<&**gpio1 2 1**>;**

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

pinctrl**-**0 **=** **<&**pinctrl\_hummingboard\_gpio1\_2**>;**

**};**

gpio1**:** gpio@0209c000 **{**

compatible **=** "fsl,imx6q-gpio"**,** "fsl,imx35-gpio"**;**

reg **=** **<**0x0209c000 0x4000**>;**

interrupts **=** **<**0 66 IRQ\_TYPE\_LEVEL\_HIGH**>,**

**<**0 67 IRQ\_TYPE\_LEVEL\_HIGH**>;**

gpio**-**controller**;**

#gpio-cells = <2>;

interrupt**-**controller**;**

#interrupt-cells = <2>;

**};**

包含则是最基本的方式，比如我们要在i2c1接口添加一个i2c外设，那么就必须要在i2c1下面添加一个节点，比如上面代码中的rtc: pcf8523@68 {}。

**&**i2c1 **{**

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

pinctrl**-**0 **=** **<&**pinctrl\_hummingboard\_i2c1**>;**

rtc**:** pcf8523@68 **{**

compatible **=** "nxp,pcf8523"**;**

reg **=** **<**0x68**>;**

**};**

**};**

除了名字和地址外，节点前边还可以有一个标签(label)，这个标签不是必须的，一般只有在别个地方需要引用这个节点时才会用标签标示这个节点，因为如果用全路径太繁琐了。如“i2c\_1: i2c@12C70000”中的i2c\_1就是一个标签。

如果一个节点是属性节点（即仅仅是作为属性被其他节点调用），那么它定义在哪里其实无所谓，重要的是调用的位置，比如lcd屏幕的时序，其实我们完全可以把它定义在其他犄角旮旯，然后在lcd节点下用&来调用它，这也是可以的。这有点类似于函数：在哪定义不重要，重要的是在哪调用。

### 4.2 s3c2440**-**tq2440**-**dt**.**dts

#### 4.2.1 头文件包含

#include "s3c2440.dtsi"

#include <dt-bindings/interrupt-controller/irq.h>

#include <dt-bindings/gpio/gpio.h>

其中irq.h内容如下：

#define IRQ\_TYPE\_NONE 0

#define IRQ\_TYPE\_EDGE\_RISING 1

#define IRQ\_TYPE\_EDGE\_FALLING 2

#define IRQ\_TYPE\_EDGE\_BOTH (IRQ\_TYPE\_EDGE\_FALLING | IRQ\_TYPE\_EDGE\_RISING)

#define IRQ\_TYPE\_LEVEL\_HIGH 4

#define IRQ\_TYPE\_LEVEL\_LOW 8

gpio.h内容如下：

#define GPIO\_ACTIVE\_HIGH 0

#define GPIO\_ACTIVE\_LOW 1

#define GPIO\_PUSH\_PULL 0

#define GPIO\_SINGLE\_ENDED 2

#define GPIO\_OPEN\_DRAIN (GPIO\_SINGLE\_ENDED | GPIO\_ACTIVE\_LOW)

#define GPIO\_OPEN\_SOURCE (GPIO\_SINGLE\_ENDED | GPIO\_ACTIVE\_HIGH)

#### 4.2.2 根目录

先对根目录的开头部分进行分析。

**/** **{**

//机型名称

model **=** "TQ2440"**;**

//用于系统启动匹配设备树

compatible **=** "samsung,s3c2440"**,** "samsung,tq2440"**;**

//别名，也就是&i2c\_gpio\_1就是i2c1

aliases **{**

i2c1 **=** **&**i2c\_gpio\_1**;**

**};**

//内存地址范围

memory **{**

reg **=** **<**0x30000000 0x34000000**>;**

**};**

//linux启动参数

chosen **{**

bootargs **=** "noinitrd root=/dev/mtdblock3 init=/linuxrc console=ttySAC0,115200n8 earlyprintk"**;**

**};**

**....**

**};**

#### 4.2.3 外部晶振频率设置

clocks **{**

compatible **=** "simple-bus"**;**

#address-cells = <1>;

#size-cells = <0>;

xti**:** oscillator@0 **{**

compatible **=** "fixed-clock"**;**

reg **=** **<**0**>;**

clock**-**frequency **=** **<**12000000**>;**

clock**-**output**-**names **=** "xti"**;**

#clock-cells = <0>;

**};**

**};**

这里会详细分析下这段dts内容如何和代码进行适配。

compatible = "simple-bus";这段代表什么意思呢，这里需要从初始化开始说起。

const struct of\_device\_id of\_default\_bus\_match\_table**[]** **=** **{**

**{** **.**compatible **=** "simple-bus"**,** **},**

**{** **.**compatible **=** "simple-mfd"**,** **},**

**{** **.**compatible **=** "isa"**,** **},**

#ifdef CONFIG\_ARM\_AMBA

**{** **.**compatible **=** "arm,amba-bus"**,** **},**

#endif /\* CONFIG\_ARM\_AMBA \*/

**{}** /\* Empty terminated list \*/

**};**

int of\_platform\_default\_populate**(**struct device\_node **\***root**,**const struct of\_dev\_auxdata **\***lookup**,**struct device **\***parent**)**

**{**

**return** of\_platform\_populate**(**root**,** of\_default\_bus\_match\_table**,** lookup**,** parent**);**

**}**

static int \_\_init of\_platform\_default\_populate\_init**(**void**)**

**{**

**...**

of\_platform\_default\_populate**(NULL,** **NULL,** **NULL);**

**...**

**return** 0**;**

**}**

arch\_initcall\_sync**(**of\_platform\_default\_populate\_init**);**

of\_platform\_populate函数的作用为遍历of\_allnodes中的所有节点，生成并初始化platform\_device结构。但是当compatible属性为simple-bus时，需要继续遍历子节点，并创建对应的platfrom\_device。流程如下：

of\_platform\_populate

1.of\_find\_node\_by\_path**(**"/"**);** //寻找到根节点

2.遍历根节点下的所有的节点，执行of\_platform\_bus\_create

2.1 **!**of\_get\_property**(**bus**,** "compatible"**,** **NULL)**

//忽略没有compatible属性的节点

2.2 of\_platform\_device\_create\_pdata //生成platform\_device

2.3 **!**of\_match\_node**(**matches**,** bus**)** //如果compatible属性不是"simple-bus"和"arm,amba-bus"则在返回，不继续遍历子节点。

2.4 对于"simple-bus"和"arm,amba-bus"设备要继续遍历子节点，并创建对应的platform\_device

回到dts代码片段，该代码会创建compatible属性为fixed-clock的设备，那么接下来需要寻找compatible属性为fixed-clock的driver。

static const struct of\_device\_id of\_fixed\_clk\_ids**[]** **=** **{**

**{** **.**compatible **=** "fixed-clock" **},**

**{** **}**

**};**

static struct platform\_driver of\_fixed\_clk\_driver **=** **{**

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

**.**name **=** "of\_fixed\_clk"**,**

**.**of\_match\_table **=** of\_fixed\_clk\_ids**,**

**},**

**.**probe **=** of\_fixed\_clk\_probe**,**

**.**remove **=** of\_fixed\_clk\_remove**,**

**};**

可以发现有fixed-clock的platform\_device和platfrom\_driver相匹配，probe函数被调用。

static int of\_fixed\_clk\_probe**(**struct platform\_device **\***pdev**)**

**{**

struct clk **\***clk**;**

clk **=** \_of\_fixed\_clk\_setup**(**pdev**->**dev**.**of\_node**);**

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

**return** 0**;**

**}**

static struct clk **\***\_of\_fixed\_clk\_setup**(**struct device\_node **\***node**)**

**{**

struct clk **\***clk**;**

const char **\***clk\_name **=** node**->**name**;**

u32 rate**;**

u32 accuracy **=** 0**;**

int ret**;**

//clock-frequency = <12000000>

//读取dts中的clock-frequency属性

//这个属性必须有

**if** **(**of\_property\_read\_u32**(**node**,** "clock-frequency"**,** **&**rate**))**

**return** ERR\_PTR**(-**EIO**);**

//dts中没有该属性

of\_property\_read\_u32**(**node**,** "clock-accuracy"**,** **&**accuracy**);**

//clock-output-names = "xti"

//读取dts中的clock-output-names属性

of\_property\_read\_string**(**node**,** "clock-output-names"**,** **&**clk\_name**);**

//注册频率固定的时钟

clk **=** clk\_register\_fixed\_rate\_with\_accuracy**(NULL,** clk\_name**,** **NULL,**

0**,** rate**,** accuracy**);**

//把整个时钟注册进全局链表of\_clk\_providers

ret **=** of\_clk\_add\_provider**(**node**,** of\_clk\_src\_simple\_get**,** clk**);**

**return** clk**;**

**}**

从代码中可以看出，根据dts中的时钟信息，platform\_driver从dts中获取到参数并进行配置。

在dts中，基本的数据类型如下，可以通过of\_property\_read系列函数读取。

text string（以null结束），以双引号括起来，如：

string**-**property **=** "a string"

cells 是32位无符号整形数，以尖括号括起来，如

cell**-**property **=** **<**0xbeef 123 0xabcd1234**>**

binary data 以方括号括起来，如：

binary**-**property **=** **[**0x01 0x23 0x45 0x67**];**

不同类型数据可以在同一个属性中存在，以逗号分格，如：

mixed**-**property **=** "a string"**,** **[**0x01 0x23 0x45 0x67**],** **<**0x12345678**>;**

多个字符串组成的列表也使用逗号分格，如：

string**-**list **=** "red fish"**,**"blue fish"**;**

主要的操作函数有：

of\_property\_read\_u32

of\_property\_read\_bool

of\_property\_read\_string

of\_property\_read\_u32\_array

#### 4.2.4 dm9000网卡配置

srom-cs4@20000000 {

compatible = "simple-bus";

#address-cells = <1>;

#size-cells = <1>;

reg = <0x20000000 0x8000000>;

ranges;

ethernet@20000000 {

compatible = "davicom,dm9000";

reg = <0x20000000 0x2 0x20000004 0x2>;

interrupt-parent = <&gpf>;

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

local-mac-address = [00 00 de ad be ef];

davicom,no-eeprom;

};

};

dm9000接在内存总线的第四个片选上。地址范围为0x20000000开始的的128M。根据4.1.3的经验，找到davicom,dm9000定义的platform\_driver。

static const struct of\_device\_id dm9000\_of\_matches**[]** **=** **{**

**{** **.**compatible **=** "davicom,dm9000"**,** **},**

**{** /\* sentinel \*/ **}**

**};**

static struct platform\_driver dm9000\_driver **=** **{**

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

**.**name **=** "dm9000"**,**

**.**pm **=** **&**dm9000\_drv\_pm\_ops**,**

**.**of\_match\_table **=** of\_match\_ptr**(**dm9000\_of\_matches**),**

**},**

**.**probe **=** dm9000\_probe**,**

**.**remove **=** dm9000\_drv\_remove**,**

**}**

当匹配后，probe函数调用，probe里面哪些代码和dts相关呢？

static int dm9000\_probe**(**struct platform\_device **\***pdev**)**

**{**

//dm9000\_parse\_dt

**if** **(**of\_find\_property**(**np**,** "davicom,no-eeprom"**,** **NULL))**

pdata**->**flags **|=** DM9000\_PLATF\_NO\_EEPROM**;**

//of\_get\_mac\_address

addr **=** of\_get\_mac\_addr**(**np**,** "local-mac-address"**);**

//获取寄存器地址，一个用于传输addr，一个用于传输data

db**->**addr\_res **=** platform\_get\_resource**(**pdev**,** IORESOURCE\_MEM**,** 0**);**

db**->**data\_res **=** platform\_get\_resource**(**pdev**,** IORESOURCE\_MEM**,** 1**);**

//获取中断信息

ndev**->**irq **=** platform\_get\_irq**(**pdev**,** 0**);**

**return** ret**;**

**}**

获取寄存器地址dts代码解析：

#address-cells = <1>;

#size-cells = <1>;

reg **=** **<**0x20000000 0x2 0x20000004 0x2**>**

从描述可以知道address的长度为1个32位整数，length长度也为1个32位整数。

所以这里可以得到两段地址：

**(**1**)**0x20000000**~**0x20000002

**(**2**)**0x20000004**~**0x20000006

中断相关dts代码解析：

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

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

上面表示dm9000使用了GPIOF7作为中断源。当调用platform\_get\_irq得到的是GPIO7对应的外部中断号。获取该中断号需要经历两个步骤，先从gpio-controller中获取相关信息，再将信息转换给interrupt-controller。真正的中断信息需要4个字节才能表示。**这个转换过程后面再分析**。

interrupt**-**parent这是一个设备节点的属性，用于表明当前设备的中断是属于哪一个interrupt**-**controller的，如果没有这个属性，则继承其父节点的interrupt**-**parent属性

interrupts 这是一个设备节点的属性，他是中断说明符列表，每一个中断说明符 表示此设备的一个中断信号输出。

上级中断dts代码为：

gpf**:** gpf **{**

gpio**-**controller**;**

#gpio-cells = <2>;

interrupt**-**controller**;**

#interrupt-cells = <2>;

**};**

intc**:**interrupt**-**controller@4a000000 **{**

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

reg **=** **<**0x4a000000 0x100**>;**

interrupt**-**controller**;**

#interrupt-cells = <4>;

**};**

interrupt-controller 这个属性没有值，他表示这个节点是一个接收中断信号的设备。

interrupt-cells这个属性是一个interrupt-controller节点的属性，他说明了这个interrupt-controller的每个中断说明符（interrupt specifier）有几个cells，类似#address-cells 与 #size-cells的作用。

interrupt-cells（在interrupt-controller@10140000节点中）的值是2，所以，每个中断说明符由两个cell数据组成。这个例子中用的是最常见的中断说明符 形式，第一个cell表示中断线的序号，第二个cell表示终端类型的flag（表示高有效，低有效。。。），对于不同的终端控制器，需要阅读对应的binding document来得知其 中断说明符 的格式。

标签"intc:"被加到了中断控制器的节点上，这个标签在父节点上创建了一个phandle，这个phandle就是父节点的interrupt-parent。所以这个中断控制器就成了系统所有子节点的默认终端控制器，只有当子节点明确的声明了其interrupt-parent才会被覆盖。

#### 4.2.5 模拟i2c

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

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

#address-cells = <1>;

#size-cells = <0>;

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

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

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

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

eeprom@50 **{**

compatible **=** "24c02"**;**

reg **=** **<**0x50**>;**

pagesize **=** **<**32**>;**

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

**};**

**};**

同样的套路，先找到compatible属性为i2c-gpio的platfrom\_driver。

static const struct of\_device\_id i2c\_gpio\_dt\_ids**[]** **=** **{**

**{** **.**compatible **=** "i2c-gpio"**,** **},**

**{** /\* sentinel \*/ **}**

**};**

static struct platform\_driver i2c\_gpio\_driver **=** **{**

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

**.**name **=** "i2c-gpio"**,**

**.**of\_match\_table **=** of\_match\_ptr**(**i2c\_gpio\_dt\_ids**),**

**},**

**.**probe **=** i2c\_gpio\_probe**,**

**.**remove **=** i2c\_gpio\_remove**,**

**};**

接下来probe函数被调用，需要分析probe函数哪些内容和dts相关。

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

**{**

//获取SDA SCL对应的gpio编号 然后再申请gpio

//of\_i2c\_gpio\_get\_pins

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

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

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

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

//获取模拟i2c时钟周期

of\_property\_read\_u32**(**np**,** "i2c-gpio,delay-us"**,** **&**pdata**->**udelay**);**

//注意模拟i2c的添加多了个bit 传输算法结构体使用i2c\_bit\_algo

i2c\_bit\_add\_numbered\_bus**(**adap**);**

//函数of\_i2c\_register\_devices中会遍历这个adapter对应的device node的child device node，

//这些child device node对应的就是挂载i2c bus上的板级外设的硬件信息

//然后调用of\_i2c\_register\_device，这个函数根据每个child device node的信息构造i2c\_board\_info，并调用i2c\_new\_device

of\_i2c\_register\_devices**(**adap**);**

**}**

这里有两个重点需要关注，如何根据gpios的内容获取gpio对应的number，**这个后面进行分析**，第二个是i2c adpater下设备是如何被添加的。

基于设备树的i2c adapter下的i2c设备节点添加过程如下：

i2c\_add\_numbered\_adapter

i2c\_add\_adapter

i2c\_register\_adapter

of\_i2c\_register\_devices

of\_i2c\_register\_device

of\_i2c\_register\_device是如何根据dts里面的信息创建i2c device的呢？

static struct i2c\_client **\***of\_i2c\_register\_device**(**struct i2c\_adapter **\***adap**,** struct device\_node **\***node**)**

**{**

//从compatible获取i2c设备的名称

of\_modalias\_node**(**node**,** info**.**type**,** **sizeof(**info**.**type**)**

//获取i2c设备的地址

of\_get\_property**(**node**,** "reg"**,** **&**len**);**

//根据获取到的信息注册i2c device

i2c\_new\_device**(**adap**,** **&**info**);**

**}**

#### 4.2.6 硬件i2c

**&**i2c **{**

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

samsung**,**i2c**-**max**-**bus**-**freq **=** **<**200000**>;**

eeprom@50 **{**

compatible **=** "24c02"**;**

reg **=** **<**0x50**>;**

pagesize **=** **<**32**>;**

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

**};**

**};**

i2c@54000000 **{**

compatible **=** "samsung,s3c2440-i2c"**;**

clocks **=** **<&**clocks PCLK\_I2C**>;**

clock**-**names **=** "i2c"**;**

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

pinctrl**-**0 **=** **<&**i2c0\_bus**>;**

**};**

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

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

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

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

#address-cells = <1>;

#size-cells = <0>;

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

**};**

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

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

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

**};**

硬件i2c的dts描述分散在几个文件中。在这里又出现一些不太清楚的设置。

首先是clocks设置，表示该device需要一个clock，名字为i2c,i2c取自clocks的PCLK\_I2C，代码如何根据这些信息得到clk信息。

然后是pinctrl设置，表示该device只有一种管脚配置，管脚配置的信息在i2c0\_bus里面，代码如何根据这些信息配置对应的gpio？

最后是interrupts配置，4个参数代表的是什么？如何根据这4个参数找到对应的中断。

这些疑问需要后面对clock/pinctrl/中断子系统进行分析，暂时了解使用过程。

这里还注意到status的属性为disabled，表示该节点对应的platform\_device并不会创建。不是dtsi文件中所有的节点都会被注册，在注册总线和设备时，会对dts节点的状态作一个判断，如果节点里面的status属性没有被定义，或者status属性被定义了并且值被设为“ok”或者“okay”，其他情况则不被注册到系统中。

回到代码，需要找到compatible属性为samsung,s3c2410-i2c的platform\_driver。

static const struct of\_device\_id s3c24xx\_i2c\_match**[]** **=** **{**

**{** **.**compatible **=** "samsung,s3c2410-i2c"**,** **.**data **=** **(**void **\*)**0 **},**

**{** **.**compatible **=** "samsung,s3c2440-i2c"**,** **.**data **=** **(**void **\*)**QUIRK\_S3C2440 **},**

**{},**

**};**

static struct platform\_driver s3c24xx\_i2c\_driver **=** **{**

**.**probe **=** s3c24xx\_i2c\_probe**,**

**.**remove **=** s3c24xx\_i2c\_remove**,**

**.**id\_table **=** s3c24xx\_driver\_ids**,**

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

**.**name **=** "s3c-i2c"**,**

**.**pm **=** S3C24XX\_DEV\_PM\_OPS**,**

**.**of\_match\_table **=** of\_match\_ptr**(**s3c24xx\_i2c\_match**),**

**},**

**};**

static int \_\_init i2c\_adap\_s3c\_init**(**void**)**

**{**

**return** platform\_driver\_register**(&**s3c24xx\_i2c\_driver**);**

**}**

同样的进入到probe函数并分析哪些代码和dts配置相关。

static int s3c24xx\_i2c\_probe**(**struct platform\_device **\***pdev**)**

**{**

//获取i2c支持的最大频率

of\_property\_read\_u32**(**np**,** "samsung,i2c-max-bus-freq"**,(**u32 **\*)&**pdata**->**frequency**);**

//获取名称为i2c对应的clk

devm\_clk\_get**(&**pdev**->**dev**,** "i2c"**);**

//获取reg中的寄存器地址并使用

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

i2c**->**regs **=** devm\_ioremap\_resource**(&**pdev**->**dev**,** res**);**

//获取pinctrl的default配置 并设置

//等同于devm\_pinctrl\_get\_select(dev, "default");

i2c**->**pctrl **=** devm\_pinctrl\_get\_select\_default**(**i2c**->**dev**);**

//获取interrupts = <0 0 27 3>对应的中断

i2c**->**irq **=** ret **=** platform\_get\_irq**(**pdev**,** 0**);**

devm\_request\_irq**(&**pdev**->**dev**,** i2c**->**irq**,** s3c24xx\_i2c\_irq**,**0**,** dev\_name**(&**pdev**->**dev**),** i2c**);**

**}**

#### 4.2.7 nandflash

**&**nand0 **{**

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

nand**,**tacls **=** **<**0xa**>;**

nand**,**twrph0 **=** **<**0x19**>;**

nand**,**twrph1 **=** **<**0xa**>;**

#address-cells = <1>;

#size-cells = <1>;

partitions **{**

/\* MTD partition table \*/

#address-cells = <1>;

#size-cells = <1>;

nr**-**chips **=** **<**1**>;**

set**-**name **=** "tq2440-0"**;**

partition@0 **{**

label **=** "SPL"**;**

reg **=** **<**0x0000000 0x100000**>;**

read**-**only**;**

**};**

partition@100000 **{**

label **=** "U-BOOT"**;**

reg **=** **<**0x0100000 0x100000**>;**

read**-**only**;**

**};**

partition@200000 **{**

label **=** "KERNEL"**;**

reg **=** **<**0x200000 0x300000**>;**

read**-**only**;**

**};**

partition@500000 **{**

label **=** "ROOTFS"**;**

reg **=** **<**0x500000 0x0**>;**

**};**

**};**

**};**

nand0**:** nand@4e000000 **{**

compatible **=** "**samsung,s3c2440-nand**"**;**

reg **=** **<**0x4e000000 0x40**>;**

interrupts **=** **<**0 0 24 3**>;**

clocks **=** **<&**clocks HCLK\_NAND**>;**

clock**-**names **=** "nand"**;**

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

pinctrl**-**0 **=** **<&**nand\_pinctrl**>;**

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

**};**

nand\_pinctrl**:** nand\_pinctrl **{**

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

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

samsung**,**pin**-**function **=** **<**1**>;**

**};**

compatible为samsung,s3c2440-nand对应的platform\_driver代码如下：

static const struct of\_device\_id s3c24xx\_nand\_ids\_of\_match**[]** **=** **{**

**{** **.**compatible **=** "samsung,s3c2410-nand"**,** **.**data **=** **(**void **\*)**TYPE\_S3C2410 **},**

**{** **.**compatible **=** "samsung,s3c2440-nand"**,** **.**data **=** **(**void **\*)**TYPE\_S3C2440 **},**

**{** **.**compatible **=** "samsung,s3c2412-nand"**,** **.**data **=** **(**void **\*)**TYPE\_S3C2412 **},**

**{** **.**compatible **=** "samsung,s3c6400-nand"**,** **.**data **=** **(**void **\*)**TYPE\_S3C2412 **},**

**{},**

**};**

static struct platform\_driver s3c24xx\_nand\_driver **=** **{**

**.**probe **=** s3c24xx\_nand\_probe**,**

**.**remove **=** s3c24xx\_nand\_remove**,**

**.**suspend **=** s3c24xx\_nand\_suspend**,**

**.**resume **=** s3c24xx\_nand\_resume**,**

**.**id\_table **=** s3c24xx\_driver\_ids**,**

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

**.**name **=** "s3c24xx-nand"**,**

#ifdef CONFIG\_OF

**.**of\_match\_table **=** of\_match\_ptr**(**s3c24xx\_nand\_ids\_of\_match**),**

#endif

**},**

**};**

module\_platform\_driver**(**s3c24xx\_nand\_driver**);**

当match后进入probe函数，依然需要分析probe函数如何解析dts中的相关信息。

static int s3c24xx\_nand\_probe**(**struct platform\_device **\***pdev**)**

**{**

//获取nandflash设置的参数

of\_property\_read\_u32**(**np**,** "nand,tacls"**,** **&**val**);**

of\_property\_read\_u32**(**np**,** "nand,twrph0"**,** **&**val**);**

of\_property\_read\_u32**(**np**,** "nand,twrph1"**,** **&**val**);**

//处理nandflash分区信息

ofpart\_node **=** of\_get\_child\_by\_name**(**mtd\_node**,** "partitions"**);**

of\_property\_read\_u32**(**ofpart\_node**,** "nr-chips"**,** **&**val**);**

of\_property\_read\_string\_index**(**ofpart\_node**,** "set-name"**,** 0**,** **&**name**);**

//对每个nandflash分区进行解析

for\_each\_child\_of\_node**(**ofpart\_node**,** pp**)**

**{**

//获取分区的末尾地址

of\_get\_property**(**pp**,** "reg"**,** **&**len**);**

//判断末尾地址是否4对齐

a\_cells **=** of\_n\_addr\_cells**(**pp**);**

s\_cells **=** of\_n\_size\_cells**(**pp**);**

**if** **(**len **/** 4 **!=** a\_cells **+** s\_cells**){**

**goto** ofpart\_fail**;**

**}**

//读出nandflash的offset和size

parts**[**i**].**offset **=** of\_read\_number**(**reg**,** a\_cells**);**

parts**[**i**].**size **=** of\_read\_number**(**reg **+** a\_cells**,** s\_cells**);**

//获取分区名字

partname **=** of\_get\_property**(**pp**,** "label"**,** **&**len**);**

//判断该分区是否只读

**if** **(**of\_get\_property**(**pp**,** "read-only"**,** **&**len**))**

parts**[**i**].**mask\_flags **|=** MTD\_WRITEABLE**;**

**}**

//获取nandflash使用的clock

info**->**clk **=** devm\_clk\_get**(&**pdev**->**dev**,** "nand"**);**

//获取dts中的寄存器地址

res **=** pdev**->**resource**;**

size **=** resource\_size**(**res**);**

info**->**regs **=** devm\_ioremap\_resource**(&**pdev**->**dev**,** res**);**

//并没有发现pinctrl的相关设置，因为在probe之前系统会默认设置default的pinctrl配置

//详情请见really\_probe--->pinctrl\_bind\_pins

**}**

#### 4.2.8 clocks

clocks: clock-controller@4c000000 {

compatible = "samsung,s3c2440-clock";

reg = <0x4c000000 0x20>;

#clock-cells = <1>;

};

详细分析见文章Linux驱动\_clk子系统。

#### 4.2.9

pinctrl如何保证gpio之间不冲突。

pinctrl 中断 gpio clock spi

pinctrl@56000000 {

compatible = "samsung,s3c2440-pinctrl";

};

http://blog.csdn.net/zqixiao\_09/article/details/50822753